DeSmet C

Development Package

Published and Distributed by

C WARE CORPORATION

Sunnyvale, California

DeSmet C

Development Package

Mark DeSmet

Published and Distributed by

C Ware Corporation

Sunnyvale, California

DeSmet C Development Package

Version 2.5 — October, 1985
Version 2.4 — October, 1984
Version 2.3 — April, 1984

Published by: C Ware Corporation
P.O. Box C
Sunnyvale, CA 94087
USA
(408) 720-9696
Telex 358185 C WARE SNVLE

DISCLAIMER OF WARRANTIES AND LIMITATION OF LIABILITIES

The author has taken due care in preparing this book and the programs and data on the electronic media accompanying this book including research, development, and testing to ascertain their effectiveness. The author and the publisher make no expressed or implied warranty of any kind with regard to these programs nor the supplemental documentation in this book. In no event shall the author or C Ware Corporation be liable for incidental or consequential damages in connection with or arising out of the furnishing, performance or use of any of these programs.

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 available. 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 Mike Ouye (pronounced Oh´Way) for converting the manual into MacWrite format, and both Glen Emigh and Pacific Data Works for proofreading its many revisions.

We especially thank the following for repeatedly throwing their bodies in front of and on top of the more explosive parts of the Version 2.5 Beta release — Clopper Almon, Dave Auslander, Glen Emigh, Panos Galidas, Scott Guthery, Bill Hunt, George O'Neal Keyes II, Scott Lewis, Greg Mansfield, Bill Morrison, and Jim Rome, as well as the folks at Kris Jamsa Software (Las Vegas, NV), Nanosoft (Ft. Belvoir, VA), and Pacific Data Works (Santa Monica, CA).

Table of Contents

5. The ASM88 8088/8086 Assembler

6. The BIND Object File Linker

7. The LIB88 Object File Librarian

8. The D88 C Language Debugger

9. Utility Programs

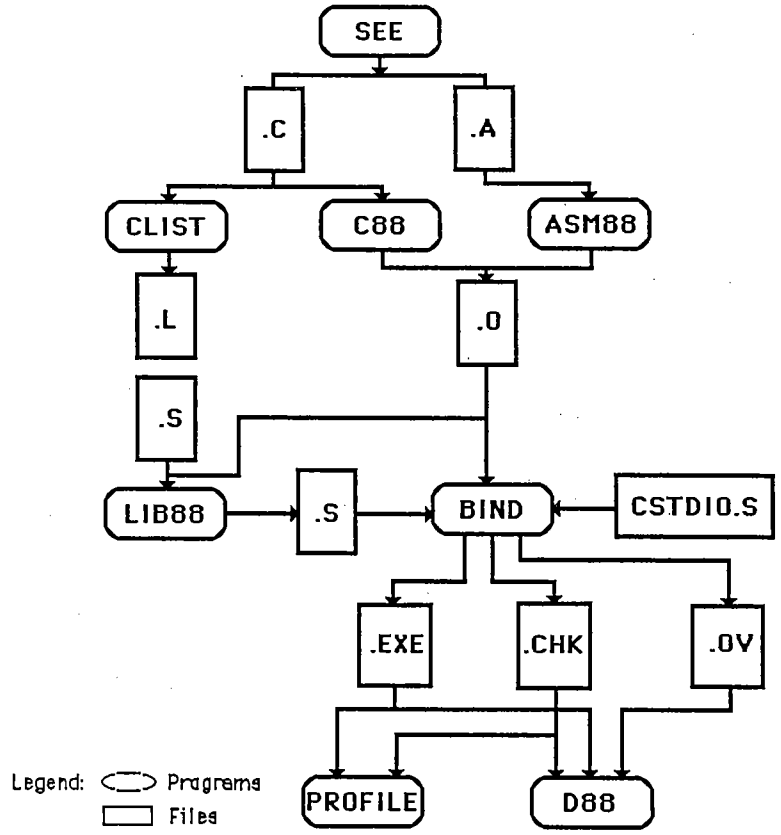
10. The CSTDIO.S Standard Library

Appendix A: Messages

Appendix B: The ASM88 Assembly Language

1. Introduction

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 MS-DOS, 1.xx, 2.xx, and 3.xx. The program execution profiler requires the use of MS-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]

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.

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].

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).

2. Getting Started

First things first. Copy all of the files from the distribution disks onto a set of working floppy diskettes or hard disk. The package is not copy-protected so the MS-DOS copy command can be used to copy the files. The package is distributed on two DOS 2 double-sided (360KB) diskettes. If your machine has single-sided drives, or only supports DOS 1, the package is available on "flippies" (each side of each diskette is formatted as a single-sided diskette and the diskette must be physically flipped over to access the other side). The distribution diskettes should never be used, they should be kept as the backup copy of the package.

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 and D88 must be configured before they can be used. See the notes in the file CONFIG.C on Disk #2 for configuration details.

2.1 Distribution Disks

The package is distributed on two 5 1/4 inch floppy diskettes, labeled Disk #1 and Disk #2. If you requested "flippies", Disks #3 and #4 are on the back sides of Disk #1 and #2, respectively. Disk #3 contains the rest of Disk #1, while Disk #4 completes Disk #2.

Disk #1:

C88.EXE:	The first pass of the C compiler.
GEN.EXE:	The second pass of the C compiler.
ASM88.EXE:	The assembler and third pass of the C compiler.
SEE.EXE:	The full-screen editor.
STDIO.H:	Include file for the Standard I/O package.
MATH.H:	Include file for the Standard Math package.
BIND.EXE:	The object file linker.
LIB88.EXE:	The object file librarian.
EXEC.O:	The Exec() and Chain() functions.
CSTDIO.S:	The standard C function library with software floating-point support.
CSTDIO7.S:	The standard C function library with 8087 support. To use this library, rename it to CSTDIO.S.

Disk #2:

RAM.COM:	RAM Disk driver for DOS 2.0 and later operating systems.
PCIO.A:	Source code for the PC screen functions.
BUF128.EXE:	128 byte type-ahead buffer program.
BUF128.A:	Source code for BUF128.EXE.
DUMP.EXE:	The hex file display utility.
DUMP.C:	Source code for DUMP.EXE.
LIFE.EXE:	Full screen game of Life.
LIFE.C:	Source code for LIFE.EXE.
BUGS!.EXE:	Arcade game (use 'BUGS! c' for color displays).
LATER.C:	Source code for a file modification date checking program.
CB.C:	Source code for a brace matching program.
CLIST.EXE	The C listing and cross-reference utility.
PROFILE.EXE:	The program execution profiler.
PROFSTAR.EXE:	Used by PROFILE.EXE.
PROFEND.EXE:	Used by PROFILE.EXE.
COMPARE.EXE:	The source code comparison utility.
D88.EXE:	The C source-level symbolic debugger.

Disk #2 also contains various object files and configuration files for generating the SEE editor and D88 debugger for systems with displays other than the IBM PC. CONFIG.C contains the display and keyboard dependent routines. See the comments in the CONFIG.C file for instructions on customizing SEE and D88.

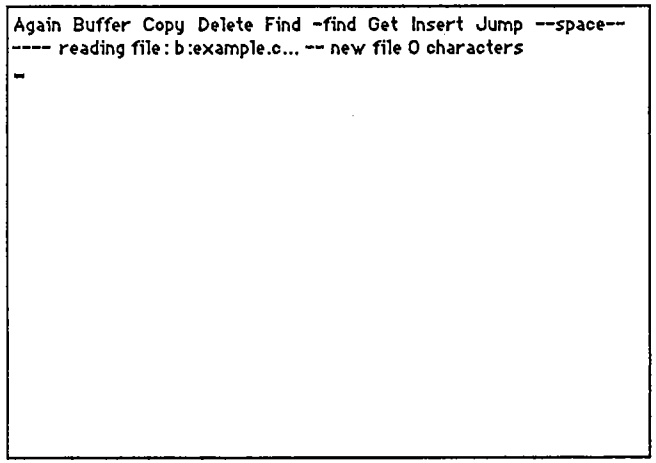
2. 2 A Short Example

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:



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);
    }
-
```

Now that the program is entered, it must be saved to the disk. Since the name of the file was specified on the command line, just type the sequence of characters, 'Q' for Quit and 'S' for Save-exit. This will store the copy of the file in memory onto the disk and exit from the editor.

To compile the program just entered, type:

c88 b:example

and the computer will display:

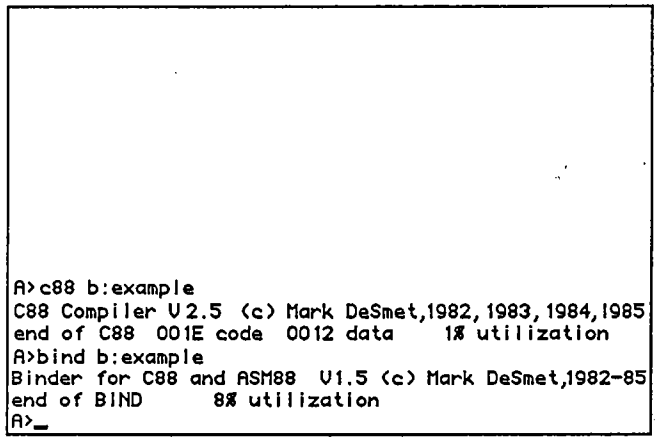
```
    }



A>c88 b:example
C88 Compiler V2.5 (c) Mark DeSmet,1982,1983,1984,1985
end of C88  001E code  0012 data    1% utilization
A>_
```

If there were errors during the compilation, go back to the editor and make sure that the program was entered correctly. (Note that the .C extension was not needed on the filename given to the compiler. When no extension is given, the compiler automatically assumes .C as the extension, just as the assembler assumes .A and the binder assumes .O).

Now it's time to bind (link) the program to the standard library and create the application program. Do this by typing:

bind b:example

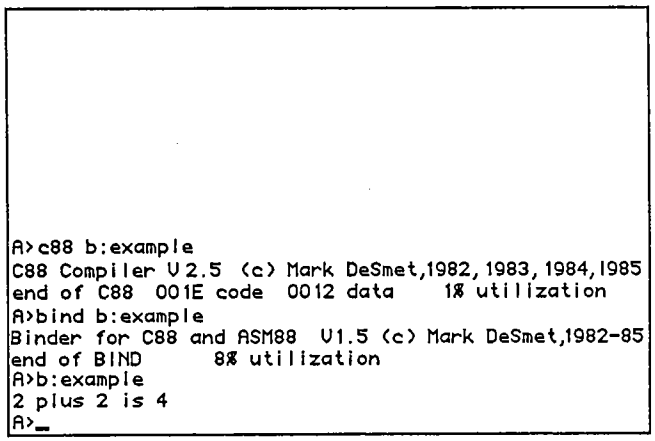
The binder will automatically search the CSTDIO.S library so it should not be included in the command line. When the binder successfully links the program, the screen should appear as follows:



The application is now ready to run. Type:

b:example

and the computer should respond with:



Don't worry if some of the utilization numbers are different from those shown in the example. These numbers will vary depending on the system being used (i.e. amount of memory, RAM disk installed, etc.).

2.3 Setting Up DOS 2.xx, 3.xx, ...

For systems utilizing DOS 2.x or later versions of the operating systems, make sure that the ASCII text file CONFIG.SYS exists on your boot disk. If it doesn't exist, you can create it with SEE. The file must contain the line:

FILES=20

since C88 supports 20 open files — stdin, stdout, stderr, and 17 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.

2.3.1 RAM DISK

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

copy the RAM.COM file from the distribution diskette to the system disk and add the line:

DEVICE=RAM.COM n

to the CONFIG.SYS file. The parameter, n, is a decimal number from 32 to 650 indicating the size of the Ram Disk. The value is specified in units of one kilobyte (1024).

Re-boot the system to install the Ram Disk. The drive letter used for this 'disk drive' is dependent on the configuration of the system. DOS will install the Ram Disk at the first free device "slot". For an IBM PC with two floppies, this will probably be drive C:. For an XT, it will probably be drive D:. Sanyo 550/5 reserves the first four slots for its floppies, so the Ram Disk is drive E:. To find where DOS has installed the Ram Disk, use

chkdsk x:

where x takes on the values c, d, ... You will get either a disk error, or a return showing the size of the Ram Disk. Once you find it, the Ram Disk will always be the same until you add other device drivers before it in the CONFIG.SYS file.

2.3.2 Completion Codes

The C88, ASM88, BIND and LIB88 programs set the completion code to:
zero if no warnings or errors occurred,
one if warnings were issued, and
two 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 LATER 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.

3. The SEE™ Text Editor

3.1 Introduction

SEE is a general purpose full-screen text editor designed for program entry rather than word processing. It handles files larger than memory and can edit two files simultaneously. Its macro facility allows you to capture a series of keystrokes and replay them to ease repetitive tasks. It also features automatic indentation and 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 MS-DOS but have different keyboard and/or screen interfaces than the IBM-PC (see Section 3.6).

3.2. Getting Started

3.2.1 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 block of text is deleted with the Delete command, the text is placed in a special area known as the copy buffer. 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.

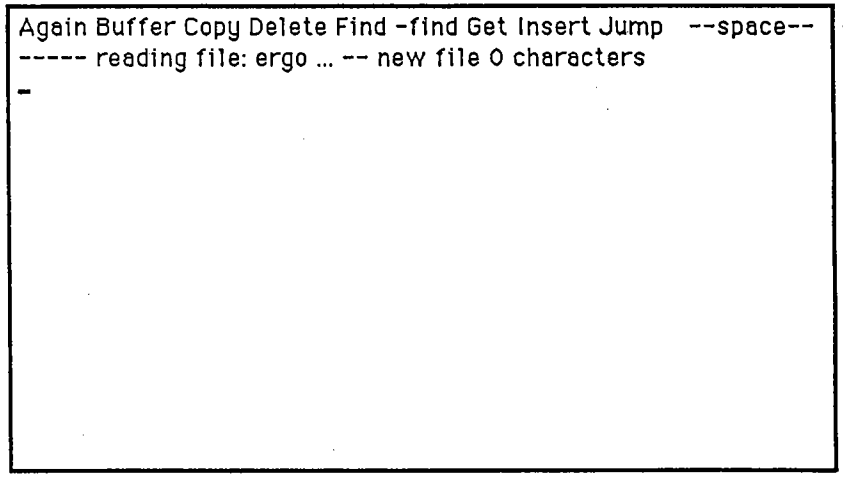
The cursor indicates the location where all action will occur. It will be in one of three states: a double-bar cursor indicating command mode, a single-bar cursor indicating Insert mode or a block cursor indicating Exchange mode. The cursor is always visible on the screen. As the cursor is moved to an 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.

3.2.2 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:



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--

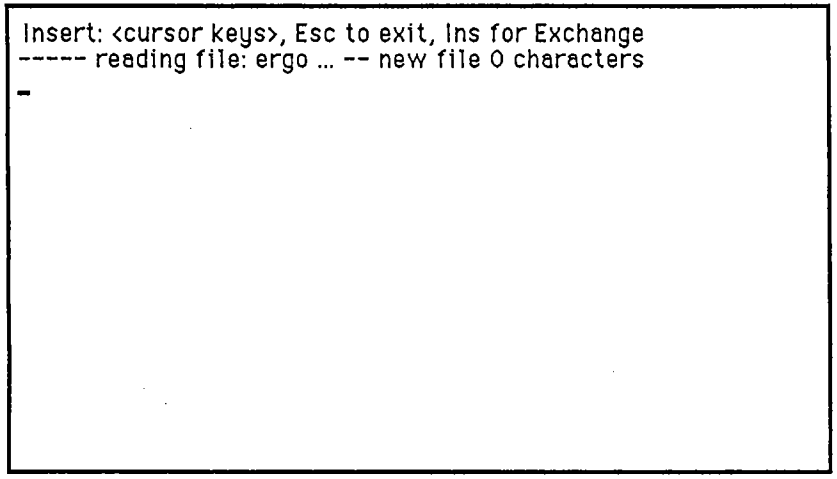
Hitting the space bar again will redisplay the first set of commands. The commands are fully described in Section 3.5 of this manual. Each command may be executed by typing the first letter of its menu item; for example, A for Again, B for Buffer, etc. The case of the command letter is not important.

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.

3.2.3 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:

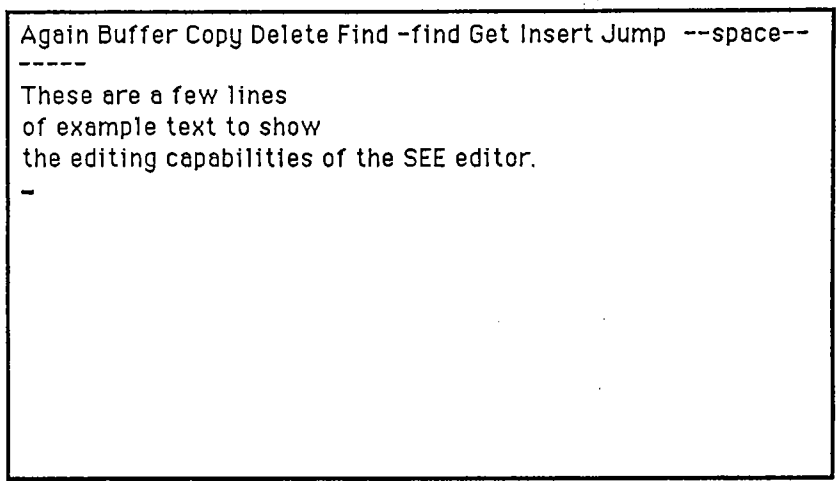


Note that the menu line has changed to indicate the types of actions, other than inserting text, that may be performed. Any character now typed, except for one of the special keys described in Section 4, will now be inserted into the text at the cursor location, just prior to the character that the cursor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

Now type in the lines:

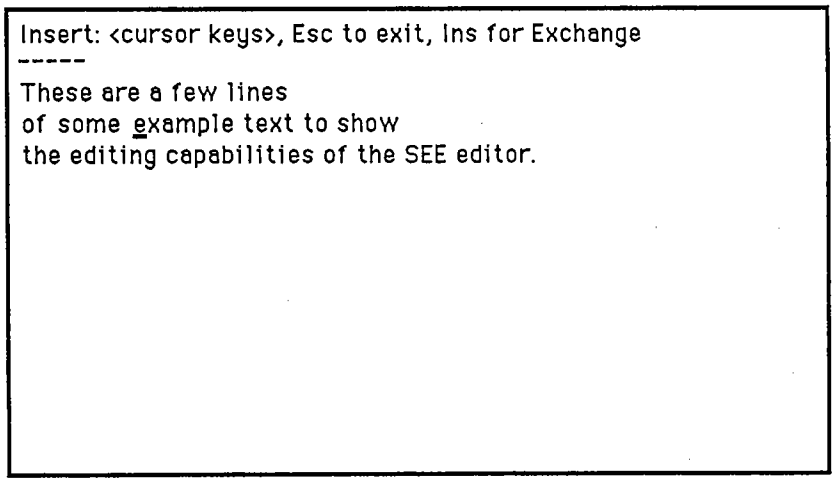
These are a few lines <Return>
of example text to shoe<Backspace>w<Return>
the editing capabilities of the SEE editor. <Return>
<Esc>

The screen should now look as follows:



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:



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 type 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.
@-
```

3.2.4 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 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.

3.2.5 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.

3.3. 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.

3.4. 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.

3.4.1 Cursor Movement Keys

In the following descriptions, the up-arrow (^) character preceding the name of the key implies 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 the cursor to the fourth line of the previous screenful of text. The next screen starts from the current screens fourth line from the bottom.

PgDn: Moves the cursor to the fourth line of the next screenful of text. The previous screen overlaps the current screen with the first four lines of the current screen.

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 last 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.

3.4.2 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 end of line sequence is removed and the two lines are joined to form a single line.

Del: The delete key deletes the character under 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.

^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.

3.4.3 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

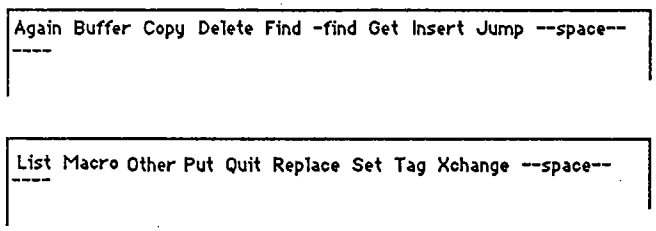
exit

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.

3.5. 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.



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.

<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 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:

Buffer: Esc to exit 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: Esc to exit 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, enter 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 also 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.

<rep> List

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 may be listed many times if it contains the search string more than once. 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.

Macro

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 typed, the macro associated with the key is replayed. Note that macro keys and the Macro command cannot 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 dirctory 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.

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: Esc to exit 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 the backquote 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 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 this is a new file being edited, no .BAK file is created.

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.

<rep | ?> Replace

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 next 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.

Set

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 the menu and message line:

```
Set: Auto-ins Case Indent PC Right-col Spill Tabs Word-wrap '{'-indent
---- off      yes  yes         72       @   4    off       off
```

The message line (the line below the menu line) contains the current settings of the controls. 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 backquote 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.

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:

Cursor-height Foreground-color Background-color

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 to disable this feature. With the color graphics adapter, enter 8, with the monochrome adapter, enter 12.

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-col: The Right-col 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.

Spill: The Spill control determines the drive on which the editor's spill files will be created. The '@' indicates the use of the current default

drive. To set the drive, simply select this item by typing 'S' and the prompt:

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

will appear. 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: The Tabs control determines the expansion factor of tab characters in the text. 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 nine indicating that the tab stop locations will be separated by one to nine 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 (set by the Right-col control).

'{'-indent: This is a special indentation mode for assisting in C programming. 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");
                }
        }
```

To change the brace indentation mode, type the left brace ({) character and type:

0: to turn brace indentation off
1: to set indentation mode 1
2: to set indentation mode 2

Tag

The Tag command is used to set markers in the text file. Once the 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 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 should not be entered. The repetition count determines the number of times the command will prompt for input.

3.6 Configuration

Distributed with the package, are a number of files used to reconfigure the editor to run on other MS-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.

4. The C88 C Compiler

4.1 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 model employed by the compiler efficiently utilizes the 8088/8086 architecture but limits a program to 64KB code and 64KB of data.

4.2 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.

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'.

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.

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 section 4.4.1 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 from the optional DOS LINK package (not included with the DeSmet C Development Package) must be in the same drive/directory as the GEN.EXE and ASM88.EXE files.

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 begining of the program. To set <defname> to one, 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'.

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 temporaries. Also, if the RAM Disk has been installed, placing the temporary files there will drastically cut the amount of time needed to compile a program.

4.3 Examples

```
C88 blip
```

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

```
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 -ic:\inc\ -a -nNewVersion -nNYear=1985
```

compiles the file named blip.c. Include files are taken from the directory c:\inc\. An assembly language file is generated named blip.a. The 'N' options are equivalent to adding

```
#define NewVersion 1
#define NYear      1985
```

to the start of blip.c

4.4 The C88 Language

C88 compiles C programs that conform to the standard definition of the C language as described in the book *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie. The following documentation describes the implementation.

4.4.1 Preprocessor directives

`#define`
defines a macro with or without parameters.

`#undef`, **`#ifdef`** and **`#ifndef`**
tests the status of the #defined macros

`#include`
includes other files into the program. #include's can be nested to a maximum depth of 3.

#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 DOS 2.X/3.X SET command) is searched for the variable **INCLUDE** . If the variable is found, it is assumed

to contain a set of directory prefixes separated by semi-colons. For example, if INCLUDE is set as follows

```
C>set include=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
```

`#if, #else, #endif`
conditionally includes or excludes source statements.

4.4.2 Data Types

`char` Unsigned byte with a range of 0 to 255.

`int`
`short` Signed Integer with a range of -32768 to 32767.

`unsigned` Unsigned integer with a range of 0 to 65535.

`long` Signed integer with a range of -2147483648 to 2147483647.

`float` Four byte floating point value. A float number has about 7 digits of precision and has a range of about 1.E-36 to 1.E+36. The floating point formats are defined by the IEEE floating-point standard.

`double` Eight byte floating point value. A double number has about 13 digits of precision and a range of about 1.E-303 to 1.E+303.

`(pointer)` pointers are two bytes, limiting total data space to 64KB.

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.

4.4.3 Extensions

The UNIX Version 7 extensions — enumerated types, extended member name-space, and structure assignment are fully supported.

Enumerated types provide 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 = green;
```

is equivalent to

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

Extended member name-space relaxes the K&R requirement that if a member name appeared in more than one structure or union, then it had to have the same data type and offset in every appearance. Now, the only restriction is that the member name must be unique within the structure or union. If a member name is used that is not in the referenced structure, the warning

`member not in structure`

is issued. As a loophole, a pointer to `char` may be used as an anonymous pointer to any structure. **CAUTION:** D88 doesn't know about the extended member name-space yet — it might use the wrong one.

```
struct {int i, j, k;} zip;
struct {int j; char i;} zap, *zp;
char *cp;
zip.i = 1;      /* OK */
zap.i = 1;      /* OK */
zap.k = 1;      /* WARNING */
zp->i = 1;      /* OK */
zp->k = 1;      /* WARNING */
cp->k = 1;      /* OK, ANONYMOUS */
```

Structures can be assigned, used as parameters, or returned from a function.
CAUTION: this may create problems with existing programs because previous versions of C88 converted the name of a structure in a parameter list to a pointer to that structure, while the current release pushes the entire structure. To highlight this potential problem, C88 will issue the following warning

structure assignment

when structures are passed by value, and the warning

returns structure

when a function returns a structure. These warnings will be removed in a future release.
CAUTION: D88 doesn't support structure assignment.

```
struct z {int i, j;} zip, zap, zxax();
main(){
     zip = zap;  /* structure assignment */
     zap = zmax(zip, zap);
     }
struct z zmax(a,b) /* func returns struct */
struct z a, b; {
     if(a.i > b.i)
          return a;
     return b;
     }
```

Variable names are significant to 31 characters instead of 8.

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. The next line beginning with the "#' character, ends the in-line assembly code. For example:

```
_move(count, src, tar)
int count; char *src, *tar; {
#asm
     MOV       CX,[BP+4] ;count
     MOV       SI,[BP+6] ;src
     MOV       DI,[BP+8] ;dst
     MOV       AX,DS
     MOV       ES,AX
     CLD
     REP MOVSB
#
     }
```

4.4.4 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;
    }
```

4.4.5 Externs

The rules for 'extern' declarations are:

1. Statements that are global to the source file, like 'int blip;' may be in several different files that are linked together. The BINDer will allocate 2 bytes for the global integer variable *blip*. This is an extension to the standard rule that restrict global declarations to a single file and require all the other declarations to be `extern`.

2. 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:

If the variable is not initialized (which means it will be initialized with zeros) either include the declaration:

```
int blip;
```

in every file or include the declaration:

```
extern int blip;
```

in every file and add the declaration:

```
int blip;
```

to one of the files to actually allocate the storage.

If the variable needs initialization, the second approach must be used. Include the declaration:

```
extern int blip;
```

in the include file. Initialize the value in only one file:

```
int blip = 1985;
```

These rules are about the same as Version 7 UNIX. Extern rules are an area of C that are currently controversial. System V UNIX tried to tighten up the rules but enough people complained that 5.2 is back to normal.

4.4.6 Macros

Macro arguments are not replaced within a quoted string. For example, in *The C Puzzle Book* by Alan Feuer the macros in <defs.h> use the following construct to customize printf() calls.

```
#define PR(fmt,v)printf("value=%fmt%t",v);
```

This does not work with the C88 compiler. Instead add the following defines to <defs.h>:

```
#define D "value = %d%t"
#define F "value = %f%t"
#define C "value = %c%t"
#define G "value = %g%t"
```

and change the PR define to

```
#define PR(fmt,v)printf(fmt,(v));
```

Statements of the type

```
PRINT1(d,x);
```

must be changed to

```
PRINT1(D,x);
```

in the programs. Lower case D, F, C, and G's would allow the programs to remain unchanged but variables c and g are used in structure one and variable g is used in structures two and three.

4.4.7 Strings

Literal character strings delimited by quotes ('\"') cannot contain the NUL character ('\0'). The compiler terminates the string at the NUL character, even though it checks that the string has a terminating quote character. If you want NUL characters in the string for initialization purposes, use an array assignment.

```
char init1[]="abcdef@xyz@012", *ip=init1;

while(ip = index(ip, '@'))
     *ip = '\0';
```

5. The ASM88 8088 Assembler

5.1 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.

5.2 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.

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.

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.

5.3 Examples

```
asm88 blip
```

assembles the file named blip.a and produces an 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

6. The BIND Object File Linker

6.1 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.

6.2 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. The '-S' option can be used to limit this size for use with `exec()`.

-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. The underscore option restores these *publics* to the listing. This option is useful when you need to see all the modules bound to your program.

6.3 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.

6.4 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.

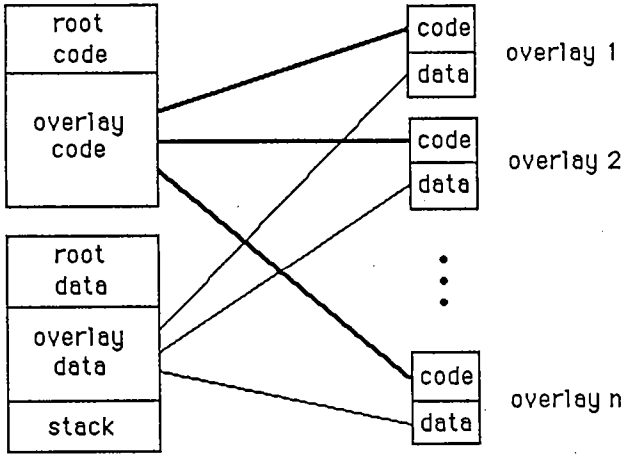
If BIND reports that the code limit is exceeded, look in the publics map for the scanf() and printf() routines. These are relatively large routines (around 2KB each) and also link in the floating-point routines. Eliminating the use of these routines can result in a large savings. If scanf() and/or printf() 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.

6.5 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.

6.6 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.

7. The LIB88 Object File Librarian

7.1 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.

7.2 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.

7.3 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.

7.4 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.

8. The D88 C Language Debugger

8.1 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 or DDT86 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.

8.2 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 in 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.

8.3 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 printed.

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.
rubout	Deletes the character to the left of the cursor.

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

8.4 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`

A subset 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.

8.5 D88 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 all of 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 causes the user screen to be restored and D88 exits. If the program has been EXITed or interrupted with control break, you cannot `Quit` or `Initialize`. Otherwise, typing 'I' causes 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+9
```

9. Utility Programs

9.1 Dump

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......*
```

9.2 CLIST

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 number sign, '#', following the 'main' symbol in the cross-reference listing indicates that the symbol was declared on that line. Unfortunately, clist is stupid and only supplies the declaration line information for procedures or data declarations which begin in the first column.

9.3 PROFILE

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 programs 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 information. 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.

10. The CSTDIO.S Standard Library

10.1 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 MS-DOS Operating System.

All of the routines are in the CSTDIO.S library file provided on the distribution disk. This file must be 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 for BIND to execute correctly.

There is a CSTDIO7.S library that has the same functions as the CSTDIO.S library but assumes the availability of the 8087 math chip to perform the floating-point operations. To use the 8087, rename CSTDIO7.S to CSTDIO.S.

10.2 Names

Public names starting with the underline character ('_') are reserved for C88 internal routines and should be avoided. Names of this form are also employed for user callable routines such as `_move()` whose names might conflict with user names.

C88 automatically appends the underline character ('_') to the end of 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. BIND ignores the case of publics, unlike UNIX, so `puts_` matches the name `PutS_`.

10.3 Program Initialization

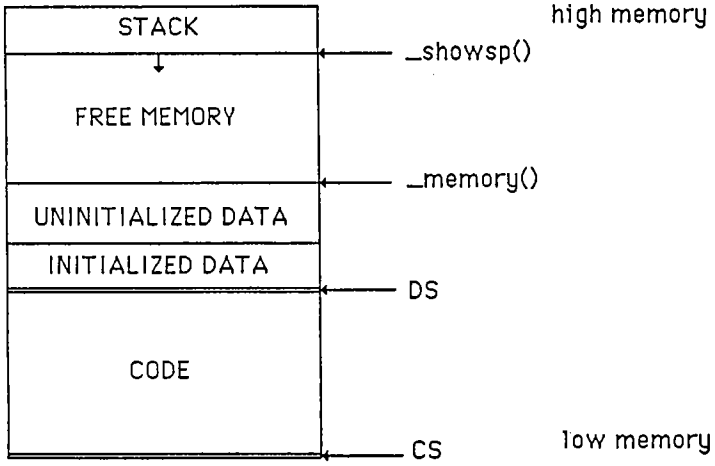
BIND inserts a `jmp _csetup` as the first executable instruction in the program. _CSETUP performs the following initialization functions:

1. Sets the data/stack segment size to the lower of: available memory, 64K, or the size of the static data area plus the BIND -S option,
2. Formats `argc` and `argv[]` from the Program Segment Prefix,
3. Zeros the Uninitialized Data Area, and
4. Calls `main(argc, argv)`

Assembly language main programs that require normal initialization should contain the following:

```
        PUBLIC MAIN_
MAIN_:
```

The initialization code will set the SS, DS and SP registers so that there is the largest possible stack unless the BIND '-S' option is used to restrict the stack size. The stack grows towards the unitialized data area. The figure below shows the memory layout after the initialization code has run:



The memory between the end of the uninitialized area and the stack is not normally used by C88 or the program (unless the program needs an inordinate amount of stack). This area is considered to be free memory. This memory area can be accessed directly by using the `_memory()` and `_showsp()` routines to calculate its extent. Another way to access this memory is to use the `malloc()` routine. Do <u>not</u> use both methods. Remember to leave enough space for the stack to grow.

The -A option of BIND inhibits the call to `_csetup`. Execution commences with the first instruction of the first filename specified to BIND. 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()
```

10.4 Calling Conventions

Arguments to a function are pushed on the stack, rightmost argument first. It is the responsibility of the calling function to clean up the stack. For example

```
int i;

zip(i, 6);
```

would generate the following code

```
mov        ax,6
push       ax
push       word i_
public zip_
call       zip_
add        sp,4
```

The `word` modifier is required because C88 allocates two bytes for `int`s. The `add sp,4` removes the two words that were pushed as parameters to `zip_`. Note the C88 appended '_' on names. If there had been no local variables defined in the 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` pushed as a word
`unsigned`

```
push WORD data_
```

`long` pushed 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_
PUBLIC _FLOADE  ; load float
call _FLOADE
PUBLIC _FPUSH   ; push double
call _FPUSH
```

`double` pushed with least-significant word pushed last

```
push WORD data_[6]
push WORD data_[4]
push WORD data_[2]
push WORD data_[0]
```

`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
mov  ax,ds     ; setup
mov  es,ax     ;    es
cld            ; set direction up
rep  movsw     ; copy to stack
```

10.5 LIBRARY CONVENTIONS

Called functions are responsible for preserving `CS`, `DS`, `SS`, `SP`, and `BP` across the function call. All other registers need not be maintained. 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 [bp+4], 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 [bp+4]   ; x_
mov  ax,WORD [bp+6]   ; lsw of y_
mov  dx,WORD [bp+8]   ; msw of y_
lea  si,[bp+10]       ; addr of z_
```

For functions that do return structures, [bp+4] contains a pointer to where the structure should be returned. So if the above fragment was

```
struct foo blip(x, y, z)
```

the parameters would be

```
mov  cx,WORD [bp+06]  ; x_
mov  ax,WORD [bp+08]  ; lsw of y_
mov  dx,WORD [bp+10]  ; msw of y_
lea  si,[bp+12]       ; addr of z_
```

Local variables are allocated below the current frame pointer regardless of 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`	returned in `AX`. `char` values are returned in `AL` with `AH`
`int`	set to zero
`unsigned`	
`long`	returned in `DX:AX`. (AX contains lsw)
`double`	returned on floating point stack (s/w or 8087).
`float`	
`struct`	returned to address in `[bp+4]`

10.6 Disk Input/Output Routines

In this implementation of C I/O, procedures like `getc()` are functions rather than macros and a file identifier `FILE` is simply an integer containing the file number rather than a pointer to a structure. This change means that `read()` and `getc()` calls may be intermingled and there is little difference between `open()` and `fopen()` on a file.

In UNIX there is a distinction between file and stream I/O. Crudely stated, the functions that have 'f' as their first letter (`fopen()`, `fread()` etc.) deal with streams, and other primitives (`open()`, `read()` etc.) access files. These two forms of I/O are compatible -- `fopen()` may be used to open a file and then `read()` used to read it -- but it is best to use either the stream or file primitives only for a particular file for portability. The `FILE` type is defined in the `stdio.h` include file and is simply an `int` type. This `int` contains the file number — the same number returned by `open()` and `creat()`. To use the stream routines with a file opened with the `open()`, merely pass the file number.

The stream style of I/O primitives are: `fopen()` to open a file, `fread()`, `fgets()` or `fgetc()` [`getc()`] to read, `fwrite()`, `fputs()` or `fputc()` [`putc()`] to write, `fseek()` to seek, `fflush()` to write out internal buffers, and `fclose()` to close.

The file type I/O primitives are: `open()`, `creat()`, `read()`, `write()`, `lseek()`, and `close()`.

The maximum number of files that can be open at one time is either 20, or the number specified in CONFIG.SYS, whichever is less. See section 2.3 for details about CONFIG.SYS. New files are `creat()`'d and old files are `open()`'d.

A closed file may be `rename()`'d or `unlink()`'d (deleted).

Three predefined file numbers may be used to read from or write to the console. To use them, include the following defines in the program:

```
#define stdin   0
#define stdout  1
#define stderr  2
```

10.7 Math Routines

If any of the transcendental or sqrt() functions are used, *include* the file `math.h` or the equivalent declarations to specify them as returning `double`'s.

`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 */
```

10.8 IBM-PC Screen and Keyboard Interface

A number of functions have been written to 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.

10.9 Alphabetical Function Index

Type	Function	Library	Reference
int	abs(int)	CSTDIO.S	MATH-LIMIT
double	acos(double)	CSTDIO.S	MATH-TRIG
double	asin(double)	CSTDIO.S	MATH-TRIG
double	atan(double)	CSTDIO.S	MATH-TRIG
int	atoi(char *)	CSTDIO.S	STRING-CONVERSION
double	atof(char *)	CSTDIO.S	STRING-CONVERSION
long	atol(char *)	CSTDIO.S	STRING-CONVERSION
char *	calloc(unsigned, unsigned)	CSTDIO.S	MEMORY-ALLOCATION
double	ceil(double)	CSTDIO.S	MATH-LIMIT
void	chain(char *, char *)	CSTDIO.S	OS-TASK CONTROL
char	ci()	CSTDIO.S	CONSOLE-I/O
int	close(int)	CSTDIO.S	FILE-OPEN/CLOSE
void	co(ch)	CSTDIO.S	CONSOLE-I/O
double	cos(double)	CSTDIO.S	MATH-TRIG
double	cot(double)	CSTDIO.S	MATH-TRIG
int	creat(char *)	CSTDIO.S	FILE-OPEN/CLOSE
char	csts()	CSTDIO.S	CONSOLE-I/O
void	dates(char *)	CSTDIO.S	OS-SYSTEM INTERFACE
double	exp(char *)	CSTDIO.S	MATH-TRANSCENDENTAL
double	exp10(double)	CSTDIO.S	MATH-TRANSCENDENTAL
void	exit(char)	CSTDIO.S	OS-TASK CONTROL

double	fabs(double)	CSTDIO.S	MATH-LIMIT
int	fclose(int)	CSTDIO.S	FILE-OPEN/CLOSE
int	fgetc(int)	CSTDIO.S	CHARACTER-I/O
char *	fgets(char *, int, int)	CSTDIO.S	STRING-I/O
double	floor(double)	CSTDIO.S	MATH-LIMIT
int	fopen(char *, char)	CSTDIO.S	FILE-OPEN/CLOSE
int	fprintf(int, char *, ...)	CSTDIO.S	FORMATTED OUTPUT
int	fputc(char, int)	CSTDIO.S	CHARACTER-I/O
int	fputs(char *, int)	CSTDIO.S	STRING-I/O
double	frand()	CSTDIO.S	RANDOM NUMBERS
int	fread(cahr *, int, int, int)	CSTDIO.S	FILE-I/O
int	free(char *)	CSTDIO.S	MEMORY ALLOCATION
double	frexp(double, *int)	CSTDIO.S	MATH-COMPONENTS
int	fscanf(int, char *, ...)	CSTDIO.S	FORMATTED INPUT
long	fseek(int, long, char)	CSTDIO.S	FILE-I/O
int	fwrite(char *, int, int, int)	CSTDIO.S	FILE-I/O
char	getchar()	CSTDIO.S	CHARACTER-I/O
char *	gets(char *)	CSTDIO.S	STRING-I/O
int	getc(int)	CSTDIO.S	CHARACTER-I/O
int	getw(int)	CSTDIO.S	CHARACTER-I/O
char *	index(char *, char)	CSTDIO.S	STRING-OPERATIONS
int	isalnum(ch)	CSTDIO.S	STRING-TYPING
int	isalpha(ch)	CSTDIO.S	STRING-TYPING
int	isascii(ch)	CSTDIO.S	STRING-TYPING
int	iscntrl(ch)	CSTDIO.S	STRING-TYPING
int	isdigit(ch)	CSTDIO.S	STRING-TYPING
int	islower(ch)	CSTDIO.S	STRING-TYPING
int	isprint(ch)	CSTDIO.S	STRING-TYPING
int	ispunct(ch)	CSTDIO.S	STRING-TYPING
int	isspace(ch)	CSTDIO.S	STRING-TYPING
int	isupper(ch)	CSTDIO.S	STRING-TYPING
double	ldexp(double, int)	CSTDIO.S	MATH-COMPONENTS
void	_lmove(int,int,int,int,int)	CSTDIO.S	CHARACTER-ACCESS
double	log(double)	CSTDIO.S	MATH-TRANSCENDENTAL
double	log10(double)	CSTDIO.S	MATH-TRANSCENDENTAL
void	longjump(char *, int)	CSTDIO.S	GLOBAL GOTO
long	lseek(int, long, char)	CSTDIO.S	FILE-I/O
char *	malloc(unsigned)	CSTDIO.S	MEMORY ALLOCATION
double	modf(double, *double)	CSTDIO.S	MATH-COMPONENTS
void	move(int, char *, char *)	CSTDIO.S	CHARACTER-ACCESS

int	open(char *, char)	CSTDIO.S	FILE-OPEN/CLOSE
double	pow(double, double)	CSTDIO.S	MATH-TRANSCENDENTAL
int	printf(char *, ...)	CSTDIO.S	FORMATTED OUTPUT
void	puts(char *)	CSTDIO.S	STRING-I/O
int	putc(char, int)	CSTDIO.S	CHARACTER-I/O
void	putchar(char)	CSTDIO.S	CHARACTER-I/O
int	putw(unsigned, int)	CSTDIO.S	CHARACTER-I/O
void	qsort(char *, int, int, (*)())	CSTDIO.S	SORT
unsigned	rand()	CSTDIO.S	RANDOM NUMBERS
int	read(int, char *, int)	CSTDIO.S	FILE-I/O
char *	realloc(char *, unsigned)	CSTDIO.S	MEMORY ALLOCATION
int	rename(char *, char *)	CSTDIO.S	FILE-DIRECTORY
long	rewind(int)	CSTDIO.S	FILE-I/O
char *	rindex(char *, char)	CSTDIO.S	STRING-OPERATIONS
void	scanf(char *, ...)	CSTDIO.S	FORMATTED-INPUT
void	scr_aputs(char *, char)	PCIO.A	CONSOLE-IBM PC
char	scr_ci()	PCIO.A	CONSOLE-IBM PC
void	scr_clr()	PCIO.A	CONSOLE-IBM PC
void	scr_clrl()	PCIO.A	CONSOLE-IBM PC
void	scr_cls()	PCIO.A	CONSOLE-IBM PC
void	scr_co(char)	PCIO.A	CONSOLE-IBM PC
char	scr_csts()	PCIO.A	CONSOLE-IBM PC
void	scr_cursoff()	PCIO.A	CONSOLE-IBM PC
void	scr_curson()	PCIO.A	CONSOLE-IBM PC
void	scr_rowcol(row,col)	PCIO.A	CONSOLE-IBM PC
void	scr_scdn()	PCIO.A	CONSOLE-IBM PC
void	scr_scrdn(int,int,int,int,int)	PCIO.A	CONSOLE-IBM PC
void	scr_scrup(int,int,int,int,int)	PCIO.A	CONSOLE-IBM PC
void	scr_scup()	PCIO.A	CONSOLE-IBM PC
void	scr_setmode(char)	PCIO.A	CONSOLE-IBM PC
void	scr_setup()	PCIO.A	CONSOLE-IBM PC
char	scr_sinp()	PCIO.A	CONSOLE-IBM PC
int	setjump(char *)	CSTDIO.S	GLOBAL GOTO
void	_setmem(char *, int, char)	CSTDIO.S	CHARACTER-MOVING
double	sin(double)	CSTDIO.S	MATH-TRIG
void	sprintf(char *, char *, ...)	CSTDIO.S	FORMATTED OUTPUT
double	sqrt(double)	CSTDIO.S	MATH-TRANSCENDENTAL
void	srand(int)	CSTDIO.S	RANDOM NUMBERS
void	sscanf(char *, char *, ...)	CSTDIO.S	FORMATTED INPUT

char *	strcat(char *, char *)	CSTDIO.S	STRING-OPERATIONS
int	strcmp(char *, char *)	CSTDIO.S	STRING-OPERATIONS
char *	strcpy(char *, char *)	CSTDIO.S	STRING-OPERATIONS
unsigned	strlen(char *)	CSTDIO.S	STRING-OPERATIONS
char *	strncat(char *, char *, int)	CSTDIO.S	STRING-OPERATIONS
int	strncmp(char *, char *, int)	CSTDIO.S	STRING-OPERATIONS
char *	strncpy(char *, char *, int)	CSTDIO.S	STRING-OPERATIONS
double	tan(double)	CSTDIO.S	MATH-TRIG
void	times(char *)	CSTDIO.S	OS-SYSTEM INTERFACE
char	tolower(char)	CSTDIO.S	CHARACTER-TYPING
char	toupper(char)	CSTDIO.S	CHARACTER-TYPING
char	ungetc(char, int)	CSTDIO.S	CHARACTER-I/O
int	unlink(char *)	CSTDIO.S	FILE-DIRECTORY
int	write(int, char *, unsigned)	CSTDIO.S	FILE-I/O

CHARACTER — ACCESS

```
char ch, sp, tp;
unsigned port, wd, num, sseg, tseg;
char _inb(), _peek();
unsigned _inw();
void _outb(), _outw(), _lmove(), _poke();

     ch = _inb(port);
     wd = _inw(port);
     _outb(ch, port);
     _outw(wd, port);
     _lmove(number, sp, sseg, tp, tseg);
     ch = _peek(sp, sseg);
     _poke(ch, tp, tseg);
```

_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*.

_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());
```

_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* .

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`.

CHARACTER — I/O

```
int fp, w, data;
char ch;
int getchar(), getc(), fgetc(), getw(), ungetc(),
    putchar(), putc(), fputc(), putw();

    data = getchar();
    data = putchar(ch);

    data = getc(fp)
    data = putc(ch, fp);
    data = fgetc(fp)
    data = fputc(ch, fp);

    data = getw(fp)
    data = putw(w, fp);

    data = ungetc(ch, fp)
```

getchar returns the next character from *stdin*, or -1 if an error, end of file, or a CTRL-Z was found. *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. *putchar* returns *ch*, or -1 on error.

getc and *fgetc* return the next character from the file *fp*, or -1 if an error, or end of file was sensed. *putc* and *fputc* write *ch* to the file *fp*. *fp* must have been opened prior to the call. The functions return *ch*, or -1 on error.

getw returns the next `int` from the file *fp*, or -1 if an error, or end of file was sensed. *putw* writes the `int` *w* to the file *fp*. *fp* must have been opened prior to the call. *putw* returns *w*, or -1 on error.

ungetc pushes the character *ch* back onto the file *fp*. The next call to *getc* or *fgetc* will return *ch*.. *ungetc* returns *ch*, or -1 if it can't push the character back. *fseek* clears all pushed characters. EOF (-1) can't be pushed.

SEE ALSO: `scanf(), fread(), printf(), fwrite()`

NOTE: *getchar*, *getc* and *putchar* are functions rather than macros.

getchar will hangup reading redirected input under DOS 2.X and higher. Use `getc(stdin)` if the input could be redirected.

There is no way to distinguish the return from `putw(-1, fp)` from an error.

CHARACTER — MOVING

```
unsigned  number;
char  *sourcePtr,  *targetPtr,  ch;
void  _setmem,  _move();

      _setmem(targetPtr,  number,  ch);
      _move(number,  sourcePtr,  targetPtr);
```

_setmem sets *number* bytes of memory starting at *targetPtr* to the byte value *ch* .

_move moves *number* bytes from *sourcePtr* to *targetPtr* .

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.

CHARACTER — TYPING

```
char  c,  ch;
int  tval;
int  isalnum(),  isalpha(),  isascii(),
      iscntrl(),  isdigit(),  islower(),
      isupper(),  isprint(),  ispunct(),
     isspace();
char  tolower(),  toupper();

    tval  =  isalnum(c);
          .  .  .
    ch  =  tolower(c);
    ch  =  toupper(c);
```

The *is****** functions return either *true* (non-zero) or *false* (zero) accordingly as *c* can be classified as follows:

`isalnum(c)`	c is either a letter or a digit
`isalpha(c)`	c is a letter
`isascii(c)`	c is less than `0x80`
`iscntrl(c)`	c is either `0x7F`, or less than `0x20` (space)
`isdigit(c)`	c is a digit
`islower(c)`	c is a lower-case letter
`isupper(c)`	c is an upper-case letter
`isprint(c)`	c is a printing character, `0x20` (space) through `0x7E` ('~')
`ispunct(c)`	c is neither a control nor an alphanumeric character
`isspace(c)`	c is a `0x20` (space), `'\t'` (tab), `'\r'` (carriage return), `'\n'` (linefeed), or `'\f'` (formfeed).

tolower transforms upper-case letters to lower-case while *toupper* performs the opposite transform. Both functions return *c* unchanged if it isn't the correct case.

NOTE: These are functions rather than the usual macro implementation.

CONSOLE — IBM-PC

```
char ch, newMode, attr, *string;
int lines, frow, fcol, trow, tcol;
char scr_ci(), scr_csts(), scr_sinp()
void scr_clr(), scr_clrl(), scr_cls(),
     scr_co(), scr_aputs(); scr_cursoff(),
     scr_curson(), scr_rowcol(),
     scr_scdn(),scr_scrdn(),
     scr_scrup(),
     scr_setmode(), scr_setup();

     scr_setup();
     ch = scr_ci();
     ch = scr_csts();
     ch = scr_sinp();
     scr_clr();
     scr_clrl();
     scr_cls();
     scr_co('A');
     scr_aputs(string, attr);
     scr_cursoff();
     scr_curson();
     scr_rowcol(trow, tcol);
     scr_scdn();
     scr_scrdn(lines, frow, fcol, trow, tcol);
     scr_scup();
     scr_scrup(lines, frow, fcol, trow, tcol);
     scr_setmode(newMode);
```

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.

scr_ci is like `ci()` but uses its own translation table for command characters. It returns the next character from the input queue.

scr_csts returns the next character from the queue, or 0 if no character is available. If a character is present, it is equivalent to a `csts()` followed by a `ci()`.

scr_sinp returns the character under the cursor on the screen.

scr_clr erases the entire screen and sets the cursor to the home (0,0) location.

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.

CONSOLE — IBM-PC

scr_co is the same as `co()` — it writes a single character out to the screen.

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.

scr_cursoff turns the cursor off; *scr_curson* turns it back on.

scr_rowcol moves the cursor to row *trow* and column *tcol*.

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 character locations (*frow*, *fcol*), (*trow*, *tcol*).

scr_scup scrolls the screen up one line, but leaves the top two lines alone.

scr_scrup scrolls the area up *lines*. The area is defined by the character locations (*frow*,*fcol*), (*trow*,*tcol*).

scr_setmode sets the mode of the color card. *newMode* must be between 0 and 6. See the Note below.

NOTE: All of the above 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 */
char scr_attr; /* current character attribute.
                  Normally 7 for white on black but
                  can be set for any attributes
                  see Technical Reference Manual */
```

CONSOLE — I/O

```
char  ch,  ci(),  csts();
void  co();

    ch  =  ci();
    co(ch);
    ch  =  csts();
```

ci reads the next character from the keyboard. If one is not available, *ci* waits until one is entered. There is no check for CTRL-C.

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. No test for CTRL-C is performed.

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_co(), scr_csts()`

NOTES: *ci*, and *csts* return a zero as the first character of an extended key sequence, and return the extended key code on the next call.

In order to decode an extended key sequence, use the `scr_` equivalents. They map the extended key sequences into `char` values between `0x80` and `0xFF`. See the files CONFIG.C and PCIO.A for the mapping.

FILE — DIRECTORY

```
char  *oldFile,  *newFile;
int  stat;
int  rename(),  unlink();

     stat  =  rename(oldFile,  newFile);
     stat  =  unlink(oldFile);
```

rename changes the file name *oldFile* to *newFile*. Under DOS 2.X and higher, *oldFile* may contain a path specification. Returns -1 if *oldFile* is open, or if an error is detected.

unlink deletes the file *oldFile*. Under DOS 2.X and higher, *oldFile* may contain a path specification. Returns -1 if *oldFile* doesn't exist, is open, or if an error is detected.

FILE — I/O

```
unsigned num, nitems, count;
char *buf;
int fp, stat, mode;
int fread(), read(), fwrite(), write(),
    fflush();
long offset, lval;
long fseek(), lseek(), rewind();

    num = fread(ip, sizeof(*ip), nitems, fp);
    num = read(fp, buf, count);
    num = fwrite(ip, sizeof(*ip), nitems, fp);
    num = write(fp, buf, count);
    stat = fflush(fp);
    lval = fseek(fp, offset, mode)
    lval = lseek(fp, offset, mode)
    lval = rewind(fp);
```

fread reads into an area starting at *ip*, *nitems* of data of type **ip*, from the file *fp* ; It returns *num*, the number of items actually read, or 0 if an error occurred.

read reads *count* bytes into *buf* from the file *fp*. It returns *num*, the number of bytes actually read, or -1 if an error occurred.

fwrite appends from an area starting at *ip*, at most *nitems* of data of type **ip*, to the file *fp*; It returns *num* , the number of items actually written.

write appends from *buf*, at most *count* bytes of data to the file *fp*. It returns *num*, the number of bytes actually written, or -1 if an error occurred.

fflush writes any buffered data for the file `fp` to that file. The file remains open. It returns `-1` if `fp` is not open, or if an error occurred writing the buffered data to that file.

fseek and *lseek* set the location of the next input or output operation on the file *fp* to the signed *offset* bytes from the beginning, the current location, or the end of the file accordingly as *mode* has the value 0, 1, or 2. Both functions return the current location, or -1 if there was an error.

rewind is the same as `fseek(fp,0L,0)` — it seeks to the begining of the file *fp* .

SEE ALSO: `fgetc(), fputc(), fgets(), fputs(), scanf(), printf()`

NOTE: `#define ftell(fp)  fseek(fp, 0L, 1)`

Only disk files are buffered, so *fflush* does nothing on non-disk files.

FILE — OPEN/CLOSE

```
int  fp,  stat;
char  *name,  *method,  mode;
int  creat(),  fopen(),  open(),  fclose(),  close();

    fp  =  creat(name)
    fp  =  fopen(name,  method)
    fp  =  open(name,  mode)
    stat  =  fclose(fp);
    stat  =  close(fp);
```

creat creates the file *name* and returns an `int` that is used to reference it in future file operations, or -1 if the file can't be opened. If the file *name* already exists, it is deleted.

fopen and *open* open the file *name* and return an `int` that identifies the file in future file operations. *fopen* returns zero, and *open* returns -1 if the file can't be opened.

method is a `char` string having one of the following values: "r" = open for reading,"w" = open for writing, "a" = open for append — open for writing at end of file, or create for writing.

mode is a `char` having one of the following values: 0 = open for reading, 1 = open for writing, 2 = open for update, and 3 = open for reading (CTRLZ indicates EOF).

fclose and *close* write any buffered data for the file *fp* to that file, and close the file. They return `-1` if *fp* is not open, or if an error occurred writing the buffered data to that file.

NOTES: *creat* opens a file in update mode so that after a file is written, a program can seek to the begining of the file and read it without closing the file and reopening it.

fopen for read access is the same as *open* for update; write access is the same as *creat* for the file; append causes a seek to the end of an existing file, or the creation of a new file as required.

Any of the functions can open the console (`"CON"`) or printer (`"PRN"`).

FORMATTED INPUT

```
char  fcs[],  buf[];
int  fp,  num;
int  scanf(),  fscanf(),  sscanf();

     num = scanf(fcs  [, ptr ] ... );
     num = fscanf(fp,  fcs  [, ptr ] ... );
     num = sscanf(buf,  fcs  [, ptr ] ... );
```

scanf reads from the *stdin* , *fscanf* reads from the file *fp* , and *sscanf* reads from the string *buf* . Each function returns the number of items successfully scanned or -1 on end of input or error (*fscanf*). 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* , *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.

```
|  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
| 255 | → scanf("%u", &uns)  →  255
|-255 | → scanf("%ld", &lng) → -255L
```

FORMATTED INPUT

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 is not 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);
```

FORMATTED OUTPUT

```
char fcs[], *cp, buf[];
int status, fp;
void printf(), sprintf();
int fprintf();

printf(fcs [, arg ] ... );
status = fprintf(fp, fcs, [, arg ] ... );
sprintf(buf, fcs [, arg ] ... );
```

printf formats the output to the file *stdout*. *fprintf* formats the output to the file *fp*, which must have been opened prior to the *fprintf* call — *fprintf* returns -1 on error. *sprintf* formats the output in the character array *buf*, terminated with the character `'\0'`.

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|
```

FORMATTED OUTPUT

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("%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()`.

GLOBAL GOTO

```
#include  <setjmp.h>

int  val;
jmp_buf  env;
int  setjmp();
void  longjmp();

      val  =  setjmp(env);
      longjmp(env,  val);
```

jmp_buf is defined in *<setjmp.h>* . It creates an environment used by *setjmp* for future use by *longjmp* . *jmp_buf* is defined as

```
      typedef char jmp_buf[6];
```

setjmp saves the current SP, BP, and return address in *env.* It returns a zero.

longjmp restores SP, BP, and return address from *env* and returns *val* . *val* cannot be zero.

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.

setjmp 's caller can tell by the returned value if control was returned from *setjmp* (0), or from *longjmp* (!=0).

MATH—COMPONENTS

```
#include  <math.h>

double  dval,  value,  *ipart;
int  exp,  *eptr;
double  frexp(),  ldexp(),  modf();

        dval  =  frexp(value,  eptr);
        dval  =  ldexp(value,  exp);
        dval  =  modf(value,  ipart)
```

frexp returns the mantissa of *value* as a fraction (< 1.0), and the base 2 exponent of *value* as an integer at **eptr*

ldexp returns the quantity *value* * 2^{exp}.

modf returns the positive fractional part of *value* and stores the integer part at **ipart*.

MATH—LIMITS

```
#include  <math.h>

double    dval, x;
double    fabs(), floor(), ceil();
unsigned  uval, y;
unsigned  abs();

          dval = fabs(x);
          uval = abs(y);
          dval = floor(x);
          dval = ceil(x);
```

fabs returns $|x|$ (absolute value).

abs returns $|y|$ (absolute value).

floor returns the largest integer value not greater than `x`.

ceil returns the smallest integer value not less than `x`.

MATH—TRANSCENDENTALS

```
#include  <math.h>

double  x, y, dval;
double  exp(), exp10(), log(), log10(), pow(),
        sqrt();

        dval  =  exp(x);
        dval  =  exp10(x);
        dval  =  log(x);
        dval  =  log10(x);
        dval  =  pow(x, y);
        dval  =  sqrt(x);
```

exp returns the exponential function of *x*; *exp10* returns the base 10 exponent.

log returns the natural logarithm of *x*; *log10* returns the base 10 logarithm.

pow returns x^y.

sqrt returns the square root of *x*.

NOTE: **EDOM** indicates an invalid argument, i.e., sqrt(-1.0). **ERANGE** indicates that the result would under/overflow, i.e., tan(PI/2.0). **EDOM** and **ERANGE** are defined in *math.h.*

exp and *pow* return a very large value when the result would overflow; *errno* is set to **ERANGE**. *pow* returns zero and sets *errno* to **EDOM** when the second argument is negative and not integral, or when both arguments are zero.

log and *log10* return zero when x is zero or negative; *errno* is set to **EDOM**.

sqrt returns zero when *x* is negative; *errno* is set to **EDOM**.

MATH—TRIG

```
#include  <math.h>

double  x,  y;
double  sin(),  cos(),  tan(),
          asin(),  acos(),  atan();

        y  =  sin(x);
        y  =  cos(x);
        y  =  tan(x);
        y  =  asin(x);
        y  =  acos(x);
        y  =  atan(x);
```

sin , *cos* , and *tan* are trigonometric functions of radian argument x. The meaningfulness of the result depends upon the magnitude of the argument.

asin returns arc sin in the range $-\pi/2$ to $\pi/2$.

acos returns the arc cosine in the range 0 to π.

atan returns the arc tangent in the range of $-\pi/2$ to $\pi/2$.

NOTE: **EDOM** indicates an invalid argument, i.e., sqrt(-1.0). **ERANGE** indicates that the result would under/overflow, i.e., tan(PI/2.0). **EDOM** and **ERANGE** are defined in *math.h.*

asin and *acos* return 0.0 and set *errno* to **EDOM** for $x > 1.0$. *tan* returns a huge number and sets *errno* to **ERANGE** at its singular points.

MEMORY ALLOCATION

```
char *cp, *np, *op;
unsigned seg, num, size, stack;
void freeall(), _setsp();
char *_memory(), _showsp(), *malloc(), * calloc(),
     *realloc();
unsigned *_showcs(), *_showds();

     freeall(stack);
     cp = _memory();
     cp = _showsp();
     np = malloc(size);
     np = calloc(num, size);
     np = realloc(op, size);
     cp = free(op);
     seg = _showcs();
     seg = _showds();
     _setsp(stack);
```

freeall initializes the memory allocation area between the address returned by *memory*, to *stack* bytes below the contents of the stack pointer, *_showsp*.

malloc returns a pointer to a block of *size* bytes, or 0 if it couldn't allocate the memory. If *freeall* hadn't been called, the first call to *malloc* will call it to set up the memory allocation area and reserve 1024 bytes of stack space.

calloc returns a pointer to a block of *num* * *size* bytes, or 0 if it couldn't allocate the memory. If *freeall* hadn't been called, the first call to *calloc* will call it to set up the memory allocation area and reserve 1024 bytes of stack space.

realloc changes the size of the block at *op* to *size* bytes and 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.

free marks the block at *cp* as unallocated.

_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).

_setsp sets the stack pointer (the SP register) to *stack*.

MEMORY ALLOCATION

NOTE: The memory allocation 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 (0xC6). *size* is the size of *data* in bytes. The address of *data* is returned by *malloc*, *calloc*, and *realloc*, and used by *free*.

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* .

freeall sets up the area with two blocks — an unallocated block at *_memory* of a size determined by the current value of SP and the amount of *stack* passed to *freeall*, and the end-of-area block of size zero.

The following function prints out a map of the memory allocation area.

```
char *cp;

cp = _memory();
while(cp->status != 0xC6) {
    printf("%5u %2salloc bytes at %u\n",
        cp->size,
        cp->status == 0xAB ? "" : "un",
        cp);
    cp = cp->data + cp->size;
    }
```

OS — OVERLAY

```
int  overlayNumber,  status;
char  *overlayFilename;

      status  =  overlay_init(overlayFilename);
      status  =  overlay(overlayNumber);
      status  =  moverlay(overlayNumber);
```

overlay_init must be called prior to the first *overlay* call and must not be used when the -M option of BIND is used. *overlayFilename* contains the overlays. On MS-DOS 2.0 and later, 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 driver number, e.g. "B:X.OV". -1 is returned if the file could not be found.

overlay loads overlay *overlayNumber* as created by the BIND command with the -V option. It must be called before any reference or call to data or code in the overlay. Overlays are <u>not</u> automatically loaded by referencing a value in the overlay. -1 will be returned if the overlay_init routine has not been called successfully, if the .OV file is bad, or if *overlayNumber* does not correspond to an existing overlay.

moverlay loads the indicated overlays created by the -M option in the BIND command. It works the same as the overlay function described above.

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 mode in an overlay, it is placed in the root. If a data item is initialized in a root module, it is also placed in the root. If it is initialized in an overlay, it is placed in the overlay.

OS — SYSTEM INTERFACE

```
char *buf, inum, ival;
unsigned arg;

extern unsigned _rax, _rbx, _rcx, _rdx, _rsi, _rdi,
    _res, _rds;
extern char _carrf, _zerof;
unsigned _os();
void _doint(), dates(), times();

    ival = _os(inum, arg);
    _doint(inum);
    dates(buf);
    times(buf);
```

_rax - *_rds* contain the values of the corresponding 8088 internal registers that are loaded and saved by *_doint*. *_carrf* is the carry flag; *_zerof* is the zero flag

_os provides an elementary interface to the BIOS. On DOS, *inum* goes into AH and *arg* into DX, and an `int 21H`, is executed. On CP/M-86, *inum* goes into CL and *arg* into DX, and an `int 224` is executed. The value in AL is returned.

_doint will cause a software interrupt and may be used to call whatever routines are available in the particular machine. If *_rds* is set to -1, the current value of the DS register is used.

dates formats the string *buf* with the current date as "`mm/dd/yy`".

times formats the string *buf* with the current time as "`hh-mm-ss`"

NOTE: The following example, on an IBM-PC, would use interrupt 10H to print the string.

```
extern unsigned _rax, _rbx, _rds;
extern char _carryf;
char str[] = "Hello There!!";
int i;

for (i = 0; i < sizeof(str); i++) {
    _rax = (14 << 8) + str[i];
    _rbx = 0;
    _rds = -1;
    _doint(0x10);
    }
```

OS — TASK CONTROL

```
char  *filename,  *commandTail,  Code;

      Code  =  exec(filename,  commandTail)
      chain(filename,  commandTail)
      exit(Code)
```

exec will load and execute an arbitrary program — *filename* is the complete pathname of the program (including the .EXE or .COM suffix). *commandTail* contains the arguments to the program. *exec* will return the completion code from the program or -1 if an error occurred loading the program. *exec* is in the EXEC.O file provided on the distribution disks.

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
```

exit terminates the program with completion code *Code*. A `main()` procedure can also *exit* with a completion code of zero by returning or by "falling" through the end of the procedure. *exit* does not close open files.

NOTE: Completion codes are set for programs running under MS-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 IBM 2.xx 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:

```
exec("\\command.com", "/cxxx");
```

where *xxx* is one of the DOS built-in commands ('dir', 'copy', ...) or the name of a batch file, including the trailing .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, type

```
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.

RANDOM NUMBERS

```
int  seed,  ival;
double  dval;
int  rand();
double  frand();
void  srand();

      ival  =  rand();
      dval  =  frand();
      srand(seed);
```

rand returns the next pseudo-random number in the range from 0 to 2^{15}- 1.

frand returns the next pseudo-random number in the range from 0.0 to 1.0.

srand sets the seed for the multiplicative congruential random number generator to *seed* .

NOTE: The initial seed for the generator is a `long`. *srand* only sets the low order word, so the generator cannot be restarted. Solution: seed the generator with your own integer before any calls to *rand* or *frand* .

SORT

```
char  array[];
int  num,  width,  (*compare)();
void  qsort();

     qsort(array,  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.

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 may not produce the expected results.

STRING CONVERSION

```
char *cp;
int ival, atoi();
double dval, atof();
long lval, atol();

    ival = atoi(cp);
    dval = atof(cp);
    lval = atol(cp);
```

These functions convert the `char` array at *cp* to `int`, `double`, and `long` values respectively. 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.

Function	Valid character Sequences
`atoi()`	{*whitespace* }[*sign*]{*digit* }
`atof()`	{*whitespace* }[*sign*]{*digit* }['.'][{*digit* }[*E* [*sign*]{*digit* }]]
`atol()`	{*whitespace* }[*sign*]{*digit* }

SEE ALSO: `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;
    }
```

STRING — I/O

```
char *cp, buf[];
int err, len, fp;
char *gets(), *fgets();
int _gets(), puts(), fputs()

    cp  =  gets(buf);
    len =  _gets(buf, sizeof(buf));
    cp  =  fgets(buf, len, fp);
    err =  puts(buf);
    err =  fputs(buf, fp);
```

gets obtains 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 zero. *gets* returns its argument, or 0 on end of file or an error.

_gets obtains not more than `sizeof(`*buf* `)` - 1 characters from the console into *buf*. Editing proceeds as with *gets* . *_gets* returns the number of characters obtained, or 0 on end of file or an error.

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 zero. *fgets* returns *buf*, or 0 on end of file or an error.

puts copies the null terminated string *buf* to the console (*stdout*). *puts* returns a -1 on error.

fputs copies the null terminated string *buf* to the file *fp*. *fputs* returns a -1 on error.

SEE ALSO: `fscanf(), fread(), fprintf(), fwrite()`

NOTE: *gets* and *_gets* don't return the CR character, *fgets* does.

puts doesn't append a newline.

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. Each output will check for a CTRL-C entry, and terminate the program if it was pressed.

STRING — OPERATIONS

```
char *cp, *src, *dst, *s1, *s2, ch;
int max, len, cmp;
char *strcat(), *strncat(),
     *strcpy(), *strncpy(),
     *index(),  *rindex();
int strcmp(), strncmp(), strlen();

    cp = strcat(dst, src);
    cp = strncat(dst, src, max);
    cp = strcpy(dst, src);
    cp = strncpy(dst, src, len);
    cp = index(src, ch);
    cp = rindex(src, ch);
   cmp = strcmp(s1, s2);
   cmp = strncmp(s1, s2, max);
   len = strlen(src);
```

All functions work on null-terminated strings. There is no test for overflow.

strcat appends a copy of *src* to the end of *dst*. *strncat* appends, at most, *max* bytes. Both functions return *dst*.

strcpy copies *src* to *dst*, stopping after the null byte has been transfered. *strncpy* copies exactly *len* bytes, truncating or null-padding as required; *dst* may not be null-terminated if the `strlen(src) >= len`.

index returns a pointer to the first occurrence of *ch* in *src*, or 0 if *ch* doesn't occur. *rindex* returns a pointer to the last occurrence, or 0.

strcmp compares the two strings and returns +1, 0, or -1 accordingly as *s1* is lexically greater than, equal to, or less than *s2*. *strncmp* makes the same comparison, but of, at most, *max* bytes.

strlen returns the number of non-null characters in *src*.

Appendix A: Messages

A.1 SEE Messages

A.1.1 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 V2.4: Copyright 1982,83,84,85  Michael Ouye
```

When the editor is exited, the message line prints the message:

```
bye! <filename>
```

A.1.2 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'.

A.2 C88 Compiler Messages

A.2.1 Banner and Termination Messages

```
>C88 Compiler   V2.5   (c) Mark DeSmet,1982,83,84,85
end of C68    001A code    0012 data     1% utilization
```

OR

```
>C88 Compiler   V2.5   (c) Mark DeSmet,1982,83,84,85
```
(various error messages)
```
    Number of Warnings = 2     Number of Errors = 5
```

The first form of termination means the compile 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 compile 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, compilation will end normally and produce a .O file.

A.2.2 Messages

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. C88 reports all such errors and then quits. Warnings are produced by correctable errors and the compiler continues. Since the compiler uses ASM88 as pass 3, 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.

A.2.2.1 C88 Fatal Errors

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.

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 —C88 must be followed by the name of the source file when invoked.

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.

too many cases — currently, a switch statement can only contain 128 case statements.

too many fors/too many whiles — whiles, do-whiles, switches and for statements can only be nested 10 deep.

stuck <register> — same as 'pushed'.

too many externals — the compiler currently has a limit of 500 static's or extern's.

A.2.2.2 C88 Errors

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.

bad control — the # control is illegal.

bad declaration — the declaration of a variable was illegal.

bad include — the #include must be followed by "name" or <name>.

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.

defines too deep — #define may reference another, but there is a limit. When #defines are expanded, the number of active #defines plus the number of #define arguments referenced by each cannot exceed 30.

duplicate argument — an attempt was made to declare an argument twice.

duplicate label — two labels have the same name.

EOF within comment — end of file was found inside a comment. A '*/' is missing.

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 define — a #define has unmatched parentheses or the #define parameters are illegally specified.

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 indirection — caused by trying to use a char, int, unsigned, float or double as a pointer. longs can be used as pointers but the other types cannot.

include nesting too deep — includes can only be nested three deep

illegal use of float — floating point numbers cannot be used as pointers.

indirect call — there is an expression of the form `x(...)` and *x* is not defined as a function or a pointer to a function. To call a pointer to a function use `(*x)(...)`.

line must be constant — a #line control must be followed by a decimal constant.

line too long — the maximum line length is 128.

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','TEXT') can only contain one to four characters.

missing argument — 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 return float — a float or double function must end with a return statement that returns a value. **Note:** The compiler is too stupid to accept.

```
double x(){if (1) return 1.;}.
```

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 constant — a 'case' prefix must be followed by an integer constant expression.

need constant after #if — a #if control must be followed by a 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 — the name prior to a '.' operator is not the name of a struct or union.

not an identifier — #ifdef, #ifndef and #undef controls must reference a #define value;.

not defined — #undef controls must reference a #define value;.

only STATIC and EXTERN allowed at this level — an attempt was made to declare an 'auto' outside of a procedure.

parameters cannot span lines — the arguments to a #define must all be on the same line.

return lacks argument — if a function is declared as returning a value then "return;" is illegal. Use "return 0;" if the value is unimportant.

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 200 characters. Usually means that a "" is missing.

undefined structure — a pointer to an undefined structure cannot be added to.

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 #define was used with the wrong number of arguments.

A.2.2.3 C88 Warnings

Warnings indicate a change in syntax (as in the case of structures), or suspicious code that is probably OK.

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 `(char *)` pointer will select any member.

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.

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.

undefined variable — the variable has not been defined. It is assumed to be an auto int.

A.2.2.4 ASM88 Errors (from C88 Execution)

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.

A.3 ASM88 Assembler Messages

A.3.1 Banner and Termination Messages

```
ASM88 8088 Assembler V1.3 (c) Mark DeSmet, 1982,83,84,85
```

(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
```

A.3.2 Messages Produced by ASM88

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.

A.3.2.1 Fatal Errors From ASM88

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 names and labels are allowed.

too many symbols — the assembler ran out of symbol space. The source program should be broken into smaller modules.

A.3.2.2 Errors from ASM88

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.

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.

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.

A.4 BIND Messages

A.4.1 Banner and Termination Messages

```
Binder for C88 and ASM88     V1.9     (c) Mark DeSmet, 1982,83,84,85
end of BIND          9% utilization
```

A.4.2 Warnings from BIND

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.

A.4.3 Fatal Errors from BIND

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 segment has over 64K of code. See the description of BIND overlay support.

over 64K data — the resultant 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.

A.5 LIB88 Messages

A.5.1 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.

A.5.2 Warnings from LIB88

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.

A.5.3 Fatal Errors from LIB88

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.

A.6 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 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.

A.7 CLIST Messages

A.7.1 Banner and Termination

```
CLIST   V1.3  (c) Mark DeSmet, 1982,83,84
end of CLIST
```

A.7.2 Messages Produced by CLIST

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.

B. The ASM88 Assembly Language

B.1 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.

B.2 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 `dddd0` or `ddddq` or `dddd0` 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.

B.3 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.

B.4 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.
```

B.5 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.

B.6 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
```

B.7 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]
```

B.8 Comments

A non-quoted semi-colon causes the rest of a line to be ignored.

B.9 Assembler Directives

Directives may be in either upper or lower case.

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
```

Include: INCLUDE "filename"

The indicated file is included in the source.

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.

Public: PUBLIC identifier [:BYTE|WORD etc.] [,identifier...]

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.

Dseg and Cseg: DSEG or CSEG

The DSEG directive indicates that data follows and the CSEG directive indicates that code follows. 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. There is no support for more than these two segments.

End: END

The END statement is optional and does nothing.

B.10 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 either binary (without a period or 'E' exponent) or single precision floating point. 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.

B.11 Differences Between Intel ASM86 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.

B.12 8086 Instructions

B.12.1 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.

B.12.2 Instructions

AAA ascii adjust for addition. This instruction fixes AL after two ascii digits have been added.

AAD ascii adjust for division. AH is multiplied by 10 and added to AL.

AAM ascii adjust after multiply. AL is divided by 10. The result goes in AH and the remainder into AL.

AAS ascii adjust after subtract. Repairs AL when AL is the result of ascii subtraction.

ADD Adds the right operand to the left operand. The flags are set.

ADD AX | AL,constant
ADD regrm,reg | constant
ADD reg,regrm

```
add  ax,ax
add  al,harry[bp+55]
add  word [bp+5],0
```

ADC Adds the right operand and the carry bit to the left operand. The flags are set.

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
```

CBW sign extend AL into AX.

CLC clear carry flag.

CLD clear direction flag.

CLI clear interrupt enable flag. Disables interrupts.

CMC complement carry flag.

CMP Compares operands. The flags are set the same as for SUB.

CMP AX | AL,constant
CMP reg,regrm

```
cmp   ax,ax
cmp   al,harry[bp+55]
cmp   word [bp+5],0
cmp   ax,ax
```

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.

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.

CALL label | regrm

```
call laba
call bx
call word es:[bx]
```

CWD sign extend AX into DX:AX.

DAA decimal adjust after add. Adjusts AL after packed addition.

DAS decimal adjust after subtract. Adjusts AL after packed subtraction.

DEC Decrements the operand. The flags other than carry are set.

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.

DIV regrm

```
div  cx
div  vara
```

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.

ESC constant,rm

HLT stops the processor. The processor stops until an external interrupt occurs.

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.

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.

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.

IN AL | AX,constant
IN AL | AX,DX

```
in   al,44
in   ax,dx
```

INC Increment the operand.

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 DDT86 or DEBUG can use an 'int 3' to call the debugger.

INT constant

```
int  0C1H
int  3
```

INTO interrupt on overflow. Cause an interrupt 4 if the overflow bit is set.

IRET return from an interrupt.

JB/C/NAE jump if below/carry/not above or equal.
JBE/NA jump if below or equal/not above.
JE/Z jump if equal/zero.
JL/NGE jump if lower/not greater than or equal to.
JLE/NG jump if not less than or equal to/not greater.
JNB/AE jump if not below/above or equal.
JNBE/A jump if not below or equal/above.
JNE/NZ jump if not equal/not zero.
JNL/GE jump if not lower/greater than or equal to.
JNLE/G jump if not less than or equal to/greater.
JNO jump if no overflow.
JNP/PO jump if parity odd.
JNS jump if not signed (positive).
JO jump if overflow.
JP/PE jump if parity even.
JS jump if signed (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.

JCXZ jump if CX is not equal to zero.

JCXZ label

JMP jump. 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. 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.

LCALL rm

```
lcall laba[bx]
```

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.

LDS is the same except that DS is loaded instead of ES. As always, the offset value in memory must precede the segment value.

LDS wreg,regrm
LES wreg,regrm

```
lds  bx,vara
les  di,vara
```

LEA loads the offset of the referenced memory location into a register.

LEA wreg,rm

```
lea  ax,[si+di+44]
lea  ax,vara
mov  ax,offset vara   ; same effect as above
```

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.

LJMP label

LOCK Lock the bus. LOCK demands a bus lock for the following instruction. Usually used with XCHG to implement semaphores.

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.

```
lodsb
rep lodsw
```

LOOP decrement CX and jump if CX not equal to zero.
LOOPZ/E decrement CX and jump if CX not zero and the zero flag is set.
LOOPNZ/E 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.

LOOP label
LOOPZ/E label
LOOPNZ/E 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.

LRET | constant

MOV The contents of the right operand are moved to the left operand.

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.

```
     movsb
rep  movsw
```

MUL Multiply AL by byte operand with result in AX or multiply AX by word operand with result in DX:AX.

MUL regrm

```
mul  CX
mul  vara
```

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.

OUT constant,AL | AX
OUT DX,AL | AX

```
out  dx,ax
out  33,al
```

NEG Negate the operand.

NEG regrm

```
neg  ax
neg  vara
```

NOP do nothing in three cycles.

NOT Invert the bits of the operand.

NOT regrm

```
not  ax
not  vara
```

OR logical or of the operands. Flags are set.

OR AX | AL,constant
OR regrm,constant | reg
OR reg,regrm

```
or   ax,ax
or   al,harry[bp+55]
```

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.

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.

PUSH wreg | regrm | segreg

```
push ax
push total
```

PUSHF The flags are pushed onto the stack.

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.

REP instruction
REPZ instruction
REPNZ instruction

```
rep  movsb
repz stosw
```

RCL rotate left through carry. The carry bit ends up as the new low bit and the high bit becomes the carry bit.
RCR rotate right through carry. The carry bit ends up as the new high bit and the low bit becomes the carry bit.
ROL rotate left. The high bit ends up in carry and as the new low bit.
ROR rotate right. The low bit ends up in carry and as the new high bit.
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.
SAR shift arithmetic right. The low bit becomes the carry bit, the high bit is left alone (i.e. the sign remains the same).
SHR shift right. The low bit goes to carry, the new high bit is zero.

RCL regrm,1 | CL
RCR regrm,1 | CL
ROL regrm,1 | CL
ROR regrm,1 | CL
SAL regrm,1 | CL
SHL regrm,1 | CL
SAR regrm,1 | CL
SHR regrm,1 | CL

```
shr  al,1
mov  cl,4
shr  vara,cl
```

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.

RET | constant

```
ret  5
ret
```

SAHF

New flags are loaded from AH. The format of AH is:

SF	ZF	X	AF	X	PF	X	CF

SBB Subtract the right operand and the carry bit from the left operand.

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.

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.

STC set the carry flag.

STD set the direction flag.

STI set interrupts enabled.

TEST logically ands the operands and sets the zero flag if no bits remain on. The operands are unchanged.

```
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.

XCHG The contents of the two operands are exchanged. XCHG is often used to implement semaphores.

```
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.

```
XLAT  rm
```

XOR Performs an exclusive or on the operands. The result replaces the left one. Flags are set.

XOR AX | AL,constant
XOR regrm,constant | reg
XOR reg,regrm

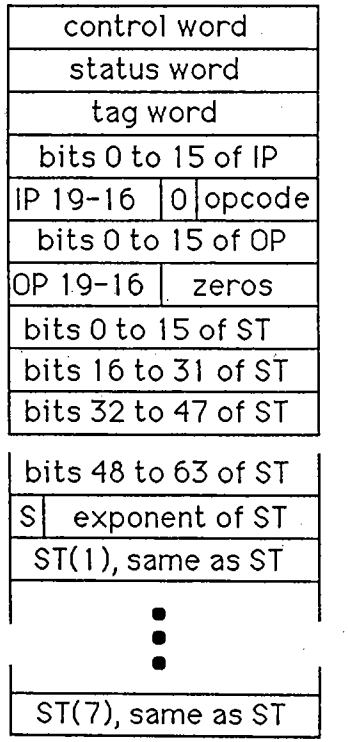
```
xor   ax,ax
xor   al,harry[bp+55]
xor   word [bp+5],0
xor   ax,ax
```

B.13 Floating Point

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.

B.13.1 Control Word

The control word can be loaded with FLDCW and stored with FSTCW and has the following format:

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.

B.13.2 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.

B.13.3 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.

B.13.4 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.

B.13.5 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
```

Large Case Option

Introduction

This section describes the Large Case Option of the DeSmet C Development Package. Its features include:

Full 1-megabyte addressability via 32-bit pointers.

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

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.

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.

Disk Contents

The Large Case Option Disk contains:

C88.EXE	Compiler — Pass 1
GEN.EXE	Compiler — Pass 2
ASM88.EXE	Assembler (Compiler — Pass 3)
BBIND.EXE	Object File Linker
LIB88.EXE	Object File Librarian
PROFILE.EXE	Program Execution Profiler
BEXEC.O	`exec()` and `chain()` functions
BCSTDIO.S	Standard Library with software floating point
BCSTDIO7.S	Standard Library with 8087 support
PCIO.A	Standard Screen interface (Large Case Assembler)

If you have purchased the Source Debugger Option, you will receive a 2nd disk with:

D88.EXE	Source Level Debugger

Installing the Large Case Option

To install the Large Case Option, copy the contents of the Large Case Option distribution disk to your working directory. The compiler passes will replace those of your current release.

If you have an 8087, delete the file BCSTDIO.S from your working directory and rename BCSTDIO7.S to BCSTDIO.S. Otherwise, delete BCSTDIO7.S from your working directory.

To test the installation, use the file LIFE.C from your original distribution disk #2. Compile LIFE.C by entering

```
c88 life -b
```

The -b switch will compile in the *big* mode. To bind with the Large Case PCIO.A, enter

```
asm88 pcio -b
bbind life pcio
```

The -b switch instructs the assembler to emit the correct object code. If you forget the -b switch, BBIND will complain about mixing large and small case object files.

Converting To Large Case

With most programs, just recompile with the -b switch on, and re-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 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()`.

Large Case C88

The Large Case Option 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 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.

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.

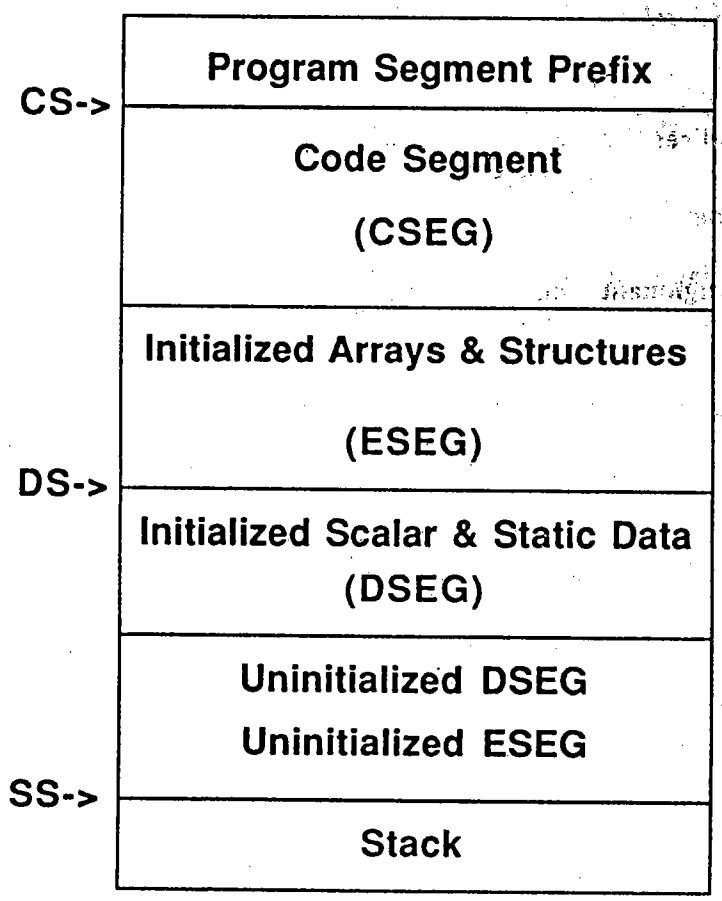
In addition to the standard CSEG (Code Segment) and DSEG (Data Segment) directives, there is a ESEG (Extra Segment) directive. CSEG and ESEG can be any size, while DSEG is restricted to 64K. CSEG is addressed with CS, DSEG with DS, and ESEG with either DS or ES. The Stack is a separate segment whose 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
```

The Large Case Option is loaded into memory as follows:



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

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.

Large Case LIB88

The Large Case LIB88 will create Large Case and Small Case libraries, but won't mix Large and Small Case object files in the same library.

Large Case Libraries

The Large Case libraries, CLIBL.LIB and BCSTDIO.S, are the same as their Small Case counterparts except for the deletion of `overlay_init()`, `overlay()`, `moverlay()` [no overlay support in Large Case] and `_memory()` [no static heap].

Large Case D88 & Profiler

Large Case D88 & Profiler are the same for both Large and Small Case programs.

Large Case DOS Link Support

There is no Large Case DOS Link Support.