SOFTWARE DEVELOPMENT SERIES

MANX™

Aztec C Version 4.10c

for PC-DOS and MS-DOS

Release Document

Information about version 4.10c of Aztec C86 is contained in the following places:

* The release document for v4.10a. It describes all the features of version 4.10a that weren't discussed in the manual, and describes the files in version 4.10a. These files are also in versions 4.10b and 4.10c of the software.
* This document. It describes the features that are new in version 4.10c. It also includes information from the 4.10b release document.
* A "read.me" file on the v4.10c disks, which lists the bugs that have been fixed in going from version 4.10a to v4.10b to v4.10c and lists known bugs.

New Features

The following describes the new features and enhancements found in versions 4.10b and 4.10c. Descriptions of the new programs are appended to this release document.

BUGS FIXED

Several bugs have been fixed. These are described in the **readme** file.

NEW COMPILER FEATURES

A new option, **+lf**, causes the compiler to make all global data **far**.

NEW LINKER FEATURES

When generating a symbol table in response to the **-t** option, the linker now by default outputs, for each symbol, the relative paragraph number of the segment that contains the symbol in addition to the offset of the symbol within that segment.

A new option, **+p**, causes the linker, when generating a symbol table, to output just the offset component of each symbol's address.

NEW UTILITIES

This release contains five new utility programs:

cpp	Generates an output file with your preprocessor statements.
ctoeng	Generates an English description for a specified C declaration.
engtoc	Given an English description of a desired C declaration, **engtoc** generates the C declaration.
hd	hex dump utility
proto	Generates a file containing C prototype definitions for the functions that are in a specified C source module.

Documentation Corrections

The following utilities are included only in the Developer and Commercial Packages:

z
ctags
proto

Packaging

As stated, the Aztec C86 Developer system is a superset of the *Professional*, and the *Commercial* system is a superset of the *Developer*.

This section first lists the files that are common to all three systems. It then lists the files that are in the *Developer* and *Commercial* systems, but not in the *Professional*. It then lists the files that are only in the *Commercial* system.

FILES IN ALL VERSIONS OF AZTEC C86

Executable Programs

The following executable programs are in all versions of Aztec C86:

CC.EXE	C Compiler, pass 1
CGEN.EXE	C Compiler, pass 2
AS.EXE	Assembler
LN.EXE	Linker
OBJ.EXE	Aztec-to-Microsoft object convertor
DB.EXE	Assembly language Debugger
SDB.EXE	Source level debugger
LB.EXE	Object file librarian
ORD.EXE	Object library generation utility
CNM.EXE	Object file utility
OBD.EXE	Object file utility
SQZ.EXE	Object file utility
CRC.EXE	CRC utility
ARCV.COM	Source archive utility
C.EXE	C driver
TERM.EXE	Terminal emulator program
Z.EXE	Text editor
CTAGS.COM	Text editor utility
CTOENG.EXE	C-to-English translator
ENGTOC.EXE	English-to-C translator
CPP.EXE	C Pre Processor
HD.EXE	HexDump Utility

Libraries

The following libraries are in all versions of Aztec C86. Each uses the small code and small data memory model.

C.LIB	Library of non-floating point functions
M.LIB	Library of floating point functions (non-8087 version)

M87.LIB	Library of floating point functions (8087 version)
M87S.LIB	Library of floating point functions (sensing version)
S.LIB	Screen functions
G.LIB	Graphics functions

All Aztec C86 systems also contain a 'large code', 'large data' version of each of the above libraries. The name of a 'large code', 'large data' version of a library is derived by appending the letter 'l' to the name of the 'small code', 'small data' version of the library. For example, the name of the 'large code', 'large data' version of **c.lib** is **cl.lib**.

Object Modules

The following object modules are in all versions of Aztec C86:

OVLD.O, OVLDPATH.O, OVBGN.O	Object modules for overlay support
CRT0.OBJ	Object module of Startup routine for programs linked with Microsoft libraries

New Header File

All Aztec C86 systems include several header files, which can be included in C programs. These files have extension *.h*.

STDLIB.H	declares **size_t** and gives Prototypes for standard functions.

Source Archives

The following source archives are in all versions of Aztec C86. They can ɔe unpacked using the **arcv** program.

S.ARC	Screen functions
G.ARC	Graphics functions
TERM.ARC	terminal emulator programs

Miscellaneous

The source file **stksiz.c** controls the size of a program's stack and heap, and he relative positioning of these two areas. For details, see the Programming)rganization section of the **Technical Information** chapter.

The runtime variable **_agetc_mask** determines the appropriate mask for igetc. This used to be 0x7f. If you did an agetc and the high bit was on, it got urned off. In this release it is set to 0xff, and the high bit remains intact. This llows Aztec C to handle 8-bit characters in strings and quotes. For more nformation refer to the Lib86 section of your 4.1 Release Document.

The file **exmpl.c** contains source to a sample C program.

FILES ONLY IN *DEVELOPER* AND *COMMERCIAL* SYSTEMS

Executable Programs

The following programs are only in the *Developer* and *Commercial* versions of Aztec C86:

MAKE.EXE	Program maintenance utility
DIFF.EXE	Source file comparator
GREP.EXE	Pattern matcher
LS.COM	File listing utility
PROF.EXE	Program profiler
PROTO.EXE	Prototype Generator

Libraries

In addition to 'small code', 'small data' and 'large code', 'large data' versions of each library, the *Developer* and *Commercial* versions of Aztec C86 contains a version that uses the 'large code', 'small data' memory model and a version that uses the 'small code', 'large data' memory model. The name of the 'large code', 'small data' version of a library is derived by appending the letters **lc** to the name of the 'small code', 'small data' library, while the name of the 'small code', 'large data' version is derived by appending **ld**.

For example, the name of the 'large code', 'small data' version of **c.lib** is **clc.lib**.

FILES ONLY IN THE *COMMERCIAL* SYSTEM

Libraries

The *Commercial* version of Aztec C86 contains a library, **c86.lib**, for generating programs that run on CP-M/86.

Source Archives

The following source archives are only in the *Commercial* version of Aztec C86. They can be unpacked using the *arcv* program.

STDIO.ARC	Standard I/O functions
MISC.ARC	Miscellaneous functions
MCH86.ARC	Miscellaneous functions
MATH.ARC	Floating point functions
DOS20.ARC	DOS 2.x functions
CPM86.ARC	CP/M-86 functions
BUILD.ARC	Makefiles & related files, for library generation

UNPACK.BAT	Batch file for dearchiving source archives

Files for Creating ROMable Code

The following *Commercial* system files are used to generate ROMable code:

HEX86.EXE	Intel hex record generator
ROM.O	Object module of Startup routine for ROMable programs that use 'small code, small data'
LROM.O	Object module of Startup routine for ROMable programs that use 'large code, large data'
LCROM.O	Object module of Startup routine for ROMable programs that use 'large code, small data'
LDROM.O	Object module of Startup routine for ROMable programs that use 'small code, large data'

CHECKING THE FILES

The file **crclist** contains the CRC values for the files. You can compute the CRC values of the files we sent you and then compare them with their expected values, using the program **crc**. For example, entering

```
crc *.*
```

computes the CRC of all the files on the current directory of the default drive.

Common Problems

Stray Pointers

If a program does not behave consistently or corrupts the operating system, the program may contain stray pointers. (Stray pointers are defined as those variables that are not pointing to the proper memory given their assigned values and that are directed to storage in an improper location.) This may occur when the user variable fails to initialize a local variable properly.

To locate a stray pointer, use **sdb** or **db**.

Array Index Out of Bounds

Memory may be corrupted if an array index is out of bounds.

To avoid this problem, be sure that all subscripts are valid. This can be done using the `assert` function.

Calls to Library Functions

If you encounter an error in a large program that you think might be related to calling in a library function, test the function call first in a small program.

Library Function Return Values

When you call one or more library functions, be sure to check their return values, if any, if your program does not work.

Assignment vs. Equals

Remember that the following code:

```
A = B
```

represents "assignment," not "equal."

Use

```
A == B
```

to compare A and B.

Function Return Values

The program will not run if the return value or the parameters of a function are wrong.

To avoid this problem, use prototypes. The **proto** can be used to generate prototypes. For information on **proto**, see the manual page for **PROTO** in this release document.

TECHNICAL SUPPORT

While we do our best to ship problem-free software, problems sometimes occur. Manx has a Technical Support staff ready to help you out if you should encounter problems while using our software. A Problem Report form is located in this section to assist you in describing your problem. In addition, our Technical Support Staff is available from Monday through Friday 10am-12Noon and 2pm-5pm EST. They can be reached at (201) 542-1795.

NAME

ctoeng - C-to-English Translator

SYNOPSIS

ctoeng [C declaration]

DESCRIPTION

ctoeng takes a C declaration from standard input (keyboard) or the command line and writes the corresponding English description to standard output.

Only valid C declarations should be used because of the limited capability for syntax error checking.

EXAMPLES

char *(*id[3]);

generates

a three element arrray of pointers to char

while

char *(*id) ();

generates

a pointer to function returning pointer to char

SUPPORT

To a limited degree, **ctoeng** *does* support scalar variables, array declarations, and function definitions. It *does not* support structs, unions, or enums.

NAME

engtoc - English-to-C translator

SYNOPSIS

engtoc [english description]

DESCRIPTION

engtoc takes an English description of a C declaration from standard input or the command line and writes the C declaration to standard output.

An English description can be a standard C type specifier; e.g. `char, int, long, float, double, short, void, volatile.`

If *desc* is an English description, the following are, too (brackets indicate optional entries):

pointer [to] *desc*
function [returning] *desc*
[a] <number> element [array of] *desc*

The following keywords can be abbreviated to their first three characters: array, char, double, element, float, function, long, pointer, short, signed, void, and volatile.

Make the following keywords plural, if necessary, by adding "s" to the end of the word: array, char, double, element, float, function, int, long, pointer, short, and signed.

EXAMPLES

Input:

```
a 3 element arr of pointers to char
```

Output:

```
char *id[3];
```

Input:

```
poi to 3 ele arr of unsigned long
```

Output:

```
unsigned long (* id ) [3];
```

NAME

CPP - C-Pre-Processor

SYNOPSIS

cpp [-k -3 -d<id>=<macro> -o <filename> -n]

DESCRIPTION

cpp is a C preprocessor which uses command line options and your C program as input. The output file generated substitutes your #defines, #includes, #ifdefs, and line directives with their actual values.

NAMING CONVENTIONS

prog.c

```
generates an output file
```

prog.i

OPTIONS

-k suppress extended keywords.

-3 use old preprocessor rules

-d<id><Macro> allows you to define a macro

-n suppress the #line directive

-b specify pathname of the include file directory.

-o<filename> name of the output file

LIMITATIONS

All constant expressions evaluate to zero on #IF statements.

NAME

proto - function prototype declarations

SYNOPSIS

proto [-options] file1 [file2...]

DESCRIPTION

The **proto** utility program creates a file of function prototype declarations for the functions that are defined in the C source files **file1, file2, ...**

Input Files

The specified filenames can contain "wildcard" characters. Also, the extension on a file name is optional; if not specified, it's assumed to be ".c". For example,

***.c**	All files in the current directory whose extension is .c
f*.c	Files in the current directory whose name begins with "f" and whose extension is .c
t*	Files in the current directory whose name begins with t and whose extension is .c.

proto also automatically generates prototypes for functions defined in a C source file's `#include` files.

Output file

By default, **proto** writes generated prototypes to the file **proto.h**. This can be overriden using the **-o** option. This option is followed by the name of the desired output file. For example, the following command writes prototypes for the C source file **prog.c** to **prog.pro**:

proto -o prog.pro prog.c

By default, **proto** creates a new file before writing the prototypes to it. The **-a** option tells **proto** to append the prototypes to an existing file.

Options

-a	Append prototypes to the output file, if it exists.
-i path	Search for `#include` files in the directory defined by **path**, in addition to the directories defined by the INCLUDE environment variable.
-o outfile	Write prototypes to **outfile** instead of **proto.h**.
-r	Ignore register class in formal parameter with register attribute (default: put storage class "register" in prototype).
-b	Generate prototypes for all functions (default: dont generate prototypes for static functions)
-c	Suppress comments in the output file

NAME

hd - Hex Dump Utility

SYNOPSIS

hd [-r] [+n[.]] file1 [+n[.]] file2 ...

DESCRIPTION

hd displays the contents of one or more files in hex and ascii to its standard output.

file1, file2,... are the names of the files to be displayed.

+n specifies the offset into the file where the display is to start, and defaults to the beginning of the file.

if **+n** is followed by a period, **n** is assumed to be a decimal number; otherwise it is assumed to be hexidecimal. Each file will be displayed beginning at the last specified offset.

EXAMPLES

hd +16b oldtest newtest +0 junk

displays the data forks of the files **oldtest** and **newtest**, beginning at offset 0x16b, and of the file named **junk** beginning at its first byte.

hd -r +1000. tstfil

displays the contents of the resource fork of **tstfil** beginning at byte 1000

NAME

strsrt, stricmp, strrev, strlwr

SYNOPSIS

```
char *strstr(s1,s2)
const char *s1, *s2;
int stricmp(s1,s2)
const char *s1, *s2;
char *strrev(s1);
char *s1;
char *strlwr(s1);
char *s1;
```

DESCRIPTION

Strstr finds the first substring in **s1** identical to **s2**. If it is found, **strstr** returns a pointer to it, else it returns a NULL pointer.

Stricmp does a string compare with no case sensitivity, e.g. the letters "A" and "a" are considered the same. **Stricmp** returns either less than zero, zero, or greater than zero, depending on whether **s1** is lexigraphically greater than, equal to, or less than **s2**.

Strrev takes a pointer to a string and reverses the order of it, overwriting the original string. **Strrev** returns a pointer to the new string.

Strlwr takes a pointer to a string and converts it to lower case letters, thus overwriting the original string. **Strlwr** returns a pointer to the new string.

EXAMPLES

```
main()
        {
        printf("%s\n", strstr("hello world", "world"));
        printf("%d\n", stricmp("aBc", "abc");
        printf("%s\n", strrev("hello world");
        printf("%s\n", strlwr("This Had Caps"));
        }

        generates the following output:

        world
        0
        dlrow olleh
        this had caps
```

Aztec C86, version 4.10
for PC-DOS and MS-DOS, versions 2.x and 3.0
Release Document

This release document introduces the features of Aztec C86, version 4.10 for PC-DOS and MS-DOS. It's divided into the following sections:

1. *Description of the Package*
2. *Information for New Users*
3. *New Features*
4. *Packaging*
5. *Technical Support*
6. *Additional Documentation*

This release document contains updates made to the manual since it was printed as well as new enhancements to our software environment.

1. Description of the Package

Aztec C86, version 4.10, consists of software and a manual for developing programs in the C language using PC-DOS or MS-DOS.

There are three Aztec C86 systems available: *Professional*, *Developer*, and *Commercial*. The *Developer* system is a superset of the *Professional*, and the *Commercial* is a superset of the *Developer*.

The systems are supplied on floppy disks whose contents are described in the *Packaging* section of this document.

If you're a new user of Aztec C86, your package also includes a manual.

There are three sources of documentation for Aztec C86: (1) the manual; (2) documentation that's appended to this release document, which describes features that have been added to Aztec C86 since the manual was last printed; and when appropriate (3) a read.me file on the disks, which describes features that have been added since the release document was printed. Taken together, this documentation describes all the features of the *Commercial* version of Aztec C86. If you have the *Professional* or *Developer* system, your package doesn't have all the documented features.

1.1 Differences between the three systems

As mentioned above, the *Developer* system is a superset of the *Professional*, and the *Commercial* is a superset of the *Developer*. In this section we're going to describe the main differences between the three systems.

The main components of the *Professional* system are these:

* The compilers, assembler, and linker;
* Object module utilities;
* 8087/80287 support;
* Libraries supporting two memory models: one for the small code, small data memory model; and the other for large code, large data.
* The debuggers *sdb* and *db*;
* The Z text editor;
* The C driver program, *c*.

The *Developer* system contains all the components of the *Professional*, plus the following:

* The "Unitools" programs *make*, *grep*, *diff*, and *ls*;
* *pcz*, a memory-mapped version of Z for IBM PCs and true compatibles;
* Libraries for the other two memory models (large code, small data; and small code, large data);
* The *prof* profiler.

The *Commercial* system contains all the components of the *Developer*, plus the following:

* Source to the library functions;
* Support for generation of ROMable code;
* Libraries for creating CP/M-86 programs;
* One year of updates.

2. Information for New Users

The best way to acquaint yourself with our package is to go through the overview and tutorial sections in the manual. This will provide an introduction to your C programming environment by walking you through the commands needed to compile, assemble, and link the sample program provided. The sections on the compiler, assembler, linker, and libraries will provide you with additional information and options to allow you to make the most out of the product that you have received.

Another section in the manual to read is the style chapter. This chapter explains some common pitfalls and things to watch out for.

3. New features

This section summarizes the features that have been added to Aztec C86 in going from version 3.40b to version 4.10a. Complete descriptions of these features are appended to this release document.

3.1 The compiler

The following list describes the changes that have been made to the compiler in going from version 3.40b to 4.10a:

3.1.1 Two passes

The compiler has been divided into two passes. Pass 1 is named *cc.exe*, and pass 2 is named *cgen.exe*. *cc* automatically starts *cgen*.

3.1.2 ANSI support

Aztec C86 now supports the following features of the proposed ANSI standard: (1) function prototypes, (2) the ANSI preprocessor, (3) the keywords *const* and *volatile*, (4) expression evaluation using value-preserving rules, and (5) switches whose expression is of type *long*.

There are two new compiler options related to the new ANSI features: *-ansi* and *-3*. The *-ansi* option makes the compiler behave as much as possible like an ANSI compiler, leaving the ANSI features enabled and disabling any non-ANSI features. The *-3* option makes the compiler accept programs written for version 3.40b of the Aztec C86 compiler; this requires the compiler to disable some of the ANSI features.

Another new option, *-c*, causes the compiler to issue a warning message when it automatically converts an argument to a prototyped function.

The first formal review period for the draft proposed ANSI standard has ended, and there will probably be a second review period later this year. The proposed standard will probably become a real standard in the first half of 1988.

3.1.3 Inline 8087 support

The compiler can now generate inline 8087 code. This is enabled using one of two options:

+e	When necessary, save 8087/80287 registers between subroutine calls.
+ef	Don't save 8087/80287 registers between subroutine calls.

3.1.4 Compiler support for inline 80287 instructions

If you specify the compiler options that enable 80286 support (the *+2* option) and inline-floating point code (the *+e* or *+ef* option) the compiler will automatically generate code that supports the 80287, by starting the assembler with the new *-2* option.

3.1.5 New keywords: *near, far, huge*

Aztec C86 now allows you to explicitly define the addressing technique used to access specific data items and functions, using the

keywords *near*, *far*, and *huge*. Data items and functions for which these keywords aren't used are accessed using the addressing technique associated with the program's memory model.

Support for these keywords is disabled by the new option *-k*, and by the *-ansi* option.

By default, these keywords 'bind' just like the ANSI keywords *const* and *volatile*. The new option *-Ze* causes them to bind as specified by the Microsoft compiler.

3.1.6 New keywords: *fortran, pascal, cdecl*

These keywords are reserved, but are not yet functional.

Support for these keywords is disabled by the *-k* option, and by the *-ansi* option.

3.2 The assembler

To support the 8087 and 80287 math coprocessors, the following features have been added to the assembler.

* Support for the 8087 and 80287 instructions, by implementing their codemacros.
* Support for the codemacro parameter specifiers F and T, which define a floating point register.
* Support for the floating point codemacros *rfix*, *rfixm*, *rnfix*, *rnfixm*, and *rwfix*.
* Support for the *qword* and *tbyte* data items.

In addition, the new *-2* option enables the assembler's support for the 80287 chip, by preventing the assembler from generating a WAIT instruction when it processes an instruction for a math co-processor.

3.3 SDB: new features

sdb supports the following new features:

* Disassembly of floating point instructions;
* New options for the *bs* command, which allow a breakpoint to be set to a function's return address. If the function isn't currently active, *sdb* will automatically set the breakpoint on entry to the function.
* Enhancements to the *bd* command, to support the changes made to the *bs* command.
* A new command , 'P', that is like 'p', except that it prints the address of displayed items.
* The print command's 'format override' specifier, '@', is now optional.

* When printing a string, the print command can now optionally display non-printable characters using the standard backslash notation.
* You can now specify to the print command the number of characters to be printed in a string.
* The precision specifier, a period followed by a number, now applies just to the next string- or float-specifier, and not to subsequent specifiers.
* In assembly mode, the 't' command now causes *sdb* to skip over function calls, instead of single-stepping into the called function.
* When *sdb* is in source mode and enters a function for which no source information exists, it will suspend source mode and enter assembly mode until it gets back to a function for which source information exists.
* Variable names can be qualified, specifying a function from which they're visible, or a file that contains them.
* *sdb* can search for an unqualified name, in the current module and in the file whose source is currently being displayed with the *df* command.
* When displaying a source statement, *sdb* now displays the comments that precede the statement rather than those that follow it.
* User variable names take precedence over register names.
* While *sdb* is active, you can execute a DOS command, using *sdb*'s new '!' command.
* *sdb* can make use of two screens: one for itself, the other for the program being debugged.

3.4 Z - new features

Z has been enhanced to allow editing of large files, to support the EGA 43-line mode, and to support color displays.

3.5 The *printf* functions

The *printf* functions have been enhanced to support the ANSI definition, and to support pointer arguments that use the *near*, *far*, and *huge* keywords.

3.6 New functions: *lmalloc*, *lcalloc*, *lrealloc*, *lfree*

Several new memory allocation functions are provided in this release: *lmalloc*, *lcalloc*, *lrealloc*, and *lfree*. These are similar to the UNIX-compatible functions *malloc*, *calloc*, *realloc*, and *free*, except that they can allocate a buffer that's larger than 64kb.

3.7 Standard I/O access of the preopened auxiliary and printer devices

The preopened auxiliary and printer devices can now be accessed by programs via the standard I/O functions. To do so, the program must #define the symbol MSDOS before #including *stdio.h*, and must refer to these devices using the names *stdaux* and *stdprt*.

3.8 New features of the *agetc* function

Before returning a character, the *agetc* function masks it with the contents of the global *int* *_agetc*. A program can change this field, thereby changing the mask.

3.9 Linking old and new object modules together

You can link together object modules that have been created using Aztec C86 version 3.4 and version 4.10a, since the object module format hasn't changed. However, you must use the new object module libraries, since several internal library functions have been added and/or changed.

4. Packaging

The Aztec C86 *Developer* system is a superset of the *Professional*, and the *Commercial* system is a superset of the *Developer*.

This section first lists the files that are common to all three systems. It then lists the files that are in the *Developer* and *Commercial* systems, but not in the *Professional*. It then lists the files that are only in the *Commercial* system.

4.1 Files that are in all versions of Aztec C86

4.1.1 Executable programs

The following executable programs are in all versions of Aztec C86:

CC.EXE	Optimizing C Compiler, pass 1
CGEN.EXE	Optimizing C Compiler, pass 2
CCB.EXE	Non-optimizing C Compiler
AS.EXE	Assembler
LN.EXE	Linker
OBJ.EXE	Aztec C-to-Microsoft object convertor
DB.EXE	Assembly language Debugger
SDB.EXE	Source level debugger
LB.EXE	Object file librarian
ORD.EXE	Object library generation utility
CNM.EXE	Object file utility
OBD.EXE	Object file utility
SQZ.EXE	Object file utility
CRC.EXE	CRC utility
ARCV.COM	Source archive utility
C.EXE	C driver
TERM.EXE	Terminal emulator program

Z.EXE	Text editor (non-memory mapped)
CTAGS.COM	Text editor utility

4.1.2 Libraries

The following libraries are in all versions of Aztec C86. Each uses the 'small code' and 'small data' memory model.

C.LIB	Library of non-floating point functions
M.LIB	Library of floating point functions (non-8087 version)
M87.LIB	Library of floating point functions (8087 version)
M87S.LIB	Library of floating point functions (sensing version)
S.LIB	Screen functions
G.LIB	Graphics functions

All Aztec C86 systems also contain a 'large code', 'large data' version of each of the above libraries. The name of a 'large code', 'large data' version of a library is derived by appending the letter 'l' to the name of the 'small code', 'small data' version of the library. For example, the name of the 'large code', 'large data' version of *c.lib* is *cl.lib*.

4.1.3 Object modules

The following object modules are in all versions of Aztec C86:

OVLD.O, OVLDPATH.O, OVBGN.O	Object modules for overlay support
CRT0.OBJ	Object module of Startup routine for programs linked with Microsoft libraries

4.1.4 Header files

All Aztec C86 systems include several header files, which can be included in C programs. These files have extension *.h*.

4.1.5 Source archives

The following source archives are in all versions of Aztec C86. They can be unpacked using the *arcv* program.

S.ARC	Screen functions
G.ARC	Graphics functions
TERM.ARC	terminal emulator programs

4.1.6 Miscellaneous

The source file *stksiz.c* controls the size of a program's stack and heap, and the relative positioning of these two areas. For details, see the Programming Organization section of the Technical Information

chapter.

The file *exmpl.c* contains source to a sample C program.

4.2 Files that are only in *Developer* and *Commercial* systems

4.2.1 Executable programs

The following programs are only in the *Developer* and *Commercial* versions of Aztec C86:

MAKE.EXE	Program maintenance utility
DIFF.EXE	Source file comparator
GREP.EXE	Pattern matcher
LS.EXE	File listing utility
PCZ.EXE	Text editor (PC memory mapped)
PROF.EXE	Program profiler

4.2.2 Libraries

In addition to 'small code', 'small data' and 'large code', 'large data' versions of each library, the *Developer* and *Commercial* versions of Aztec C86 contains a version that uses the 'large code', 'small data' memory model and a version that uses the 'small code', 'large data' memory model. The name of the 'large code', 'small data' version of a library is derived by appending the letters *lc* to the name of the 'small code', 'small data' library, while the name of the 'small code', 'large data' version is derived by appending *ld.*

For example, the name of the 'large code', 'small data' version of *c.lib* is *clc.lib.*

4.3 Files that are only in the *Commercial* System

4.3.1 Source archives

The following source archives are only in the *Commercial* version of Aztec C86. They can be unpacked using the *arcv* program.

STDIO.ARC	Standard I/O functions
MISC.ARC	Miscellaneous functions
MCH86.ARC	Miscellaneous functions
MATH.ARC	Floating point functions
DOS20.ARC	DOS 2.x functions
CPM86.ARC	CP/M-86 functions
DOS11.ARC	PC-DOS/MS-DOS 1.1 functions

4.3.2 Files for creating ROMable code

The following *Commercial* system files are used to generate ROMable code:

HEX86.EXE	Intel hex record generator
SROM.O	Object module of Startup routine for ROMable programs that use 'small code, small data'

LROM.O	Object module of Startup routine for ROMable programs that use 'large code, large data'
LCROM.O	Object module of Startup routine for ROMable programs that use 'large code, small data'
LDROM.O	Object module of Startup routine for ROMable programs that use 'small code, large data'

4.4 Checking the files

The file *crclist* contains the CRC values for the files. You can compute the CRC values of the files we sent you and then compare them with their expected values, using the program CRC. For example, entering

```
crc *.*
```

computes the CRC of all the files on the current directory of the default drive.

5. Technical support information

While we do our best to ship problem free software, problems sometimes occur. Manx has a technical support staff ready to help you out if you should encounter problems while using our software. At the very end of this document is a discussion of how to make the most out of the technical support that Manx offers. In addition, we have added problem report forms for the reporting of any problems you may encounter with our software.

6. Additional Documentation

This section contains documentation of features that have been added to Aztec C86 since edition 4 of the manual was printed. In particular, it discusses the following topics:

* New compiler features
* The ANSI preprocessor
* Defining the memory model of individual items
* New assembler features
* New linker features
* New Z features
* New features of the *printf* functions.
* New *lmalloc*, *lcalloc*, *lrealloc*, and *lfree* functions.
* Support for stdprt and stdaux
* New features of the *agetc* function
* New *sdb* features
* *sdb* tutorial
* New C driver program, *c*.
* Additions to the manual's description of the *open* function.
* New *filelock* function.
* Corrections to the manual's description of the *make* program.

* Corrections to the manual's description of the *scdir* function.
* Technical Support Information.

The description of new features of the compiler, assembler, linker, and *sdb* can be added to the end of the corresponding manual chapters.

The descriptions of Z's new features can be appended to the Unitools chapter.

The descriptions of the new features of the *printf* functions can be appended to the manual's System Independent Functions chapter.

The description of the *lmalloc*, *lcalloc*, *lrealloc*, and *lfree* functions can be added to the "8086 Functions" chapter.

The description of *agetc*'s new features can be added to the "8086 Functions" chapter.

The description of the support for Standard I/O access of the preopened auxiliary and printer devices can be added to the chapter entitled "Library Functions Overview: 8086 Information".

The *sdb* tutorial takes you through the startup and some of the more commonly used commands of the source level debugger. You can place this tutorial at the front of the *sdb* chapter.

The C driver, *c*, is a program that generates a program. It invokes the compiler and assembler to generate the program's object modules, and then invokes the linker to combine the modules into an executable program. To put it in perspective, it's similar in function to the UNIX *cc* command, more powerful and flexible than the PCDOS/MSDOS batch facility, and less powerful (but easier to use) than the *make* program. You can place the description of the C driver at the end of the manual's *Unitools* chapter.

The description of the new options available to the *open* function can be placed after the description of *open* in the manual's System Independent Functions chapter.

The description of the *filelock* function can be added to the chapter entitled "8086 Functions".

The discussion of *MANX technical support* and the problem report forms can be put at the very end of your manual.

The Compiler: new features

The following list describes features that have been added to the compiler since the manual was last printed. The description of each feature lists the version of Aztec C86 in which the feature was introduced.

1. Two passes

The compiler is divided into two passes. Pass 1 is named *cc* and pass 2 is named *cgen*. *cc* automatically starts *cgen*.

The compiler was first split into two passes for version 4.10a.

2. ANSI support

Aztec C86 supports some of the features of the proposed ANSI standard.

The compiler supports two options related to the new ANSI features: *-ansi* and *-3*. The *-ansi* option makes the compiler behave as much as possible like an ANSI compiler, leaving the ANSI features enabled and disabling any non-ANSI features. The *-3* option makes the compiler accept programs written for version 3.40b of the Aztec C86 compiler; this requires the compiler to disable some of the ANSI features. These options are described in detail later in this section.

ANSI support was first available in version 4.10a of Aztec C86.

2.1 Function prototypes

The compiler supports function prototypes, as defined in the proposed ANSI standard. A function prototype defines the types of a function's arguments in addition to the type of its return value. For example, the following is a prototype definition of the function *subr*, which returns a double as its value, and which is passed three arguments: a double, a pointer to a long int, and a pointer to a structure of the typedef-ed type FILE:

```
double subr(double, long *, FILE *);
```

When the compiler encounters a call to *subr()*, it will examine the arguments being passed to *subr*:

* If there's too few or too many arguments, the compiler will log an error message.
* If the type of an actual argument differs from that of the corresponding prototype argument, and if conversion of the actual argument's type to the prototype argument's type is valid, the compiler will automatically generate code that performs the conversion.

By default, the compiler will issue warning messages when it generates code to convert the type of an argument to a prototyped function. The new *-c* option toggles the flag that specifies whether these messages are displayed; and the *-ansi* option turns off this flag.

2.2 Support for ANSI preprocessing directives

Aztec C86 by default supports the ANSI preprocessor. The features of this preprocessor are described in a separate appendix to this release document.

The *-3* option disables the ANSI preprocessor and enables the UNIX-compatible preprocessor that was supported by earlier versions of Aztec C86.

2.3 Expression evaluation using ANSI-defined, 'value-preserving' rules

Aztec C86 by default generates code that computes expressions using the ANSI-defined rules, which are called 'value-preserving'. With these rules all data types are assigned a rank. When two operands are used in an expression, the operand having lesser rank is promoted to the type of the other operand. The ranking of data types, ordered from lowest to highest is as follows: char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, double.

The rules used by previous versions of the compiler were called 'unsigned-preserving'. With these rules, all signed data types are assigned a rank. When two operands are used in an expression, the expression is first evaluated as if both operands are signed, after promoting the lesser-ranked operand to the type of the other operand. Then, if either operand is unsigned, the type of the expression is set to 'unsigned'.

As an example of the difference in these rules, consider the addition of an unsigned char to a signed int: for 'value-preserving rules', the type of the sum is signed int, while for 'unsigned preserving' rules, it's unsigned int.

The option *-3* causes the compiler to evaluate expressions as it did in version 3.40b; i.e. using unsigned-preserving rather than value-preserving rules.

2.4 Evaluation of shift operations using ANSI-defined rules

Aztec C86 generates code that evaluates shift operations as defined by the proposed ANSI standard. According to the standard, the type of the shift operation is determined by the type of the left operand. The type of the right operand has no effect on the resultant type; and in fact the right operand is converted to an int before the shift occurs.

In previous versions of the compiler, both operands participated in determining the type of a shift operation, using unsigned-preserving rules.

The *-3* option has no effect on the evaluation of shift expressions: the compiler always evaluates shift expressions using ANSI rules.

2.5 Support for the type specifiers *volatile* and *const*

Aztec C86 supports the type specifiers *volatile* and *const*. A *volatile* data item may be modified in ways unknown to the compiler, so the compiler won't perform register tracking on it. For example, volatile variables can be used as synchronization flags between separate programs.

A *const* data item cannot be written to, and the compiler can do register tracking on it.

It's possible for a data item to be both *const* and *volatile*. Such an item may be modified by hardware, but it can't be assigned to, incremented, or decremented.

3. Inline 8087 support

The compiler can generate inline 8087 code. This is enabled using one of two options:

+e	When necessary, save 8087/80287 registers between subroutine calls.
+ef	Don't save 8087/80287 registers between subroutine calls.

The *+ef* option is provided for Microsoft compatibility. It generates faster code than the *+e* option, but the resultant program may not work correctly: there aren't many 8087 registers, and the program may reuse some that are already in use.

Inline 8087 support was first available in version 4.10a of Aztec C86.

4. Compiler support for inline 80287 instructions

If you specify the compiler options that enable 80286 support (the *+2* option) and inline-floating point code (the *+e* or *+ef* option) the compiler will automatically generate code that supports the 80287, by starting the assembler with the *-2* option. For information on the *-2* option, see the discussion of the assembler.

80287 support was first available in version 4.10a of Aztec C86.

5. Defining the memory model for selected items

The Aztec C86 compiler now allows you to explicitly define, using keywords *near*, *far*, and *huge*, the memory model used by select data items and functions, thus overriding the default memory model.

Support for these keywords is disabled by the option *-k*, and by the *-ansi* option.

This topic is discussed in a separate appendix to this release document.

6. The *fortran, pascal, cdecl* keywords

These keywords are reserved, but are not yet functional.

Support for these keywords is disabled by the *-k* option, and by the *-ansi* option.

Support for these keywords was first available in version 4.10a of Aztec C86.

7. The *-ansi* option

The *-ansi* option makes the compiler look, as much as possible, like an ANSI compiler, by enabling the ANSI features and disabling non-ANSI extensions. Specifically, the *-ansi* option:

* Enables the ANSI preprocessor;
* Turns off the flag that specifies whether warning messages are displayed when an argument to a prototyped function is automatically converted;
* Enables expression evaluation using 'value-preserving' rules;
* Disables the following keywords: *near*, *far*, *huge*, *pascal*, *fortran*, *cdecl*.

Support for the *-ansi* option was first available in version 4.10a of Aztec C86.

8. The *-3* option

The *-3* option makes the compiler accept programs that were previously compiled using the version 3.40b compiler. This option doesn't affect those features of the compiler that were not available in the version 3.40b compiler.

Specifically, the *-3* option:

* Enables the v3.40b preprocessor and disables the ANSI preprocessor;
* Enables expression evaluation using 'unsigned-preserving' rather than 'value-preserving' rules.

Support for the *-3* option was first available in version 4.10a of Aztec C86.

9. Code generator improvements

To generate better code, the following changes have been made to the compiler in version 4.10a:

* Register tracking has been improved;
* The floating point code generator has been completely reworked;
* The bit field code generator has been completely reworked;
* If an expression has no side effects (e.g. it doesn't modify memory), the compiler doesn't generate code for it;
* For control structures whose test is known in advance (e.g. while (1) and if (0)) the compiler generates a jmp instruction rather than a test and jmp.

10. Support for *sdb*, the source-level debugger

The compiler by default generates information used by *sdb*, the Aztec source-level debugger. The *-n* option tells the compiler not to collect and save this information thus resulting in an increase in compilation speed.

Support for *sdb* was first available in version 3.40a of Aztec C86.

11. Placement of string constants

By default, a program's string constants are put in the program's data segment. For a module that uses the 'large data' memory model (i.e. that has been compiled with the +l or +ld option), the compiler's *+s* option causes string constants to be placed instead in the code segment. This option is useful if you are creating ROMable code.

The *-s* option was first available in version 3.40a of Aztec C86.

12. Stack overflow

The *+b* option tells the compiler to generate code for a program that will, on entry to a function, see if the program's stack has overflowed its area. If so, a message ("stack overflow, raise stack size") is output. This message is contained in a module, *_stkover*, in *c.lib*; it can be modified by the user if different behavior is desired.

The *+b* option was first available in version 3.40a of Aztec C86.

13. Summary of new compiler options

The following list summarizes the compiler options that are not listed in the manual.

-3	Make the compiler revert to its v3.40b behavior.
-ansi	Enable the ANSI features, and disables non-ANSI extensions.
-c	Toggles the flag that specifies whether to issue

warning messages when an argument to a prototyped function is automatically converted.

-k	Disable support for the keywords *near*, *far*, *huge*, *pascal*, *fortran*, *cdecl*.
-ze	Make the keywords *near*, *far*, and *huge* behave as defined by Microsoft.
+e	Generate in-line 8087/80287 floating point instructions. When necessary, save and restore floating point registers between function calls.
+ef	Generate in-line 8087/80287 floating point instructions. Don't save and restore floating point registers between function calls.
-n	Don't generate *sdb* information
+s	Put string constants in the code segment.
+b	Generate code that checks for stack overflow.

The ANSI C Proprocessor

Version 4.10a of the Aztec C86 compiler supports two preprocessors: the preprocessor defined in the proposed ANSI standard, and the UNIX-compatible processor that was supported by previous versions of Aztec C86. This section highlights the features of the ANSI preprocessor. For a complete description of the ANSI preprocessor, see the proposed ANSI standard. For a description of the UNIX preprocessor, see the Compiler section of the Aztec C86 manual.

This section discusses the following features of the ANSI preprocessor:

1. Directive syntax;
2. The *#include* directive;
3. The *#if* directive;
4. The *#define* directive.
5. Miscellaneous directives;

1. Directive syntax

ANSI preprocessing directives occupy a single source file line, and have the following format:

```
# dirname [operands]
```

where *dirname* is the name of the directive and *operands* are its operands. The '#' need not be the first character on the line; whitespace characters (space, tab, comments) can precede it. Whitespace can separate the '#' from the directive name. Whitespace must separate the directive name from its operands.

The preprocessor supports macro substitution, as described below, but the directive name cannot itself be the result of a macro expansion. Thus, you can't create your own preprocessor directives.

2. The *#include* Directive

As with the UNIX preprocessor, the *#include* directive causes the ANSI preprocessor to suspend the compilation of one source file, compile another, and then continue compilation of the suspended file. *#include* statements can be nested.

The *#include* directive still supports the angle-bracket and double-quote syntax for specifying include files. For example, the first of the following two *#include* directives causes the preprocessor to search for *stdio.h* in the directories defined in the compiler's -I option and in the INCLUDE environment variable. The second causes the preprocessor to search for *myhdr.h* in the current directory, then in the -I

directories, and finally in the INCLUDE directories.

```
#include <stdio.h>
#include "myhdr.h"
```

If the operand to the *#include* directive isn't in angle-bracket or quoted-string form, the ANSI preprocessor will treat the remainder of the line (i.e. the part that follows the *#include*) as normal text, and perform macro expansions. The resultant text then *must* be an angle-bracket or double-quote specification of a file name. For example, the following statements cause the preprocessor to include the statements that are in *vars.h*.

```
#define hdr <vars.h>
#include hdr
```

3. The *#if* Directive

There are several *#if* directives: *#if*, *#ifdef*, *#ifndef*, *#elif*, and *#endif*. Of these, only *#elif* was not supported by previous versions of Aztec C86. This new directive is syntactically similar to *#if* and has the same purpose as *else if* does in the rest of the language, removing the necessity for nesting *#if*'s to obtain a simple selection. Except for the treatment of the constant expression following *#if* and *#elif*, all of the *#if* directives behave as in the UNIX preprocessor. Six levels of nesting is guaranteed.

The constant expression in an *#if* or *#elif* is evaluated following normal C rules, with the following exceptions:

* The expression must have an integer value.
* It can't use the *sizeof* operator, casts, or enumeration constants.
* All integer constants in the expression are treated as if they were followed by 'L'.
* Undefined symbols are replaced by the value 0.

The ANSI preprocessor supports a new unary operator, named *defined*, which has the following two forms:

```
defined identifier
defined ( identifier )
```

defined evaluates to one if the macro name *identifier* is currently defined as a macro, otherwise zero. Thus *#ifdef name* can be thought of as being equivalent to *#if defined(name)*, and *#ifndef name* as equivalent to *#if !defined(name)*

4. The *#define* directive

As with the UNIX preprocessor, the ANSI processor supports macro definition using the *#define* macro, and macros undefinition using the *#undef* directive. The syntax of these directives is still the

same: in particular, *#define* supports the definition of function-like macros that have parameters, and of object-like macros that don't.

Two definitions of the same macro are not permitted unless (1) there is an intervening *#undef* of the macro or (2) the two definitions are identical except for white space. This prohibits stacking of definitions, but permits "benign" redefinition.

An invocation of a function-like macro is not recognized unless the macro name is followed by an open parenthesis, nor are macros recognized in string literals or character constants. For example, in the following example, only the second occurrence of *mac* is recognized as an invocation of the *mac* macro.

```
#define mac(x) x
int mac;
char a[]="hello mac(world)";
int mac(c);    /* expands to int c; */
```

Macro invocations, as opposed to definitions, may occupy more than one line of source, since a newline is considered as just another white space character within invocations.

Arguments to a macro invocation are separated by commas. An argument can be parenthesized, but the parentheses must be matched. Commas inside of a parenthesized argument are not considered to be argument-separators. For example, the arguments to the following macro call are *(a, b)* and *c*:

```
m((a,b),c)
```

4.1 The 'stringize' operator,

The UNIX preprocessor performs substitution of macro arguments that occur within a macro body's quoted strings, but the ANSI preprocessor doesn't. For example, consider the following statements:

```
#define pr(x) printf("x=%d", x)
pr(a+b);
```

The UNIX preprocessor expands the second statement to

```
printf("a+b=%d", a+b);
```

The ANSI preprocessor expands it to

```
printf("x=%d", a+b);
```

To allow creation of strings containing macro arguments, the ANSI preprocessor provides the 'stringize' operator, #. If, during expansion of a macro, the preprocessor finds a # followed by a macro argument in the macro body, it replaces the # and the argument with a character string consisting of the argument value surrounded by double quotes. When combined with the rule that string literals, separated only by white-space, are treated as a single string literal, this allows you to

build strings that contain macro arguments.

For example, the *#define* used above could be rewritten as:

```
#define pr(x) printf(#x "=%d", x)
```

When # is used, the original spelling of the argument is retained. For example, using the *pr* macro defined above, the statement *pr(0x0001)* causes *0x0001=1* to be written to stdout.

Normally, when the ANSI preprocessor is expanding a macro function and finds an argument in the macro body, it performs macro expansion on the argument and substitutes the resulting value into the macro body. However, when the argument is preceded by the stringize operator, #, the argument is not first macro-expanded. For example, the following macro call expands to *a*, not *b*:

```
#define m(x) #x
#define a b
m(a);
```

4.2 The 'concatenate' operator,

The ANSI preprocessor supports the concatenate operator, ##, which is used with the body of a macro to concatenate macro arguments. This operator isn't supported by the UNIX preprocessor. For example, the following invocation of the *concat* macro expands to *ab* since *ab* is not an argument of *concat*:

```
#define mac(a,b)  ab
mac(_,func)
```

The next invocation of *concat* expands to *_func*, since the ## operator allows the preprocessor to identify the two arguments:

```
#define mac1(a,b)  a##b
mac(_,func)
```

As with the 'stringize' operator, arguments used with the concatenate operator are not macro-expanded before they are substituted into the macro body.

4.3 Recursion

In the ANSI preprocessor, macro invocations are not recursive, even indirectly, although the replacement string after expansion is examined for invocations of other macros. Of course, if a macro's invocation has an invocation of itself as an argument, the argument is expanded.

For example, the following code expands to *x = *sin(3.14)*:

```
#define sin(x)  *sin(x)
x = sin(3.14);
```

The next example expands to *mac(((3)+1)-1)*:

```
#define mac(a)  macb((a)+1)
#define macb(a)  mac((a)-1)
mac(3)
```

The next example expands to *((3)+1)+1*:

```
#define add(x) (x)+1
add(add(3))
```

4.4 Predefined macros

The ANSI preprocessor predefines the following macros

Macro	*Value*
___LINE___	A decimal constant representing the number of the current source line;
___FILE___	A string literal containing the name of the current source file;
___DATE___	The compilation date, in "Feb 5 1987" form;
___TIME___	The compilation time, in "13:01:22" form;
___STDC___	A decimal constant one, indicating conformance.

None of these macros can be #*undef*'ed, nor can any other macro identifiers be predefined.

5. Miscellaneous Directives

The miscellaneous directives are these:

Directive	*Meaning*
#*line*	Redefines the compiler's notion of the number of the current source line and optionally the name of the current source file;
#*error*	Causes the preprocessor to produce a diagnostic message that includes the processed remainder of the line. This is a convenient means of obtaining an error from the preprocessing phase of compilation.
#*pragma*	Aztec C86 doesn't currently support any pragmas.
#	Ignored (present for historical reasons).

Defining the memory model of selected items

By default, the memory model used by a module defines the memory model used by its functions and data; i.e. the segments in which functions and data are located, and the addressing technique used to access them. For example, if a module uses the 'small code', 'small data' memory model, then (1) its functions, and the functions that it calls, are in a single code segment, (2) the data items that it accesses are in a single data segment, and (3) functions and data are accessed using 16-bit addresses.

Aztec C86 now allows you to explicitly define the memory model for specific functions and data, thus overriding their default memory model. This is done using the keywords *near*, *far*, and *huge*.

For example, suppose that your program has modest needs except for one *char* array that must be 75kb long. There's no limit to the size of a *huge* array, so you could keep the size of the program down, its speed up, and still allow the existence of this large array by specifying that the array is *huge* and that the rest of the program is to use the small code, small data memory model.

Support for these keywords is disabled by the option *-k*, and by the *-ansi* option.

By default, these keywords bind just like the ANSI keywords *const* and *volatile*. The new *-Ze* option makes them bind like the Microsoft compiler.

Support for *near*, *far*, and *huge* keywords was first available in version 4.10a of Aztec C86.

1. Near, far, and huge data items

For a data item, the keywords have the following meanings:

near A data item of type *near* is in a program's standard data segment. This segment can be up to 64kb long. It contains many things (as defined in the manual's Tech Info chapter), so defining a data item to be *near* limits its size to something less (usually much less) than 64kb. A *near* data item can be accessed using 16-bit addresses, since the program's DS segment register always points to the standard data segment.

far A data item of type *far* is in its own segment. The maximum size of this segment, and thus of the only data item that it contains, is 64kb. A *far* data item is accessed using 32-bit addresses. To determine the address of an item in the segment, the segment

component is set to that of the segment's beginning address, and the offset component is computed.

huge A data item of type *huge* is in its own segment. The size of this segment is limited only by the size of available memory. A *huge* data item is accessed using 32-bit addresses. To determine the address of an item in the segment, both segment and offset components must be computed.

Summarizing, the size of a *near* data item is less (usually much less) than 64kb; a *far* data item is limited to 64kb; and a *huge* data item is limited only by memory size. A *near* data item is accessed fastest and with the least amount of code; a *far* item slower and with more code; and a *huge* item slowest and with the most code.

We recommend that you use *huge* data items only when necessary, and when *far* data items won't suffice, because of the resultant performance degradation.

2. Near, far, and huge functions

For functions, the keywords have the following meanings:

near The function is accessed using 16-bit addresses.

far The function is accessed using 32-bit addresses.

huge Same as *far* (*huge* is mainly used for making huge data items).

Unlike *far* and *huge* data items, a *far* function is not put in its own segment: it remains in its module's code segment.

Using *near* and *far* functions, you can create programs whose modules use different code memory models. In this case, the code for the modules that have been compiled to use the 'small code' memory model are gathered together into one segment, while the code for each 'large code' module is in its own segment. Each of these code segments can be up to 64kb long.

For example, suppose you have an object library whose modules have been compiled to use the large code memory model, and that you want to call these functions from some of your own modules, which have been compiled to use the small code memory model. In your modules, simply define the library's functions as type *far*. When the program is linked, the code for your 'small code' modules will be in one segment, and the code for each of the library's 'large code' modules that is pulled into the program will be in its own segment; the maximum size of each of these segments is 64kb.

Continuing with this example, suppose that the library functions call one of your functions (you define this function to the library by

passing its address to the library functions). This function would be defined as type *far*. When called by a library function, it could still call other functions, including those of type *near*. Essentially, it provides a gateway between the library's *far* functions and your own *near* functions.

3. Example 1

This example creates an array named *arr* of 30000 *char*s. *arr* is in its own segment (which could be up to 64kb long). An array element is accessed using a 32-bit address, whose segment component is set to that of *arr*'s first element, and whose offset component is computed.

```
char far arr[30000];
```

The *far* keyword binds to *char*, which means that each element of *arr* has the *far* attribute. This attribute tells the compiler where to place the array (in its own segment), and how to compute element's addresses (32-bit addresses, single segment, computation only of elements' offset component).

4. Example 2

This example creates an array *arr* of 70000 *int*s. *arr* is in its own segment (size limited only by the size of memory). An array element is accessed using a 32-bit address, whose segment and offset components must both be computed.

```
int huge arr[70000];
```

The *huge* keyword binds to *int*, giving it the *huge* attribute. This attribute tells the compiler where to place the array, and how to compute an element's address.

5. Example 3

This example creates a *near* array of 32-bit pointers to *far* chars. That is, the array of pointers is in the standard data segment, each element of which is a 32-bit pointer to a *char* that is in its own segment. Such a segment can be at most 64kb long, so the address of another *char* in a segment is determined by setting the segment component to that of the segment's beginning address, and by computing the offset component.

```
char far * arr[100];
```

The *far* keyword binds to the *char* keyword, giving it the *far* attribute. The attribute of the * isn't specified, so it defaults to *near*.

arr is thus an array of pointers having the *near* attribute, which means that the array is in the standard data segment and can be accessed using the DS segment register and a computed 16-bit offset.

Each array element points at a *char* having the *far* attribute, which means that the pointers are 32 bits long, that a referenced segment can be at most 64kb long, and that the address of other fields in a segment are determined by setting the segment component to that of the segment's beginning address and by computing the offset component.

6. Example 4

This example demonstrates the binding of *far* to *.

This example creates a *far* array of pointers to *near* chars. That is, the array is in its own segment, which can be up to 64kb long; each pointer is 16 bits long and points to a char that's in the standard data segment.

```
char * far arr[100];
```

The *far* keyword binds to *, giving * the *far* attribute. The attribute of *char* is not specified, and hence defaults to *near*.

arr is thus an array of pointers, each having the *far* attribute. This means that the array is in its own segment, that the segment is at most 64kb long, and that the address of an array element is determined by setting the segment component to that of the array's beginning address and by computing the offset component.

Each array element points at a *char* having the *near* attribute. This means that each pointer is 16 bits long, pointing to a *char* in the standard data segment.

7. Example 5

This example defines a *far* function named *func*. It's accessed using 32-bit addresses and returns an *int* as its value. The compiler will translate C-language call statements to *func* into assembly language *far call* statements.

```
int far func();
```

The *far* keyword binds to *int*, giving it the *far* attribute. By definition, a function that returns a value having the *far* attribute is accessed using 32-bit addresses and assembly language *far call* instructions.

8. Example 6

This example defines *fp*, a *near* array of pointers to *far* functions, each of which returns a pointer to a *far char*.

```
char far * far (*fp[])();
```

The attribute of the rightmost * is *near*, since its attribute isn't explicitly specified. Thus, *fp* is an array of pointers each having the *near* attribute, which means that the array is in the standard data segment and its elements can be accessed using 16-bit addresses.

The rightmost parentheses specify that each array element is a pointer to a function.

The *far* to the right of * binds to the *. Together, they say that a function pointed at by an element of *fp* returns a pointer having the *far* attribute. By definition, this means that each function is accessed using 32-bit addresses and by assembly language *far call* instructions. Since the *fp* array elements point to functions, this in turn means that the elements of the *fp* array are 32 bits long.

The *far* to the right of *char* binds to *char*. Together, they say that a pointer returned by a function references a *char* having the *far* attribute. This means that a referenced *char* is in its own segment, that the segment can be at most 64kb long, and that the address of other fields in the segment are determined by setting the segment component to that of the segment's beginning address and by computing the offset component.

9. Example 7

The following example creates a typedef *fi* for a *far int*. It then creates an array *arr* of *fi* objects. The array is thus in its own segment, and its elements are accessed using 32-bit pointers, whose segment component is that of the segment's beginning address, and whose offset component is computed.

It also creates *xp*, a pointer to an *fi* object. *xp* is in the standard data segment. The object it points at is in its own segment. This segment is at most 64kb long. The address of fields in the segment are determined by setting the segment component to that of the segment's beginning address, and by computing the offset component.

```
typedef far int fi;
fi arr[100], *xp;
```

The *far* could be to either the left or right of *int*; it binds to *int* in either case.

10. Example 8

This example first creates *func*, a typedef for a *far* function that returns an int. It then defines *f*, a *far* function that returns an int. It also define *fp*, a *near* variable that points at a *far* function that returns an int.

```
typedef int far func();
func f, *fp;
```

The Assembler: new features

Several features have been added to the assembler, to support the 8087 and 80287 math coprocessors. These features are discussed in the following paragraphs.

1. 80287 support

The *-2* option enables the assembler's support for the 80287 chip, by preventing the assembler from generating a WAIT instruction when it processes an instruction for a math coprocessor.

2. New codemacro parameter specifiers: F and T

There are two new codemacro parameter specifiers:

F	Matches a reference to a floating point stack element; e.g. ST or ST(i).
T	Matches a reference to the top of the floating point stack; e.g. ST or ST(0).

3. New types for data items

The assembler supports the following additional types for data items:

QWORD	Eight bytes long
TBYTE	Ten bytes long

4. New codemacro directives

The assembler supports the following additional codemacro directives:

4.1 RFIX

rfix has a single argument, which can be either an absolute number or the name of a formal parameter whose specifier is D.

When the assembler is generating 8086/8087 code (i.e. it was started without the -2 option) *rfix* generates two bytes, of which the first is a *wait* instruction. The second is an *escape* instruction's first byte; its low-order three bits are defined by the *rfix* parameter.

When the assembler is generating 80286/80287 code (i.e. it was started with the -2 option) *rfix* behaves like it does when generating 8086/8087 code, except that it doesn't generate the leading *wait* instruction.

For example, here's the codemacro for the *fld1* instruction:

```
codemacro fld1
rfix    001B
db      11101000B
endm
```

When the assembler is started without the -2 option, the source statement *fld1* generates:

```
10011011 11011001 11101000
```

The first byte is the *wait* instruction. The second is the *escape* instruction's first byte. The low-order three bits of the second byte and the bits in the third byte identify this as an *fld1* instruction.

When the assembler is started with the -2 option, the statement *fld1* generates:

```
11011001 11101000
```

4.2 RFIXM

rfixm has the following format:

```
rfixm esc, memloc
```

where *esc* is either an absolute number or the name of a formal parameter with specifier D; and *memloc* is the name of a formal parameter that represents a memory address (e.g. its specifier is E, M, or X).

When the assembler is generating 8086/8087 code (i.e. it was started without the -2 option), *rfixm* generates two bytes that are the same as those generated by *rfix*. When necessary, it also generates a segment-override byte to access the specified memory address; this byte is generated after the *wait* instruction and before the *escape* instruction's first byte.

When the assembler is generating 80286/80287 code (i.e. it was started with the -2 option), *rfixm* behaves like it does when generating 8086/8087 code, except that it doesn't generate the leading *wait* instruction.

For example, here's the codemacro for the *fadd* instruction:

```
codemacro    fadd    memloc:Mq
rfixm   100B, memloc
modrm 0, memloc
endm
```

When the assembler is generating 8086/8087 code, the source statement *fadd quad ptr es:10[bx]* generates the following bytes:

```
10011011 00100110 11011100 00001010
```

The first byte is the *wait* instruction. The second is the segment override byte, specifying the ES register. The third is the *escape*

instruction's first byte. It's low-order three bits and the bits in the fourth and fifth bytes identify this as an *fadd* instruction with memory operand ten bytes beyond the location pointed at by BX.

4.3 RNFIX

rnfix has the following format:

```
rnfix   op
```

where *op* is either an absolute number or the name of a formal parameter with specifier D.

rnfix generates two bytes. The first byte is a *nop* instruction. The second is an *escape* instruction whose low-order three bits are set to the value defined by *rnfix*'s parameter.

For example, here's the codemacro for the *fnclex* instruction:

```
codemacro fnclex
rnfix   011B
db      11100010B
```

The source statement *fnclex* generates the following three bytes:

```
10010000 11011011 11100010.
```

4.4 RNFIXM

rnfixm has the following format:

```
rnfixm esc, memloc
```

where *esc* is either an absolute number or a the name of a formal parameter with specifier D, and *memloc* is the name of a formal parameter that represents a memory address (i.e. its specifier is E, M, or X).

rnfixm generates two bytes that are the same as those generated by *rnfix*. When necessary, it also generates a segment-override byte to access the specified memory address; this byte is generated after the *nop* instruction and before the *escape* instruction's first byte.

For example, here's the codemacro for the *fnsave* instruction:

```
codemacro       fnsave  memloc:M
rnfixm  101B, memloc
modrm   110B, memloc
endm
```

The source statement *fnsave word ptr ss:[bx]* generates the following bytes:

```
10010000 00110110 11011101 00110111
```

4.5 RWFIX

rwfix generates a *wait* instruction (10011011B). Its format is:

```
rwfix
```

The Linker: new features

The following list describes features that have been added to the linker since the manual was last printed. The description of each feature lists the version of Aztec C86 in which the feature was introduced.

1. The CLIB environment variable

The CLIB environment variable can now define a list of directories to be searched for libraries that are specified using the *-l* option. As described in the following section, this list of directories can also be searched for object modules and libraries that are specified by name. Each CLIB list element defines a directory to be searched, and the elements are separated by semicolons or spaces.

A list element, which specifies a directory, consists of a path to the directory, followed (usually) by a trailing backslash. The trailing backslash is required because of the way the linker generates a file name from a directory specifier: it prepends the directory specifier to the base file name. A null list element refers to the current directory.

For example, the following setting of CLIB causes the linker to search for libraries and object modules in the *path1* directory (which is a subdirectory of the current directory), then the *\path2* directory, and finally in the current directory.

```
set CLIB=path1\;\path2\;
```

The following setting of CLIB causes the linker to search for libraries and object modules in the current directory, then in the current directory of the d: drive, and finally in the *\path2* directory.

```
set CLIB=;d:;\path2\
```

Support for CLIB directory lists was introduced in version 3.40b of Aztec C86.

2. Searching for object modules and libraries

The linker can now search for object modules and libraries that are specified by name. Given a file name, here's how the linker tries to find the file:

1. Using the specified name, the linker tries to open the file.
2. If an extension wasn't specified, the linker appends ".o" to the specified name and tries to open that file.
3. Using either the specified name (if an extension was specified) or the name with an appended ".o" (if an extension wasn't specified), the linker searches for the file in the directories defined by the CLIB environment variable.

Support for this searching was introduced in version 3.40b of Aztec C86.

3. Linking programs that have lots of symbols

Version 3.40b of Aztec C86 contained two versions of the Aztec C86 linker. One of these was *ln*, which was limited in the number of symbols that a linked program could have, but which ran fast. The other was *bln*, which had much higher limits on the number of symbols that a program could have, but which ran slower than *ln*.

We found that *bln*'s speed wasn't significantly slower than *ln*'s, so version 4.10a of Aztec C86 has just the *bln*-type linker. Its name is *ln*.

4. Support for *sdb*

Two new options *-g*, and *-q* have been added to the linker to enable and disable, respectively, the collection of the symbol table information needed by the source level debugger *sdb*. This information is put into a file whose name is derived from that of the program file, with extension changed to *.dbg*. When *sdb* loads a program, it automatically looks for an associated *.dbg* file.

These options allow you to collect source level debugger information for selected files. When *ln* encounters a *-g* option, it begins collecting *sdb* information, and continues doing so until it either finds a *-q* option or reaches the end of the argument list. Similarly, when it finds a *-q* option, it stops collecting *sdb* information, and doesn't start again until and unless it finds a *-g* option.

Support for *sdb* was introduced in version 3.40a of Aztec C86.

Z - new features

This section describes the changes that have been made to the Z text editor.

1. Editing large files

The size of file that Z can edit is now limited only by the amount of memory. Z will initially allocate a 16kb edit buffer; when necessary, it will allocate additional 16kb buffers.

2. Support for the EGA 43-line display

On an EGA display, Z can now optionally display 43 lines of text. This feature is enabled and disabled by specifying the operands *43=1* and *43=0*, respectively, in an *:se* command or in the ZOPT file.

3. Support for color displays

Z now allows you to define the color of a display's text area and status line, by specifying, as operands to the *:se* command or in the ZOPT file, the display characters' attribute bytes.

The operands to define the attribute byte for characters in the display's text area and status line are, respectively:

co=*val*
sc=*val*

where *val* is the decimal value of the attribute byte.

By default, the attribute byte of text area characters is 7 (white characters on black background), and those of status line characters is 112 (black characters on white background). For a definition of screen attribute bytes, see the PC technical reference manual.

4. ZOPT enhancements

The ZOPT environment variable can now define either the file in which *:se* options are found or the options themselves.

The syntax for defining the ZOPT file is unchanged. For example, the following command says that the options are in the file *zopt.cmd*:

```
set ZOPT=zopt.cmd
```

To define options in the ZOPT variable, set the first character of the variable to a colon, ':' and list the options, with each pair of options separated by spaces. If an option normally requires an equals character followed by a value (e.g. co=7), enter a colon instead of the equals (e.g. co:7). (DOS doesn't allow equals characters in an environment variable's value). For example, the following command

is equivalent to entering the command *:se co=31 sc=32* to Z:

```
set ZOPT=:co:31 sc:32
```

New features of the printf functions

Enhancements have been made to the *printf*-type functions (*printf*, *fprintf*, *sprintf*, and *format*) to support the argument conversion specifications defined by the ANSI standard, and to support the *near*, *far*, and *huge* keywords. The following paragraphs discuss these changes.

* For the *d*, *i*, *o*, *u*, *x*, *X* conversions, the *precision* indicator defines the minimum number of digits to be displayed.
* To indicate that a *d*, *i*, *o*, *u*, *x*, or *X* conversion specifier applies to a *short int* or *unsigned short int*, the specifier can optionally be preceded by *h*.
* To indicate that a *d*, *i*, *o*, *u*, *x*, or *X* conversion specifier applies to a *long int* or *unsigned long int*, the specifier can optionally be preceded by *l*.
* To indicate that a *e*, *E*, *f*, *F*, *g*, or *G* conversion specifier applies to a *long double*, the specifier can optionally be preceded by *L*.
* If *h*, *l*, or *L* precedes any other conversion specifier, it is ignored.
* The *field width* indicator can optionally be preceded by a flag character. The flag characters and their meanings are:

-	Left-justify the result in its field.
+	Prepend a sign character (plus or minus) to the result of a signed conversion.
space	If the first character of a signed conversion is not a sign, prepend a space character to the result. If both *space* and + flags are specified, the *space* flag is ignored.
#	Convert the result to an alternate form, as follows: For *c*, *d*, *i*, *s*, and *u* conversions, this flag has no effect. For an *o* conversion, increase the precision to force the first digit of the result to be a zero. For *x* and *X* conversions, prepend the result with *0x* or *0X*.

* There are several new conversion specifiers:

i	The same as *d*.
X	The same as *x*, except upper case letters (ABCDEF) are used instead of lower

(abcdef).

E The same as *e*, except the exponent is preceded by E instead of e.

G The same as *g* except that when exponential form is used, the exponent is preceded by E instead of e.

n The argument is a pointer to an integer into which is written the number of characters written to the output so far by this call.

p The argument is a pointer to *void*. The value of the pointer is printed.

* A conversion specifier (such as *s*) that takes a pointer as an argument can be preceded by an *N* or *F*, indicating that the pointer is *near* or *far*, respectively. A *near* pointer is a 16-bit offset to a field in the programs' standard data segment. A *far* pointer is a 32-bit value that can point at any location in memory.

* For a numerical conversion (%d, %x, %o, etc.), the *field width* indicator in a conversion specification defines the minumum number of characters that the converted value will use, not (as the Aztec C86 manual says) the exact number of characters. For a string conversion (%s), the *field width* defines the maximum number of characters in the generated string.

NAME

lmalloc, lcalloc, lrealloc, lfree - memory allocation

SYNOPSIS

```
huge void * lmalloc(size)
long size;

huge void * lcalloc(nelem, elemsize)
long nelem, elemsize;

huge void * lrealloc(ptr, size)
huge void *ptr;
long size;

lfree(ptr)
huge void *ptr;
```

DESCRIPTION

Introduction

These functions allocate and free blocks of memory, from the area above the program. They are similar to the UNIX-compatible functions *malloc*, *calloc*, *realloc*, and *free*, except that (1) the size of a block can be larger than 64kb; and (2) for a program that uses the 'small data' memory model, and whose stack is above its heap, the area of memory managed by these functions is completely separate from that managed by *malloc*, etc.

When called by programs that use the 'large data' memory model, the area managed by these functions is the same as that managed by the *malloc*, *free*, *sbrk*, and *brk* functions. In this case, in fact, these functions call *sbrk* to allocate and free buffers, just as do the *malloc* and *free* functions.

These functions can be used by any program, except one that uses the 'small data' memory model and whose stack is below the heap. Programs can mix calls to these functions, the *malloc* and *free* functions, and the *sbrk* and *brk* functions.

Function Descriptions

lmalloc allocates a block of *size* bytes, and returns a pointer to it.

lcalloc allocates a single block of memory which can contain *nelem* elements, each *elemsize* bytes big, and returns a pointer to the beginning of the block. Thus, the allocated block will contain (*nelem* * *elemsize*) bytes. The block is initialized to zeroes.

lrealloc changes the size of the block pointed at by *ptr* to *size* bytes, returning a pointer to the block. If necessary, a new block will be allocated of the requested size, and the data from the original block moved into it. The block passed to *lrealloc* can have been freed, provided that no intervening calls to *lcalloc*,

lmalloc, or *lrealloc* have been made.

lfree deallocates a block of memory which was previously allocated by *lmalloc*, *lcalloc*, or *lrealloc*; this space is then available for reallocation. The argument *ptr* to *lfree* is a pointer to the block.

Technical Information

These functions maintain a circular list of free blocks. When called, *lmalloc* searches this list, beginning with the last block freed or allocated, coalescing adjacent free blocks as it searches. It allocates a buffer from the first large enough free block that it encounters. If this search fails, it calls *sbrk* or DOS to get more memory for use by these functions.

SEE ALSO

Memory Usage (O), break (S)

DIAGNOSTICS

lmalloc, *lcalloc* and *lrealloc* return a null pointer (0) if there is no available block of memory.

lfree returns -1 if it's passed an invalid pointer.

Standard I/O and the auxiliary and printer devices

When a program starts, five devices are preopened for it. Three of these have always been accessible via the standard I/O functions: *stdin*, *stdout*, and *stderr*.

Now, if the symbol MSDOS is #defined before the header file is #included in a program, the other two preopened devices are also accessible via the standard I/O functions: the auxiliary device is accessed using the name *stdaux*, and the printer is accessed using the name *stdprt*.

For example, the following program writes a message to the printer:

```
#define MSDOS
#include <stdio.h>
main()
{
        fprintf(stdprt, "hello, world");
}
```

The agetc function: new features

Before returning a character, the *agetc* function ands it with the contents of the global *int* named *__agetc_mask*. This field by default contains 0x7f, which causes the standard I/O function *agetc* to turn off the high-order bit of all characters it returns. By setting this field to 0xff, you can prevent *agetc* from turning off this bit.

For example, to read a file created by a word processor, in which the most significant bit of each byte contains control information, a program would set *__agetc_mask* to 0xff, thus preventing the *agetc* function from stripping off these control bits.

SDB: new features

The following paragraphs describe the features that have been added to *sdb* in going from version 3.40b to 4.10a.

1. Floating point disassembly

sdb now supports disassembly of floating point instructions.

2. New addressing modes for *bs* command

The *bs* command, which is used to set or modify a breakpoint, has two new addressing modes, @func and @. @func causes a temporary breakpoint to be set at the return address of the specified function, whenever the function is called. The breakpoint is dynamically set when the function is entered, and is removed once taken; thus, the actual breakpoint address can differ for each call to the specified function.

@ causes a permanent breakpoint to be set at the return address of the current function. The breakpoint must be explicitly removed using the *bc* command.

3. New display information for the *bd* command

The *bd* command, which lists breakpoints that have been set with the *bs* command, has been enhanced to support the *bs @func* command. Such a breakpoint can have one or more entries in the breakpoint list. One entry always exists for such a breakpoint: its address field contains the function address, preceded by the @ character. The other entries follow the @ entry, and define actual addresses at which temporary breakpoints will be taken on return from the function; the address field for these entries contains the actual breakpoint address, preceded by the ^ character.

4. Changes to the print command

The following paragraphs describe changes that have been made to the print command.

4.1 The P command

A 'P' command has been added, which is just like the 'p' command, except it prints the address of the display items.

4.2 Format override character @ is now optional

The print command can display the same field in several formats. If you don't specify a format, the field will be displayed in its default format. In previous versions of *sdb*, you had to precede the characters that defined the override format with a @ character. This preceding

@ is now optional. It is also obsolete, and will be removed in the next version of *sdb*.

4.3 Changes to the string format specifier, 's'

The 's' format has been changed slightly. It now causes the non-printable characters in a character string to be displayed using standard backslash notation. For example, if a character string is defined as follows:

```
char arr[]="abc\tdef\n";
```

then the command "ps arr" prints "abc\tdef\n".

4.4 Precision specification for strings

The number of characters to be printed from a character string can be defined by preceding the 's' or 'S' specifier with a period followed by a number. For example, the command "p.8s arr" prints the first 8 characters from arr.

4.5 Precision specifier

The precision specifier, a period followed by a number, now applies just to the next string- or float-specifier, and not to subsequent specifiers. For example, if you enter "p.5f f1" and then "pf f2", *f1* will be displayed using five digits of precision, and *f2* using the default 7 digits of precision.

4.6 Repeating a print command

Typing 'p' without operands causes *sdb* to recompute the address used in the last print command. Typing the return key causes *sdb* to display the next value in a list of values.

For example, suppose that *ip* is a pointer to an array of *int*s. Then the following command displays the *int* pointed at by *ip*:

```
p*d ip
```

If you then type return, *sdb* will display the second *int* in the array. If you then type 'p' without operands, *sdb* will display the first *int* again.

5. Changes to the 't' command

In assembly mode, *sdb* will now skip over calls, instead of single-stepping into the called function.

6. Suspension of source mode

When *sdb* is in source mode and enters a function for which no source information exists, it will suspend source mode and enter assembly mode until it gets back to a function for which source information exists.

7. Qualifying names and expressions

7.1 Name qualifiers

Previous version of *sdb* have only allowed you to access variables that are in the current function and the module that contains it. Using name qualifiers, you can now access automatic and static variables that are located in other functions and modules.

There are two types of qualifiers: file qualifiers and function qualifiers. File qualifiers are used to specify the file that contains file-scope statics; i.e. static variables that are defined outside of any function. Function qualifiers are used to access automatic and static variables that are visible to an active function; i.e. to a function that has been called but that hasn't yet returned.

A qualifier to a variable name precedes the name, and is separated from it by a period.

For example, suppose that a program has taken a breakpoint in the function *s4()*, and that the active functions, and the order in which they were called, are *main()*, *s1()*, *s2()*, *s3()*, and *s4()*. Then you could refer to the automatic or register variable *av* in function *s2()* using the function-qualified name *s2.av*. If *av* is a static that is visible from *s2()*, you could also refer to it using the function-qualified name *s2.av*.

Suppose further that the program has a module *mod.c*, and that it contains the file-scope static *fss*. You could refer to it using the file-qualified name *mod.c.fss*.

7.2 Expression qualifiers

You can also qualify expressions, to define the function from which the expressions elements are visible or the file in which they are defined. The qualifier precedes the parenthesized expression, and is separated from it by a period.

Continuing the above example, suppose that *s1()* contains the automatic variables *v1* and *v2*. Then you could display the value of *v1-v2* using the command

```
=s1.(v1-v2)
```

If the file *mod.c* contained the file-scope statics *x1* and *x2*, you could display the sum of *x1* and *x2* using the command

```
=mod.c.(x1+x2)
```

7.3 Disambiguating qualified names and expressions

There are cases where *sdb* can't tell where the qualifier ends and the qualified name begins. To clearly identify the qualifier from the qualified name, surround the qualifier with backquotes.

For example,

= 'my-prog.c'.av

8. Searching for unqualified names

If you specify an unqualified name, *sdb* will first search for it in the current function and module. If not found, it will then look for a file-scope static of that name in the file, if any, that's currently being examined using the *df* (display file) command.

9. Displaying comments

When *sdb* displays a statement, it will display the comments that immediately precede it. The previous version of *sdb* displayed the comments that followed a displayed statement.

10. User names and register names

User names take precedence over register names. For example, if your program has a variable named *ax*, the command "dw ax" will display the contents of that variable, and not of the AX register.

11. Executing another command

While *sdb* is active, you can execute a DOS command, batch file, or program named *cmd* by entering:

!cmd

12. Separate screens for programs and *sdb* (the -w option)

When used on an IBM PC or an equivalent, *sdb* can optionally maintain separate screens for itself and for a program that is being debugged. With this feature enabled, a program-generated screen is displayed while a program is executing, and a screen of operator-*sdb* interactions is displayed while *sdb* is executing.

By default, this feature is implemented as follows: when a program encounters a breakpoint, *sdb* saves the contents of the screen and displays the debug screen; similarly, when *sdb* continues a program, it saves the debug screen and restores the program screen.

Alternatively, if your system has a display adaptor that supports multiple pages, you can tell *sdb* to display its information on a specified page. This will speed up *sdb*, since it won't have to save and restore screens.

To enable the use of separate screens for programs and *sdb*, where *sdb* will save and restore the *sdb* and program screens, specify the *-w* option when you start *sdb*. This option must precede the name of the program that is to be debugged. To have *sdb* use a specific page of display memory for its display, follow the *-w* option with the number

of the page that is to be used for *sdb*'s information.

Two *sdb* commands are related to separate program/debug screens:

* The *w* command causes *sdb* to toggle between displaying the debug and program screen.
* The *W* command disables screen saving and restoring.

13. Using two screens (the -2 option)

The *-w* option described above allows *sdb* to separate its displayed information from that of a program, using just one actual display. If you have two displays, you can have the program's information displayed on the default display (the one that was active when *sdb* was started), and *sdb*'s information displayed on the other by specifying the option *-2* when you start *sdb*.

SDB Tutorial

This document forms a brief tutorial on use of the source-level debugger, detailed reference documentation is provided in the SDB manual supplied with this package.

1. Getting Started

Since source-level debug mode is the default mode for the compiler, no changes need be made to your compiling directives in order to make use of the debugger. The linker, however, only produces the *.dbg* file needed by *sdb* if the *-g* option is specified. Note: the *-g* option must be specified on the link command line before any object files upon which you wish to be able to run *sdb*.

For general use of the debugger, you should make use of the help screens, which can be obtained by entering ?<return>. Futher detail is provided for many of the commands by entering the first letter of the command followed by ?<return>. Some brief directions for using *sdb* follow:

2. Starting the *sdb*

The command for starting *sdb* is of the form:

sdb [options] [progfile] [arg1 arg2 ...]

The optional parameter *[progfile]* is the name of a file containing a program to be debugged, and the optional parameters *arg1*, *arg2*, ..., are character strings to be passed to the program. [options] is any of the following:

-s*dir1;dir2;...* Search for source files in the directories *dir1*, *dir2*, ...

-i*dir1;dir2;...* Same as the *-s* option.

-w[*page*] Maintain separate screens for the debugger and the program that's being debugged. If *page* isn't specified, *sdb* will save and restore the two screens on entry and exit from *sdb*. For display adaptors that support multiple screen pages, *page* can be used to define the page that *sdb* will use for its own display; in this case, *sdb* will simply switch screens upon entry and exit, instead of saving and restoring screen information.

-2 System has two physical displays. Use the primary for the program's output, and the alternate for *sdb*'s.

-a start debugger in assembly mode - default is C source mode.

When invoked, *sdb* will load the program to be debugged, start it, and then stop at the entry point to the main function, displaying that line.

3. Displaying sections of the source file.

Two commands, *c* and *df*, are provided for displaying the user's source file. *c* displays the current source line together with the five lines that precede and follow it. It takes no arguments.

The format of the *df* command is:

df [FILENAME,] RANGE

where FILENAME is the name of a file to be displayed. FILENAME is optional and defaults to the current file if none is specified. FILENAME can be used to display lines from a file that is not the current one.

RANGE is one of the following:

LINE
LINE .. LINE
LINE,COUNT

LINE and COUNT are expressions yielding an integer result.

If LINE is specified alone, the line indicated is displayed together with the next nine lines from the file.

As with all display commands, hitting <return> after issuing the command will cause the next ten lines of source to be displayed.

4. Running the program.

Once the program has been loaded, debugging consists of setting breakpoints and running the program. Breakpoints can be one-time breakpoints or can be set as permanent breakpoints.

If you don't wish to set permanent breakpoints, you can contol execution of the program using the single step and go commands.

The format of a single step command is:

[COUNT]s
[COUNT]S
[COUNT]t
[COUNT]T

where COUNT is a positive integer which causes the debugger to single step COUNT times, printing information for each breakpoint, before stopping.

The difference between *s* and *t* is that *s* single steps into calls while *t* single steps across calls. In effect, *t* treats a call as a single line.

S and *T* will step COUNT times displaying information only for the last breakpoint taken. *S* will step into calls just like *s* while *T* will treat calls as a single line.

In source level mode (the default), single step is by source line. In assembly mode, (set by typing z<return>) single step is by instruction.

If single stepping encounters a function for which no source line information is available (a library function for example), single stepping into the function will cause the debugger to step over the call just like *t*.

To set a permanent breakpoint type:

[COUNT]bs ADDR[,COMMAND]

where ADDR is:

[FILENAME].LINE
FUNCTION[.LINE]
ADDRESSEXPRESSION

COUNT is an integer expression. COMMAND is a set of debugger commands separated by semicolons. For example, you can enter:

bs linkmain.c.39

This will set a breakpoint on line 39 of linkmain.c. Alternatively,

bs getfld.10

will set a breakpoint at line 10 in the function getfld.

bs getfld+10

will set a breakpoint at ten bytes beyond the start of getfld.

Finally, to make the program go, you simply type *g<return>*. The program will execute until it encounters a breakpoint or terminates.

The format of the go command is:

g [@][ADDR]

where ADDR is as described above and '@' means go until hitting a return after ADDR. ADDR is set as a temporary breakpoint and won't be remembered after execution of the command. For example:

g@

means go until the current function returns.

g linkmain.c.39

means go to line 39 in linkmain.c

5. Displaying the trace of calls

It is often useful to know where the program is in a set of nested calls, and what the arguments and local variables are to each of the calls. to display the nested stack of calls in *sdb*, simply type *ds<return>*. Each of the calls currently active will be displayed together with its arguments. Each argument is displayed in a form appropriate to its type. Typing *dS<return>* causes the functions to be displayed with the types, names, and values of each function argument, and auto variables.

For example:

```
ds
```

might cause the following display

```
main__(1,0xFF35)
Croot__()
```

while

```
dS
```

could cause:

```
main(int argc = 1, char **argv = 0xFF35)
        int i = 0
        long j = -1
        char name[8] = "hello."

Croot__()
```

6. Displaying values and computing expressions.

sdb provides some simple, but powerful facilities for displaying variables, arrays, and structures. These are the *p* (print) and the *e* (evaluate) commands. For example, to print a structure named symbol you simply enter:

```
p symbol
```

which might result in:

```
struct symboltb symbol = {
        int s__flag = 10
        char *s__name = 0xFF42
        int s__value[2] = {
        10,5
        }
}
```

Suppose then you want to display the string pointed to by symbol.s__name. You simply type:

```
ps *symbol.s_name
```

So the result might be:

```
"pointer"
```

In addition to the print command, the debugger has an evaluate command, so you can perform general C expression evaluation, including calls to C functions, assignment, pre- and post- increment and decrement, casts, and conditionals.

```
e  c = getchar()
```

might result in

```
=10
```

7. Walking up and down the frames.

In using the expressions shown above the user is generally limited to refering to variables that are visible by C rules at the point where execution stopped. That is, you cannot refer to local variables of functions that are not active or are not the "current" function. (Statics, however, may be referred to by qualifying them with the name of the file or function they were declared in. e.g. linkmain.c.name or main.name).

In order to refer to names in other active functions you can change sdb's notion of what the current function is by walking up and down the call frames, using the commands *fu* for frame up, and *fd* for frame down. These commands walk up the call frames, displaying the line from which the next frame down was called and making visible all of the current call's local variables.

8. Displaying assembly.

Finally, you can display the assembly code at any address with the unassemble command. Its format is:

```
u ADDR
U ADDR
```

where ADDR is as described above.

u does a disassembly with symbols substituted where possible for global and local variables. *U* disassembles without symbol substitution but with the hex for the code shown as well as the assembly.

NAME

c - c driver

SYNOPSIS

c [options] file1 [file2 file3 ...] [-lmylib1 -lmylib2 ...]

DESCRIPTION

The driver is designed to compile, assemble and link using one command. You can specify a single file, multiple files, or a file which contains a list of files that are to be compiled, assembled and linked together to create one executable file. Filenames can optionally specify multiple files, using the "wildcard" characters (? and *). The name of the executable is derived by taking the first file specified and appending *.exe* unless you use the *-o* linker option.

The actions taken by the driver are dependent on the options given and on the filenames and extensions.

* A *.c* extension will cause the file to compiled, assembled, and linked.
* A *.asm* will cause the file to be assembled and linked.
* A *.o* will cause the file to be linked only.

Specifying the library names to the linker is optional. The c library, *c.lib*, will always be linked in with the appropriate memory model version based on the options to the compiler. For example, typing the following linc:

c foo.c fum.asm

will cause the driver to compile and assemble foo, assemble fum, link both programs together with c.lib and create an executable called foo.exe.

To compile and link using the large code, large data model, and the math library, you would type the following:

c +l foo.c fum.asm -lm

will cause the driver to compile and assemble foo, assemble fum, link both programs together with cl.lib and ml.lib and create an executable called foo.exe. This assumes that *fum.asm* has been previously compiled using the large model option. If not, a mixed model error will result.

If you have more than one file that must be compiled, and/or assembled, and linked to create an executable, you could use the *-f* option. This option allows you to include a file containing the names of the modules desired. For example, to include a file *filelist* and link in the graphics library:

c -f filelist -lg

OPTIONS

1. **Driver Options**

-f file	Read command arguments from file.
-c	Don't invoke the linker.

2. **Compiler Options**

-a	Prevents the compiler from starting assembler.
-b	Don't pause after every fifth error.
-d	Defines a symbol for the preprocessor. example: -dmaxlen=100 prog.c
-e	Specifies the size of expression table. example: -e100
-i	Defines an area to be searched for files specified in a #include statement. example: -ib:urceinc
-L	Specifies the size of the local symbol table. example: -L200
-n	Do not collect source level debugger information.
-s	Don't print warning messages.
-T	Include c source statements in the assembly code output as comments.
-y	Specifies the maximum number of outstanding cases allowed in a switch. example: -y100
-z	Specifies the size of the table for literal strings. example: -z100
+l	Generate code that uses the 'large code','large data' memory model.
+lc	Generate code that uses the 'large code','small data' memory model.
+ld	Generate code that uses the 'small code','large data' memory model.

3. Linker Options

-B addr When linking a DOS .com file, set the program's base address to the hex value addr.

-C addr When linking an .exe program that will be burned into ROM, set the starting paragraph number of the program's code segments to the hex value addr.

-D addr When linking an .exe program that will be burned into ROM, set the starting paragraph number of the program's data segments to the hex value addr.

-g Enables the collection of source level debugger information for all files listed after it on the command line. (This information is put into a *.dbg* file.)

-lname Search the library name.lib for needed modules.

-m Don't issue 'multiply defined symbol' warning messages.

-N Don't abort if there are undefined symbols.

-o file Write executable code to the file named file.

-q Disables the collection of source level debugger information for all files listed after it on the command line.

-S size Tell DOS not to load the program unless at least size bytes is available for its stack and heap.

example: -S 400 i.e. 400 is hex value.

-t Generate a symbol table file (for DB).

-U addr When linking a DOS .com file, set the starting offset of the program's uninitialized data segment to the hex value addr.

-v Be verbose.

-x size Tell DOS to allocate memory to the program so that the program doesn't have more than size paragraphs for its stack and heap.

example: -x 400 i.e. 400 is hex value.

Enhancements to MAKE

The following enhancements have been added to the *make* utility:

1. Dependencies on files in directories and drives other than the current one are now supported.
2. Command line macros now supported.
3. Command line options are now order independent.

These enhancements are discussed in detail below.

1. Different directory and drive dependencies

make has been enhanced to permit dependencies on files in directories and drives other than the current directory and drive. In previous versions of make, the following dependency line was illegal:

foo.o: a:../front/cc.h

make will now allow the drive name in a directory specification, but the following rules must be used :

* The colon (':') is used in two places, one to mark a dependency, and one to specify another disk drive. A colon that is marking a dependency **must** have a tab or a space on any side of it, otherwise it is taken to be an alternate drive specification.
* If you specify a file to be in a different drive, do not refer to the file later with a different path name. This will confuse *make* and the consequences from it are undefined.

  Here is an example of what NOT to do.

  foo.c : a:../src/foo.o

  and then later name it by ...

  a:/root/src/foo.c

  The forward slash or the backward slash may be used in any combination to specify other directories.

  a:dir/sub/foo.c IS NOT THE SAME AS a:/dir/sub/foo.c.

  The first name will look for foo.c starting with the current directory drive a is set to, the second name will look for foo.c in the path starting at drive a's top directory, even if this was all taking place from a makefile on drive b. This is because MS-DOS

remembers a current directory for each drive.

2. Command line macros

You can also define Macros from the command line. Here are some examples.

```
make COPT=-DDEBUG
make OBJ=foo.o
```

However, you cannot specify a macro with whitespace in it.

```
make NOGOOD=foo.o slug.o splat.o
```

This would define the macro NOGOOD to be *foo.o* only, and then try and "make" the files *slug.o* and *splat.o.*

Any command line macro specified will override any macro with the same name in the makefile, for the duration of that execution of *make.*

3. Command line argument order

Command line arguments may now be specified in any order.

NAME

open - additional modes for file sharing

DESCRIPTION

open supports DOS 3.1 file sharing. The header file *fcntl.h* contains these additional *modes*:

mode	*meaning*
O_DENYRW	Deny read, deny write
O_DENYW	Deny write
O_DENYR	Deny read
O_DENYN	Deny nothing
O_INHER	Inherit attributes
O_COMP	Compatibility mode

These modes are different from the other modes allowed by *open* in that they must be or'd and not added.

NOTE: no support is provided for *fopen*ing shared files. However, you can get shared file support for stream I/O through the use of *fdopen* which converts a file opened by *open* into a stream.

SEE ALSO

For more information on file sharing, refer to the section on Interrupts, DOS function call 0x3D, in the PCDOS Technical Reference manual, version 3.00.

EXAMPLES

To create, open, and restrict the read and write priviledges on a file, testfile:

```
fd = open("testfile", O_CREAT | O_DENRW);
```

To use the standard I/O functions on a shared file it must first be opened by *open* and then by *fdopen*:

```
#include "fcntl.h"
FILE *fp, *fdopen();
int fd;

fd = open("testfile", O_CREAT | O_DENRW);

fp = fdopen(fd, "r+");
```

NAME

filelock - lock a region within a file

SYNOPSIS

```
int filelock(fd, flag, offset, length)
int fd, flag;
long offset, length;
```

DESCRIPTION

filelock locks or unlocks a region of a file that has been opened for unbuffered I/O.

fd is the file descriptor associated with the file.

flag indicates the action to be done: 0 = lock, 1 = unlock.

filelock will lock/unlock a region starting at *offset* bytes from the beginning of a file for *length* bytes.

If *filelock* is successful, it will return a zero.

NOTE: care should be exercised when locking files used by stdio streams due to some buffering that takes place.

SEE ALSO

Unbuffered I/O (O).

For more information on file locking, refer to the section on Interrupts, DOS function call 0x5C, in the PCDOS Technical Reference manual, version 3.00.

DIAGNOSTICS

If *filelock* fails, it will return -1 as its value and set an error code in the global integer *errno*.

Using MANX Technical Support

We have put together a set of guidelines to help you take the most advantage of the technical support service offered by MANX. We ask that you read and follow these guidelines to enable us to continue to give you quality technical support.

Have everything with you.

Try to be organized. When using our phone support, have everything you need with you at the time you call. Our goal is to get you the help you need without keeping you on the phone too long. This can save you a lot of time, and if we can keep the calls as short as possible we can take more calls in the day. This can be to your advantage on days when we are busy and it's hard to get through. Also, *have the following information ready* when you call technical support. We will ask you for this information first.

* *Your name.* This is necessary in case we need to get back to you with additional information.
* *Phone number.* In case we have additional information we will be able to contact you. This will never be given to anyone, so you need not worry.
* The *product* you are using, and the *serial number.* If you have a cross compiler please tell us both host and target, even if the problem is with just one side of the system.
* The *revision of the product* you are using. This should include a letter after the number: i.e. 3.20d or 1.06d. THIS IS VERY IMPORTANT. The full version number may be found on your distribution disks or when you run the COMPILER.
* The *operating system* you are using, and also the version.
* The *type of machine* you are using.
* Anything interesting about your machine configuration. ie. ram disk, hard disk, disk cache software etc.

Know what questions you wish to ask.

If you call with a usage question please try to have your questions narrowed down as much as possible. It is easier and quicker for all to answer a specific question than general ones.

Isolate the code that caused the problem.

If you think you have found a bug in our software, try and create a small program that reproduces the problem. If this program is small enough we will take it over the phone, otherwise we would prefer that you mail it to us, using the supplied problem report, or leave it on one of our bbs systems. Once we receive a "bug report" we will attempt to reproduce the problem and if successful we will try to have it fixed in the next release. If we can not reproduce the problem we will contact you for more information.

Use your C language book and technical manuals first.

We have no qualms about helping you with your general C programming questions, but please check with a C language programming book first. This may answer your question quicker and more thoroughly. Also, if you have questions about machine specific code, i.e. interrupts or dos calls, check with that machine's technical reference manual and/or operating system manual.

When to expect an answer.

A normal turn around time for a question is anywhere from 2 minutes to 2 days, depending on the nature of the question. A few questions like tracing compiler bugs may take a little longer. If you can call us back the next day, or when the person you talk to in technical support recommends, we will have an in-depth answer for you. But normally we can answer your questions immediately.

Utilize our mail-in service.

It is always easier for us to answer your question if you mail us a letter (We have included copies of our problem report form for your use). This is especially true if you've found a bug with our compiler or other software in our package. If you do mail your question in, try to include all of the above information, and/or a disk with the problem. Again, please write small test programs to reproduce possible bugs. The address for mail-in reports is P.O. Box 55, Shrewsbury, N.J. 07701. If you have questions/problems concerning C Prime or Apprentice C, mail them to P.O. Box 8, Shrewsbury, N.J. 07701.

Updates, Availability, Prices.

If you have any questions about updates, availability of software, or prices, please call our order desk. They can help you better and faster. You can reach them at...

Outside N.J. --> 1-800-221-0440
Inside N.J. --> 1-201-542-2121 (also for outside the U.S.A.)

Bulletin board system.

For users of Aztec C we have a bulletin board system available. The number is ...

1-(201)-542-2793 This is at 300/1200 bps. (all products)

Answer the questions that will be asked after you are connected. When this is done you will be on the system with limited access. To gain a higher access level send mail to SYSOP. Include in this information your serial number and what product you have. Within approximately 24 hours you should have a higher access level, provided the serial number is valid. This will allow you to look at the various information files and upload/download files.

To use the bulletin board best, please do not put large (> 8 lines) source files onto the news system, which we use for an open forum question/answer area. Instead, upload the files to the appropriate area, and post a news item explaining the problem you are having. Also, the smaller the test program, the quicker and easier it is for us to look into the problem, not to mention the savings of phone time.

When you do post a news item, please date it and sign it. This will be very helpful in keeping track of questions. Try to do the same with uploaded source files.

Phone support, number and hours.

Technical support for Aztec C is available between 10-12 am and 2-6 pm eastern standard time at 1-(201)-542-1795. Phone support is available to registered users of Aztec C with the exception of the Apprentice C and C Prime products. For those products, please use the mail-in support service and send questions/problems to P.O. Box 8, Shrewsbury, N.J. 07701.

These guidelines will aid us in helping you quickly through any roadblocks you may find in your development. Thanks for your cooperation.

MANX Problem Report

Date:_____/_____/_____

Name:_____

Phone #:1-(_____)-_____-_____

Company :_____

Address :_____

Product : c86-PC_____ c86-CPM86_____ c68k_____
c68k-Am_____ cII_____ c80_____
c65-ProDos_____ c65-Dos3.3_____
cross:_____

VERSION #:_____ Serial #:_____

Op. - sys.:_____ Machine Config.:_____

Send this form to :

Manx Software Systems
P.O. Box 55
Shrewsbury, N.J. 07701

(C Prime/Apprentice C only):
MANX Software Systems
P.O. Box 8
Shrewsbury, N.J. 07701

or call tech support at 1-201-542-1795 between 10-12 am and 2-6 pm EST.
Sorry, phone support not available for the C Prime/Apprentice C product.)

Description of problem --
(include what has already been attempted to fix it)
(use the reverse side of this sheet if needed.)

MANX Problem Report

Date:_____/_____/_____

Name:_____

Phone #:1-(_____)-_____-_____

Company :_____

Address :_____

Product : c86-PC_____ c86-CPM86_____ c68k_____
c68k-Am_____ cII_____ c80_____
c65-ProDos_____ c65-Dos3.3_____
cross:_____

VERSION #:_____ Serial #:_____

Op. - sys.:_____ Machine Config.:_____

Send this form to :

Manx Software Systems
P.O. Box 55
Shrewsbury, N.J. 07701

(C Prime/Apprentice C only):
MANX Software Systems
P.O. Box 8
Shrewsbury, N.J. 07701

or call tech support at 1-201-542-1795 between 10-12 am and 2-6 pm EST.
(Sorry, phone support not available for the C Prime/Apprentice C product.)

Description of problem --
(include what has already been attempted to fix it)
(use the reverse side of this sheet if needed.)

Aztec C86
for
PCDOS, MSDOS, CP/M-86,
and the
8086 Family of ROM Systems

version 3.2

Distributed by:
Manx Software Systems, Inc.
P.O. Box 55
Shrewsbury, N.J. 07701

USE RESTRICTIONS

The components of Aztec C86 are licensed software products. Manx Software Systems reserves all distribution rights to these products. Use of these products is prohibited without a valid license agreement. The license agreement is provided with each package. Before using any of these products the license agreement must be signed and mailed to:

Manx Software Systems
P. O. Box 55
Shrewsbury, N. J 07701

The license agreement limits use of these products to one machine. Any uses of these products that might lead to the creation of or distribution of unauthorized copies of these products will be a breach of the licensing agreement and Manx Software Systems will excercise its right to reclaim the original and any and all copies derived in whole or in part from first or later generations and to pursue any appropriate legal actions.

Software that is developed with the Aztec C86 can be run on machines that are not licensed for these products as long as no part of the Aztec C software, libraries, supporting files, or documentation is distributed with or required by the software. In the latter case a licensed copy of the appropriate Aztec C software is required for each machine utilizing the software. There is no licensing required for executable modules that include runtime library routines.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013. DAC #84-1, 1 March 1984. DOD Far Supplement.

COPYRIGHT

DISCLAIMER

Manx Software Systems makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Manx Software Systems reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Manx Software Systems to notify any person of such revision or changes.

TRADEMARKS

Aztec C, Manx AS, and Manx LN are trademarks of Manx Software Systems. CP/M-80 and CP/M-86 are trademarks of Digital Research. MSDOS is a trademark of Microsoft. PCDOS is a trademark of IBM. UNIX is a trademark of Bell Laboratories. Macintosh is a trademark of Apple Computer.

Manual Revision History

February 1983 ..... First Edition
December 1984 ..... Second Edition
May 1985 ..... Third Edition
February 1986 ..... Fourth Edition

Summary of Contents

8086-specific chapters

title	*code*
Overview	ov
Tutorial Introduction	tut
The Compiler	cc
The Assembler	as
The Linker	ln
Utility Programs	util
Library Functions Overview: 8086 Information	libov86
8086 Functions	lib86
Technical Information	tech
Unitools	unitools
Source Level Debugger	sdb
Assembly Language Debugger	db

System Independent Chapters

Overview of Library Functions	libov
System-Independent Functions	lib
Style	style
Compiler Error Messages	err

Index

Index	index

Contents

Overview ..... ov

Tutorial Introduction ..... tut
1. Installing Aztec C86 3
2. Creating an executable program 7
3. Where to go from here 8

The Compiler cc
1. Operating Instructions 5
1.1 The C Source File 6
1.2 The Output Files 7
1.2.1 Creating an Object Code File 7
1.2.2 Creating just an Assembly Language Source File 8
1.3 Searching for #include Files 9
1.3.1 The -I Option 9
1.3.2 The INCLUDE Environment Variable 10
1.3.3 The Search Order for #include Files 10
1.4 Memory Models 11
1.4.1 How a Memory Model is Selected 12
1.4.2 Multi-module Programs 13
1.4.3 Program Organization 15
1.4.4 'Large model' versus Overlays 15
1.4.5 Implementation of the Memory Models 16
2. Compiler Options 19
2.1 Summary of the Options 19
2.1.1 Machine-Independent Utility Options 19
2.1.2 Table Manipulation Options 19
2.1.3 8086 Options for the Optimizing Compilers 20
2.1.4 8086 Options for the Non-optimizing Compilers 21
2.2 The Options 22
2.2.1 Machine-Independent Utility Options 22
2.2.2 Table Manipulation Options 23
2.2.3 8086 Options for the Optimizing Compilers 26
2.2.4 8086 Options for the Non-optimizing Compilers 29
3. Programming Information 31
3.1 Supported Language Features 31
3.1.1 Preprocessor Statements 31
3.1.1.1 Macros 31

#define 31
3.1.1.2 Conditional Compilation 34
#ifdef 35
#ifndef 35
#if 35
3.1.1.3 More Preprocessor Statements 36
#include 36
#line 36
#asm and #endasm 36
3.1.2 More Features 37
Structure Assignment 37
Line Continuation 37
The *void* Data Type 38
Special Symbols 38
___FILE___ 38
___LINE___ 38
___FUNC___ 38
3.1.3 Special Features of Aztec C 39
String Merging 39
Reserved Words 39
Global Variables 39
3.2 Data Formats 41
3.2.1 char 41
3.2.2 Pointer 41
3.2.3 int, short 41
3.2.4 long 41
3.2.5 float and double 42
3.3 Floating Point Exceptions 42
3.4 Writing Machine-Independent Code 42
3.4.1 Compatibility between Aztec C Products 42
3.4.2 Sign Extension for *char* variables 43
3.4.3 The MPU... Symbols 43
3.5 Using Long Pointers 44
3.5.1 Passing Long Pointers Between Functions 44
3.5.2 Expressions involving Long Pointers 46
3.5.3 Creating and Accessing Huge Arrays 48
4. Error Messages 50

The Assembler ..... as
1. Operating Instructions 5
1.1 The Source File 5
1.2 The Object Code File 6
1.3 The Listing File 6
1.4 Searching for 'include' Files 6
2. Assembler Options 9
3. Programmer Information 10
3.1 Syntax 10
3.2 Symbols 11

3.3 Segmentation 13
3.3.1 The SEGMENT and ENDS Directives 13
3.3.2 Multiple Definitions for a Segment 14
3.3.3 Nested Segments 14
3.3.4 The Default Segment 15
3.3.5 The ASSUME Directive 15
3.3.6 Using the Uninitialized Data Segment 16
3.4 Globally-accessible Symbols 16
3.4.1 The PUBLIC Directive 16
3.4.2 The GLOBAL Directive 17
3.4.3 The EXTRN Directive 17
3.4.4 Interactions of GLOBAL, PUBLIC, and EXTRN 18
3.5 Operands and Expressions 19
3.5.1 Registers 19
3.5.2 Immediate Operands 19
3.5.3 Memory Operands 20
3.5.4 Operand Expressions 23
3.5.5 The Arithmetic Operators 23
HIGH and LOW 23
Addition and Subtraction 23
Multiplication and Division 23
The Shift Operators 24
The Relational Operators 24
The Logical Operators 24
3.5.6 Attribute-overriding Operators 25
Segment Override 25
PTR 25
SHORT 27
3.5.7 Attribute-value Operators 27
THIS 27
SEG 28
OFFSET 28
TYPE 29
LENGTH 29
SIZE 29
3.5.8 Operator Precedence 30
3.6 Instructions 30
3.7 Directives 31
ASSUME 31
BSS 31
DB, DW, and DD 32
END 34
EQU 35
= 36
EVEN 36
EXTRN 37
GLOBAL 37
GROUP 37

INCLUDE 37
LABEL 38
LARGECODE 38
MOD186 39
NAME 39
ORG 39
PROC and ENDP 39
PUBLIC 42
RECORD 42
SEGMENT 44
3.8 Macro Directives 44
3.8.1 Local Symbols 46
3.8.2 Concatenating Parameters to Text 47
3.8.3 Concatenating Parameters to Parameters 48
3.8.4 Parameter Substitution in Quoted Strings 49
3.8.5 Passing a Symbol's Value to a Macro 50
3.8.6 Passing Comma-containing Parameters to a Macro 50
3.8.7 Nesting Macros 51
3.8.8 Repeatedly Assembling a Block of Statements 53
3.8.9 Summary of the Macro Directives 56
ENDM 56
EXITM 56
IRP 56
IRPC 56
LOCAL 56
MACRO 56
PURGE 57
REPT 57
3.9 Conditional Directives 57
IF 58
IFE 59
IF1 59
IF2 59
IFDEF 59
IFNDEF 59
IFB 59
IFNB 59
IFIDN 60
IFDIF 60
ELSE 60
ENDIF 60
3.10 Codemacros 60
3.10.1 Specifiers 62
3.10.2 Modifiers 63
3.10.3 Range Specifiers 63
3.10.4 The Codemacro Directives 64
SEGFIX 64
NOSEGFIX 65

MODRM 65
RELB 66
RELW 66
DB, DW, and DD 66
User-defined Record Directives 67
3.10.5 The Dotshift operator 67
3.10.6 The PROCLEN Symbol 68
3.10.7 Matching Codemacros to Instructions 69

The Linker ..... ln
1. Introduction to linking 3
2. Using the Linker 7
3. Linker Options 9
4. Linker Error Messages 17

Utility Programs ..... util
arcv (Source dearchiver) 4
cnm (Object file utility) 5
crc (File verificator) 9
hex86 (ROM Hex Generator) 10
lb (Object module librarian) 11
ls (list directory contents 22
obd (Object file utility) 25
obj (MSDOS/PCDOS Object code generator) 26
ord (Object library generation utility) 27
prof (Execution profiler) 28
sqz (Object file utility) 29
term (terminal emulator for IBM PC) 30

Library Overview: 8086 Information libov86

8086 Functions lib86
Index 5
The functions 7

Technical Information ..... tech
1. Program Organization 4
1.1 The program areas 5
1.2 Factors affecting Program Organization 7
1.3 Symbols related to Program Organization 13
1.4 Startup routine Termination Codes 14
2. Overlay Support 15
2.1 Introduction to Overlays 15
2.2 Programmer Information 19
3. Libraries 25
4. Cross Development 26
5. Using the PCDOS/MSDOS Linker 27
6. Assembly Language Functions 30
6.1 Conventions for C-callable Functions 30

6.2 Assembly Language Macros 33
6.3 Embedded Assembler Source 39
7. Generating ROMable code 41
7.1 Features of ROMable Programs 41
7.2 Special ROM-related Programs 42
7.3 The Procedure 42
7.4 Description of hex86 43

Unitools ..... unitools
diff (Source File Comparator) 6
grep (Pattern Matcher) 10
make (Program Maintainance Utility) 16
1. The Basics 16
1.1 What MAKE does 17
1.2 The makefile 17
1.3 Rules 19
1.3.1 MAKE's use of rules 20
1.3.2 An Example 20
1.3.3 Interaction or rules and dependency entries 21
2. Advanced Features 21
2.1 Dependent files 21
2.2 Macros 22
2.2.1 Using Macros 22
2.2.2 Defining macros in a makefile 22
2.2.3 Defining macros in a command line 23
2.2.4 Macros used by built-in rules 23
2.2.5 Special macros 23
2.3 Rules 24
2.3.1 Rule definition 24
2.3.2 Built-in rules 25
2.4 Commands 26
2.4.1 Allowed commands 26
2.4.2 Logging commands and aborting MAKE 26
2.4.3 Long command lines 26
2.5 Makefile syntax 27
2.5.1 Comments 27
2.5.2 Line continuation 27
2.6 Starting MAKE 28
2.6.1 The command line 28
2.6.2 MAKE's standard output 29
2.7 Executing commands 29
2.8 Differences between the Manx and UNIX MAKEs 29
3. Examples 30
3.1 Example 1 30
3.2 Example 2 31
Z - the text editor 34
1. Getting Started 37
1.1 Creating a new file 37

1.2 Editing an existing file 40
2. More commands 45
2.1 Introduction 46
2.2. Paging and scrolling 48
2.3. Searching for strings 49
2.3.1 The other string search commands 49
2.3.2 Regular expressions 49
2.3.3 Disabling extended pattern matching 50
2.4. Local moves 52
2.4.1 Moving around on the screen 52
2.4.2 Moving within a line 52
2.4.3 Word movements 53
2.4.4 Moves within C programs 53
2.4.5 Marking and returning 54
2.4.6 Adjusting the screen 55
2.5. Making changes 56
2.5.1 Small changes 56
2.5.2 Operators for deleting and changing text 56
2.5.3 Deleting and changing lines 57
2.5.4 Moving blocks of text 57
2.5.5 Duplicating blocks of text 58
2.5.6 Named buffers 59
2.5.7 Moving text between files 60
2.5.8 Shifting text 60
2.5.9 Undoing and redoing changes 60
2.6. Inserting text 61
2.6.1 Additional commands 61
2.6.2 Insert mode commands 61
2.7. Macros 63
2.7.1 Immediate macro definition 63
2.7.2 Examples 63
2.7.3 Indirect macro definition 64
2.7.4 Re-executing macros 65
2.8 The Ex-like commands 67
2.8.1 Addresses in Ex commands 67
2.8.2 The 'substitutute' command 68
2.8.3 The '&' (repeat last substitution) command 69
2.9. Starting and stopping Z 70
2.10. Accessing files 73
2.10.1 File names 73
2.10.2 Writing files 73
2.10.3 Reading files 74
2.10.4 Editing another file 74
2.10.5 File lists 76
2.10.6 Tags 76
2.10.7 The CTAGS utility 77
2.11. Executing system commands 79
2.12. Options 80

2.13. Z vs. Vi 81
2.14. System dependent features 82
2.14.1 IBM PC features 82
3. Command Summary 85

Source Level Debugger ..... sdb
1. Overview 5
1.1 Basic Commands 5
1.2 Names 6
1.2.1 Code and Data Symbols 6
1.2.2 Operator Usage of Names 6
1.3 Loading programs and symbols 6
1.4 Breakpoints 7
1.5 Memory-change breakpoints 8
1.6 Separate screens for programs and *sdb* 8
1.7 Trace mode 9
1.8 Backtracing 9
1.9 Macros 9
1.10 Displaying source files 9
1.11 Other features 10
2. Using SDB 11
2.1 Starting SDB 11
2.2 Commands 11
2.2.1 Definitions 11
2.3 Command descriptions 13
2.3.1 The BREAKPOINT (b) commands 13
2.3.2 The DISPLAY (d) commands 16
2.3.3 The 'Find source string' (/) command 20
2.3.4 The FRAME (f) commands 21
2.3.5 The GO (g) commands 21
2.3.6 The INPUT (i) commands 22
2.3.7 The LOAD (l) commands 23
2.3.8 The MODIFY MEMORY (m) commands 24
2.3.9 The OUTPUT (o) commands 25
2.3.10 The PRINT (p) command 25
2.3.11 The QUIT (q) command 32
2.3.12 The REGISTER (r) command 33
2.3.13 The SINGLE STEP (s) commands 33
2.3.14 The UNASSEMBLE (u) commands 34
2.3.15 The VARIABLE (v) command 34
2.3.16 The MACRO (x) commands 35
2.3.17 The EXPRESSION commands 35
2.3.18 The 'Redirect command input' (<) commands 36
2.3.19 The HELP (?) command 36
2.3.20 The 'Change Mode' (z) command 37
3. Command Summary 38

Assembly Language Debugger ..... db

1. Overview 5
1.1 Basic Commands 5
1.2 Names 5
1.2.1 Code and Data Symbols 6
1.2.2 Operator Usage of Names 6
1.3 Loading programs and symbols 6
1.4 Breakpoints 7
1.5 Memory-change breakpoints 8
1.6 Separate screens for programs and *db* 8
1.7 Trace mode 9
1.8 Backtracing 9
1.9 Macros 9
1.10 Displaying source files 9
1.11 Other features 9
2. Using DB 11
2.1 Starting DB 11
2.2 Commands 11
2.2.1 Definitions 11
2.3 Command descriptions 16
2.3.1 The BREAKPOINT (b) commands 16
2.3.2 The CLEAR (c) commands 19
2.3.3 The DISPLAY (d) commands 19
2.3.4 The 'Find source string' (f) command 23
2.3.5 The GO (g) commands 24
2.3.6 The INPUT (i) commands 25
2.3.7 The LOAD (l) commands 25
2.3.8 The MODIFY MEMORY (m) commands 27
2.3.9 The OUTPUT (o) commands 29
2.3.10 The PRINT (p) command 29
2.3.11 The QUIT (q) command 35
2.3.12 The REGISTER (r) command 36
2.3.13 The SINGLE STEP (s) commands 36
2.3.14 The UNASSEMBLE (u) commands 37
2.3.15 The VARIABLE (v) commands 37
2.3.16 The MACRO (x) command 38
2.3.17 The 'Display Expression' command 38
2.3.18 The 'Redirect command input' (<) command 39
2.3.19 The HELP (?) command 39
3. Command Summary 40

Overview of Library Functions ..... libov
1. I/O Overview 4
1.1 Pre-opened devices, command line args 4
1.2 File I/O 6
1.2.1 Sequential I/O 6
1.2.2 Random I/O 6
1.2.3 Opening Files 6
1.3 Device I/O 7

1.3.1 Console I/O 7
1.3.2 I/O to Other Devices 7
1.4 Mixing unbuffered and standard I/O calls 7
2. Standard I/O Overview 9
2.1 Opening files and devices 9
2.2 Closing Streams 9
2.3 Sequential I/O 10
2.4 Random I/O 10
2.5 Buffering 10
2.6 Errors 11
2.7 The standard I/O functions 12
3. Unbuffered I/O Overview 14
3.1 File I/O 15
3.2 Device I/O 15
3.2.1 Unbuffered I/O to the Console 15
3.2.2 Unbuffered I/O to Non-Console Devices 16
4. Console I/O Overview 17
4.1 Line-oriented input 17
4.2 Character-oriented input 18
4.3 Using ioctl 19
4.4 The sgtty fields 19
4.5 Examples 20
5. Dynamic Buffer Allocation 22
6. Error Processing Overview 23

System Independent Functions ..... lib
Index 5
The functions 8

Style ..... style
1. Introduction 3
2. Structured Programming 7
3. Top-down Programming 8
4. Defensive Programming and Debugging 10
5. Things to watch out for 15

Compiler Error Codes ..... err
1. Summary 4
2. Explanations 7
3. Fatal Error Messages 35

OVERVIEW

Overview

Aztec C86 is a set of programs for translating programs written in the C programming language into a form which can be executed on 8086 systems running PC-DOS, MS-DOS, or CP/M-86.

Aztec C86 can also be used to create programs that will run on a ROM-based 8086 system.

The development can be done on a PC-DOS, MS-DOS, or CP/M-86 system; it can also be done on several other types of systems, as described below, and the executable code downloaded to the target machine.

There are several different Aztec C86 Systems, providing different features. The following is a list of features that are in the Aztec C86 Commercial System. Not all of these features are supported by the other Aztec C86 Systems.

* The full C language, as defined in the book *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, is supported. This now includes the bit field data type and many of the System 5 extensions to the C language.
* Floating point operations can be performed either by software routines or by the 8087 math chip. The choice can be made when a program is linked, by linking the program with the appropriate version of the math library. The choice can be made dynamically, when the program is started, by linking the program with the 'sensing' math library; in this case, the 8087 will be used if it's on the machine on which the program is currently running, and the software routines will be used if it's not.
* Programs can be created that use the 80186 instructions.
* An extensive set of user-callable functions is provided;
* Programs can be created that use the 'small code' or 'large code' memory model and/or the 'small data' or 'large data' memory model. 'Large code' allows a program to have as much memory-resident executable code as desired. 'Large data' allows a program to have 64K bytes of global and static data, 64K bytes of stack space for automatic variables, and unlimited space for dynamically-allocated buffers.
* Overlays are supported, allowing programs to be created and executed that are larger than available memory;

* Object modules and libraries created with the Aztec C86 compiler and assembler can be linked together with either the Aztec linker or the PC-DOS/MS-DOS linker, *link*.
* Aztec C86-compiled and assembled object modules and libraries can be linked with object modules and libraries that have been created using other manufacturers' compilers and assemblers, such as those from Lattice and Computer Innovations.
* With some versions of Aztec C86, several utility programs are provided that are similar to UNIX programs: *Z*, a text editor, which is like the UNIX *vi* editor; *make*, which automates some of the steps in program development and maintainance; *grep*, a pattern-matcher; *diff*, a program that determines the difference in source files;
* Modular programming is supported, allowing the components of a program to be compiled and assembled separately, and then linked together;
* Assembly language code can either be combined in-line with C source code, or placed in separate modules which are then linked with C modules;
* A powerful symbolic debugger is provided.

There are two classes of user-callable functions: system independent and system dependent. The system-independent functions are compatible with their UNIX counterparts and with the system-independent functions provided with Aztec C packages for other systems. Use of these functions allows programs to be recompiled for use on UNIX-based systems or on other systems supported by Aztec C with little or no change.

The system-dependent functions allow programs to take advantage of special features of a system.

Versions

Several Aztec C86 packages are available, for use in different environments:

* The PCDOS/MSDOS package and the code it generates run on systems using PCDOS or MSDOS, version 2.0 or later. With some versions of this package you can generate programs that will run on systems using CP/M-86, or on PCDOS/MSDOS version 1.1.
* The CP/M-86 package and the code it generates run on CP/M-86. With some versions of this package you can generate programs that will run on systems using MSDOS/PCDOS.

* The UNIX package runs on a system which uses the UNIX operating system, and generates code which runs on PCDOS or MSDOS systems or on CP/M-86 systems.

Components

Aztec C86 contains the following components:

* The compiler, assembler, linker, and object file librarian;
* Object libraries containing user-callable functions and support functions;
* Several utility programs, including, with some packages, programs similar to the UNIX programs *make*, *grep*, *diff*, and a symbolic debugger.

Preview

This manual is divided into two sections, each of which if in turn divided into chapters. The first section presents 8086-specific information; the second describes features that are common to all Aztec C packages. Each chapter is identified by a symbol.

The 8086-specific chapters and their identifying codes are:

tut describes how to get started with Aztec C86: it discusses the installation of Aztec C86 and gives an overview of the process for turning a C source program into an executable form;

cc, *as*, and *ln* present detailed information on the compiler, assembler, and linker;

util describes some of the utility programs provided with Aztec C86;

libov86 presents 8086-specific overview information;

lib86 describes the 8086-specific functions provided with Aztec C86;

tech discusses several miscellaneous topics, including program organization, overlays, cross development, libraries provided with Aztec C86, using the Microsoft linker, generating ROMable code, and writing assembly language functions that can call and be called by C functions.

unitools describes the utility programs *z*, *make*, *grep*, and *diff*, which are similar to UNIX programs.

sdb describes the source level debugger;

db describes the assembly language debugger;

The System-independent chapters and their codes are:

libov presents an overview of the system-independent features

lib describes the system-independent functions provided with Aztec C86;

style discusses several topics related to the development of C programs;

err lists and describes the error messages which are generated by the compiler and linker.

TUTORIAL INTRODUCTION

Chapter Contents

Tutorial Introduction ..... tut
1. Installing Aztec C86 3
2. Creating an executable program 7
3. Where to go from here 8

Tutorial Introduction

This chapter describes how to start using your Aztec C86 software. The chapter has three sections: the first describes how to install Aztec C86; the second describes the procedure to create an executable program from a simple C source program; and the third introduces some features of Aztec C86 that you may want to investigate further.

1. Installing Aztec C86

Manx has sent you your Aztec C86 software on one or more floppy disks. If your system supports only single-sided disk drives, we may have used reversible disks, and put files on both surfaces of a single disk. In this case, each side of a disk that we have used will have a label. To access the files on a particular side of a reversible disk, put the disk in the drive with that side facing the drive's read/write heads.

The disks we have sent you are not bootable; that is, they don't contain the operating system (DOS or CP/M-86). In order to use them, you will have to boot the operating system from one of your own disks.

Back up the disks

The first thing you should do with your Aztec C disks is make a copy of them. This can be done using the standard operating system copy utility.

Check the Files

Before you start using your Aztec C software, you should verify that all the files are there, by comparing a directory of the provided files with a list that's in the release document.

The *crc* utility that's on your disk computes a unique number for a file, called its crc. If you wonder whether a file is corrupted, you can compare its crc number with its correct value, which is listed in the file *crclist*.

A complete description of *crc* is provided in the Utility Programs chapter. To compute the crc of all files on the b: drive, enter

```
crc b:*.*
```

Create a working disk

A 'working disk' is a disk that contains just the most frequently-used Aztec C files. When developing programs, you will have the working disk in one drive, and a disk containing the programs being

developed in another drive.

Thus, before you can begin using Aztec C, you must create a working disk.

You can link object modules together using either the Aztec linker *ln* or the PCDOS/MSDOS linker *link*. For now, we'll assume that you want to use the Aztec linker. At the end of this section, we'll show you how to modify the working disk for use with the PCDOS/MSDOS linker.

Here are the steps to create a working disk:

1. The first file you need on your working disk contains the C compiler. Some Aztec C86 packages contain two C compilers, with the versions differing in the speed of compilation, the optimization of the generated code, and support for the 80186 and 80286 processors. To get started, you can just copy the *cc* compiler to the working disk. Later on, you can select another compiler, if desired. The compiler chapter and the release document describe the features of the compilers.

   The name of the file containing a program is derived from the name of the program by appending an extension to the program name: the extension is *.exe* for PCDOS/MSDOS, and *.cmd* for CP/M-86. Thus, the name of the file containing the *cc* compiler is *cc.exe* on PCDOS/MSDOS and *cc.cmd* on CP/M-86.

2. Next, copy the *as* assembler and the *ln* linker to the working disk.

3. Now copy any header files you need to the working disk. A "header file" is a file containing C source code which another program includes in itself with the #include statement. The header files provided with the Aztec package have extension ".h".

4. Finally, copy the object libraries you need to the working disk. the libraries that you will need initially are *c.lib* and *m.lib*, which contain the non-floating-point functions and floating-point functions, respectively. These libraries use the 'small code' and 'small data' memory model. Other versions of these libraries that support other memory models are provided with some packages. See the Libraries section of the Technical Information chapter for details.

If your working disk can't hold all these files, we recommend that you put the compiler, assembler, and header files on one disk, and the linker and libraries on another disk.

The working disk and the PCDOS/MSDOS linker

If you want to use the PCDOS/MSDOS linker instead of the Aztec linker you must modify your working disk, as follows:

1. Replace the Aztec linker *ln* on your working disk with the PCDOS/MSDOS linker.
2. Copy the Aztec utility program *obj* onto the working disk. This program converts object modules from Aztec format into PCDOS/MSDOS format.
3. Copy the special startup routine *crt0.obj* to the working disk.
4. Using *obj*, convert *c.lib* and *m.lib* to PCDOS/MSDOS format by entering:

   ```
   obj c.lib mc.lib
   obj m.lib mm.lib
   ```

   The converted libraries are in the files *mc.lib* and *mm.lib*. You can now remove *c.lib* and *m.lib* from the working disk.

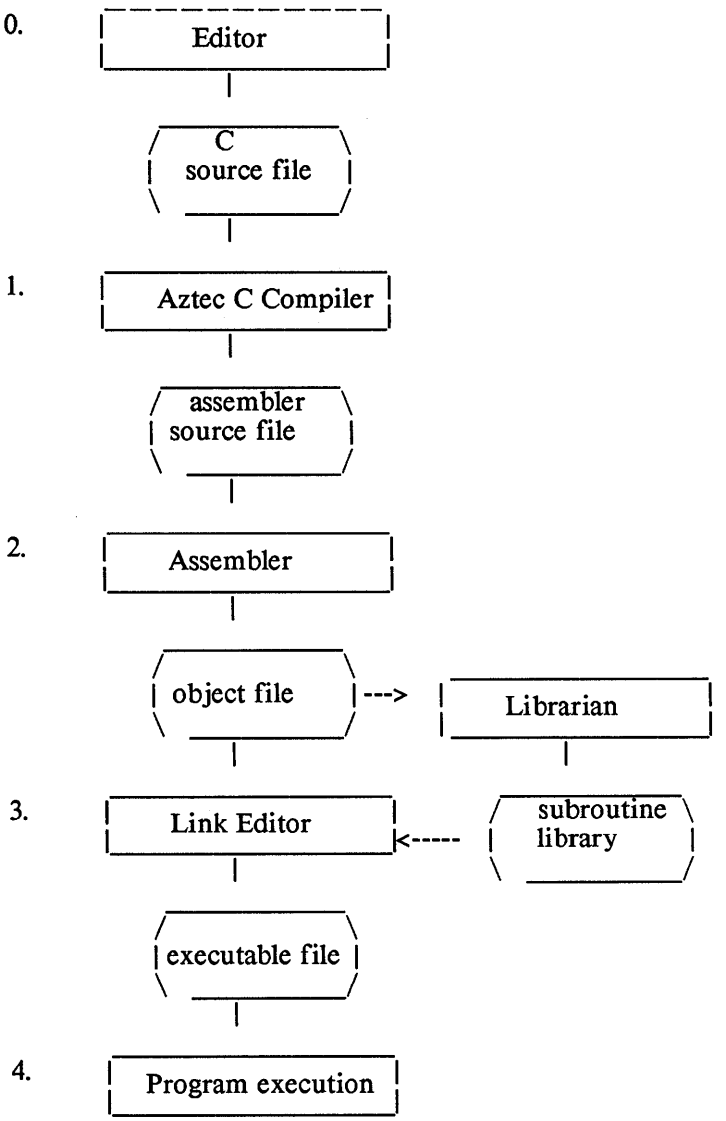


Figure 1: Developing C programs with Aztec C

2. Creating an Executable Program

Figure 1 graphically depicts the steps to create an executable program from a C source program.

The following paragraphs will first present the actual commands to create an executable version of the sample program *exmpl.c* that is on the distribution disk, when using the Aztec linker. Then follows a brief description of each of the commands. For complete descriptions of these programs, see subsequent chapters in this manual.

At the end of this section is a description of the commands for converting *exmpl.c* into an executable program when using the PCDOS/MSDOS linker.

Typically, when developing programs, you will have your working disk in the a: drive and a disk containing your own files in the b: drive. The following discussion assume that this is the case. The commands to generate an executable version of *exmpl.c*, when using the Aztec linker, are:

```
cc b:exmpl.c          step 1 & 2: compile and assemble
ln b:exmpl.o c.lib    step 3: link
```

Step 0: Create the Source Program

Any text editor can be used to create C source programs.

Step 1: Compile

cc translates a C source program into an assembly language program. This translation is called compilation. The command to compile the C source that's in the file *exmpl.c* which is in the current user area on the b: drive is:

```
cc b:exmpl.c
```

cc writes the assembly language source for the C program to a temporary file in the current area on the b: drive, and then starts the assembler. The 'current area' on PCDOS/MSDOS is the current directory and on CP/M-86 is the current user area.

Step 2: Assemble

as, which is automatically started by the compiler, unless you specify otherwise, translates an assembly language source program into relocatable object code.

It writes the object code to the file *exmpl.o* in the current area on the b: drive, and then automatically deletes the temporary assembler source file, since it is no longer needed.

Step 3: Link

The object module version of the *exmpl* program must next be linked to needed functions that are in the *c.lib* library of object

modules and converted into an executable form. This is done by entering:

```
ln b:exmpl.o c.lib
```

The output of the linker is sent to the file *exmpl.exe* on PCDOS/MSDOS, and to *exmpl.cmd* on CP/M-86, in the current area on the b: drive.

During the link step, the linker will search libraries specified to it; when it finds a module containing a needed function, it will include the module in the executable file it's building.

All C programs need to be linked with *c.lib* (or an equivalent, if the program uses a memory model other than 'small code'/ 'small data'. This library contains the non-floating point functions which are defined in the functions chapter, *lib* of this manual. It also contains functions which are called by compiler-generated code.

If a program performs floating point operations, it must also be linked with a math library. The math library that you will use when getting familiar with Aztec C is *m.lib*. This uses 'small code', 'small data', and uses software routines to perform the floating point operations. Other versions of the math library are available. See the Libraries section of the Technical Information chapter for details.

When a program is linked with a math library, that library must be specified before *c.lib*. For example, if *exmpl.c* performed floating point, the following would link it:

```
ln b:exmpl.o m.lib c.lib
```

Creating an executable program, when using the PCDOS/MSDOS linker

If you are using the PCDOS/MSDOS linker, the following commands will convert *exmpl.c* into an executable program:

```
cc exmpl.c
obj exmpl.o
link crt0+exmpl,exmpl,exmpl,,mc
```

3. Where to go from here

You now have a working disk and have used it to create an executable program from a C source file. Since the C language supported by Aztec C is fully UNIX-compatible, and since Aztec C86 supports a large subset of the standard UNIX functions, you can proceed to develop programs without immediately reading the rest of this manual, with assurance that that Aztec C behaves like the version of UNIX with which you are familiar, or like the textbook on UNIX and C that you are reading.

To be able to make full use of Aztec C, however, you must read the rest of this manual. Some topics of interest:

* You should peruse the Library Functions chapters. We have provided a lot of functions, including 8086-specific ones, and an awareness of them may save you from reimplementing some of them.

* As your programs get larger, you may want to either use one of the large memory model options or partition them into overlays. Memory models are discussed in the Operator Information section of the Compiler chapter, and Overlays in the Technical Information chapter. Use of alternate memory models for a program requires use of alternate versions of the *c.lib* and *m.lib*. The libraries provided with Aztec C are discussed in the Libraries section of the Technical Information chapter.

* You have control over several factors that determine how a program is organized in memory. Some of the most important of these determine the size of the program's stack and heap areas, and whether a program's stack is above or below the heap. (the heap is the area from which buffers are dynamically allocated). With the heap above the stack, the heap can grow and contract dynamically, and with it the space allocated to the program, to satisfy requests for buffers. For a discussion of program organization, see the Technical Information chapter.

* The version of the math library that we have discussed in this chapter, *m.lib*, performs floating point operations using software routines, and requires the program to use the 'small code' and 'small data' memory models. Other versions of the math library are provided that use the 8087 to perform floating point operations, and that use different memory models. With some of the math libraries, the choice of software- or 8087-execution of floating point operations is made when the program is linked. For others, the choice is made when the program is started: an 8087 will be used if available, and software routines otherwise. The Libraries section of the Technical Information chapter has the details.

* Use of the PCDOS/MSDOS linker instead of the Aztec linker is discussed in detail in the section "Using the PCDOS/MSDOS Linker" section of the Technical Information chapter.

* You may want to create modules that can call functions in other manufacturers' libraries; for example, those of Lattice or Computer Inovations. Or you may want to create modules or libraries that can be called by modules that have been created with other manufacturers' compiler and assembler. The compiler options relevant to this are discussed in the

Options section of the Compiler chapter. In this case, you must use the PCDOS/MSDOS linker *link*. This is discussed in the section "Using the PCDOS/MSDOS Linker", in the Technical Information chapter.

* The creation of ROMable code is discussed in the Technical Information chapter.
* If you want to write assembly language routines that can call and be called by C functions, see the "Assembler Functions" section of the Technical Information chapter and the Programmer Information section of the Assembler chapter.
* The compiler has an option that causes it to generate code that uses 80186 instructions. If you have a machine that has and 80186 or 80286 microprocessor, you may want to compile your programs using this option, and recompile the libraries using this option. See the Options section of the Compiler chapter for details.

THE COMPILER

Chapter Contents

The Compiler cc
1. Operating Instructions 5
1.1 The C Source File 6
1.2 The Output Files 7
1.2.1 Creating an Object Code File 7
1.2.2 Creating just an Assembly Language Source File 8
1.3 Searching for #include Files 9
1.3.1 The -I Option 9
1.3.2 The INCLUDE Environment Variable 10
1.3.3 The Search Order for #include Files 10
1.4 Memory Models 11
1.4.1 How a Memory Model is Selected 12
1.4.2 Multi-module Programs 13
1.4.3 Program Organization 15
1.4.4 'Large model' versus Overlays 15
1.4.5 Implementation of the Memory Models 16
2. Compiler Options 19
2.1 Summary of the Options 19
2.1.1 Machine-Independent Utility Options 19
2.1.2 Table Manipulation Options 19
2.1.3 8086 Options for the Optimizing Compilers 20
2.1.4 8086 Options for the Non-optimizing Compilers 21
2.2 The Options 22
2.2.1 Machine-Independent Utility Options 22
2.2.2 Table Manipulation Options 23
2.2.3 8086 Options for the Optimizing Compilers 26
2.2.4 8086 Options for the Non-optimizing Compilers 29
3. Programming Information 31
3.1 Supported Language Features 31
3.1.1 Preprocessor Statements 31
3.1.1.1 Macros 31
#define 31
3.1.1.2 Conditional Compilation 34
#ifdef 35
#ifndef 35
#if 35
3.1.1.3 More Preprocessor Statements 36
#include 36
#line 36
#asm and #endasm 36
3.1.2 More Features 37
Structure Assignment 37

Line Continuation 37
The *void* Data Type 38
Special Symbols 38
___FILE___ 38
___LINE___ 38
___FUNC___ 38
3.1.3 Special Features of Aztec C 39
String Merging 39
Reserved Words 39
Global Variables 39
3.2 Data Formats 41
3.2.1 char 41
3.2.2 Pointer 41
3.2.3 int, short 41
3.2.4 long 41
3.2.5 float and double 42
3.3 Floating Point Exceptions 42
3.4 Writing Machine-Independent Code 42
3.4.1 Compatibility between Aztec C Products 42
3.4.2 Sign Extension for *char* variables 43
3.4.3 The MPU... Symbols 43
3.5 Using Long Pointers 44
3.5.1 Passing Long Pointers Between Functions 44
3.5.2 Expressions involving Long Pointers 46
3.5.3 Creating and Accessing Huge Arrays 48
4. Error Messages 50

The Compiler

This chapter describes how to use the Aztec C compiler. It is not intended to be a complete guide to the C language; for that, you must consult other texts. One such text is *The C Programming Language*, by Kernighan and Ritchie. The Aztec C compiler was implemented according to the language description in the Kernighan and Ritchie book.

As mentioned in the Tutorial chapter, some Aztec C86 Systems provide two C compilers, supporting different features. The *cc* compiler in the Developer and Commercial Aztec C86 Systems supports the full C language (now including bit fields), generates optimized code, can optionally generate code that takes advantage of the 80186 and 80286 processors, and supports the large memory models. The *ccb* compiler in the Developer and Commercial Systems and the *cc* compiler in the Personal System support the full C language except for bit fields, quickly generate non-optimized code, can't generate 80186 and 80286 code, and don't support the large memory models. All the compilers are operationally the same, with the exception of the 8086-specific options. Modules that are compiled with different compilers can be linked together into one program. The only place in this chapter where we make a distinction between the compilers is in the discussion of the 8086-specific options.

This chapter has four major sections: the first describes how to use the compiler, the second describes the compiler options, the third describes information related to the writing of programs, and the fourth describes error processing,

1. Compiler Operating Instructions

The compiler is invoked by a command of the format:

```
cc [-options] filename.c
```

where [-options] specify optional parameters, and *filename.c* is the name of the file containing the C source program. Options can appear either before or after the name of the C source file.

The compiler reads C source statements from the input file, translates them to assembly language source, and writes the result to another file.

When the compiler is done, it activates the Manx assembler, unless it's told not to. The assembler translates the assembly language source to relocatable object code, writes the result to another file, and deletes

the assembly language source file. The compiler -A option tells the compiler not to start the assembler.

1.1 The C source file

On the command line, the name of the file containing the C source can optionally specify the drive on which the file is located. If not specified, it's assumed to be on the default drive.

1.1.1 Source files on MSDOS and PCDOS.

On MSDOS and PCDOS, the source file name can optionally specify a path to the directory containing the file. By default, it's assumed to be in the current directory on the specified drive. For example, with the following command the compiler looks for *filename.c* on drive *a:*, directory *\source\subs*:

```
cc a:\source\subs\filename.c
```

and for the following command, with *b:* as the default drive and *\modules* as the current directory, the compiler looks for *filename.c* on the *b:* drive, directory *\modules*:

```
cc filename.c
```

1.1.2 Source files on CP/M-86.

On CP/M-86, the source filename can optionally specify the user area containing the file. If not present, it's assumed to be in the current user area on the specified drive. For example, with the following command, the compiler will look for *clock.c* on drive *h:*, user 5:

```
cc 5/b:clock.c
```

As shown in this example, a CP/M-86 filename consists of (1) optionally, a user area followed by a backslash, (2) optionally, a drive identifier, followed by a colon, (3) the file name, and (4) optionally, a period followed by an extension. On CP/M-86, any file name passed to a Manx program has this format.

For another CP/M-86 example, if the default drive is c:, and the current user area is 8, with the following command, the compiler will look for *generate.c* on drive *c:*, user 8:

```
cc generate.c
```

1.1.3 More source file information.

The extension on the source file name is optional. If not specified, it's assumed to be *.c*. For example, with the following command, the compiler will assume the file name is text.c:

```
cc text
```

The compiler will append *.c* to the source file name only if it doesn't

find a period in the file name. So if the name of the source file really doesn't have an extension, you must compile it like this:

```
cc filename.
```

The period in the name prevents the compiler from tacking on *.c* to the name.

1.2 The output files

1.2.1 Creating an object code file

Normally, when you compile a C program you are interested in the relocatable object code for the program, and not in its assembly language source. Because of this, the compiler by default writes the assembly language source for a C program to an intermediate file and then automatically starts the assembler. The assembler then translates the assembly language source to relocatable object code, writes this code to a file, and erases the intermediate file.

By default, the object code generated by a compiler-started assembler is sent to a file whose name is derived from that of the file containing the C source by changing its extension to *.o*. This file is placed in the area that contains the C source file. On MSDOS and PCDOS, the area is the directory containing the source file, and on CP/M-86 it's the user area containing the source file. For example, if the compiler is started with the command

```
cc prog.c
```

the file *prog.o* will be created, containing the relocatable object code for the program.

The name of the file containing the object code created by a compiler-started assembler can also be explicitly specified when the compiler is started, using the compiler's -O option. For example, the command

```
cc -O myobj.rel prog.c
```

compiles and assembles the C source that's in the file *prog.c*, writing the object code to the file *myobj.rel*.

When the compiler is going to automatically start the assembler, it by default writes the assembly language source to a temporary file named *ctmpxxx.xxx*, where the x's are replaced by digits in such a way that the name becomes unique.

On MSDOS and PCDOS the temporary file is placed in the drive and directory specified by the environment variable *CCTEMP*. If this variable doesn't exist, the file is placed in the current directory on the default drive.

The format of *CCTEMP* is

[d:][path\]

where the brackets indicate an optional field. The fields have the following meanings:

* *d:* is the identifier of the drive on which the file is to be placed; if not specified, the default drive is used.
* *path* specifies the directories that must be passed through to reach the directory in which the temporary file is to be placed. If not specified, the file is placed in the root directory on the selected drive. When *path* is specified, it must have a trailing backslash character.

For example, the first command that follows sets *CCTEMP* so that the temporary file is placed in the root directory on the *c:* drive; the second causes it to be placed in the *compile\temp* directory on the default drive; the third causes it to be placed in the *tmp* directory on the *d:* drive:

```
set CCTEMP=c:
set CCTEMP=compile\temp\
set CCTEMP=d:tmp\
```

Note that a terminating backslash is required when a subdirectory is explicitly specified, but not when just the drive is specified.

On CP/M-86 the temporary file is always placed in the current user area of the default drive.

If you are interested in the assembly language source, but still want the compiler to start the assembler, specify the option -T when you start the compiler. This will cause the compiler to send the assembly language source to a file whose name is derived from that of the file containing the C source by changing its extension to *.asm*. The C source statements will be included as comments in the assembly language source. For example, the command

```
cc -T prog.c
```

compiles and assembles *prog.c*, creating the files *prog.asm* and *prog.o*.

1.2.2 Creating just an assembly language file

There are some programs for which you don't want the compiler to automatically start the assembler. For example, you may want to modify the assembly language generated by the compiler for a particular program. Or you may want the assembly language source sent to a location, such as a RAM disk, where it wouldn't normally be sent when the compiler activates the assembler.

In such cases, you can use the compiler's *-A* option, which prevents the compiler from starting the assembler.

When you compile a program using the *-A* option, you can tell the compiler the name and location of the file to which it should write the

assembly language source, using the *-O* option.

If you don't use the *-O* option but do use the *-A* option, the compiler will choose the name and location of the assembly language source file: it will send the assembly language source to a file whose name is derived from that of the C source file by changing the extension to *.asm*, and place this file in the same area as the one that contains the C source file. On MSDOS and PCDOS, the area is the directory containing the source file, and on CP/M-86 it's the user area on the drive containing the source file.

For example, the command

```
cc -A prog.c
```

compiles, without assembling, the C source that's in *prog.c*, sending the assembly language source to *prog.asm*.

As another example, the command

```
cc -A -O e:temp.asm prog.c
```

compiles, without assembling, the C source that's in *prog.c*, sending the assembly language source to the file *temp.asm* on the drive *e:*.

When the -A option is used, the option -T causes the compiler to include the C source statements as comments in the assembly language source.

1.3 Searching for *#include* files

You can make the compiler search for *#include* files in a sequence of areas, thus allowing source files and *#include* files to be contained in different areas.

Areas can be specified with the -I compiler option, and, on MSDOS and PCDOS, with the INCLUDE environment variable. The compiler itself also selects a few areas to search. The maximum number of searched areas is eight.

If the file name in the #include statement specifies a drive id, user area, or path, only the single area specified in the statement is searched.

1.3.1 The -I option.

A -I option defines a single area to be searched. The area descriptor follows the -I, with no intervening blanks.

1.3.1.1 The -I option on MSDOS and PCDOS

On MSDOS and PCDOS, the -I option looks just like you'd expect:

```
-Ib:\incfiles
```

defines the directory *\incfiles* on drive b:.

1.3.1.2 The -I option on CP/M-86

On CP/M-86, the area descriptor following the -I consists of (1) an optional user number followed by a slash, and (2) an optional drive identifier. For example, the following defines user area 5 on drive c:

```
-I5/c:
```

The user number is optional, and defaults to the current user number:

```
-Id:
```

defines the current user area on the d: drive. The drive id is also optional, and defaults to the default drive:

```
-I4/
```

defines user area 4 on the default drive.

1.3.2 The INCLUDE environment variable.

On MSDOS and PCDOS, the INCLUDE environment variable also defines directories to be searched for #include files. This variable has the same format as the PATH environment variable. That is, something like the following, which defines three areas to be searched:

```
set INCLUDE=b:\incl;c:\cc\inc2;a:
```

1.3.3 The search order for include files

1.3.3.1 The search order on MSDOS and PCDOS.

On MSDOS and PCDOS, directories are searched in the following order:

1. If the #include statement delimited the file name with the double quote character, ", the current directory on the default drive is searched. If delimited by angle brackets, < and >, this area isn't automatically searched.
2. The directories defined in -I options are searched, in the order listed on the command line.
3. The directories defined in the INCLUDE environment variable are searched, in the order listed.

1.3.3.2 The search order on CP/M-86.

On CP/M-86, user areas are searched in the following order:

1. If the #include statement delimited the file name with the double quote character ("), the current user area on the default drive is searched. If delimited by angle brackets, < and >, this area isn't automatically searched.
2. The directories specified in -I options are searched, in the order listed on the command line.

3. If the current user number isn't zero, user area 0 on the default drive is searched.

4. If the default drive isn't A:, and if the A: drive is logged in, that is, has been accessed, user area 0 on the A: drive is searched.

1.4 Memory models

This section discusses the different memory models supported by Aztec C86. The *cc* compiler that is in the Developer and Commercial Aztec C86 Systems allows you to select the memory model that a program will use. The *ccb* compiler that is provided with these systems and the *cc* compiler that is provided with the Personal Aztec C86 System supports just one model: 'small code' and 'small data'.

A program created by Aztec C86 is organized into several sections. The memory model selected for a program determines how large the program's sections can be.

The sections of a program are these:

* *code*, containing the program's executable code;
* *data*, containing the program's global and static data;
* *stack*, containing its automatic variables, control information, and temporary variables;
* *heap*, an area from which buffers are dynamically allocated.

There are two attributes to a program's memory model. One determines the amount of executable code the program can have. This attribute can specify that a program is to have *small code* or *large code*:

* *small code* limits a program to 64K bytes of code, if the program isn't partitioned into overlays;
* *large code* means that there's no limit to the size of a program's code.

The other attribute to a program's memory model determines the amount of data the program can have. This attribute can specify that a program is to have *small data* or *large data*:

* *small data* limits the sum of the sizes of a program's data, stack, and heap sections to 64K bytes;
* *large data* allows the program's data section to be up to 64K bytes long, its stack to be up to 64K bytes long, and its heap to be any size up to the remaining amount of memory.

Even with 'large data' there is a limitation on the size of data objects such as arrays and dynamically allocated buffers: they can't normally contain more than 64K bytes. Bigger arrays and buffers can be created, but their fields can be accessed only by using 8086-specific code.

One other important characteristic of 'large data' programs is that they can directly access any memory location, since such a program uses long pointers (four bytes containing segment and offset components) to data objects. A 'small data' program uses short pointers (two bytes containing just an offset within the program's data area) to data objects, and hence can't directly access memory outside its data area.

When a program uses either 'large code' or 'large data' it obviously has one advantage over a 'small code', 'small data' version of the same program: it can be bigger. It also has some disadvantages:

* The program is larger;
* It takes longer to load the program;
* For 'large code', function calls and returns take longer;
* For 'large data', it takes longer to access a variable via a pointer;
* For 'large data', the ability to access any location in memory means that the program can accidentally destroy any location in memory, possibly resulting in unexplained crashes or other anti-social behavior.

Thus, indiscriminate use of the 'large code' and/or 'large data' memory model options for programs is not recommended.

Only *.exe* programs that run on PC-DOS or MS-DOS, version 2.0 or later can use the 'large code' or 'large data' memory model. DOS 2.x *.com* programs, DOS 1.1 programs, and CP/M-86 programs must use 'small code' and 'small data'.

1.4.1 Selecting a module's memory model

The memory model to be used by a module is selected when the module is compiled. With the *cc* compiler that is in the Developer and Commercial Aztec C86 Systems, you can explicitly select a module's memory model using the following compiler options:

+LC	Large code, small data;
+LD	Small code, large data;
+L	Large code, large data.

With this compiler, if a module is compiled without the specification of a memory model, it will have the 'small code', 'small data' memory model.

With the *ccb* compiler that is in the Develop and Commercial Systems, and the *cc* compiler that is in the Personal System, a module always has the 'small code', 'small data' memory model.

For example, the following commands compile *prog.c* to use different memory models;

command	*memory model*
cc prog	small code, small data
cc +LC prog	large code, small data
cc +LD prog	small code, large data
cc +LC +LD prog	large code, large data
cc +L prog	large code, large data

Compiling a module to use the 'large data' memory model just causes the module's data object pointers to be long pointers. In order for the program to actually have the large data areas, it must be linked with a 'large data' version of *c.lib*, which includes a special startup routine that sets up the program's data areas. This is discussed below.

1.4.2 Multi-module programs

A C program contains multiple modules, which are linked together to form the executable program. In this section we want to discuss the relationship of the memory models that are used by a program's modules.

1.4.2.1 You can't mix 'large code' and 'small code' modules

All modules that are linked together to form an executable program must use the same memory model code option. That is, they must either all use the 'small code' or all use the 'large code' memory model option.

1.4.2.2 Mixing 'large data' and 'small data' modules

There are two characteristics to a program's data memory model: the maximum size of its memory-resident data areas, and the size of pointers to data objects. As mentioned above, the former characteristic is given to a program by the startup routine with which the program is linked; this in turn depends on whether the program is linked with a 'large data' or 'small data' version of *c.lib*. The latter characteristic is given to an individual module when it is compiled.

Usually, you'll want a program to have either large data areas and long data object pointers, or small data areas and short data object pointers. And you will prefer for a program to use small data areas and short data object pointers whenever possible, since the use of long pointers makes a program larger and slower.

It is possible and occasionally useful, however, to mix together in the same program modules that use different memory model data options. This allows you to keep a program's size down and its speed up, by compiling most of its modules to use the 'small data' memory model, and by linking it with a 'small data' version of *c.lib*, while still allowing the program to directly access any location in memory, via modules that have been compiled to use the 'large data' memory model.

For example, if you are writing a driver that can be called by other programs, you could place the functions that initialize the interrupt vector table and that access the caller's memory space in 'large data' modules and the rest of the program's functions in 'small data' modules.

The only requirements for a program that mixes modules that use different data memory models are:

* It must be linked with a version of *c.lib* that uses the 'small data' memory model.
* When a pointer to a data object is passed between two functions or when a global pointer is referenced by two functions, the functions must use the same memory model data option.

1.4.2.3 Libraries

The rules presented above concerning the mixing of modules that use different memory models also apply when some of the modules are in libraries. Thus, it's possible that you may need up to four versions of a library, corresponding to the four possible combinations of memory model options. In a particular version of a library, all modules must have been generated to use the same memory model options.

For example, there are four possible versions of *c.lib*, whose modules use the following combinations of memory models:

* 'small code', 'small data',
* 'large code', 'large data',
* 'small code', 'large data'
* 'large code, 'small data'

Similarly, there are four possible versions of each of the other libraries provided with Aztec C86.

For most libraries, the only difference between a 'small data' and 'large data' version of the library is that for the 'small data' library the modules are compiled to use short pointers to data objects, and for the 'large data' library they are compiled to use long pointers to data objects. For *c.lib*, the versions also contain a different startup routine: a 'small data' version contains *sbegin*, while a 'large data' version contains *lbegin*. *sbegin* gives a program the 'small data' memory organization (ie, a single physical data segment), while *lbegin* gives a program the 'large data' memory organization (ie, separate data and stack segments, and separate heap space).

The following rules define the libraries that you should use:

* Use a 'small code' or 'large code' version of a library, depending on whether your modules use 'small code' or 'large code';

* Use a 'small data' version of a library if *any* of your modules uses 'small data'. Use a 'large data' version of a library only if *all* of your modules use 'large data'.

Since it is illegal to link together modules that use different memory model code options, the linker will generate an error message in this case. However, since it is legal to link together modules that use different memory model data options, the linker won't generate a message in this case. Thus, you must be careful when mixing modules that use different memory model data options, since the linker can't know whether it was intentional, or whether the resulting code is correct.

There are several Aztec C86 packages available. They don't all provide all four memory model versions of each library. For more information, see the Libraries section of the Technical Information chapter and the release document.

1.4.3 Program Organization

The memory model that is selected for a program affects how the program is organized in memory. For a discussion of this, see the section "Program Organization" in the Technical Information chapter.

1.4.4 'large model' versus overlays

Normally, when a program is created by the Manx Linker, the entire program resides in memory. If you have a big program you can select one of the large memory models for the program. The program has then acquired the negative features noted above; in addition, the machine on which the program is to run must have enough memory for the entire program to reside in memory at once.

An alternative way to create the program is to partition it into overlays. When a program is partitioned into overlays, only those parts that are actually being executed need to be in memory at once. Thus, a program containing overlays can be as large as needed, regardless of the amount of memory available on the machine on which the program is to run.

There is degradation in performance of an overlaid program compared to a non-overlaid version of the same program, since the overlays must be loaded into memory from disk before they can be executed. But with judicious partitioning of the program, the affect of the loading of overlays can sometimes be minimized.

A program cannot use both overlays and a large memory model.

1.4.5 Implementation of the memory models

The following paragraphs discuss the memory models supported by Aztec C86 in more detail than was discussed above. You don't need to read this discussion in order to create programs that use a large

memory model.

1.4.5.1 Small code

The executable code for a program that uses the 'small code' memory model is contained in a single logical segment, and all functions are 'near' procedures. The CS register is set up to point to the beginning of this segment, and is never changed. All references to functions, such as function calls and pointers to functions, are represented by two bytes, which contain the offset of the function from the beginning of the code segment. These offsets are determined when the program is linked, and hence don't require adjustment when the program is loaded.

1.4.5.2 Large code

The executable code for a program that uses the 'large code' memory model is contained in multiple logical segments, each containing the functions declared in a single module. When a function is active, the CS segment register points to the beginning of the segment containing the function.

Thus, on entry to or exit from a function in a program that is using large code option, the CS segment register must be modified. This is not the case for programs that use the small code option. Hence, function calls and returns are executed slightly faster for 'small code' programs for 'large code' programs.

When a program uses the large code option, the address of a function is contained in four bytes in a 'call' instruction or in a variable that points to a function: two bytes contain the paragraph number of the beginning of the segment containing the function, and the other two contain the offset of the function's entry point from the beginning of the segment. As noted above, function addresses are represented by two bytes in programs using the small code option. Hence, a program will be larger if it uses the large code option than if it uses the small code option.

If a program uses the large code option, the paragraph numbers of the logical segments that contain functions aren't known until the program is loaded. Thus, fields within a large code program that contain function addresses (that is, function calls and variables that point to functions) must be modified when the program is loaded into memory. Only the paragraph number of a function reference needs to be modified; the offset of the function within its segment is correctly determined when the program is linked. This modification isn't necessary for 'small code' programs. Hence, a program will take longer to load if it uses the 'large code' memory model than if it uses the 'small code' memory model.

1.4.5.3 Small data

The data-containing sections of a program that has been linked with a 'small data' version of *c.lib* are organized into a single section of memory, as described in the Program Organization section of the Technical Information chapter. The DS, ES, and SS segment registers are initialized to the beginning of this block when the program is loaded, and don't change. Note that while ES is initialized to the same value as DS, no Aztec function or program requires it to have a particular value.

Data object pointers in modules that have been compiled to use the 'small data' memory model are two bytes long, consisting of the offset of the object from the beginning of the physical data segment. The linker determines these offsets; hence they don't have to be adjusted when the program is loaded into memory.

1.4.5.4 Large data

The data-containing sections of a program that has been linked with a 'large data' version of *c.lib* are organized into three separate blocks of memory, as described in the Program Organization section of the Technical Information chapter.

For a 'large data' program, all its modules must be compiled to use the 'large data' memory model. This causes its data object pointers to be four bytes long; two bytes contain the paragraph number of the beginning of the segment that contains the item, and the other two bytes contain the offset of the item within this segment.

When a program accesses a variable in one of the data sections directly (that is, not via a pointer), the access is as fast when the program uses 'large data' as when it uses 'small data'. This is because the compiler-generated code knows which segment the item is in and which segment register points to this segment, and hence can generate code that accesses the item without having to load a segment register with the paragraph number of the segment.

When a program accesses a variable via a pointer, the access is slower if the program uses 'large data' than if it uses 'small data'. The reason for this is that a 'large data' program must load a four-byte pointer into registers, while a 'small data' program must only load a two-byte pointer.

If a program pre-initializes a pointer to a data item with a declaration such as

```
char *cp=&a;
```

the pointer must be adjusted when the program is loaded into memory. The reason for this is that the starting address of the segment containing the pointed-at item is not known until the program is loaded. The offset of the item within its segment is determined when

the program is linked, and hence need not be modified when the program is loaded. For a 'small data' program, pre-initialized pointers to data items don't have to be adjusted when the program is loaded; Hence, a program will take longer to load if it uses 'large data' than if it uses 'small data'.

2. Compiler Options

2.1 Summary of options

There are two types of options in Aztec C compilers: machine independent and machine dependent. The machine-independent options are provided on all Aztec C compilers. They are identified by a leading minus sign.

The Aztec C compiler for each target system has its own, machine-dependent, options. Such options are identified by a leading plus sign.

There is one set of 8086-specific options for the *cc* compiler that is in the Developer and Commercial Aztec C86 Systems. There is another set of 8086-specific options for the *ccb* compiler that is in these systems and for the *cc* compiler that is in the Personal System. In the description of options that follow, we refer to the *cc* compiler that is in the Developer and Commercial Systems as the 'Optimizing Compiler'. We refer to the *ccb* compiler that's in these systems and the *cc* compiler that's in the Personal System as 'Non-Optimizing Compilers'.

2.1.1 Machine-independent utility Options

-A Prevents the compiler from starting the assembler.

-D Defines a symbol for the preprocessor.

-I Defines an area to be searched for files specified in a #include statement.

-O Used to specify an alternate output file.

-S Don't print warning messages.

-T Include C source statements in the assembly code output as comments. Each source statement appears before the assembly code it generates.

-B Don't pause after every fifth error to ask if the compiler should continue. See the Errors section of this chapter for details.

2.1.2 Table Manipulation Options

The following options allow you to specify the size of the tables used by the compiler. They are preceded by a minus sign, indicating that they are common to all Aztec compilers.

-E Specifies the size of the expression table.

-L Specifies the size of the local symbol table.

-Y Specifies the maximum number of outstanding cases allowed in a switch.

-Z Specifies the size of the table for literal strings.

2.1.3 8086 options for the Optimizing Compiler

The following options are available for the optimizing compiler, *cc*, that is provided with the Developer and Commercial Aztec C86 Systems.

+F Generate fast, rather than compact, code.

+C Generate compact, rather than fast, code.

+N Save register variables before all function calls, and restore them afterwards. This option must be specified when calling functions in modules that have been compiled by other compilers, such as Lattice and CI/C86.

+D Module calls Lattice-compiled function that returns a long. The compiler will generate code to convert long values returned by called functions to Aztec format.

+DF Module is being called by Lattice-compiled function. The compiler will generate code to return long values in Lattice format.

+U Convert uninitialized global variables into externs.

+R Disable register tracking between statements.

+0 Generate code for an 8086 or 8088 processor.

+1 Generate code for an 80186 processor.

+2 Generate code for an 80286 processor.

+LC Generate code that uses the 'large code', 'small data' memory model. (for more information on memory models, see the Operator Information section of this chapter).

+LD Generate code that uses the 'small code', 'large data' memory model.

+L Generate code that uses the 'large code', 'large data' memory model.

+A The module being compiled does not use 'aliases' of the form **ptr* for a named variable when assigning values to the variable.

+M When a statement is encountered that requires multiplication by a constant, always use the 8086 'multiply' instruction.

2.1.4 8086 options for the Non-optimizing Compilers

The following 8086 options are provided for non-optimizing Aztec C86 compilers. As mentioned above, these compilers are the *cc* compiler that is provided with the Personal Aztec C86 System, and the *ccb* compiler that is provided with the Developer and Commercial Aztec C86 Systems.

+F	Forces frame allocation to take place in-line rather than through a call to a library function.
+U	Same as the optimizing compilers' +U option.
+J	Generate short, rather than long, conditional jump instructions.

2.2 Detailed description of the options

2.2.1 Machine-independent utility options

The -D Option (Define a macro)

The *-D* option defines a symbol in the same way as the preprocessor directive, *#define*. Its usage is as follows:

```
cc -Dmacro[=text] prog.c
```

For example,

```
cc -DMAXLEN=1000 prog.c
```

is equivalent to inserting the following line at the beginning of the program:

```
#define  MAXLEN  1000
```

Since the *-D* option causes a symbol to be defined for the preprocessor, this can be used in conjunction with the preprocessor directive, *#ifdef*, to selectively include code in a compilation. A common example is code such as the following:

```
#ifdef  DEBUG
    printf("value: %d\n", i);
#endif
```

This debugging code would be included in the compiled source by the following command:

```
cc -dDEBUG program.c
```

When no substitution text is specified, the symbol is defined as the numerical value, one.

The -I Option (Include another source file)

The *-I* option causes the compiler to search in a specified area for files included in the source code. On MSDOS and PCDOS, the area is a directory on a drive; on CP/M-86, it's a user area on a drive.

The name of the area immediately follows the -I, with no intervening spaces. For example, on MSDOS and PCDOS, the following defines directory \source\inc on drive b: as a search area:

```
-Ib:\source\inc
```

On CP/M-86, the area consists of (1) optionally, a user number and slash, (2) optionally, a drive id. The user number defaults to the current user number, and the drive defaults to the default drive. For example, the following defines user 8 on drive c: as a search area:

```
-I8/c:
```

For more details, see the Compiler Operating Instructions, above.

The -S Option (Be Silent)

The compiler considers some errors to be genuine errors and others to be possible errors. For the first type of error, the compiler always generates an error message. For the second, it generates a warning message. The *-S* option causes the compiler to not print warning messages.

2.2.2 Table Manipulation Options

The compiler has several memory-resident tables in which to store information about a program it is compiling. Some of these tables are used to keep track of the symbols defined within the program, and some as a "scratch pad" for temporarily storing information.

The compiler uses the following tables: macro/global symbol table, local symbol table, label table, string table, expression work table, and case statement work table.

The sizes of these tables are determined when the compiler starts. For all tables except the macro/global symbol table, the size can be specified by the user with a command line option; if the user doesn't specify the size of one of these tables, the compiler sets it to a default value.

The macro/global symbol table is located in memory above all the other tables. Its size is set after all the other table sizes have been set, so that it uses all the rest of available memory. Hence, the user can't set the size of this table.

If a table overflows, the compiler will print an error message and stop. If any table except the macro/global symbol table overflowed, the compilation can be restarted, using a different size for the table which overflowed. If the macro/global symbol table overflowed, the compilation can be restarted, using smaller sizes for one or more of the other tables.

2.2.2.1 The Macro/Global Symbol Table

This table is where macros defined with the #define statement are remembered. It also contains information about all global symbols.

If this table overflows, the message

Out of Memory!

will be printed.

2.2.2.2 The Local Symbol Table and the -L Option

New symbols can be declared after any open brace. Most commonly, a declaration list appears at the beginning of a function body. The symbols declared here are added to the local symbol table. If a variable is declared in the body of, say, a *for* loop, it is added to the table. When the compiler has finished compiling the loop, that entry in

the table is freed up. And when it has finished the function, the table will be empty.

The default size of the table is 40 entries. Since each entry consumes 26 bytes, the table begins at 520 bytes. If the table overflows, the compiler will send a message to the screen and stop.

The number of entries in the table can be adjusted with the -L option. The following compilation will use a table of 75 entries, or almost 2000 bytes:

```
cc -L75 program.c
```

2.2.2.3 The Expression Table and the -E Option

This is the area where the "current" expression is handled. It is the compiler's work space as it interprets a line of C code. The various parts of the line are stored here while the statement is being compiled. When the compiler moves on to the next expression, this space is again freed for use.

The default value for -E is 80 entries. Each "entry" in the table consumes 14 bytes in memory. So the expression table starts at 840 bytes. Each operand and operator in an expression is one entry in the expression table-- another fourteen bytes. The term, "operator", includes each function and each comma in an argument list, as well as the symbols you would normally expect (+, &, ~, etc.). There are some other rules for determining the number of entries an expression will require. Since they are not straightforward and are subject to change, they will not be discussed here.

The following expression uses 15 entries in the table:

```
a = b + function( a + 7, b, d) * x
```

Everything is an entry except for the ")", including the commas which separate the function arguments.

If the expression table overflows, the compiler will generate error number 36, "no more expression space."

This command will reserve space for 100 entries (1400 bytes) in the expression table:

```
cc -E100 filename
```

The option must be given before the filename. There can be no space between the option letter and the value.

2.2.2.4 The Case Table and the -Y Option

When the compiler looks at a switch statement, it builds a table of the cases in it. When it "leaves" the switch statement, it frees up the entries for that switch. For example, the following will use a maximum of four entries in the case table:

```
switch (a) {
case 0:                    /* one */
    a += 1;
    break;
case 1:                    /* two */
    switch (x) {
    case 'a':              /* three */
        func1 (a);
        break;
    case 'b':              /* four */
        func2 (b);
        break;
    }                  /* release the last two */
    a = 5;
case 3:                  /* total ends at three */
    func2 (a);
    break;
}
```

The table defaults to 100 entries, each using up four bytes. If the compiler returns with an error 76 ("case table exhausted"), you will have to recompile with a new size, as in:

```
cc -Y100 file
```

2.2.2.5 The String Table and the -Z Option

This is where the compiler saves "literals", or strings. The size of this area defaults to 2000 bytes. Each string occupies a number of bytes equal to the size of the string. The size of a string is just the number of characters in it plus one (for the null terminator).

If the string table overflows, the compiler will generate error 2, "string space exhausted". The following command will reserve 3000 bytes for the string table:

```
cc -Z3000 file
```

2.2.3 8086 options for the Optimizing Compiler

The +F Option (Generate Fast Code)

The *+F* option causes the compiler to select code sequences that yield the fastest possible execution time, even at the cost of increased program size.

The +C Option (Generate Small Code)

The *+C* option causes the compiler to select code sequences that yield the smallest resultant program size, even at the cost of reduced execution speed.

The +N Option (Foreign Functions)

The *+N* option causes the compiler to generate code for function calls that saves registers that contain register variables before issuing the function call and that restores the registers afterwards.

This option is not needed when calling functions that are in modules that have been compiled by Aztec C, since such modules preserve the caller's register variables.

It's only needed when calling functions that are in modules that have been compiled by other C compilers, such as Lattice and CI/C86, if those functions don't preserve register variables.

The +D Option (Lattice Interface)

The *+D* option must be specified for modules that call Lattice-compiled functions that return a long int. It causes the compiler to generate code that fetches the returned value from the registers in which Lattice-compiled functions return long ints, (AX and BX) rather than from those in which Aztec-compiled functions return long ints (AX and DX). This option need not be used when calling CI/C86-compiled functions that return longs, since they return longs in the same registers as Aztec C86-compiled functions.

If this option is specified when compiling a module, all functions called by the module that return a long must return the value in AX and BX. That is, the module can't call both Lattice-compiled functions and Aztec C86-compiled functions that return longs.

The +DF Option (Lattice Interface)

The *+DF* option must be specified when compiling a module containing a function that returns a long, and that will either be called by a Lattice-compiled function or be called by a function that has been compiled with the Aztec compiler using the *+D* option. The option causes the compiler to return the long value in the registers in which the Lattice function will expect to find it (AX and BX) rather than the registers that Aztec C86-compiled functions normally use (AX and DX).

The +0, +1, and +2 Options (Processor Selection)

The +0, +1, and +2 options define the type of processor for which the compiler is to generate code:

option	*processor*
+0	8086/8088
+1	80186
+2	80286

If these options aren't specified, the *cc* compiler that is supplied with the Developer and Commercial Aztec C86 Systems will generate code for the 8086/8088 processors.

The +R option (Forget Registers)

The *+R* option causes the compiler to forget the contents of registers between statements. If the +R option is not used, the compiler will track contents of registers throughout each user function, attempting to minimize register reloads.

The +U Option (Globals to Externs)

The *+U* option converts *global* variables into *externs*. For example, if a program is compiled with the +U option, *int i* outside any function becomes *extern int i*. This option is useful when compiling modules that will be linked with the MS-DOS/PC-DOS linker.

The universal way of defining a global integer, i, is to have the statement, *int i*, in one file and the statement, *extern int i* in all other program files in which the variable is used. The *int i* is a "definition" of the variable since it causes space to be reserved in memory for the variable. The *extern* causes no memory to be reserved; it says, "This variable is defined somewhere else but it is going to be used in this file of the program."

When using the Aztec assembler and linker, the only requirement is that a global variable must be defined at least once. So in this example, it is also possible to have *int i* in every file; the "extern" keyword is not extremely significant in this case. Although there may turn out to be more than one global *int i* in the program, memory will be allocated for just one. This is also the behavior under UNIX.

The situation is slightly different when employing the MS-DOS/PC-DOS linker. In this case, a global variable must be defined exactly once. That is, *extern int i* must appear in every declaration except one, which must be an *int i*. This is where the +U option is useful. By specifying it for all but a single source file, you will not have to worry about having too many or not enough externs; the "externs" can be left off entirely since they will be tacked on under the +U option.

A global initialization is immune to the +U option. Hence, *int i = 3;* is unchanged by it. Initializing a global variable to zero will cause it to be ignored by +U. This is one means for forcing a data definition when using this option.

The +a option

The *+a* option tells the compiler that, in the module being compiled, there are no assignments of the form

```
*ptr= ...
```

where *ptr* is a pointer to a named variable. With this information, the compiler can generate better code for the module.

For example, suppose a module contains the declarations

```
int i, *ip;
```

The following assignment is allowed in a module that's compiled with the *+a* option:

```
i = 1;
```

But if *ip* points to the named variable *i*, the following assignment prevents the module from being compiled with the *+a* option:

```
*ip=1;
```

A module can be compiled with the *+a* option and still contain assignments of the form **ptr=...* providing that *ptr* doesn't point to a named variable. For example, if *ptr* points to an element of a statically- or dynamically-allocated array, the assignment of values to the array elements using statements of the form **ptr=...* do not preclude the compilation of the module using the *+a* option.

As implied by the above paragraph, statically-allocated arrays are not considered to be named variables; hence, values can be assigned to their elements using **ptr=...* or *a[i]=...* statements without preventing the compilation of the module with the *+a* option.

Structures, whether statically- or dynamically-allocated, are also not considered to be named variables. Thus, the following statements do not prevent the module from being compiled with the *+a* option:

```
struct xx s, *sp;
sp=&s;
*sp=...;
```

The compilers normally track registers, that is, remember that a register contains a constant or the value of a named variable, so that they don't unneccessarily generate code to load a register with a constant or variable whose value is already in a register. The compilers also assume by default that an assignment of the form **ptr=* may assign a value to a named variable. Because they can't know the

named variable, if any, that such an assignment affects, they must, upon encountering such an assignment, forget the contents of all registers that they thought contained values of named variables.

When a module is compiled with the *+a* option, the compilers assume that assignments of the form **ptr*= don't affect the values of named variables. Thus, they can generate better code, since they need not forget that a register contains the value of a named variable when they encounter a **ptr*=... assignment.

The +m option

When the compiler encounters a statement that requires multiplication of a value by a constant, it will normally either generate code that performs the operation by a sequence of 8086 'shift' and 'add' instructions, or generate an 8086 'multiply' instruction, with the choice depending on the number of instructions in the sequence (if six or fewer instructions, use the sequence, otherwise use the multiply instruction). A 'shift/add' sequence executes faster than the 'multiply' instruction, but requires more code.

The *+m* option causes the compiler to always generate a 'multiply' instruction, thus resulting in code that executes slower but that is smaller.

Like *+m*, the *+c* option also causes the compiler to always generate a 'multiply' instruction when a non floating point value needs to be multiplied by a constant. However, the *+c* option also causes the compiler to generate other code that reduces code size at the expense of increased execution time, including the following: at the beginning of a function, it generates a call to an internal subroutine, rather than in-line code, that performs function initialization; when a switch statement is encountered, it generates a call to an internal subroutine, rather than generating in-line code, that process the switch.

For some programs, such as those that perform lots of operations on arrays of structures, the generation of a 'multiply' instruction than a 'shift/add' sequence to process multiplication of a variable by a constant can dramatically decrease the size of the program, and is worth the increased execution time; the decreased code size caused by the other *+c* selections, on the other hand, are not deemed worth the increased execution time. For such programs, the *+m* option was invented.

2.2.4 8086 options for the non-optimizing compilers

The +F Option (Generate fast code)

The *+F* option for the non-optimizing C86 compiler causes function entry code to be generated in-line. Normally, every compiled C function begins with a call to a routine in *c.lib*. This option replaces this call with the equivalent code.

This results in a small savings in execution speed every time the compiled function is called. If the function is called repeatedly, the savings can add up to a large difference in the execution time of the program. As a side effect, this option will slightly increase the size of the compiled code.

The +J Option (Generate short branches)

The **+J** option for the non-optimizing C86 compiler, which must be used with care, causes the compiler to generate code that is somewhat faster and smaller, by creating short conditional jump instructions rather than long. By default, the compiler generates long conditional jumps.

A short jump can jump to an instruction within approximately 128 bytes of itself, whereas a long jump can jump to any instruction within the code segment.

The 8086 and 8088 don't actually have a long conditional jump instruction, so the compiler simulates one with a 'codemacro' instruction consisting of a short conditional jump, and a long unconditional jump.

This option cannot be used with all programs: it will sometimes create programs that cannot be linked. The reason for this is that the compiler, when it generates a short jump instruction, doesn't know if the destination of the jump is within range of the jump or not.

Short jumps which are out of range can be detected by the Manx assembler (backward jumps only), the linker, or the **sqz** utility.

So, use this option with care: if your program links, all short jumps were within range; otherwise, go back and recompile the unlinkable modules without the *+J* option.

The +U Option (Globals to Externs)

The *+U* option is the same for both the non-optimizing and optimizing Aztec C86 compilers. See its description in the section on options of the optimizing compilers for details.

3. Writing programs

The previous sections of this chapter discussed operational features of Aztec C86; that is, presented information that an operator would use to compile a C program. In this section, we want to present information of interest to those who are actually writing programs to be compiled with Aztec C86.

3.1 Supported language features

Aztec C86 supports the entire C language as defined in *The C Programming Language* by Kernighan and Ritchie. This now includes the bit field data type.

It also supports additional features, as described below.

3.1.1 Preprocessor statements

Aztec C86 supports the following preprocessor statements, all of which begin with a #.

3.1.1.1 Macros

A macro is a symbol that has an associated character string. When the compiler is reading a C source file and encounters a macro name, it replaces the name with its associated string.

Basic definition and use of macros

A macro can be defined in two ways: from within a C source program, using the *#define* preprocessor statement; and from the command that starts the compiler, using the -D option.

The *#define* statement within a C program defines a macro. This statement has the form

```
#define name string
```

where *name* is the name of the macro and *string* is its associated string. When the preprocessor subsequently encounters the string *name* in the source, it replaces it with the associated string, *string*.

For example, the following code defines the macro named MAXFILE. The declaration of *table* then creates an array of 8 integers.

```
#define MAXFILE 8
...
int table[MAXFILE]
```

The compiler option *-D* is used to define a macro in the command line that starts the compiler. This option has the form

```
-Dname=def
```

where *name* is the name of the macro, and *def* is its associated string. When a macro is defined in this way, spaces are not allowed in

-Dname=def; this entire string must be passed to *cc* as a single argument.

The *=def* part is optional; if not given, a string containing the single character "1" is associated with *name*.

For example, the definition of MAXFILE could have been made with the following command:

```
cc -DMAXFILE=8 prog
```

where *prog.c* is the name of the file being compiled. And if MAXFILE was to be assigned a value of 1, the compiler could have been started with either of the following commands:

```
cc -DMAXFILE=1 prog
cc -DMAXFILE prog
```

Macros having parameters

A macro can have named parameters. A macro having parameters is defined within a C program with a statement of the form

```
#define name(parm1,...,parmx) string
```

This statement defines a macro named *name* that has arguments, associating with it the string *string*. When the preprocessor encounters the string *name(arg1,...,argx)* it replaces the string with *string*; in the process, it performs parameter substitution, replacing every occurrence of *parm1* in *string* by *arg1*, and so on.

In the definition, no spaces are allowed between the name of the macro and the (. Spaces are allowed in *string*.

For example, the following code first defines a macro named *abs* that computes the absolute value of its argument. It then uses this macro to compute the absolute value of the expression *a+b*.

```
#define abs(x) ( (x) > 0 ? (x) : -(x) )
...
y = abs(a+b);
```

As with parameterless macros, parameterized macros can also be defined in the command that starts the compiler. The syntax is the same; namely

```
cc -Dname=def prog
```

As with parameterless macros that are defined with the -D option, the entire -Dname=def string must not contain any spaces.

Thus, the *abs* macro that was defined above using the *#define* statement could also be defined when the compiler is started using the statement

```
cc -Dabs(x)=((x)>0?(x):-(x)) prog
```

where *prog.c* is the name of the file to be compiled.

Undefining macros

Macros can be undefined from within a C program, with the statement

```
#undef name
```

where *name* is the name of the macro being undefined.

More features of macros

This section discusses several additional features of the definition and use of macros.

* A macro name can be any valid C name.
* When the compiler finds the beginning of a symbol name in a C source file, it reads the entire name before checking to see if the name is a macro. For example, consider the following code:

```
#define a 3
#define b 4
#define ab 56
int y=ab;
```

This creates the integer variable *y*; it's initialized to 56, not 34.

* Macros can be defined in terms of other macros.

Thus, the following statements

```
#define a y
#define b a
int b;
```

causes an integer variable named *y* to be defined.

* Macro expansion doesn't occur in *#define* or *#undef* statements.

For example, consider the following statements:

```
#define a y
#define b a
#undef a
int b;
```

This example causes an integer variable named *a* to be created. If macro expansion occurred when *b* was being defined, the name of the created variable would have been *y*.

* Macro names are not recognized in character constants or quoted strings during the processing of normal C statements.

For example,

```
#define a 12345
char y[]="a";
```

creates a character array named *y*, initializing it with the string "a", and not with the string "12345".

* Macro names are not recognized in character constants or quoted strings during macro definition.

Thus,

```
#define b 123
#define a "b"
char y[]=a;
```

creates a character array named *y*, initializing it with the string "b", and not to "123".

* Macro parameters are recognized and substituted in a macro's associated string, even when the parameters are in character constants or quoted strings.

For example,

```
#define b(a) "a"
char y[] = b(123);
```

creates a character array named *y*, initializing it to "123".

* A macro must not be defined when an attempt is made to define it in a #define statement.

For example, the following statements generate an error:

```
#define a b
#define a c
```

while these don't:

```
#define a b
#undef a
#define a c
```

3.1.1.2 Conditional compilation statements

Aztec C86 supports several preprocesssor statements that makes the compilation of blocks of statements conditional.

The simplest use of the conditional compilation statements is to begin the block with one of the *#if...* statements and end it with the *#endif* statement. For example, in the following code the block of statements within the *#ifdef* and *#endif* statements will be compiled if and only if the symbol DEBUG has been defined:

```
#ifdef DEBUG
 /* the block goes here */
 /* it's compiled only if DEBUG is defined */
#endif
```

You can also have one or another block be compiled, depending on specified conditions, using the *#else* statement:

```
#ifdef LARGE
  /* this block compiled only if LARGE is defined */
#else
  /* this block compiled only if LARGE is not defined */
#endif
```

Those are the two basic conditional compilation constructs. These constructs can be nested, in which case an *#else* will pair up with the nearest preceding *#if*....

The following paragraphs define the different forms of *#if*... statements.

#ifdef *name*

This statement causes the block which follows to be compiled only if the symbol *name* is defined. The definition could have been made using the *#define* statement or using the compiler's *-D* option. If *name* was defined, and then later undefined by *#undef*, the block won't be compiled.

#ifndef *name*

This statement causes the block of statements that follows to be compiled if the symbol *name* is *not* defined. The definition of *name* that would cause the block to not be compiled could have been made using the *#define* statement or using the compiler's *-D* option. If the symbol was defined, and then later undefined by *#undef*, the block will be compiled.

#if *expression*

This statement causes the block of statements which follows to be compiled if and only if the *expression* evaluates to non-zero. *expression* must be built from constant integer values. All binary non-assignment C operators, the ?: operator, the unary operators -, !, and ~ are allowed in *expression.* Their precedence is the same as for normal C statements.

For example, the following code will be compiled only if the symbol MAXFILES is defined and has a value greater than 5:

```
#if MAXFILES > 5
/* code to be compiled if MAXFILES > 5 */
...
#endif
```

3.1.1.3 More preprocessor statements

In addition to statements for defining and undefining macros and for making the compilation of statements conditional, the preprocessor supports several other statements. These statements are discussed in this section.

#include <*filename*>

#include "filename"

Causes the contents of *filename* to be read and compiled. For more information on this statement and on the places that the compiler searches for the file, see the description on include files in section 1 of this chapter.

#line *line_number "filename"*

Causes the compiler to think that the line number of the next line to be compiled is *line_number*, and that the name of the file being compiled is *filename*. If "filename" is not given, the current file name is unchanged.

#asm and #endasm

Aztec C86 allows C programs to contain in-line assembly language source. The assembly language code begins and ends with the preprocessor directives *#asm* and *#endasm*, respectively.

When the compiler encounters a *#asm* statement, it copies lines from the C source file to the assembly language file that it's generating, until it finds a *#endasm* statement. The *#asm* and *#endasm* statements are not copied.

While the compiler is copying assembly language source, it doesn't try to process or interpret the lines that it reads. In particular, it won't perform macro substitution.

A program that uses *#asm ...#endasm* must avoid the following placing in-line assembly code immediately following an *if* block; that is, it should avoid the following code:

```
if (...){
   ...
}
#asm
  ...
 #endasm
  ...
```

The code generated by the compiler will test the condition and if false branch to the statement following the *#endasm* instead of to the beginning of the assembly language code. To have the compiler generate code that will branch to the beginning of the assembly language code, you must include a null statement between the end of the *if* block and the *asm* statement:

```
if (...){
   ...
}
;
#asm
  ...
 #endasm
  ...
```

3.1.2 More features

In addition to the preprocessor statements described above, Aztec C86 supports several language features that aren't described in the K & R text.

Structure assignment

Aztec C86 supports structure assignment. With this feature, a program can cause one structure to be copied into another using the assignment operator.

For example, if *s1* and *s2* are structures of the same type, you can say:

```
s1 = s2;
```

thus causing the contents of structure s1 to be copied into structure s2.

Unlike other operators, the assignment operator doesn't have a value when it's used to copy a structure. Thus, you can't say things like "a = b = c", or "(a=b).fld" when a, b, and c are structures.

Line continuation

If the compiler finds a source line whose last character is a backslash, \, it will consider the following line to be part of the current line, without the backslash. For example, the following statements define a character array containing the string "abcdef":

```
char array[]="ab\
cd\
ef";
```

The *void* data type

Functions that don't return a value can be declared to return a *void*. This provides a safety check on the use of such functions: if a *void* function attempts to return a value, or if a function tries to use the value returned by a *void* function, the compiler will generate an error message.

Variables can be declared to point to a *void*, and functions can be declared as returning a pointer to a *void*.

Unlike other pointers, a pointer to a *void* can be assigned to a pointer to any type of object, and vice versa. For other types of pointers, the compiler will generate a warning message if an attempt is made to assign one pointer to another, when the types of objects pointed at by the two pointers differ.

That is, the compiler will generate a warning message for the assignment statement in the following program:

```
main()
{
    char *cp;
    int *ip;
    ip = cp;
}
```

The compiler won't complain about the following program:

```
main()
{
    char *cp;
    void *getbuf();
    cp = getbuf();
}
```

Special symbols

Aztec C86 supports the following symbols:

___FILE___	Name of the file being compiled. This is a character string.
___LINE___	Number of the line currently being compiled. This is an integer.
___FUNC___	Name of the function currently being compiled. This is a character string.

In case you can't tell, these symbols begin and end with two underscore characters.

For example,

```
printf("file= %s\n", ___FILE___);
printf("line= %d\n", ___LINE___);
printf("func=%s\n", ___FUNC___);
```

3.1.3 Special features

The following features are supported by Aztec C86, but not by any of the UNIX C compilers.

String merging

The compiler will merge adjacent character strings. For example,

```
printf("file=" ___FILE___ " line= %d func= " ___FUNC___,
    ___LINE___);
```

Long names

Symbol names are significant to 31 characters. This includes external symbols, which are significant to 31 characters throughout assembly and linkage.

Reserved words

const, *signed*, and *volatile* are reserved keywords, and must not be used as symbol names in your programs.

Global variables

Aztec C supports the rule of the standard C language regarding global variables that are to be accessed by several modules. This rule requires that in the modules that want to access such a variable, exactly one module declare it without the *extern* keyword and all others declare it with the *extern* keyword.

Previous versions of Aztec C did not strictly enforce this rule when the Aztec linker *ln* was used to link programs. In these versions, the following modified version of the rule was enforced:

* multiple modules could declare the same variable, with the *extern* keyword being optional;
* when several modules declared a variable without using the *extern* keyword, the amount of space reserved for the variable was set to the largest size specified by the various declarations;
* when one module declared a variable using the *extern* keyword, at least one other module must declare the variable without using the *extern* keyword;
* at most one module could specify an initial value for a global variable;
* when a module specified an initial value for a global variable, the amount of storage reserved for the variable was set to the amount specified in the declaration that specified an initial

value, regardless of the amounts specified in the other declarations.

In order both to enforce the standard C rule regarding global variables and to provide compatibility with previous versions of Aztec C, the current Aztec linker will generate code consistent with the previous versions, but will by default generate a "multiply defined symbol" message when multiple modules are found that declare a global variable without the *extern* keyword. The *-M* linker option can be used to cause the linker to treat global variables just as they were in previous versions of Aztec C; in this case, the "multiply defined symbol" message won't occur when several modules declare the same variable without the *extern* keyword, as long as no more than one specifies an initial value for the variable. If multiple modules declare an initial value for the same variable this message will be issued, regardless of the use of the *-M* option.

Both previous and the current versions of Aztec C prevent a global symbol from being both a variable name and a function name. When such a situation arises, the linker will issue the "multiply defined symbol" message, regardless of the use of the *-M* option.

If you have programs whose modules follow the modified version of the rule regarding global variables, and you either want to link the modules using the Aztec linker without having to specify the -M linker option and without having the "multiply defined symbols" message appear, or you want to link the modules using the PCDOS/MSDOS linker, the compiler's *-U* option can be useful. When a module is compiled with this option, all the declarations of global variables that don't specify an initial value are implicitly turned into *extern* declarations. Thus, you can place the declarations of a program's global but uninitialized variables into one file, place *#include* statements for that file in the modules that need those variables, and compile one of the modules without the *-U* option, and the others with it.

There are three assembly language directives that create globally-accessible variables: *public*, which causes a variable that is defined in the module using a *db*, *dw*, or *dd* directive to be made globally-accessible; *global*, which both creates a variable in the uninitialized data area and makes it globally accessible; and *extrn*, which permits a module to access a variable that is defined in another module using *public* or *global*. When the Aztec compiler encounters a declaration of a variable outside a function, it generates a *global*, *public*, or *extrn* directive for the variable, depending on the declaration and on whether the compiler was started with the +U option:

* A *global* directive is generated if the declaration doesn't specify an initial value for the variable, and the declaration doesn't specify the *extern* keyword, and the +U option isn't

used.

* A *public* directive is generated if the declaration specifies an initial value for the variable.
* An *extrn* directive is generated if the declaration declares the variable to be an *extern*, or if the +U option is used.

For a discussion of global variables in assembly language terms, see the discussion of globally-accessible variables in the Programmer Information section of the Assembler chapter.

3.2 Data formats

3.2.1 char

Variables of type *char* are one byte long, and can be signed or unsigned. By default, a *char* variable is signed.

When a signed char variable is used in an expression, it's converted to a 16-bit integer by propagating the most significant bit. Thus, a char variable whose value is between 128 and 255 will appear to be a negative number if used in an expression.

When an unsigned char variable is used in an expression, it's converted to a 16-bit integer in the range 0 to 255.

A character in a *char* is in ASCII format.

3.2.2 pointer

Pointer variables are either two or four bytes long, depending on the memory model that the program is using.

Function pointers are two bytes long if the program uses the 'small code' memory model option, and four if it uses 'large code'.

Pointers to data items are two bytes long if the proram uses the 'small data' memory model option, and four if it uses 'large data'.

3.2.3 int, short

Variables of type *short* and *int* are two bytes long, and can be signed or unsigned.

A negative value is stored in two's complement format. A -2 stored at location 100 would look like:

location	*contents in hex*
100	FE
101	FF

3.2.4 long

Variables of type *long* occupy four bytes, and can be signed or unsigned.

Negative values are stored in two's complement representation. Longs are stored sequentially with the least significant byte stored at the lowest memory address and the most significant byte at the highest memory address.

3.2.5 float and double

Variables of type *float* are represented in 32 bits, and those of type *double* are represented in 64 bits. They are in standard 8087 format.

3.3 Floating Point Exceptions

Floating point operations are performed either by an 8087 co-processor or by software, depending on the version of *m.lib* with which a program is linked.

If software routines perform the calculations, the routines check for overflow, underflow, and division by zero. When the software floating point functions return to the caller, the global integer *flterr* indicates whether an exception has occurred, as follows:

flterr	*value returned*	*meaning*
0	computed value	no error has occurred
1	+/- TINY_VAL	underflow
2	+/- HUGE_VAL	overflow
3	+/- HUGE_VAL	division by zero

The symbols *HUGE_VAL* and *TINY_VAL* are defined in the file *math.h*.

When an 8087 performs floating point calculations, floating point exceptions are not detected, and *flterr* is not used.

3.4 Writing machine-independent code

The Aztec family of C compilers are almost entirely compatible. The degree of compatibility of the Aztec C compilers with v7 C, system 3 C, system 5 C, and XENIX C is also extremely high. There are, however, some differences. The following paragraphs discuss things you should be aware of when writing C programs that will run in a variety of environments.

If you want to write C programs that will run on different machines, don't use bit fields or enumerated data types, and don't pass structures between functions. Some compilers support these features, and some don't.

3.4.1 Compatibility Between Aztec Products

Within releases, code can be easily moved from one implementation of Aztec C to another. Where release numbers differ (i.e. 1.06 and 2.0) code is upward compatible, but some changes may be needed to move code down to a lower numbered release. The downward compatibility problems can be eliminated by not using new

features of the higher numbered releases.

3.4.2 Sign Extension For Character Variables

If the declaration of a *char* variable doesn't specify whether the variable is signed or unsigned, the code generated for some machines assumes that the variable is signed and others that it's unsigned. For example, none of the 8 bit implementations of Aztec C sign extend characters used in arithmetic computations, whereas all 16 bit implementations do sign extend characters. This incompatibility can be corrected by declaring characters used in arithmetic computations as unsigned, or by AND'ing characters used in arithmetic expressions with 255 (0xff). For instance:

```
char a=129;
int b;
b = (a & 0xff) * 21;
```

3.4.3 The MPU... symbols

To simplify the task of writing programs that must have some system dependent code, each of the Aztec C compilers defines a symbol which identifies the machine on which the compiler-generated code will run. These symbols, and their corresponding processors, are:

symbol	*processor*
MPU68000	68000
MPU8086	8086/8088
MPU80186	80186/80286
MPU6502	6502
MPU8080	8080
MPUZ80	Z80

Only one of these symbols will be defined for a particular compiler.

For example, the following program fragment contains several machine-dependent blocks of code. When the program is compiled for execution on a particular processor, just one of these blocks will be compiled: the one containing code for that processor.

```
#ifdef MPU68000
   /* 68000 code */
#else
#ifdef MPU8086
   /* 8086 code */
#else
#ifdef MPU8080
   /* 8080 code */
#endif
#endif
#endif
```

3.5 Using long pointers

A pointer is either two or four bytes long: a two-byte pointer is called a 'short pointer', and a four-byte pointer is called a 'long pointer'. A program's function pointers will be short or long, depending on whether the program's modules use 'small code' or 'large code'. Similarly, a module's pointers to data objects will be short or long, depending on whether the module uses 'small data' or 'large data'.

There are several things of which you should be aware when using long pointers:

* You must explicitly specify the passing of a long pointer between functions, because of the difference in size of *int*s and long pointers.
* Normally, you can use long pointers in expressions just as you would a short pointer. However, if you create unusual data objects, or access data objects in unusual ways, your program may not behave as you expect.

These topics are discussed in the following sections.

3.5.1 Passing pointers between functions

This section presents rules that a program should follow when passing pointers between functions. These rules should be followed by all programs, whether they use short or long pointers; that way, if a program uses a small memory model, you can easily convert it to a large memory model.

*** Declare functions that return pointers**

If a function returns a pointer, a module that calls the function and the function itself should say so. Otherwise, the compiler will assume that the function returns an *int*, which is a two-byte value. If the called function returns a small pointer, the resulting program will work, since small pointers are also two bytes long. But if the called function returns a long pointer, the program won't work, since long pointers are four bytes long.

For example, the following code correctly specifies that the function *f()* returns a *char* pointer:

```
char *cp, *f();
 ...
cp = f();
*cp = 'a';
```

If the declaration *char *f()* was omitted, the program would work if *f()* returned a short pointer, but if it returned a long pointer the assignment *cp=f()* will set the segment component of *cp* to an incorrect value (to be specific, it sets it to a 16-bit value generated by propagating the most significant bit of the offset returned by *f()*). The

statement **cp='a'* then will set 'a' somewhere within the operating system, and the operating system will probably crash mysteriously at some later time.

*** Declare function arguments that are pointers**

If a pointer is passed as an argument to a function, the function should say so.

If it doesn't, the compiler will assume that the argument is a two-byte *int*. This assumption will do no harm, and the program will work, if the function is passed a short pointer, but if it's passed a long pointer, the program won't work.

For example, consider the following function, *f()*, which is passed a character pointer and an integer.

```
f(cp,i)
char *cp; int i;
{
    int a=i;
    x(cp,5);
    ...
}
```

Suppose that the declaration *char *cp* was missing. If the module containing *f()* used short data pointers, the function would behave correctly. But if it used long data pointers, the assignment *a=i* wouldn't work, because the compiler-generated code's idea of the location of *i* would be incorrect. And the function call *x(cp,5)* would pass only the offset part of *cp*, which would result in *x()* not being able to access whatever *cp* pointed at, and in *x()* not being able to correctly access its second argument.

*** Declare constant pointers to functions**

If a constant pointer is passed to a function, the caller should cast the constant to be a pointer. Otherwise, as usual, the call will be correctly done if a short pointer is passed, but not if a long pointer is passed.

For example, a null pointer should be passed to the function *f()* as follows:

```
f((char *) 0);
```

If it was passed using the statement *f(0)*, the call would pass only two null bytes instead of four. This would be all right if a short pointer was to be passed, but not if a long pointer was needed.

Within *stdio.h* is the definition

```
#define NULL (void *) 0
```

You should not use *NULL* to pass a null function pointer, since *NULL*

is a null pointer to a data object. If you do, and the program uses 'large data' and 'small code' or 'small data' and 'large code', an incorrect number of bytes will be passed. The following statement creates the symbol NULLFP that can be used as a null function pointer:

```
#define NULLFP (int (*)()) 0
```

3.5.2 Expressions involving long pointers to data objects

There are several facts about a 'large data' program that allows the compiler to give the program special characteristics that make it smaller and faster than it would otherwise be. These facts are:

* Long pointers are in segment/offset form, with the most significant word of a long pointer containing the starting paragraph number of a segment that contains the referenced object, and the other word containing the offset of the object from the beginning of the segment.
* The maximum size of a data object, such as an *int*, array, structure, union, or a buffer that is allocated by one of the Aztec functions, is 64K bytes. This is also the maximum size of a physical segment.
* When a call is made to one of the Aztec functions to dynamically allocate a buffer, the function returns the long pointer in 'canonical form', in which the offset component is between 0 and 15.
* Any field within a standard data object can be accessed by manipulating just the offset component of the object's base address.

The special characteristics that the compiler gives to a 'large data' program because of these facts are described below.

*** Long pointer arithmetic doesn't affect the segment number**

When a long pointer is used in an expression, the value of which is another pointer, the resultant pointer points at an object that is in the same physical segment as the pointer that's in the expression.

In other words, when an integer is added or subtracted from a long pointer, the arithmetic is performed using just the offset portion of the original pointer; the segment portion of the resultant pointer is the same as that of the original pointer.

Since a data object can't occupy more than a single physical segment, and since C does not approve of a program's generating the address of one data object from that of another, this characteristic of 'large data' programs should be satisfactory for most programs.

For programs that need data objects that are bigger than 64K bytes, and that need to move a pointer around within the entire object, see below.

*** Subtraction of two long pointers doesn't use their segment numbers**

When two long pointers are subtracted, the compiler assumes that the pointers reference objects that are in the same physical segment, and that the segment components of the two pointers are the same. The compiler then generates code that subtracts just the offset components of the pointers, and not on their segment parts.

Here are some things things relating to the subtraction of long pointers of which you should be aware:

* The number of bytes between locations referenced by two long data pointers can be determined by directly subtracting the pointers (that is, by saying something like *cp1 - cp2*) only if the locations are in the same data object (and it's a standard data object) or if the locations are either both in the program's physical data segment or both in the program's physical stack segment.

  For other cases, you can subtract two long pointers by (1) calling the function *_ptrdiff* or by (2) converting the pointers to absolute addresses, using the *ptrtoabs* function, and then directly subtracting the absolute addresses.

* When two long pointers reference fields in the same dynamically-allocated buffer, you can compare them by directly subtracting them and testing the resultant value. You can't do this if they reference fields in different dynamically-allocated buffers.

* If you allocate a non-standard data object that is bigger than 64K bytes, you can't compute the number of bytes between two arbitrary locations in the object by simply subtracting pointers to the locations.

  See below for an example of a program that accesses fields within a buffer that's bigger than 64K bytes.

*** Pointer comparisons sometimes compare the segment numbers**

When two pointers are compared, the compiler assumes that the pointers reference objects that are in the same physical segment, and that their segment components are the same, if one of the pointers is a constant or is generated by taking the address of a variable. In this case, the compiler-generated code compares just the offset components of the two pointers.

For example, the code generated for the following expression compares just the offsets of the pointers:

```
char *cp;
int i, a[10];
...
if (&a[i] < cp)
...
```

If two pointer variables are compared, the compiler makes no assumptions about the segments in which the referenced objects reside. In this case, the compiler generates code that first compares the segment components and then, when neccessary, compares the offset components.

For example, the code generated for the following expression compares first the segment numbers and then the offset components of the pointers:

```
char *cp1, *cp2;
...
if (cp1 < cp2)
...
```

If the segment component of *cp1* is less than that of *cp2*, the comparison is true. If it's greater than that of *cp2*, the comparison is false. If it's equal, then the value of the comparison depends on the relationship of the offset components.

The code generated by the compiler to compare two pointer variables is suitable for most programs. However, if your program itself manipulates the segment component of a pointer variable, you must be careful when you compare the value of that variable to other pointer variables.

3.5.3 Creating and accessing huge arrays

Aztec C86 provides several functions that allow a program to easily access arrays that contain more than 64K bytes. These are:

function	*purpose*
ptrtoabs	convert long ptr to absolute address
abstoptr	convert absolute address to long ptr
_ptradd	add a long value to a long pointer
_ptrdiff	subtract two long pointers

An absolute address is a 20-bit value that uniquely defines a location in memory. Thus, one way to use these functions to access the elements of a huge array is to keep the address of the element being accessed in absolute format, converting it to pointer format only when necessary. For example, the following program, which uses the large data memory model, dynamically allocates a 100K-byte array and then goes into a loop, calling the function *process* to process each word of the array.

```
void *abstoptr(),*sbrk();
long ptrtoabs();
long bv, lv, ev;
unsigned seg, off;
main()
{
    bv=ptrtoabs(sbrk(0)); /* get ptr to array */
    if ((brk(abstoptr(bv+100000))) /* allocate array */
        exit(3);   /* exit if array can't be allocated */
    ev = bv+100000; /* end of array */
    for (lv=bv; lv<ev; lv +=2)
        process(abstoptr(lv));
}
```

To allocate a 100K-byte buffer, the program has to be somewhat devious, since the normal buffer-allocation functions can't allocate buffers that are bigger than 64K bytes. It first calls *sbrk* to get the pointer to the current top of allocated heap space. It converts this pointer to a 20-bit absolute address, adds 100,000 to this address to get the ending address of the 100K-byte buffer, converts the ending address to a segment/offset pointer, and calls *brk* to set the top of allocated heap space to this address.

The program could also have used the *__ptradd* function instead of *abstoptr* and *ptrtoabs*, but this would have been a little less efficient, since the addition of a value to a long pointer takes longer than the addition of the value to a long int.

4. Error checking

Compiler errors come in two varieties-- fatal and not fatal. Fatal errors cause the compiler to make a final statement and stop. Running out of memory and finding no input are examples of fatal errors. Both kinds of errors are described in the *Errors* chapter. The non-fatal sort are introduced below.

The compiler will report any errors it finds in the source file. It will first print out a line of code, followed by a line containing the up-arrow (caret) character. The up-arrow in this line indicates where the compiler was in the source line when it detected the error. The compiler will then display a line containing the following:

* The name of the source file containing the line;
* The number of the line within the file;
* An error code;
* A message describing the error;
* The symbol which caused the error, when appropriate.

The error codes are defined and described in the *Errors* chapter.

The compiler writes error messages to its standard output. Thus, error messages normally go to the console, but they can be associated with another device or file by redirecting standard output in the usual manner. For example,

cc prog	errors sent to the console
cc prog >outerr	errors sent to the file *outerr*

The compiler normally pauses after every fifth error, and sends a message to its standard output asking you want to continue. The compiler will continue only if you enter a line beginning with the character 'y'. If you don't want the compiler to pause in this manner, (if, for example, the compiler's standard output has been redirected to a file) specify the *-B* option when you start the compiler.

The compiler is not always able to give a precise description of an error. Usually, it must proceed to the next item in the file to ascertain that an error was encountered. Once an error is found, it is not obvious how to interpret the subsequent code, since the compiler cannot second-guess the programmer's intentions. This may cause it to flag perfectly good syntax as an error.

If errors arise at compile time, it is a general rule of thumb that the very first error should be corrected first. This may clear up some of the errors which follow.

The best way to attack an error is first to look up the meaning of the error code in the back of this manual. Some hints are given there as to what the problem might be. And you will find it easier to understand the error and the message if you know why the compiler produced that particular code. The error codes indicate what the

compiler was doing when the error was found.

THE ASSEMBLER

Chapter Contents

The Assembler ..... as
1. Operating Instructions 5
1.1 The Source File 5
1.2 The Object Code File 6
1.3 The Listing File 6
1.4 Searching for 'include' Files 6
2. Assembler Options 9
3. Programmer Information 10
3.1 Syntax 10
3.2 Symbols 11
3.3 Segmentation 13
3.3.1 The SEGMENT and ENDS Directives 13
3.3.2 Multiple Definitions for a Segment 14
3.3.3 Nested Segments 14
3.3.4 The Default Segment 15
3.3.5 The ASSUME Directive 15
3.3.6 Using the Uninitialized Data Segment 16
3.4 Globally-accessible Symbols 16
3.4.1 The PUBLIC Directive 16
3.4.2 The GLOBAL Directive 17
3.4.3 The EXTRN Directive 17
3.4.4 Interactions of GLOBAL, PUBLIC, and EXTRN 18
3.5 Operands and Expressions 19
3.5.1 Registers 19
3.5.2 Immediate Operands 19
3.5.3 Memory Operands 20
3.5.4 Operand Expressions 23
3.5.5 The Arithmetic Operators 23
HIGH and LOW 23
Addition and Subtraction 23
Multiplication and Division 23
The Shift Operators 24
The Relational Operators 24
The Logical Operators 24
3.5.6 Attribute-overriding Operators 25
Segment Override 25
PTR 25
SHORT 27
3.5.7 Attribute-value Operators 27
THIS 27
SEG 28

OFFSET 28
TYPE 29
LENGTH 29
SIZE 29
3.5.8 Operator Precedence 30
3.6 Instructions 30
3.7 Directives 31
ASSUME 31
BSS 31
DB, DW, and DD 32
END 34
EQU 35
= 36
EVEN 36
EXTRN 37
GLOBAL 37
GROUP 37
INCLUDE 37
LABEL 38
LARGECODE 38
MOD186 39
NAME 39
ORG 39
PROC and ENDP 39
PUBLIC 42
RECORD 42
SEGMENT 44
3.8 Macro Directives 44
3.8.1 Local Symbols 46
3.8.2 Concatenating Parameters to Text 47
3.8.3 Concatenating Parameters to Parameters 48
3.8.4 Parameter Substitution in Quoted Strings 49
3.8.5 Passing a Symbol's Value to a Macro 50
3.8.6 Passing Comma-containing Parameters to a Macro 50
3.8.7 Nesting Macros 51
3.8.8 Repeatedly Assembling a Block of Statements 53
3.8.9 Summary of the Macro Directives 56
ENDM 56
EXITM 56
IRP 56
IRPC 56
LOCAL 56
MACRO 56
PURGE 57
REPT 57
3.9 Conditional Directives 57
IF 58
IFE 59

IF1 59
IF2 59
IFDEF 59
IFNDEF 59
IFB 59
IFNB 59
IFIDN 60
IFDIF 60
ELSE 60
ENDIF 60
3.10 Codemacros 60
3.10.1 Specifiers 62
3.10.2 Modifiers 63
3.10.3 Range Specifiers 63
3.10.4 The Codemacro Directives 64
SEGFIX 64
NOSEGFIX 65
MODRM 65
RELB 66
RELW 66
DB, DW, and DD 66
User-defined Record Directives 67
3.10.5 The Dotshift operator 67
3.10.6 The PROCLEN Symbol 68
3.10.7 Matching Codemacros to Instructions 69

The Assembler

This chapter describes the Manx AS assembler. It has three sections; the first describes how to operate the assembler, the second describes the assembler's options, and the third contains information of interest to those writing assembly language programs.

1. Operating Instructions

The assembler is activated by entering on the command line:

as [-options] filename.asm

where [-options] specify optional parameters and *filename.asm* is the name of the file to be assembled.

The assembler reads assembly language source statements from the input file, translates them to relocatable object code, and writes the result to another file. The assembler can optionally write a listing to a third file.

The following paragraphs describe the input and output files and the assembler options.

1.1 The Source File

The source file name can either specify the disk drive containing the file or not. If it's not specified, the assembler assumes the file is on the default drive.

1.1.1 Source files on MSDOS and PCDOS

On MSDOS and PCDOS, the source file name can optionally specify the directory containing the file. By default, it's assumed to be the current directory on the specified drive. For example, for the following command the assembler looks for *filename.asm* on drive a:, directory *assem**src*:*

as a:\assem\src\filename.asm

and for the following, the assembler looks for *filename.asm* on the current directory of the default drive:

as filename.asm

1.1.2 Source files on CP/M-86

On CP/M-86, the source file name can optionally specify the user area containing the file. It defaults to the current user area on the default drive. The format of a CP/M-86 filename is defined in the compiler chapter. For example, with the following command the

assembler will look for *subs.asm* on user 9, drive d:

```
as 9/d:subs.asm
```

The user number defaults to the current user and the drive defaults to the default drive.

The -ZAP option causes the assembler to delete the source file when it finishes. This option is used by the compiler, when it creates a temporary file containing assembler source and then starts the assembler.

1.2 The Object Code File

The name of the file to which the compiler writes object code to the file specified by the -O option; if this option isn't used, the assembler chooses the name and location of the object file.

When the -O option isn't used, the object file is created on the same drive and directory (for MSDOS or PCDOS) or user area (for CP/M-86) as the source file. The object file name is the same as the source file name, with the extension changed to *.o*.

When the -O option is used, the object file name follows the -O, with spaces between the -O and the file name. The file name can specify the drive and/or the directory (for MSDOS and PCDOS) or user area (for CP/M-86). For example, the following will assemble *subs.asm* and send the object code to *subs.o86*:

```
as -o subs.o86 subs.asm
```

1.3 The listing file

The -L option causes the assembler to create a file containing a listing of the program being assembled. The file is on the same drive and directory (for MSDOS and PCDOS) or user area (for CP/M-86) as the object code file. It's name is the same as that of the object code file, with the extension changed to *.lst*.

1.4 Searching for *include* files

You can make the assembler search for *include* files in a sequence of areas, thus allowing source files and *include* files to be contained in different areas. For DOS, an 'area' is a directory on a drive; for CP/M-86, it's a user area on a drive.

Areas can be specified with the -I assembler option, and, on MSDOS and PCDOS, with the INCLUDE environment variable. The assembler itself also selects a few areas to search. The maximum number of searched areas is eight.

If the file name in the *include* directive specifies a drive id, user area, or path, only the single area specified in the statement is searched.

1.4.1 The -I option.

A -I option defines a single area to be searched. The area descriptor follows the -I, with no intervening blanks.

1.4.1.1 The -I option on MSDOS and PCDOS

On MSDOS and PCDOS, the -I option looks just like you'd expect:

```
-Ib:\incfiles
```

defines the directory *incfiles* on drive b:.

1.4.1.2 The -I option on CP/M-86

On CP/M-86, the area descriptor following the -I consists of (1) an optional user number followed by a slash, and (2) an optional drive identifier. For example, the following defines user area 5 on drive c:

```
-I5/c:
```

The user number is optional, and defaults to the current user number:

```
-Id:
```

defines the current user area on the d: drive. The drive id is also optional, and defaults to the default drive:

```
-I4/
```

defines user area 4 on the default drive.

1.4.2 The INCLUDE environment variable.

On MSDOS and PCDOS, the INCLUDE environment variable also defines directories to be searched for include files. This variable has the same format as the PATH environment variable. That is, something like the following, which defines three areas to be searched:

```
set INCLUDE=b:\inc1;c:\cc\inc2;a:
```

1.4.3 The search order for include files

1.4.3.1 The search order on MSDOS and PCDOS.

On MSDOS and PCDOS, directories are searched in the following order:

1. The currrent directory on the default drive is searched.
2. The directories defined in -I options are searched, in the order listed on the command line.
3. The directories defined in the INCLUDE environment variable are searched, in the order listed.

1.4.3.2 The search order on CP/M-86.

On CP/M-86, user areas are searched in the following order:

1. The current user area on the default drive is searched.
2. The directories specified in -I options are searched, in the order listed on the command line.
3. If the current user number isn't zero, user area 0 on the default drive is searched.
4. If the default drive isn't A:, and if the A: drive is logged in, that is, has been accessed, user area 0 on the A: drive is searched.

2. Assembler Options

The assembler supports the following options:

-O objfile Send object code to *objfile.*

-ZAP Delete the source file after assembling it.

-186 Enable generation of code for 80186-specific instructions.

-Sn Make 'n' squeeze passes through the file, converting long branch and jump instructions to short. If this option isn't used, the assembler makes just two passes through the file, and doesn't squeeze the code.

-I Defines an area to be searched for files specified in a #include statement. For more information, see the Operating Instructions section of the Assembler chapter, above.

-L Send a program listing to a file. All statements in a macro expansion that actually generate code are listed. The name of the file is derived from that of the file to which the object code is sent by changing the extension to '.lst'.

-LA Send a listing to a file. All statements in a macro expansion are listed, including those that aren't actually assembled due to their inclusion in a conditional block whose condition is false. The name of the file is derived in the same way as for the -L option.

-LS Send a listing to a file. The statements in a macro expansion aren't listed. The name of the listing file is derived in the same way as for the -L option.

-C Same as -L, except the listing is sent to the console.

-CA Same as -LA, except the listing is sent to the console.

-CS Same as -LS, except the listing is sent to the console.

-X Don't list false conditionals. If this option is specified and if the assembler is generating a listing, it won't list statements whose assembly is conditional, if their condition is false.

-Dsym[=const]
Creates the symbol *sym*, assigning it the constant *const.* If =*const* isn't specified, *sym* is assigned the value 1.

3. Programmer Information

as is a relocating assembler: it translates an assembly language program into relocatable object code, which must then be converted into executable machine code by a linker. You can use either the Aztec *ln* linker or, after feeding the object module through the Aztec *obj* utility, the standard PC-DOS/MS-DOS *link* linker.

as supports many of the features of the PC-DOS/MS-DOS *masm* assembler, including all the standard 8086 and 80186 instructions, macros, conditional assembly, global symbols, and many of the *masm* directives.

as allows a program to be partioned into segments in a manner that is similar yet different from the segmentation supported by *masm*.

as also supports codemacros in a manner similar to that supported by Intel's own 8086 assembler. This feature allows you to create your own assembly language instructions.

The remainder of this section discusses the following topics:

* *Syntax*, which describes the syntax of assembly language statements.
* *Symbol*, which describes the attributes of symbol names.
* *Segmentation*, which describes how you divide an assembly language program into segments.
* *Global symbols*, which describes how an assembly language module accesses symbols in other modules.
* *Operands*, which describes the operands to instructions and directives, and the operators that can be used to construct operand expressions.
* *Instructions*, which discusses the instructions supported by the assembler.
* *Directives*, which discusses the directives supported by *as*, except for those related to macro definition, conditional assembly, and codemacro definition.
* *Macros*, which describes *as*'s support for macros.
* *Conditional Assembly*, which describes how to partition an assembly language program into blocks whose assembly depends on certain conditions being met.
* *Codemacros*, which describes how to create your own assembly language instructions.

3.1 Syntax

An assembly language program consists of a sequence of statements. Each statement is on a single line, which can contain up to 256 characters. There are two types of statements: instructions, which are translated into machine code, and directives, which pass information to the assembler.

A statement has the form:

```
name   operation   operand, operand    ;comment
```

where:

* *name* is the name of the statement.
* *operation* is the name of the instruction or directive that the assembler is to perform for the statement.
* The *operand* fields are expressions, separated by commas, that the assembler is to perform the operation on.
* *;comment* is a comment, which the assembler ignores, that you use to describe the statement.

A particular statement may not need all the fields described above. For example, a statement can contain just a comment. And the statement

```
ret
```

contains just the operation field: *ret* is the name of the 8086 return instruction.

The fields in a statement can be separated by blanks or tabs, and don't have to begin in specific positions on a line.

Symbol names

A symbol name can be built up from the alphabetic characters A through Z; the numerical digits 0 through 9; and the following special characters: ? @ _ $. The first character in a name must not be a digit.

For symbols that are used as statement names, the assembler distinguishes between upper and lower case characters. For other symbols (instruction and directive names, etc), the assembler doesn't care about the case of the alphabetic characters in the symbol. For example, a statement that contains the 8086 return instruction could be coded in any of the following forms:

```
ret
Ret
RET
```

And the following statements create two distinct symbols as variable names: *Bvar* and *bvar*:

```
Bvar    db      10
bvar    db      10
```

A symbol can contain as many characters as desired. However, only the first 63 are significant.

3.2 Symbols

The *as* assembler has a very small instruction set; in fact, there are fewer instruction mnemonics than there are 8086 machine

instructions. Most instruction mnemonics can generate any of several hardware instructions; the assembler uses attributes of an instruction's operands to decide which hardware instruction to generate.

For example, there are several different hardware instructions for moving data around. There is just one instruction mnemonic for moving data around, *mov*, and the assembler uses the attributes of the operands to a particular *mov* instruction to decide which hardware move instruction to generate.

When a symbol is defined, the assembler will store its name and its attributes. Then, when the symbol is used in an instruction, the assembler will recall the symbol's attributes. There are several operand operators that allow you to obtain or to override the attributes of a symbol. These are discussed in the Operands section of this chapter.

One of the attributes of a symbol is its type. This can specify a constant, which is an absolute number; a variable, which refers to a data item in memory, or a label, which refers to a memory location that can be called or jumped to.

Variables

Another attribute of a variable or label is its *segment*, which is the starting paragraph number of the segment in which the symbol is defined.

A variable or label also has an *offset* attribute, which is the distance in bytes from the symbol to the beginning in memory of the segment in which it is defined.

There are several types of variables. They are:

* *byte* - a one-byte data item.
* *word* - a two-byte data item.
* *dword* - a four-byte data item.

A variable is defined using one of the data definition directives *db*, *dw*, *dd*, *bss*, *global*; or using the *label* directive.

Labels

There are two types of labels:

* *near* - represents a label that will be accessed by a 'near' call or jump instruction. For such an instruction, the instruction and the target label must lie in the same physical code segment. When a 'near' call or jump is made, the contents of the IP register are set to the offset of the label from the beginning of the physical segment containning it, and the CS segment register is unchanged.

* *far* - represents a label that will be accessed by a 'far' call or jump instruction. For such an instruction, the instruction and the target label need not be in the same physical code segment. When a 'far' call or jump is made, both the IP and the CS registers are changed.

A label is defined in the following ways: (1) in the name field of an instruction, followed by a ':'; (2) using a *proc* directive; (3) using a *label* directive.

3.3 Segmentation

as allows a module's code and data to be partitioned into three segments: a code segment, which contains the program's executable code and, optionally, data; an initialized data segment, that contains data but no code, and an uninitialized data segment, which contains uninitialized variables.

Variables in a module's initialized data segment can be defined to have an initial value, if desired. When a program is loaded, initialized variables in this segment will assume their specified values: variables whose initial value depends on where the program is loaded will be set by the loader; other initialized variables will have been preset by the linker. The initial value of uninitialized variables in this segment is indeterminate.

When a linked program begins execution, variables in its uninitialized data segment will automatically be cleared.

When modules are linked together, all the modules' initialized data segments are appended one to another, as are the modules' uninitialized data segments. The two resulting segments will reside in the same physical segment, the maximum size of which is 64K bytes. If the program was linked to have the 'small data' memory model, the two data segments will share the physical segment with the program's stack and heap. If the program was linked to have the 'large data' memory model, the two data segments will have the entire physical segment to themselves; the program will have a separate stack segment, and it will use as much space above the program as needed for its heap.

When modules are linked together, the code segments of the modules that use the 'small code' memory model (that is, that don't contain the *largecode* directive) will be appended one to another into a single physical segment, the maximum size of which is 64K bytes. The code segments of modules that use the 'large code' memory model will each occupy its own physical segment, whose maximum size is 64K bytes.

3.3.1 The SEGMENT and ENDS Directives

The *segment* and *ends* directives surround a sequence of statements and define the segment that is to contain the code and data generated

for the statements. The directives have the form

```
segname  segment  [align-type] [combine-type] ['cname']
         ...
segname  ends
```

segname is the name of the segment into which the surrounded code and data is to be placed. This can be either *codeseg* or *dataseg*, to specify the code segment or initialized data segment, respectively.

The *align-type* operand specifies on what type of boundary in memory the segment will be located. It can have one of the following values:

* *para* - Paragraph alignment. The segment will be on a paragraph boundary; that is, it will begin at a byte whose address is divisible by 16 (ie, an address whose least significant hexadecimal digit is 0). If *align-type* isn't specified, the segment will have *para* alignment.
* *byte* - Byte alignment. The segment can start at any location.
* *word* - Word alignment. The segment must begin at a byte whose address is even. See the *even* directive.
* *page* - Page alignment. The segment must begin at an address whose least significant two hex digits are 00.

The *combine-type* operand is provided for compatibility with other 8086 assemblers, and has no effect on the *as* assembler. If specified, this operand must have one of the following values: *public*, *common*, *stack*, *memory*, *at expr*.

The *'cname'* operand is also provided for compatibility with other 8086 assemblers, and has no effect on the *as* assembler. If specified, this operand must be a character string, surrounded by single or double quotes.

3.3.2 Multiple definitions for a segment

You may open and close a segment using the *segment* and *ends* directives within a module as many times as you want. All parts of such a segment will be joined together by the assembler.

3.3.3 Nested segments

The assembler allows segments to be 'nested'; that is, one segment can be opened and closed using the *segment* and *ends* directives while another is still open. The assembler will separate the code and data for the two segments so that the one won't be imbedded in the other when the program is actually in memory.

For example, the following code nests *dataseg* within *codeseg*:

```
codeseg  segment
         ...                ;begin assembling into codeseg
dataseg  segment
         ...                ;assemble into dataseg
dataseg  ends
         ...                ;continue assembling into codeseg
codeseg  ends
```

The assembler will extract the data defined in *dataseg* so that *dataseg* won't be contained in *codeseg* when the program is loaded into memory.

When a segment is nested within another, the nested segment must be closed before the other segment is closed. For example, the following is an error:

```
codeseg segment
        ...
        dataseg segment
                ...
codeseg ends
        dataseg ends
```

3.3.4 The default segment

If a program contains statements that aren't within an open segment, the generated code will be placed in *codeseg*.

3.3.5 The ASSUME Directive

The *assume* directive identifies to the assembler the segments that are pointed at by segment registers. It has the form:

```
assume seg-reg:segname [,seg-reg:segname ...]
```

The assembler uses this information when it is processing instructions that access memory, and which don't explicitly specify the segment register to be used in the memory access. In such a case, if the segment register that should be used is the same as the segment register that the instruction will use by default, the assembler will just output the code for the instruction. If the desired and default segment registers differ, the assembler will automatically output a prefix byte before the instruction, which will force the instruction to select the proper segment register. If the desired segment isn't pointed at by a segment register, the assembler will display an error message.

The first form of *assume* defines the contents of individual segment registers. The second form tells the assembler not to make any assumptions about the contents of the segment registers.

In the first form, *assume* is passed a list of items, separated by commas, each defining the contents of a particular segment register. An item has the form *seg-reg:segname*, where *seg-reg* is the name of

the segment register; that is, CS, DS, ES, or SS.

segname can be one of the following:

* The name of the segment whose starting paragraph number is in *seg-reg.*
* *seg name*, where *name* is the name of a variable of label that is contained in a logical segment whose starting paragraph number is in *seg-reg.*
* *nothing*, if the assembler is not to make any assumptions about the contents of *seg-reg.*

For example, the *assume* statement in the following program tells the assembler that the logical segment named *codeseg* is pointed at by CS, that segment *dataseg* is pointed at by DS and ES, and that the assembler shouldn't make any assumptions about the contents of SS:

```
        assume cs:codeseg, ds:dataseg, es:dataseg, ss:nothing
dataseg segment         para
dl      dw      ?
dataseg ends
codeseg segment         para
        mov     ax,dl
        ...
codeseg ends
```

Because of the *assume* statement in the above program, the program doesn't have to explicitly specify the segment register to be used in the *mov* instruction. Without the *assume* directive, the *mov* instruction would have had to specify the segment register that it used; that is,

```
mov ax, ds:dl
```

3.3.6 Using the Uninitialized Data Segment

The *bss* and *global* directives create variables in the uninitialized data segment. For more information on these directives, see the Directives section of this chapter.

3.4 Globally-accessible symbols

as creates object modules that can be linked together into an executable program. Each module may define 'global symbols'; that is, labels, variables, and constant symbols that other modules may use.

There are three directives relevant to the creation and use of global symbols: *public*, *global*, and *extrn.*

3.4.1 The PUBLIC Directive

The *public* directive makes symbols that are defined in a module accessible by other modules. The symbols can have been defined in the name field of an instruction, or using the *label* directive, or using

the *equ* directive.

The *public* directive has the form:

```
public  name [,name ...]
```

where the *name* operands are the names of symbols defined in the module that are to be made accessible by other modules.

For example, the following code creates the variables *var1* and *var2* and the label *lbl* and makes them accessible by other modules.

```
        public  var1, var2, lbl
dataseg segment
var1    dw ?
        ...
dataseg ends
codeseg segment
lbl:
        ...
var2    dd      ?
        ...
codeseg ends
```

3.4.2 The GLOBAL Directive

The *global* directive reserves space in the uninitialized data area, creates a variable name that refers to that space, and makes the name accessible by other modules. The directive has the form:

```
global  sym:type,size
```

where the operand defines the attributes of the created variable, as follows:

* *sym* - the name of the variable;
* *type* - its type. This can be *byte*, *word*, or *dword*.
* *size* - the number of bytes to be reserved for the variable.

For example, the following statement creates the globally-accessible variable *gbl*, whose type is *word*:

```
global  gbl:word,10
```

Ten bytes will be reserved for the variable, and it will be located in the uninitialized data segment, unless the overriden by the declaration of *gbl* in other modules' *global* and *public* directives. This overriding is discussed below.

3.4.3 The EXTRN Directive

The *extrn* directive allows a module to access global symbols, which have been defined in other modules using the *public* and/or *global* directives. The directive has one or more comma-separated operands,

each of which defines the attributes of one global symbol. It has the form:

```
extrn name:type [,name:type ...]
```

where *name* is the name of a global symbol and *type* is its type. *type* can be *byte*, *word*, *dword*, *near*, or *far*.

The *extrn* directive must be contained in the segment in which the variables are actually located, or in the *dataseg* segment if they are in the uninitialized data area.

For example, the following code demonstrates how a module can use the *extrn* directive to access the variables and labels *var*, *var1*, *var2*, and *lbl* that are defined using the *public* and *global* directives shown above:

```
dataseg segment
        extrn   var:byte, var1:word, gbl:word
        ...
dataseg ends
codeseg segment
        extrn   lbl:near, var2:dword
codeseg ends
```

3.4.4 Interactions of the GLOBAL, PUBLIC, and EXTRN Directives

A globally-accessible variable can be defined using the *global* directive in some modules, using the *extrn* directive in other modules, and using the *public* directive in at most one module. If the variable is defined using an *extrn* directive in one module, it must be defined using a *global* or *public* directive in at least one other module.

When a variable is defined using *global* directives in one or more modules and is not specified in a *public* directive, an amount of space in the uninitialized data area is reserved for the variable that is equal to the largest size specified for it in the *global* directives. For example, if the variable *var* is defined in different modules using the following declarations, it will have 20 bytes reserved for it when it is linked:

```
global  var:byte,10     ;module a's declaration
global  var:byte,20     ;module b's declaration
global  var:byte,0      ;module c's declaration
```

When a variable is specified in a module's *public* directive, the variable will be located in the segment in which it is defined, regardless of its specification in other modules' *global* directives. In this case, the *global* directives don't have any effect on the amount of space reserved for the variable; the statement in the module containing the *public* directive that actually creates the variable defines its space. For example, if a module contains the declarations

```
dataseg segment
        public  var
var     byte    5 dup (?)
dataseg ends
```

and it is linked with the three modules shown above, which define *var* using *global* directives, then 5 bytes are reserved for *var* in the *dataseg* segment.

In order to alert you to accidental duplication of globally-accessible names in different modules, the Aztec linker will issue a "multiply-defined symbol" message when it encounters a global or public symbol in one module that matches a name in another module, and then proceed to generate code as discussed above. The *-M* linker option will prevent the linker from issuing these messages for *global* and *public* definitions of symbols that obey the rules; for those that don't, it will still generate an error message.

The PCDOS/MSDOS linker *link* does not support the *global* directive; *obj*, the Aztec program that converts object modules from Aztec to PCDOS/MSDOS format so that they can be linked with *link*, translates the definition of a variable using *global* into an equivalent definition using the *public* directive, and a *db*, *dw*, or *dd* directive. These converted directives don't allow the same variable to be defined in different modules using the *public* directive. Thus, if a program is to be linked using the PCDOS/MSDOS linker, a globally-accessible variable must be defined using exactly one *global* or *public* directive in all the modules that are linked together.

3.5 Operands and expressions

3.5.1 Registers

8086 registers are referenced using their standard names: CS, DS, SS, ES, AL, AH, BL, BH, CL, CH, DL, DH, AX, BX, CX, DX, SP, BP, SI, DI

as does not support the 8087 instruction mnemonics, and the standard names for the 8087 registers, *st(0)*, ...*st(7)*, are not reserved symbols in *as*.

3.5.2 Immediate operands (constants)

as allows an instruction operand to be a constant; that is, a number that has no attributes other than its value. The following types of constants are supported:

* *Binary (base 2):* A sequence of 0's and 1's followed by the letter B. For example: 10010110B and 11B.
* *Octal (base 8):* A sequence of digits 0 through 7 followed either by the letter Q or the letter O. For example: 1777Q and 56O.

* *Decimal (base 10):* A sequence of digits 0 through 9, optionally followed by the letter D. For example: 1234 and 1234D
* *Hexadecimal (base 16):* A sequence of digits 0 through 9 and/or letters A through F, followed by the letter H. The sequence must begin with one of the digits 0 through 9. For example: 0FFFFH.
* *ASCII Character:* One or more characters surrounded by single or double quotes. When a quoted character is used as an instruction operand, the character's ASCII value is used. Character strings containing more than two characters are only valid for the *db*, *dw*, and *dd* directives.

3.5.3 Memory operands

An operand in memory is specified using an "address expression". Address expressions are specified using the standard 8086 syntax. This syntax is described in the following paragraphs.

Accessing data in memory

The simplest type of address expression that accesses data is just the name of the variable containing the data, plus or minus a constant. For example,

```
add     dx,count        ;add contents of count to DX
add     abc+4,cx        ;add CX to contents of abc+4
add     abc[4],cx       ;same as the above
```

An address expression can also specify that a data operand resides in memory at an offset computed by adding together any or all of the following:

* An 8- or 16-bit displacement
* The contents of a base register
* The contents of an index register.

If a base or index register is used, its name is surrounded by square brackets.

As an example of a memory operand that doesn't involve a register, the following instruction moves the contents of the word at location *count+6* into AX:

```
mov     ax,count+6
```

If the operand's offset is specified by just a constant, the constant must be surrounded by square brackets. For example, the next instruction moves 0 into AX, while the one after it moves the contents of the word at the beginning of the segment pointed at by the DS segment register into AX:

```
mov   ax,0
mov   ax,[0]
```

The following instructions demonstrate how an offset can be specified using just a register. The first instruction moves the contents of the word pointed at by BX into AX, while the second moves the contents of AX to the word pointed at by SI:

```
mov   ax,[bx]
mov   [si], ax
```

The next instructions use one register and a displacement to define the location of an operand. The first instruction moves into AX the contents of the word whose offset from the beginning of the data segment equals that of the variable *table* plus the contents of register SI. The second and third instructions move the contents of AX to the memory word whose offset from the beginning of the data segment equals that of the variable *data* plus 4 plus the contents of BP.

```
mov   ax,table[si]
mov   data+4[bp],cx
mov   [data+4+bp],cx
```

The following equivalent instructions use an index register, base register, and a displacement to define the location of an operand:

```
mov   ax,table+4[bp][di]
mov   ax,table+4[bp+di]      ;same as the above
mov   ax,[table+4+bp+di]     ;same as the above
add   [bx][si],32
```

Operands to jump and call instructions

A call or jump instruction can specify the address to which control is to be transferred within the instruction. For example,

```
      call    lbl
      ...
lbl:
```

It can also specify a variable which contains the address to which control is to be transferred. For example,

```
xx    dw      sub1
yy    dd      sub2
      ...
      call    xx      ;near call to sub1
      jmp     yy      ;far call to sub2
```

A jump or call instruction can transfer control to a location whose offset if in any 16-bit general-purpose, base, or index register. In this case, the register name isn't surrounded by square brackets. For example,

```
        jmp     ax      ;jump to location whose offset is in AX
```

A call or jump instruction can transfer control to a location whose offset and, optionally, its segment number are in a memory location, where the offset of the memory location is in a base or index register. In this case, the register name is surrounded by square brackets. For example

```
        call    word ptr [si]
        call    dword ptr [bx]
```

In the above example, the first instruction performs a near call to the location whose offset is contained in a word in memory. The offset of this word is contained in SI. The second instruction performs a far call to the location whose offset and segment number is contained in a doubleword in memory. The offset of this double word is contained in BX.

A jump or call instruction can specify that the target address is contained in a memory location whose offset is defined by the sum of a displacement, contents of a base register, and contents of an index register. For example,

```
table   dw      sub1, sub2, ...
        ...
        jmp     table[si]           ;near jump
        jmp     word ptr [bp][di]
```

Which segment register is used?

An instruction can only access a memory operand that is contained in a physical segment whose segment number is in one of the four segment registers. The assembler and the 8086 itself will select the segment register to be used in a memory access, if an instruction doesn't explicitly specify one, using the following rules:

* If the operand contains a variable or label name, the segment register that points to the segment containing the variable or label is used. The *assume* directive defines the contents of the segment registers.
* Otherwise, if the operand uses BP or if the instruction is a stack instruction, the SS segment register is used.
* Otherwise, the DS segment register is used.

An address expression can explicitly specify the segment register that points to the segment containing the expression's operand by preceding the expression with the name of the segment register, followed by a colon. For example, the first instruction below fetches a word from the segment pointed at by SS, while the second fetches the operand from the segment pointed at by DS:

```
mov   ax,[bp]
mov   ax,ds:[bp]
```

3.5.4 Operand Expressions

An expression that is used as an instruction's operand can be created using the operators described in the following paragraphs.

3.5.5 Arithmetic Operators

The HIGH and LOW Operators

high operand
low operand

high and *low* have as their value the most significant and least significant bytes, respectively, of *operand*. *operand* can be an expression having a constant value; in this case, the resulting value is also constant. *operand* can also be a relocatable expression; in this case, the resulting value is also relocatable.

For example, the following *mov* instruction moves the most significant byte of the address of the variable *abc* into AH:

```
abc   dw    ?
      mov   ah,high abc
```

The Addition and Subtraction Operators

Addition: *operand* + *operand*
Subtraction: *operand* - *operand*

When object modules are linked using *as*, any of the operands can be relocatable or constants. When they are linked using the PCDOS/MSDOS linker (after using *obj* to convert them to PCDOS/MSDOS format), the following restrictions apply:

* For addition, at most one of the operands can be relocatable.
* For subtraction, either or both the operands can be relocatable.

The Multiplication and Division Operators

Multiplication: *operand* * *operand*
Division: *operand* / *operand*
Modulo: *operand MOD operand*

These operators may only be used on operands that are constant expressions. The result is always a constant.

The Shift Operators

Shift right: *operand shr count*
Shift left: *operand shl count*

These operators shift *operand* the number of bits specified by *count*. Bits shifted into the operand are set to 0. Both *operand* and *count* must be expressions that evaluate to absolute numbers.

For example, the following instruction moves into AX the constant 0fh, which is derived by shifting 0fah right 4 bits.

```
add     ax,0fah shr 4
```

The Relational Operators

equal:	*op1 eq op2*
not equal:	*op1 ne op2*
less than:	*op1 lt op2*
less than or equal:	*op1 le op2*
greater than:	*op1 gt op2*
greater than or equal:	*op1 ge op2*

The relational operators compare two operands, *op1* and *op2*, returning an 8- or 16-bit value that is all ones if the relationship is true, and all zeroes if it's false.

Both operands must be expressions that evaluate to an absolute number.

For example,

```
count = 5
...
if count lt 5
...
```

The Logical Operators

op1 or op2
op1 xor op2
op1 and op2
not op1

The logical operators may only be used with expressions that evaluate to an absolute number. They return an absolute number.

or performs a logical 'or' of the two operands. For each pair of bits, the resultant bit is 0 if both operand bits are 0, and is 1 otherwise. For example,

10011B or 01010B = 11011B

xor performs an exclusive 'or' on the bits in the two operands. For each pair of bits, the resultant bit is 1 if exactly one of the operand bits is 1, and is 0 otherwise. For example,

10011B xor 01010B = 11001B

and performs a logical 'and' of the bits in the two operands. For each pair of bits, the resultant bit is 1 if both of the operand bits is 1, and is 0 otherwise. For example,

101010B and 111B = 10B

not performs a logical negation of its operand, converting 1's to 0's and 0's to 1's. For example,

not 101001B = 10110B

3.5.6 Attribute-overriding operators

The Segment Override Operator, ':'

segreg:addrexpr

The segment override operator, :, is used to explicitly specify the segment register that is to be used to access a memory operand. If this operator isn't used, the assembler will decide which segment register must be used to access the operand and, if necessary, output a segment-selection prefix for the instruction.

addrexpr is the address expression whose corresponding memory operand is to be accessed, and *segreg* is the name of the segment register to be used for the access.

One common use of this operator is to override the segment register that the hardware by default selects to access an operand. For example, the following instruction will access the memory operand using the DS segment register:

```
mov     ax,[bx]
```

If the operand is contained in a physical segment that is pointed at by ES, the following instruction could be used to access it:

```
mov     ax,es:[bx]
```

In this case, the instruction that is assembled will be preceded by a "segment override prefix" that forces the processor to use ES to access the operand.

The PTR Operator

type ptr expr

The *ptr* operator sets the type of the operand expression *expr* to *type*.

For most of the 8086 instruction mnemonics, there are several hardware instructions. When the assembler encounters an instruction, it uses the types of the instruction's operands to decide which hardware instruction it should generate. In many cases, the assembler can determine the type of an expression from the type of the symbols that are in the expression. For example:

```
wvar    dw      ?
bvar    db      ?
        ...
        inc     wvar    ;increment a two-byte field
        inc     bvar    ;increment a one-byte field
```

The assembler knows that *wvar* is the name of a word field and that *bvar* is the name of a byte field. Thus, for the first *inc* instruction it correctly generates the hardware instruction that increments a word in memory. And for the second it generates the hardware instruction that increments a byte in memory.

There are some operands for which the assembler can't determine the type of the operand. For example, the assembler can't decide whether the operand in the following instruction refers to a byte or word:

```
inc     [bx]
```

In cases like this, the *ptr* operator can be used to explicitly state the type of the operand:

```
inc     word ptr [bx]
inc     byte ptr [bx]
```

The operand of the first instruction above is stated to be a word, and that of the second instruction is stated to be a byte.

The *ptr* operator can also be used to override the assembler's idea of the type of an operand. For example, for the first instruction that follows the assembler generates code that moves the immediate value 10 into AX. And for the second instruction it generates code that moves the contents of the word at offset 10 into the segment pointed at by the DS segment register into AX.

```
mov     ax,10
mov     ax,word ptr 10
```

As another example, the following code accesses separately the two bytes of a word variable:

```
mov     al, byte ptr aword      ;get low order byte
mov     bl, byte ptr aword+1    ;get high order byte
```

The *type* field of the *ptr* operator can have the following values:

* byte
* word
* dword
* near
* far

The SHORT Operator

short lbl

The *short* operator is used within the operand of a *jmp* instruction to specify that a forward-referenced label, *lbl*, is within 127 bytes of the instruction. With this information, the assembler can generate a two-byte instead of a three-byte instruction for the *jmp*.

If despite your claim, the linker finds that *lbl* is not within 127 bytes of the *jmp*, it will report an error.

3.5.7 Attribute-value operators

The THIS Operator

this type

The *this* operator is used within an *equ* statement to create a symbol whose segment and offset components are those of the current segment and the current offset in that segment, respectively, and whose type is *type*.

this is frequently used to create an alternate name and type for a data item. For example:

```
aword  equ   this word
byte1  db    ?
byte2  db    ?
```

Using the *this* operator with the *equ* directive is equivalent to using the *label* directive. The above example could also have been coded using the *lable* directive as:

```
aword  label  word
byte1  db     ?
byte2  db     ?
```

The symbol $ is equivalent to *this near*. For example,

```
xx     equ    $
       ...
       jmp xx
```

The *type* operand to the *this* operator can have the following values:

* byte

* word
* dword
* near
* far

The SEG Operator

seg varlab

The *seg* operator has as its value the beginning paragraph number of the segment in which *name* is contained. This value is relocatable; that is, it isn't known until the program is loaded into memory. *varlab* is the name of a variable or label.

For example, the *strt* variable that follows contains the starting paragraph number of the *dataseg* segment, and the *mov* instruction moves *dataseg*'s starting paragraph number into AX:

```
dataseg segment
dl      dw      ?
strt    dw      seg dl  ;starting para no of dataseg
dataseg ends

codeseg segment
        mov     ax,seg dl       ;get dataseg base
        ...
```

The OFFSET Operator

offset varlab

The *offset* operator has as its value the offset of *varlab*, which is a variable or label, from the beginning of the segment in which *varlab* is contained.

The type of the resulting value is 'relocatable immediate'; this causes the assembler to match an instruction that uses the *offset* operator to an 'immediate', rather than a 'memory reference', version of the hardware instruction.

For example, the first instruction that follows is a memory reference instruction; it moves the contents of the memory location named *table* into bx. The second instruction is an immediate instruction; it moves the offset of the memory location named *table* from the beginning of the segment containing it into bx.

```
mov     bx, table       ;move contents of table into bx
mov     bx,offset table ;move offset of table into bx
```

The TYPE Operator

type varlab

The *type* operator has as its value an integer constant that identifies the type of the operand *varlab*, which is the name of a variable or label. This operator is useful in sequences of code that process the elements of an array or table.

The types of *varlab* and the corresponding values of the *type* operator are:

varlab	*type*
byte	1
word	2
dword	4
near	255
far	254

For an example, see the description of the *length* operator, below.

The LENGTH Operator

length var

The *length* operator returns the number of elements (bytes, words, or dwords) that have been allocated for the variable *var*. This operator is useful in instruction sequences that process the elements of a table or array.

For example, the following code processes the elements of *tbl:*

```
        mov     cx,length tbl   ;get # of elements in tbl
        mov     si,0            ;index into tbl
doone:
        mov     ax,tbl[si]      ;get current tbl element
        ...     ;process it
        add     si,type tbl     ;incr SI to next element
        loop    doone
```

The SIZE Operator

size var

The *size* operator returns the number of bytes allocated for a variable. This value is related to the values of the *length* and *type* operators as follows:

size = length * type

For example

```
wtbl    dw      100 dup (?)
btbl    db      100 dup (?)
        ...
        mov     ax,size wtbl    ;ax=200
        mov     ax,length wtbl  ;ax=100
        mov     ax,size btbl    ;ax=100
        mov     ax,length btbl  ;ax=100
```

3.5.8 Operator Precedence

The expression operators are listed below in decreasing order of precedence. An expression is evaluated from left to right, following the precedence rules. You can use parentheses to specify the order in which an expression should be evaluated.

Highest precedence

1. Square-brackets and the *length* and *size* operators.
2. *ptr*, *offset*, *seg*, *type*, *this*, and segment override (segreg:name)
3. *high* and *low*
4. *, /, *mod*, *shr*, *shl*
5. Unary + and -
6. Binary + and -
7. *eq*, *ne*, *lt*, *le*, *gt*, *ge*
8. *not*
9. *and*
10. *or* and *xor*
11. *short*

Lowest precedence

3.6 Instructions

as supports all the standard 8086, 8088, and 80186 instructions, plus some special instructions. It does not support the 8087 instructions.

as has a feature, *codemacros*, with which you can define, and then invoke, your own instructions. Codemacros are described in another section of this chapter.

Most of the special instructions supported by *as* are conditional branch instructions, whose target location can be anywhere in the current code segment. The standard conditional jump instructions require that the target address be inside a small interval of code centered around the jump instruction.

When a conditional branch instruction is assembled, the equivalent jump instruction will be generated if the target of the branch can be

reached by the jump instruction. Otherwise, the assembler will generate two hardware instructions for the branch: an unconditional jump to the target (which can access any location in the code segment), preceded by a conditional jump around the unconditional jump. This preceding conditional jump tests for a condition that is the opposite of the one specified by the branch instruction.

The special branch instructions and their corresponding jump instructions are:

branch	*jump*
beq	je
bne	jne
blt	jl
ble	jle
bgt	jg
bge	jge
blo	jb
blos	jbe
bhi	ja
bhis	jae

The other special instructions supported by *as* are *nil*, which does nothing and which generates no code; and *xlatb*, which is the same as the standard *xlat* instruction, but which doesn't require an operand, and which assumes that the translate table is in the segment pointed at by the DS segment register (ie, it won't automatically output a segment override prefix).

3.7 Directives

The ASSUME Directive

assume seg-reg:segname, ...
or
assume nothing

assume identifies to the assembler the segments that are pointed at by segment registers. This directive is discussed in the "Segmentation" section of this chapter.

The BSS Directive

bss sym:type,size

The *bss* directive creates a variable that will be placed in the program's uninitialized data area. This area immediately follows the program's *dataseg* segment, and is automatically cleared by the startup routine that is in the standard versions of *c.lib*.

The operands to *bss* define the attributes of the created variable, as follows:

* *sym* - the name of the variable;
* *type* - its type. This can be *byte*, *word*, or *dword*.
* *size* - the number of bytes to be reserved for the variable.

By default, a symbol defined with the *bss* directive is local to the module in which it is defined; that is, it can't be accessed by other modules. It can be made globally accessible using the *public* directive.

The *global* directive defines uninitialized variables that can be accessed by other programs. Normally, you should use *global* to create an uninitialized global variable.

The *bss* and *public* can be useful for creating a globally-accessible area in the uninitialized data area that can be accessed using more than one name. This can't be done using the *global* directive. For example, the following code allocates 10 bytes in the uninitialized data area, which can be accessed by the names *fred* and *susan*:

```
public  fred, susan
bss     fred:0,byte
bss     susan:10,byte
```

The DB, DW, and DD Directives

```
[var]   db      val [,val, val,...]
[var]   dw      val [,val, val,...]
[var]   dd      val [,val, val,...]
```

The *db*, *dw*, and *dd* directives reserve one or more fields of memory, optionally initializes them, and optionally defines a variable. The number of bytes per field for *db*, *dw*, and *dd* is one, two, and four, respectively.

var is a variable name, and is optional. If specified, the name is entered into the symbol table with the following attributes:

attribute	*value*
type	*byte* (for db), *word* (dw), or *dword* (dd)
segment	the starting paragraph number of the segment in which *var* is defined.
offset	the distance in bytes of *var* from the beginning of the segment.
length	the number of fields defined in the directive.
size	the number of bytes defined in the directive.

Each *val* operand causes one or more fields of memory to be reserved and optionally initialized. The assembler processes the

val operands from left to right, reserving space within the current segment at successively higher addresses.

A *val* can be one of the following:

* An expression that evaluates to a constant. In this case, a single field is reserved for the operand and is initialized to the value of the expression. For example, the first directive that follows reserves a one byte field, and initializes it to the decimal value 10. The second reserves a two-byte field, and initializes it to the hexadecimal value 1234h (with 12h in the highest-addressed byte). The third reserves a four-byte field, and initializes it to hex 1234h (with 0 in the highest-addressed word and 1234h in the lowest).

```
db      10
dw      1234h
dd      1234h
```

* An address expression, for *dw* and *dd*. That is, a relocatable expression whose type is *variable* or *label*. For *dw*, the offset attribute of the expression is set in the field. (that is, the distance of the location referenced by the expression from the beginning of the segment containing it). For *dd*, both the segment and offset components of the expression are set in the four-byte field, with the segment number in the highest-addressed two bytes.

  For example, suppose *var* is the name of a variable in the *dataseg* segment. Then the *dw* statement below reserves two bytes and places the offset of *var* in it. And the *dd* statement reserves four bytes, placing the segment number of *dataseg* in the two highest-addressed bytes and the offset of *var* from the beginning of *dataseg* in the two low-addressed bytes.

```
dw      var
dd      var
```

  The offset of *var* is determined when the program is linked. The starting paragraph number of *dataseg* isn't known until the program is loaded, so the loader has to adjust the two high-order bytes of the *dd* statement when it loads the program.

* For *dw* and *dd*, an operand can be a segment name, or an expression that evaluates to the starting paragraph number of a segment (such as *seg name*). The paragraph number is set in the highest-addressed two bytes for *dd*, and the low-addressed bytes are set to zero.

* A *db*, *dw*, or *dd* operand can be a question mark. In this case, a single field is reserved for the operand, and is not initialized. For example,

```
db      ?
dw      ?
dd      ?
```

reserve one byte, word, and double word, respectively, and don't initialize them.

* For *db*, an operand can be a character string, surrounded by a pair of single or double quotes. The characters in the string will occupy successively higher memory locations. For example,

```
db      "This is a string"
db      'as is this'
```

* An operand to *db*, *dw*, or *dd* can be a repeated reservation and initialization of the form

```
count dup (val1, val2, ...)
```

where each *val* is a legal operand of the directive containing it, and *count* is a constant. This type of operand is equivalent to replicating the operands

```
val1, val2, ...
```

count times in a statement.

For example, the following statement reserves 5 bytes of memory, without initializing it:

```
db      5 dup (?)
```

The following statement reserves 10 sets of three four-byte fields. In each set, the first four-byte field is initialized to *count*, the second to *start*, and the third is uninitialized:

```
dd      10 dup (count, start, ?)
```

dup items can contain strings, as in

```
db      5 dup ('hello', 'goodbye')
db      8 dup (5, "albert")
```

And they can contain nested *dup* fields, as in

```
db      15 dup('hello', 3 dup('goodbye'))
```

The END Directive

end [expr]

The *end* directive identifies the end of an assembly language program.

The optional parameter *expr* identifies the address at which execution will begin in a program. If several modules are linked together to form an executable program, at most one of them can specify the program's starting address. And if, when an executable program is linked, none of its modules specifies a starting address for the program, execution will begin at the first byte in the program's first code segment.

The EQU Directive

name equ expr

equ creates an entry in the symbol table for the symbol *name*, assigning it the value of *expr*.

name must not already be defined (that is, have an entry in the symbol table). If you want to create identifiers whose value can be redefined, use the = directive.

expr can be any of the following:

* A variable or label name. This name can be a forward reference, if necessary; that is, the statement defining the name can follow the *equ* statement. For example, both the *equ* statements that follow are legal:

```
d1      dw      ?
new1    equ     d1
new2    equ     d2
d2      dw      ?
```

* An integer numeric constant. For example,

```
size    equ     10
```

* A valid expression involving constants, variables and labels. For example,

```
e1      equ     2 + 3
e2      equ     e1 and 4
e3      equ     d1 + 8
d1      dw      ?
```

* An 8086 register name. For example,

```
count   equ     cx
pointer equ     bx
        mov     count,10
        mov     pointer, offset array
```

* 8086 instruction names. For example,

```
get     equ     mov
bump    equ     inc
        get     ax,bx
        bump    ax
```

* A register expression. For example,

```
arg1    equ     -4[bp]
lcl1    equ     0[bp]
        mov     ax,arg1
        mov     bx,lcl1
```

expr cannot be an external identifier (that is, an identifier defined in the *extrn* directive).

The '=' (equal sign) Directive

name = *expr*

The = directive assigns the value of the constant expression *expr* to the identifier *name.*

name can be either a new identifier or one that was previously defined using the = directive. In the latter case, the new value replaces the old.

The = directive is similar to *equ.* It differs in that it allows identifiers to be redefined, and can only be assigned a constant expression as a value.

For example, the first statement below creates the symbol *sum* and assigns it the value 0. The second statement redefines *sum* to 7. And the last statement increments the current value of *sum* by one.

```
sum     =       0
sum     =       7
sum     =       sum+1
```

An identifier created using the = directive can't have its value redefined using *equ.* For example, since in the above statements *sum* was created using =, its value couldn't be redefined using *sum equ 8.*

Similarly, an identifier created using *equ* can't have its value redefined using =.

The EVEN Directive

even

The *even* directive ensures that the data following the directive is aligned on a word boundary. If the following data would otherwise begin on an odd-numbered byte, *even* outputs a single

byte consisting of a *nop* instruction (0). If the code or data would begin on an even-numbered byte, *even* does nothing.

even can be used to speed up execution of programs that will run on 8086, 80186, or 80286 processors. The reason for this is that on these processors, but not on an 8088 processor, instructions that access a word in memory execute slightly faster if the word begins on an even-numbered address.

even can only be used in the data segment, and cannot be used if the segment is byte-aligned.

The EXTRN Directive

extrn name:type [,name:type,...]

extrn defines the names and types of symbols that have been declared to be "public" or "global" in other modules, and thus allows the program being assembled to reference those symbols. For more information, see the section entitled "Globally-accessible Symbols" in this chapter.

The GLOBAL Directive

global sym:type,size

The *global* directive creates a global variable that will be placed in the program's uninitialized data area. For more information, see the section entitled "Globally-accessible Symbols" in this chapter.

The GROUP Directive

name GROUP segname, segname, ...

The *group* directive, which is supported by the PC-DOS/MS-DOS assembler, *masm*, is accepted by *as*, but doesn't have any effect, since the assembler doesn't support Intel-style grouping of segments.

The INCLUDE Directive

include filename

The *include* directive causes *as* to suspend assembly of the file that contains the directive and to assemble the source that is in the specified file, *filename*. When the assembler finishes with the *include* file, assembly of the file containing the *include* directive continues.

The *include* statement allows you place statements that are common to several assembly language programs in one file. Other files can access these statements using an *include* statement, eliminating the need to repeat the statements in each file that

uses them.

There is no limit to the number of *include* statements that a single file can contain, and a file specified in an *include* directive can itself contain an *include* directive. The maximum depth of include file nesting is five files; this means that when one file includes another, which includes another, and so on, the total number of files in this chain can't exceed five.

The LABEL Directive

name label type

The *label* directive creates a variable or label named *name*, which has the following attributes:

* Type: the value of the *type* operand. This can be *byte, word, dword, near*, or *far*.
* Segment: the segment into which code and data are currently being assembled.
* Offset: the current offset within that segment.

label is useful when several names, possibly having different type attributes, need to be associated with the same location.

For example, the following code allows a program to easily access two consecutive bytes as both a word and as two separate bytes:

```
aword   label   word
alow    db      ?
ahigh   db      ?
```

label can also be used to define secondary entry points within a procedure that has been defined with the *proc ... endp* directives. For example,

```
main    proc    far
        ...
sec     label   far
        ...
main    endp
```

If you use the *label* directive in this way, be careful that the type of the label matches the type of the procedure in which it is contained.

The LARGECODE Directive

largecode

The *largecode* directive specifies that the module being assembled is to use the large code memory model. This causes the program's *codeseg* segment to be a separate segment when the

module is linked into a program.

If the *largecode* directive isn't specified, the module will use the small code memory model. In this case, when the module is linked into a program, its *codeseg* will be joined with the *codeseg* of all other modules that were assembled to use the small memory model, into a single segment.

The MOD186 Directive

mod186

The *mod186* directive specifies that the module will run on a 80186 or compatible processor. This allows the module to contain 80186 instructions.

If this directive isn't specified, and if the program contains 80186 instructions, the assembler will flag them as an error.

The NAME Directive

name module-name

The *name* directive defines the name of the object module generated for an assembly language source file.

A module name contains up to eight characters. *module-name* can contain any number of characters, but only the first eight are used.

The only time a module needs a name is when it's in a library. When the librarian *lb* places a named object module in a library, that name is given to the library's copy of the module. When it places an unnamed module in a library, *lb* derives the name of the library's copy of the module from the name of the input file, by removing the drive, path, and extension components of the file name.

The ORG Directive

org expr

The *org* directive sets the location counter within the current segment to *expr.*

The PROC and ENDP Directives

proc_name proc [proc_type]
...
proc_name endp

The *proc* and *endp* directives are used to delimit a related sequence of instructions, such as a subroutine.

The *proc* directive creates the label *proc_name*. The optional operand *proc_type* defines the type of the label; if specified, it can be either *near* or *far*. If the type operand is not specified, *proc_name* is given type *far* if the module is being assembled to use the large code memory model (that is, it contains the *largecode* directive), and is given type *near* otherwise.

The type assigned to *proc_name* controls the type of return instruction that is generated for each *ret* instruction within the *proc ... endp* directives: if *proc_name* is of type *near*, then near returns are generated; and if *proc_name* is of type *far*, then far returns are generated.

For example, the following program fragment defines a near procedure, *np*, and a far procedure, *fp*, and shows calls to them:

```
codeseg segment         para
np      proc    near
        ...             ;the proc's code
        ret             ;this will be a near return
np      endp
fp      proc    far
        ...             ;code for the proc
        ret             ;this will be a far return
fp      endp
        ...
        call    np      ;a near call
        call    fp      ;a far call
        ...
codeseg ends
```

There can be several labels that serve as entry points into the statements within a pair of *proc ... endp* directives. If the procedure is of type *near*, the entry point labels can be defined using the *label* directive or in the label field of an instruction. For example, the following near procedure can be entered at the labels *main*, *sec1*, or *sec2*:

```
main    proc    near
        ...
sec1    label   near
        ...
sec2:   mov     ax,bx
        ...
        ret             ;a near return
```

If the procedure is of type *far*, secondary entry points can only be defined using the *label* directive. Other labels can be defined within the procedure in the label field of instructions, but these can only be used as the target of jump instructions that are contained in the procedure, and not as labels that can be

called from outside of the procedure. For example, in the following code, the near label *t1* can be jumped to by code that is within the *subr* proc, but it can't be called from code outside of the proc. The far label *t2* can be called from outside the proc, but you probably wouldn't jump to it from code that's in the proc.

```
codeseg segment         para public
subr    proc    far
        ...
t1:     mov     ax,bx
        ...
t2      label   far
        ...
subr    endp
        call    subr    ;far call to subr
        call    t2      ;far call to t2
        call    t1      ;near call to t1 **don't do this**
codeseg ends
```

procs can be nested; that is, one *proc* can be contained within another. For example, in the following code, the far proc *main* contains the near proc *subr*:

```
codeseg segment         para public
main    proc    far
        ...
subr    proc    near
        ...             ;code for subr
        ret             ;near return from subr
subr    endp
        ...             ;code for main
        call    subr    ;near call to subr
main    endp
codeseg ends
```

There is no "block structuring" of *procs*; that is, execution can fall into a nested *proc*. "Falling into" a nested procedure is usually an error, as shown in the following erroneous example:

```
p1      proc    far
        mov     ax,bx
        add     ax,12
p2      proc    near
L1:     mov     ax,cx
        ret
p2      endp
        sub     ax,1
        ret
p1      endp
```

The programmer expected that the next instruction to be executed after the *add* in the *p1* proc would be the *sub* that follows the *p2* proc; actually, the next instruction to be executed will be the *mov* in the *p2* proc. This program will crash, because a call to the far proc *p1* will result in a near return, at the *ret* that's in *p2*.

The PUBLIC Directive

public sym [,sym ...]

The *public* directive identifies the symbols defined in the program being assembled that can be accessed by other modules. For more information, see the section entitled "Globally-accessible symbols" in this chapter.

The RECORD Directive

recname record fldname:width [=initval] [,...]

The *record* directive creates user-defined directive that can be used in a codemacro. When such a user-defined directive is encountered during the expansion of a codemacro, the assembler combines specified values into a byte or word and then outputs the result. The *record* directive also defines a template that is associated with the user-defined directive. The template for a user-defined directive has the following uses:

* it defines whether a byte or word will be generated when the directive is invoked,
* it organizes the bits in the byte or word into named fields,
* it optionally assigns a default value to the fields.

In the synopsis, *recname* is the name of the user-defined directive.

The *record* directive has one or more operands, separated by commas, each of which defines the attributes of a field within the template that is created for the user's directive. As shown in the synopsis, the operand for a field contains the following items:

* *fldname* is the field's name.
* *fldwidth* is the number of bits it contains.
* *initval* is the default value for the field. This value is optional; if not specified, it's assumed to be 0. When the user-defined directive is used, and a byte or word is generated for it, a value will be set in each of the record's fields. The user-defined directive can optionally specify the value of the fields. For those fields for which the directive doesn't specify a value, the field is set to its default value.

The *record* directive doesn't explicitly specify whether the created template contains a byte or a word. This is determined from the size of the individual fields: if the sum of the field sizes is less than 8 bits, the template will occupy one byte; if the sum is greater than 8, the template will occupy a word.

Also, the *record* directive doesn't explicitly specify the location of a template's fields. This is determined from the sizes of the template's fields: the fields in the template are contiguous and are right-justified, with the last field defined in the *record* directive that created the record's template occupying the least significant bits in the template.

For example, the following *record* directive creates a directive named *errflgs* and associates with it a template containing three fields: *ioerr*, containing 3 bits; *syserr*, containing 4 bits; and *memerr*, containing 1 bit.

```
errflgs  record  ioerr:3, syserr:4, memerr:1
```

The template contains a single byte, since that is all that is needed to hold the record. The *memerr* field will occupy the least significant bit of the template; *syserr* field will occupy the next four most significant bits; and the *ioerr* field will occupy the most significant bits. No initial value was specified for the fields; because of this, when the *errflgs* directive is used, those fields for which initial values aren't explicitly specified will be set to 0.

A template need not specify all the fields in a template. In this case, the template's defined fields will be right-justified, with the undefined bits occupying the most significant bits in the template. For example, the following *record* directive creates a directive named *partly* and associates with it a template containing two named fields: *hi*, containing 6 bits, and *low*, containing 5 bits.

```
partly  record  hi:6=32, low:5=24
```

The template is 16 bits wide, with its *low* occupying the least significant 5 bits, the *hi* field occupying the next most significant 6 bits, and the most significant 5 bits of the storage being

unnamed and unused.

In the last example, a default value was specified for each of the template's fields. If a *partly* directive is used and the directive doesn't specify the initial value of the *hi* field, the field will be initialized to 32. Similarly, the default initialization value of the *low* field is 24.

Using a User-defined Record Directive

A user-defined record directive can only be used in a codemacro. This is discussed in the Codemacro section of this chapter.

The SEGMENT and ENDS Directives

segname segment [align_type] [comb-type] ['cname']
...
segname ends

The *segment* and *ends* directive identify the logical segment containing the code and data that are defined between the directives, and define the attributes of the segment. For more information, see the "Segmentation" section of this chapter.

3.8 Macro directives

as provides support for macros. The *as* macro features are compatible with those provided by the MS-DOS/PC-DOS assembler.

A macro is a named sequence of statements, which is defined when a program is assembled. Each time a macro is "invoked" (that is, its name appears in the operation field of a source line), the macro's statements are assembled.

A macro can have named parameters, with the names appearing in the statements within the macro. When a macro having parameters is invoked, the actual parameters for the macro (that is, the operands in the invoking line) replace the parameter names and then the resulting statements of the macro are assembled.

A macro definition begins with the *macro* directive and ends with the *endm* directive. The block of statements that will be assembled whenever a macro is invoked appear between these two statements.

As an example of a parameterless macro, the following statements define the macro *begin*, which might be used when entering a subroutine:

```
begin   macro
        push    bp      ;save bp
        mov     sp,bp   ;set new frame pointer
        add     sp,10   ;reserve 10 bytes on stack for locals
        endm            ;end of macro definition
```

This macro is invoked within a program as follows:

```
input   proc    near    ;entry point for the input subr
        begin           ;set up stack & stack regs for subr
        ...             ;body of the input proc
input   endp            ;end of input
output  proc    near    ;entry point for the output subr
        begin           ;initial code for subr
        ...             ;body of the output proc
output  endp            ;end of output
```

Thus, the *input* and *output* subroutines that use macros are equivalent to the following:

```
input   proc    near    ;entry point for the input subr
        push    bp
        mov     bp,sp
        add     sp,10
        ...             ;body of the input proc
input   endp            ;end of input
output  proc    near    ;entry point for the output subr
        push    bp
        mov     bp,sp
        add     sp,10
        ...             ;body of the output proc
output  endp            ;end of output
```

As you can see, use of macros makes the source program shorter, and frequently makes it easier to understand.

For an example of a macro with parameters, let's modify the *begin* macro so that it can push a invoker-specified register and add an invoker-specified amount to the stack pointer:

```
begin   macro   reg,size
        push    bp
        push    reg     ;push specified register
        mov     bp,sp
        add     sp,size ;add specified value to stack pointer
        endm            ;end of macro definition
```

In this macro, the name of the parameter specifying the register to be pushed is *reg*, and the name of the parameter specifying the amount to be added to the stack pointer is *size*.

Now let's modify the *input* and *output* subroutines to use the modified *begin* macro. *input* will tell *begin* to push register AX and add 20 bytes to the stack pointer; *output* will tell *begin* to push register BX and add 30 bytes to the stack pointer:

```
input   proc    near    ;entry point for the input subr
        begin ax,20
        ...             ;body of input
input   endp            ;end of input
output  proc    near    ;entry point for the output subr
        begin bx,30
        ...             ;body of subr
output  endp            ;end of output
```

When the modified *begin* macro is invoked in the *input* subroutine, the assembler assembles the macro's statements, replacing each occurrence of the parameter named *reg* within the macro's statements with the character string "ax", and each occurrence of the *size* parameter with the character string "20". Similarly, when the modified *begin* macro is invoked in the *output* subroutine, the assembler assembles the macro's statements, replacing each occurrence of the parameter named *reg* with "bx", and each occurrence of the *size* parameter with "30". The macro-ized *input* and *output* subroutines are thus equivalent to the following:

```
input   proc    near    ;entry point for the input subr
        push    bp
        push    ax
        mov     bp,sp
        add     sp,20
        ...             ;body of input
input   endp            ;end of input
output  proc    near    ;entry point for the output subr
        push    bp
        push    bx
        mov     bp,sp
        add     sp,30
        ...             ;body of subr
output  endp            ;end of output
```

3.8.1 Local symbols

It is sometimes neccessary to define symbols in a macro invocation which won't conflict with other symbols defined in the program and that won't conflict with the symbols created by other invocations of the same macro. For example, the following macro creates the label *lbl* whenever it is invoked:

```
bump    macro
        test    ax,4
        jz      lbl
        add     bx,10
lbl:
        endm
```

There are two problems with this macro: first, the programmer will

have to insure that a program that uses *bump* doesn't itself contain the label *lbl*. Second, the program will only be able to call *bump* once; if it calls it more than once, *lbl* will be multiply defined.

The *local* directive can solve these problems. This directive, which must precede all other type of statements in a macro definition, creates symbols that are unique for each invocation of a macro. For example, if the statement

```
local lbl
```

is added to the beginning of the *bump* macro, then *bump* can be used as often as desired in a program, and the macro's *lbl* symbol won't conflict with a *lbl* symbol in the main body of the program.

3.8.2 Concatenating parameters to text

When the assembler is processing a macro invocation, it can unambiguously spot a parameter name within the macro's statements if the name is surrounded by delimiter characters such as newline, tab, space, brackets, etc. For example, consider the following macro definition:

```
sum     macro   a
        add     ax,a
        jmp     lbla
        endm
```

sum has one parameter, *a*. When *sum* is invoked, the assembler will spot only one occurrence of the parameter name in the macro's statements: as the second operand to the *add* statement. This occurrence of the name is separated by the delimiters ',' and the newline character. There are several other occurrences of the parameter name in the macro's statements (such as the 'a' in 'add', 'ax', and 'lbla'), but since they aren't surrounded on both ends by delimiters, they aren't considered to be parameter names. Thus, if the *sum* macro is invoked with the statement

```
sum     one
```

the assembler will generate and assemble the following statements:

```
add     ax,one
jmp     lbla
```

If a parameter name appears in a statement within a macro and the name isn't surrounded on both sides by delimiters, you can tell the assembler to spot this occurrence of the parameter name during an invocation of the macro by putting an & character at the ends of the name that aren't joined to delimiters. During the invocation the assembler will replace the parameter name and the surrounding & characters with the actual value of the parameter.

For example, let's modify the *sum* macro so that the assembler will recognize the *a* in the *jmp* statement as a parameter name:

```
sum     macro  a
        add    ax,a
        jmp    lbl&a
        endm
```

When this modified *sum* macro is invoked with the statement

```
        sum    one
```

the assembler will generate and assemble the following code:

```
        add    ax,one
        jmp    lblone
```

To demonstrate how the assembler can spot a macro name that is entirely surrounded by text, let's modify the *sum* macro again:

```
sum     macro  a
add     ax,a
jmp     lbl&a&xyz
endm
```

We still want the *a* in the *jmp* statement to be recognized by the assembler as being a macro name. Since both ends of it are delimited by text and not by delimiter characters, an & character is needed at each of its ends. When this macro is invoked with

```
        sum    one
```

the assembler will generate and assemble

```
        add    ax,one
        jmp    lblonexyz
```

3.8.3 Concatenating parameters to parameters

Using the & character, a macro statement can also specify that the value of two or more macro parameters are to be concatenated when the macro is invoked. For example:

```
space   macro  p1,p2
p1&&p2  dw     ?
        endm
```

Note that two & characters are used in the *dw* statement. This is done because of the way the assembler performs parameter substitution: when the macro is invoked, the assembler will replace p1& with the value of *p1*, and *&p2* with the value of *p2*. As a convenience, the assembler allows you to abbreviate two adjacent & characters with just one; so *p1&&p2* could have been abbreviated to *p1&p2*.

When the *space* macro is invoked with the line

```
space   reg,min
```

the following statement is generated and assembled:

```
regmin dw       ?
```

3.8.4 Parameter substitution within quoted strings

When a macro is invoked, the assembler doesn't normally replace a parameter name found in a quoted string with the parameter's value. For example, consider the *storage* macro:

```
storage macro  a
        db     "a"
        endm
```

When *storage* is invoked with the statement

```
storage abc
```

the assembler will generate and assemble the statement

```
db      "a"
```

To have the quoted *a* replaced by the value of the parameter named *a*, prefix the parameter name with an & character:

```
storage macro  a
        db     "&a"
        endm
```

With this version of the *storage* macro, the statement

```
storage abc
```

now generates

```
db      "abc"
```

Modifying the definition of *storage* to

```
storage macro  a
        db     "x&a&y"
        endm
```

And invoking it with

```
storage abc
```

generates

```
db      "xabcy"
```

The use of & to identify a macro parameter name is less flexible when the name is inside a quoted string than when it's outside: when inside, the macro name must be preceded by an &. It can have a terminating &, if needed to separate the name from text that follows, but it must always have a preceding &. Thus, the following definition of *storage* is invalid:

```
storage macro  a
        db     "a&"    ;invalid
        end
```

3.8.5 Passing a symbol's value to a macro

Normally, when a character string is specified as an operand in a macro invocation, the assembler replaces occurrences of the operand's corresponding macro parameter name with that character string as it processes the macro. If the character string is the name of a symbol that has been given a constant value, using the *equ* or = directives, you can alternatively have the parameter name replaced with the symbol's *value* instead of its *name*. To request this, prefix the symbol name with the character % in the statement that invokes the macro.

For example, consider the following macro:

```
load    macro  val
        mov    ax,val
        endm
```

If *load* is invoked with the statement

```
        load   count
```

the following statement will be generated:

```
        mov    ax,count
```

If it is invoked with the following statements:

```
count   =      1
        load   %count
```

the following statement will be generated:

```
        mov    ax,1
```

In this second invocation, the % character tells the assembler to replace occurrences of the *a* parameter with the value of the *count* symbol and not with the string *count*.

3.8.6 Passing comma-containing arguments to macros

When a macro is invoked, commas separate the arguments that are to be passed to the macro. These commas are not passed to the macro. If a comma occurs in a quoted string, the assembler will consider it to be part of the string and not an argument separator.

A macro invocation can pass a comma-containing string that isn't quoted to a macro by surrounding the string with angle brackets <>. The assembler passes the string, without the angle brackets, to the macro as a single argument.

For example, consider the macro *top*:

```
top     macro   p1,p2
        dw      p1
        db      p2
        endm
```

Invoking *top* with

```
        top     a,b,c,d
```

generates

```
        db      a
        db      b
```

Invoking *top* with

```
        top     <a,b,c>,d
```

generates

```
        dw      a,b,c
        db      d
```

3.8.7 Nesting macros

Macros can be "nested"; that is, the definition of one macro can contain a statement that invokes another macro, including the macro being defined. For example, the following code defines two macros, *outer* and *inner*. *outer* calls *inner*:

```
outer   macro   p1,p2
        db      p1
        inner   p2
        endm

inner   macro   p3
        dw      p3
        endm
```

The statement:

```
        outer   1,2
```

generates:

```
        db      1
        dw      2
```

As an example of a macro that calls itself, consider the *storage* macro, which allocates a sequence of bytes of storage, containing successively smaller values:

```
storage macro  count
        db     count
        if count-1
        storage %count-1
        endif
        endm
```

The statement:

```
        storage 3
```

generates:

```
        db     3
        db     2
        db     1
```

The expansion of a macro invocation that is nested within another macro occurs when the outer level macro is invoked. Consider, for example, the following code. First, two macros, *outer* and *inner*, are defined, where *outer* invokes *inner*. Then *outer* is invoked, *inner* is redefined, and *outer* is invoked again. Finally, *inner* is purged again.

```
inner   macro          ;define inner
        db     5
        endm

outer   macro          ;define outer
        dw     4
        inner
        endm

        outer          ;invoke outer
        purge  inner   ;delete inner

inner   macro          ;define a new inner
        db     6
        endm

        outer          ;invoke outer
        purge  inner   ;purge inner
```

This code generates the following:

```
        dw     4
        db     5
        dw     4
        db     6
```

which is what you'd expect with macro expansion occurring when a macro is invoked.

If the last purge of *inner* is omitted, the assembler will generate the following code:

```
dw      4
db      6
dw      4
db      6
```

The reason for this is that when the assembler makes its second, code-generation pass, through the source file, and encounters the first definition of *inner*, it will not process it if *inner* is still defined from the first pass.

3.8.8 Directives for repeatedly assembling a block of statements

There are three directives that cause the assembler to assemble a block of statements multiple times: *rept*, *irp*, and *irpc*. The block begins with one of these directives and ends with the *endm* directive.

These directives can be used to define a block of statements either within or without a macro definition. In the former case, the block isn't assembled until the macro in which it is defined is invoked. In the latter case, the block is assembled just at the point in the program where it is defined; it doesn't have a name, and hence can't be invoked for assembly at another point in the program.

One of the directives, *rept*, doesn't support parameter definition and substitution; its statements are simply assembled a specified number of times. The other two directives, *irp* and *irpc*, each defines a parameter name and a list of values. A block defined using one of these directives is assembled once for each of the values; during a single assembly of the block, occurrences of the parameter name in the block are replaced by the current value.

A block defined using one of these directives can use some of the features available to macros:

* It can define local symbols , using the *local* directive.
* It can prematurely exit from the assembly of the block, using the *exitm* directive.
* A parameter can be joined to text using the & character;
* The value of a constant symbol can be passed to the block as an argument rather than its name, by prefixing the name with the % character;

3.8.8.1 The REPT Directive

The *rept* directive has the form:

```
rept    const_expr
```

where *const_expr* is a constant expression that defines the number of times the directive's block of statements is to be assembled.

For example,

```
x       =       0
        rept    4
        db      x
x       =       x+1
        endm
```

generates the equivalent of the statements

```
        db      0
        db      1
        db      2
        db      3
```

If this block of statements occurs outside of a macro definition, the assembler generates the *db* statements at the point in the program where it encounters the block, and the block can never be invoked again. If it occurs inside a macro definition, the assembler generates the *db* statements when the macro is invoked. The macro can be invoked of often as desired, with its *rept* block assembled anew each time. For example, consider the macro *gen*, which contains a slightly modified version of the above *rept* block:

```
gen     macro   initval, count
x       =       initval  ;set x to the starting value, initval
        rept    count    ;generate the db statement count times
        db      x
x       =       x+1
        endm    ;end of the rept block
        endm    ;end of the gen macro
```

The *rept* block inside the *gen* macro won't be assembled until *gen* is invoked. If *gen* is invoked at one point with the statement:

```
        gen     10,3
```

the equivalent of the following statements are assembled:

```
        db      10
        db      11
        db      12
```

3.8.8.2 The IRP directive

The *irp* directive has the form:

```
        irp     param, <arglist>
```

where *param* is the name of a parameter and *<arglist>* is a list of actual arguments that are separated by commas and surrounded by angle brackets.

The *irp* directive's block of statements are assembled once for each argument in *arglist*. Each time the block is assembled, occurrences of the parameter name *param* in the block are replaced with the current

argument.

For example,

```
irp     x, <1,2,3>
db      x
endm
```

generates the following:

```
db      1
db      2
db      3
```

In the following example, the macro *bump* increments the locations whose names are passed to it by a specified value:

```
bump    macro   val, list
        irp     x, <list>
        add     x, val
        endm    ;end of irp
        endm    ;end of bump macro
```

If *bump* is called with:

```
bump    10, <count, a, ax>
```

the following code is generated:

```
add     count, 10
add     a,10
add     ax,10
```

3.8.8.3 The IRPC directive

The *irpc* directive has the form:

```
irpc    param, string
```

where *param* is the name of the directive's parameter and *string* is a character string. The directive's block of statements are assembled once for each character in *string*. Each time the block is assembled, occurrences of the parameter name *param* are replaced with the current character from *string*.

For example,

```
irpc    x, 0123
db      x
endm
```

generates:

```
db      0
db      1
db      2
db      3
```

3.8.9 Summary of the macro directives

The ENDM Directive

endm

Identifies the end of a block of statements that begins with *macro*, *rept*, *irp*, or *irpc*.

The EXITM Directive

exitm

exitm causes the assembler to terminate the expansion of a macro or repetition directive. If the block containing *exitm* is contained within another block, the outer level block continues to be expanded. A block containing *exitm* must still be terminated with the *endm* directive; *exitm* and *endm* are not interchangeable.

The IRP Directive

irp param, <arglist>

irp causes its block to be assembled several times, once for each argument in *arglist*; each time the block is assembled, occurrences of *param* in the block are replaced with the current *arglist* argument.

The IRPC Directive

irpc param, string

irpc causes its block to be assembled several times, once for each character in *string*; each time the block is assembled, occurrences of *param* in the block are replaced with the current character in *string*.

The LOCAL Directive

local name [,name ...]

local is used within a macro block to create unique names for the names *name1*, *name2*, ... When *name1*, *name2*, ... are encountered in the block, they are replaced with their unique name.

The MACRO Directive

macname macro [param [, param ...]]

macro begins the definition of a macro. It has the form:

macname *macro* *arg1, arg2, ...*

where *macname* is the macro's name and *arg1*, *arg2*, ... are the names of its arguments.

The PURGE Directive

purge name [, name ...]

purge deletes the definition of a macro, allowing its space in internal tables to be reused.

A macro cannot be redefined without first using *purge* to delete the previous definition.

The REPT Directive

rept expr

rept begins a block of statements that are to be assembled *expr* times.

3.9 Conditional directives

as supports several directives with which you can specify that parts of your program should be assembled only if certain conditions are satisfied.

To make assembly of a sequence of statements conditional, begin the block with one of the *if* directives, which specifies the condition that must be met for the block to be assembled, and terminate the block with the *endif* directive. Such a block has this form:

```
if condition
 ...        ;statements to be assembled if condition is true
endif
```

You can also specify that one block of statements is to be assembled if a condition is true and that another is to be assembled if the condition is false. The two blocks have this form:

```
if condition
 ...        ;statements to be assembled if condition is true
else
 ...        ;statements to be assembled if condition is false
endif
```

You can nest blocks of statements whose assembly is conditional, to any level. This means that a block of statements whose assembly is conditional can itself contain blocks of statements that are surrounded by the conditional assembly directives. For example

```
if cond1
 ...      ;statements to be assembled if cond1 is true
 if cond2
  ...     ;statements to be assembled if cond1 and cond2 are true
 endif    ;endif for 'if cond2'
 ...      ;more statements to be assembled if cond1 is true
endif     ;endif for 'if cond1'
```

Another example:

```
if cond1
 ...      ;statements to be assembled if cond1 is true
 if cond2
  ...     ;statements to be assembled if cond1 and cond2 are true
 else
  ...     ;statements to be assembled if cond1 is true
                ;and cond2 is false
 endif    ;endif for 'if cond2'
 ...      ;more statements to be assembled if cond1 is true
endif     ;endif for 'if cond1'
```

The *else* and *endif* directives pair up with the nearest preceding *if* directive. Because of this, a block of statements whose assembly is conditional can't be partially within and partially outside another such block; it must either be entirely within or entirely outside.

Here are the conditional directives:

The IF Directive

if const_expr

const_expr is an expression having a constant value, which is built from constants, the names of symbols having a constant value, the names of macro arguments, and operators described in the section on operand operators that act on constant arguments. When computing the value of an expression in an *if* statement, the assembler uses the value of a symbol or macro argument, and not its name.

The *if* directive's condition is true if the value of the expression is nonzero, and is false if the value is zero.

For example,

```
if count lt 5
 ...      ;statements to be assembled when count < 5
else
 ...      ;statements to be assembled when count >= 5
endif
```

The IFE Directive

ife const_expr

const_exp is an expression having a constant value, as described in the discussion of the *if* directive. The *ife* directive's condition is true if the value of the expression is zero, and is false if the value is nonzero.

The IF1 Directive

if1

if1 is true if the assembler is making its first pass through the source file, and is false otherwise.

The IF2 Directive

if2

if2 is true if the assembler is making its second pass, or a sqeeze pass, through the source file, and is false otherwise.

The IFDEF Directive

ifdef symbol

ifdef is true if *symbol* is defined or has been declared external, and is false otherwise.

The IFNDEF Directive

ifndef symbol

ifndef is true if *symbol* is not defined and has not been declared external, and is false otherwise.

The IFB Directive

ifb <arg>

ifb is true if *arg* is blank or if the entire argument to *ifb*, including the angle brackets, is not present, and is false otherwise. The angle brackets around *arg* are required.

ifb is primarily used within macros, to determine whether a particular parameter has been passed to the macro.

The IFNB Directive

ifnb <arg>

ifnb is true is *arg* is not blank, and is false otherwise. The angle brackets around *arg* are required.

As with *ifb*, *ifnb* is used primarily within macros.

The IFIDN Directive

ifidn <arg1>, <arg2>

arg1 and *arg2* are character strings. *ifidn* is true if the two strings are identical, and is false otherwise. The angle brackets around *arg1* and *arg2* are required.

ifidn is used primarily within macros to determine the value of a character string argument in a call to the macro.

The IFDIF Directive

ifdif <arg1>, <arg2>

ifdif is true if the character strings *arg1* and *arg2* are not identical, and is false otherwise. The angle brackets around the arguments are required.

As with *ifidn*, *ifdif* is used primarily within macros.

The ELSE Directive

else

The *else* directive can be used with an *if* directive to specify a block of code that is to be assembled if the *if* directive condition is false. Only one *else* directive is allowed for an *if* directive.

The ENDIF Directive

endif

The *endif* directive identifies the end of a block of statements whose assembly is conditional. *endif* terminates the most recent, unterminated *if* directive.

3.10 Codemacros

A codemacro is a named sequence of directives. When the assembler encounters a codemacro's name in the operation field of a statement, it generates code as directed by the codemacro's directives.

All the instructions supported by *as* are implemented as codemacros. You can define your own codemacros, thereby creating your own customized instruction set. For example, the following statements invoke the codemacros *mov*, *myinst*, and *add*:

```
mov    bx,mem
myinst [bx], ax
add    ax,5
```

mov and *add* are codemacros whose definitions are built into the

assembler. *myinst* is a codemacro that was defined within the program sometime prior to its invocation.

A codemacro can have parameters. When a parameterized codemacro is invoked, parameters are specified as operands to the instruction; as the assembler processes the codemacro invocation, it replaces the names of the parameters in the codemacro's definition with the parameter values. In the above code, for example, *bx* and *mem* are parameters to the *mov* instruction. As the assembler processes the *mov* codemacro's directives, it replaces occurrences of the codemacro's first parameter name with *bx* and occurrences of the second parameter name with *mem*.

Codemacros can have the same name. For example, there are eleven different codemacros that have the name *add*.

The definition of a codemacro specifies the number of operands that an invocation of the codemacro can have and the types of the operands. When a codemacro name is encountered in a statement, the assembler examines the definitions of codemacros having that name, beginning with the last such codemacro that was defined. When it finds one whose requirements are met by the invocation's actual parameters, it processes that codemacro.

A codemacro definition begins with a *codemacro* directive and ends with the *endm* directive. In between these two directives are directives that form the body of the codemacro.

The *codemacro* directive has the following form:

```
codemacro    name [param_list]
```

where *name* is the name of the codemacro, and *param_list* is a list of items, separated by commas, each of which defines the name and attributes of one of the codemacro's parameters.

A param_list item has the form:

```
pname:specifier [modifier] [range]
```

where *pname* is the name of the parameter, *specifier* is a letter defining the type that the corresponding actual parameter of an invocation of this codemacro must have, *modifier* is an optional letter that imposes further requirements on the actual parameter, and *range* is an optional expression, or pair of expressions separated by a comma, that is surrounded by parentheses and that imposes even more requirements on the actual parameter.

An alternative form of the statement that begins a codemacro is

```
codemacro    name prefx
```

This form is used to define a codemacro that is to be used as a prefix to other instructions. For example, the standard codemacros *lock* and

rep are defined using this form of the *codemacro* directive.

Only a few directives can occur within a codemacro definition. These are:

* segfix
* nosegfix
* modrm
* relb
* relw
* db
* dw
* dd
* Record initialization

These directives are discussed later in this section.

Here are some simple codemacros:

```
CodeMacro    CLC
db      0f8h
Endm

CodeMacro    POPF
db      9dh
Endm

CodeMacro    ADD  dst:Ab, src:Db
db      4
db      src
Endm
```

The first codemacro defines a codemacro named CLC. It will match a statement having CLC in its operation field, with no operands. When such a statement is found, the codemacro causes the assembler to output the byte 0f8h. Similarly, the POPF codemacro matches a statement whose instruction is POPF and that doesn't have any operands. When such a statement is found, the assembler will output the byte 9dh.

The third codemacro defines one of the eleven codemacros whose name is ADD. It has two parameters, named *dst* and *src*. This ADD codemacro will match an ADD instruction having two operands, the first being either AL or AH, and the second being a constant expression.

3.10.1 Specifiers

A parameter's specifier letter defines the type of actual parameter that will match the parameter. The letters and their associated types:

specifier *parameter type*

A	Accumulator; that is, AX or AL
C	Code; that is, a label expression only.
D	Data; that is, an immediate expression having a constant value.
E	Effective address; either an M (memory reference) or R (register).
M	Memory reference; either a variable (with or without indexing) or a bracketed register expression.
R	General Register only: not an address expression, not a register in brackets, and not a segment register.
S	Segment register only: CS, DS, ES, or SS.
X	Direct memory reference; a simple variable name with no indexing.

3.10.2 Modifiers

The optional modifier letter for a codemacro's parameter further defines the type of instruction operand that will match the parameter. The meaning of the modifier depends on the parameter's type:

* For variables, the modifier defines the size of the operand: 'b' for byte, 'w' for word, 'd' for dword.
* For labels, the modifier defines the type and distance of the operand from the invoking statement: 'b' for a near label within in a small interval surrounding the invoking statement (-128 to 127 bytes), 'w' for a near label that's outside this interval, and 'd' for a far label.
* For constants, the modifier defines the size of the constant: 'b' for -256 to 255, 'w' for constants outside this range but still between -65536 and 65535.

3.10.3 Range Specifiers

The optional range specifier for a codemacro's parameter defines even more requirements that an invoking statement's operand must meet if the statement is to match the codemacro. A range specifier consists of an expression, or a pair of expressions separated by a comma, that is surrounded by parentheses. Each expression must be a register or must evaluate to a constant number.

If a register or pair of registers is specified, an operand will match the parameter only if it is one of the range's registers. If a single constant is specified as the range, an operand must have that value in order to match the parameter. If a pair of constants is specified as the range, an operand must have a number in that interval in order to match the parameter.

For example, here are the first lines of three codemacros that use the range specifier:

```
codemacro   IN    dst:Aw, port:Rw(DX)
codemacro   ROR   dst:Ew, count:Rb(CL)
codemacro   ESC   opcode:Db(0,63), adds:Eb
```

The first *codemacro* directive begins one of the IN instruction's codemacros. For an IN statement to match this codemacro, the instruction's second operand must be the DX register.

The second *codemacro* directive begins one of the ROR instruction's codemacros. For an ROR instruction to match this codemacro, the instruction's second operand must be the CL register.

The third *codemacro* directive begins the ESC codemacro. For an instruction to match this codemacro, the instruction's first operand must be a constant expression whose value is between 0 and 63.

3.10.4 The Codemacro Directives

3.10.4.1 The SEGFIX directive

The *segfix* directive has the form:

```
segfix  param_name
```

where *param_name* is the name of a codemacro parameter. This parameter must specify a memory address; that is, its specifier must be E, M, or X.

When the assembler is generating code for an instruction, *segfix* causes the assembler to determine whether the memory operand corresponding to *param_name* can be accessed using the instruction's default, hardware-selected segment register. If not, the assembler will output a segment-override prefix as the first byte of the instruction. If the prefix isn't needed, the assembler won't output it.

The assembler knows that when a segment-override prefix isn't specified in an instruction, the 8086 hardware uses the SS segment register to access the instruction's memory operand if the memory operand uses the BP base register; otherwise, it uses the DS segment register.

The assembler decides which segment register is needed to access an instruction's memory operand as follows:

* If the operand specifies the segment register to use, using the segment selector operator, :, then of course that's the segment register that is needed.
* Otherwise, if the operand contains a variable or label name, then the needed segment register is the one that points to the segment containing the variable or label, as defined by the *assume* directive.

* Otherwise, if the operand uses the BP base register, then SS is needed.
* Otherwise, DS is needed.

3.10.4.2 The NOSEGFIX Directive

The *nosegfix* directive has the form:

```
nosegfix    segreg, param_name
```

where *segreg* is one of the segment register and *param_name* is the name of a codemacro parameter that has a memory address specifier (that is, its specifier is E, M, X).

When the assembler is generating code for an instruction, the *nosegfix* directive causes it to verify that the actual operand corresponding to *param_name* can be accessed using the *segreg* segment register. If not, the assembler reports an error.

nosegfix is used in instructions, such as CMPS, MOVS, SCAS, and STOS, where a memory operand can only be accessed by the ES segment register.

3.10.4.3 The MODRM Directive

The *modrm* directive causes the assembler to generate the ModRM byte for an instruction. If the instruction calls for an 8- or 16-bit displacement, *modrm* generates that as well. The *modrm* directive has the form

```
modrm regfld, modrmfld
```

where *regfld* defines the contents of the ModRM byte's reg field, and *modrmfld* defines the contents of the ModRM byte's mod and rm fields and, when necessary, the contents of the displacement bytes.

regfld can be either an absolute number or the name of one of the codemacro's parameters. If it's a number, that same value is always set in the instruction's reg field when the instruction is used. If it's a parameter name, then the corresponding actual parameter, which is usually a register number, is set in the instruction's reg field. An instruction can specify a register by its name, of course; the assembler will place the register's number in the modrm byte's reg field.

modrmfld is the name of a codemacro parameter. When the assembler is generating code for an instruction, it determines whether the actual parameter that corresponds to the *modrmfld* parameter is a register, variable, or indexed variable and constructs the instruction's mod and r/m fields. If the operand also needs an 8- or 16-bit displacement, the assembler generates that, too.

As an example of a codemacro that uses *modrm*, here is one of the codemacros for the *mov* instruction:

```
codemacro MOV dst:Rw, src:Ew
segfix src
db 8bh
modrm dst, src
endm
```

Because of the specifiers on the two parameters, this codemacro will match those MOV instructions whose first operand is a 16-bit general register, and whose second operand designates a 16-bit register or memory location.

Continuing with this example, the instruction

```
mov dx, [bx][si]
```

will match this codemacro, generating the two bytes

```
10001101 10010000
```

The first byte is the instruction code for moving a 16-bit value from memory or register into a register. The second byte is the ModRM byte, with the destination register, DX, encoded as 010 in bits 3-5, a Mod field of 10 in bits 6 and 7, and an RM field of 000 in bits 0-2.

3.10.4.4 The RELB and RELW Directives

The *relb* and *relw* directives are used in the codemacros for call and jump instructions. They tell the assembler to output an 8- or 16-bit displacement, respectively, from the end of the instruction being processed to the label specified in the instruction.

The directives have the form:

```
relb    param_name
relw    param_name
```

where *param_name* is the name of a codemacro parameter that has a 'Cb' or 'Cw' specifier, respectively.

For example, here are two codemacros that use *relb* and *relw*:

```
codemacro JMP place:Cw
db 0e9h
relw place
endm

codemacro JE place:Cb
db 74h
relb place
endm
```

3.10.4.5 The DB, DW, and DD Directives

When the assembler encounters a *db*, *dw*, and *dd* directive as it is generating code for an instruction, it generates a byte, word, or doubleword, respectively.

The directives have the form:

```
db      cmac_expr
dw      cmac_expr
dd      cmac_expr
```

where *cmac_expr* is either an absolute number, without forward references; the name of a codemacro parameter; or the name of a codemacro parameter with a dot-recordfield shift operator.

Unlike the use of these directives outside a codemacro, these directives when used within a codemacro cannot specify a list of values separated by commas, and cannot use the DUP construct.

3.10.4.6 User-defined Record Directives

The record initialization directive has the form

```
recname <p1, p2, ...>
```

where *recname* is the name of a record that was previously defined using the *record* directive (see the Directives section), and *p1*, *p2*, ... are operands.

When the assembler encounters a user-defined record directive while generating code for an instruction, it will put together and output a byte or word (depending on the record definition) from the values in the record's operand list.

For example, here is the definition of the record *r53*, and its use in the codemacro for the version of the *dec* instruction that decrements a 16-bit general register:

```
R53 Record RF1:5,RF2:3
codemacro dec dst:rw
   r53  <01001b, dst>
endm
```

When the assembler encounters a *dec* instruction that matches this codemacro, it will output a byte whose most significant 5 bits are 01001 and whose least significant 3 bits are the number of the register that is specified as the operand of the instruction.

3.10.5 Using the Dot operator to Shift Parameters

There is a special operator, the dot operator, that can be used in a codemacro definition on the operands of the *db*, *dw*, *dd*, and record initialization directives. The operator has the form

```
param_name.field_name
```

where *param_name* is the name of codemacro parameter whose corresponding operand will be an absolute number. *field_name* is the name of a record field.

When the assembler encounters an expression containing a dot operator as it is generating code for an instruction, it shifts the operand corrresponding to *param_name* to the right, using the shift count defined by *field_name*.

The shift count for a record field name is the number of bits between the field and the least significant bit of its record. For example, with the record definition

```
R233  record  rf6:2, mid3:3, rf7:3
```

the shift counts of the fields *rf6*, *mid3*, and *rf7* are 6, 3, and 0, respectively.

The dot-shift operator is used in the codemacros for the ESC instruction. Here is one of them:

```
R53 Record RF1:5,RF2:3
R233 Record RF6:2, Mid3:3, RF7:3

codemacro esc opcode:db(0,63), addr:Eb
        segfix  addr
        r53     <11011b, opcode.mid3>
        modrm opcode, addr
endm
```

When the assembler encounters an *esc* instruction that matches this codemacro, it will output a byte for the *r53* directive whose most significant five bits are 11011 and whose least significant 3 bits are the value of the actual *esc* operand, shifted right 3 bits.

3.10.6 The PROCLEN Symbol

The special symbol *proclen* equals 0 within a *near proc* and 0ffh inside a *far proc*. It also equals 0 outside of PROC ... *endp* blocks. This symbol is used by the codemacros for the *ret* instructions to generate the correct machine instruction to return from a *call* to a *near* or *far proc*.

For example, the codemacro for the version of the *ret* instruction that doesn't add a value to the stack is :

```
R413 Record RF8:4, RF9:1, RF10:3

codemacro Ret
        r413    <0ch, proclen, 3>
endm
```

When the assembler encounters a parameter-less *ret* instruction it will output one byte whose most significant four bits are 1100 and whose least significant three bits are 011. The bit in the middle will be 0 if the instruction is in a *near proc* and 1 if it is in a *far proc*.

3.10.7 Matching Instructions to Codemacros

When the assembler encounters an instruction, it searches the codemacros that have that name, beginning with the last one defined, looking for one for which the number of codemacro and instruction parameters are the same, and for which the attributes of each of the instruction's and codemacro's parameters match.

The following rules are used to decide if an instruction parameter matches a codemacro parameter:

Specifiers

* In pass 1, a forward reference matches C, D, E, M, X.
* AX and AL match A, E, R.
* A label matches C.
* A number matches D.
* A non-indexed variable matches E, M, X.
* An indexed variable or register expression matches E, R.
* A segment register matches S.

Modifiers

Once an instruction's parameter has matched the specifier of a codemacro's parameter, an attempt is made to match the instruction's parameter to the modifier of the codemacro's parameter. Modifier matching depends on the type of the instruction's parameter.

For a number:

* A number between -256 and 255 matches 'b' only.
* Other numbers match 'w' only.

For a label:

* A *near* label that is accessable from the current contents of the CS segment register (as defined by the *assume* directive) and that is in the range -126 to +129 from the beginning of the instruction matches only 'b'. An expression that is explicitly typed using the *short* operator also matches only 'b'.
* Another *near* label that is accessable from the current contents of CS matches only 'w'.
* A *far* label matches 'd'.

For a variable:

* A variable of type *byte* matches 'b'.
* A variable of type *word* matches 'w'.

* A variable of type *dword* matches 'd'.

A forward reference matches any modifier, except when typing information is specified in the instruction's operand (eg, *byte ptr*).

If an index-register expression is used as an operand in a multi-operand instruction and its type can't be determined from the expression itself (eg, [bx]), the operand type will be determined from that of the other operand, if possible. In this case, if the other operand is a number, the operand being matched will match only 'w'. If the instruction contains just a single operand, it matches no modifier.

Ranges

Range specifiers are legal only for number and register parameters (that is, whose specifier is A, D, R, or S). If the range is just a single value, an actual operand must be that value in order to match. If the range is a pair of values, the actual operand must be the specified range in order to match. Forward references do not match a codemacro's parameter if the parameter contains a range specifier.

As an example of codemacro matching, consider the 8086 *add* instruction: there are eleven *add* codemacros, which match the following combinations of source and destination operands:

	Destination	*Source*
1.	memory byte	immediate byte
2.	memory word	immediate byte (not between -128 & 12
3.	memory word	immediate byte (from -128 to 127)
4.	memory word	immediate word
5.	AL	immediate byte
6.	AX	immediate byte
7.	AX	immediate word
8.	memory byte or byte register	byte register
9.	memory word or word register	word register
10.	byte register	memory byte or byte register
11.	word register	memory word or word register

Here are the first lines for each of the *add* codemacro definitions, which correspond to the above operand list:

```
CodeMacro Add dst:Eb, src:Db
CodeMacro Add dst:Ew, src:Db
CodeMacro Add dst:Ew, src:Db(-128,127)
CodeMacro Add dst:Ew, src:Dw
CodeMacro Add dst:Ab, src:Db
CodeMacro Add dst:Aw, src:Db
CodeMacro Add dst:Aw, src:Dw
CodeMacro Add dst:Eb, src:Rb
CodeMacro Add dst:Ew, src:Rw
CodeMacro Add dst:Rb, src:Eb
CodeMacro Add dst:Rw, src:Ew
```

When the assembler is trying to find a codemacro that matches an *add* instruction, it begins with the last *add* codemacro defined, number 11, and searches backwards. Thus, the ordering of codemacros for an instruction is very important.

Here are some examples on the matching of actual *add* instructions to the *add* codemacros. For these examples, assume that *bvar* and *wvar* are byte and word variables, respectively:

```
add     ax,250  ;matches add codemacro # 6
add     ax,300  ;#7
add     bx,wvar;#11
add     bx,dx   ;#11
add     bvar,al ;#8
add     bvar,254        ;#1
add     wvar,cx;#9
add     dh,bvar[si]     ;#10
add     al,3    ;#5
add     wvar,35648      ;#4
add     [bx][si],ah     ;#8
add     wvar,255        ;#2
```

THE LINKER

Chapter Contents

The Linker ..... ln
1. Introduction to linking 3
2. Using the Linker 7
3. Linker Options 9
4. Linker Error Messages 17

The Linker

The Manx linker has two functions:

* It ties together the pieces of a program which have been compiled and assembled separately;
* It converts the linked pieces to a format which can be loaded and executed.

The pieces must have been created by the Manx assembler.

The first section of this chapter presents a brief introduction to linking and what the linker does. If you have had previous experience with linkage editors, you may wish to continue reading with the second section, entitled "Using the Linker." There you will find a concise description of the command format for the linker.

1. Introduction to linking

Relocatable Object Files

The object code produced by the assembler is "relocatable" because it can be loaded anywhere in memory. One task of the linker is to assign specific addresses to the parts of the program. This tells the operating system where to load the program when it is run.

Linking hello.o

It is very unusual for a C program to consist of a single, self-contained module. Let's consider a simple program which prints "hello, world" using the function, *printf*. The terminology here is precise; *printf* is a function and not an intrinsic feature of the language. It is a function which you might have written, but it already happens to be provided in the file, *c.lib*. This file is a library of all the standard i/o functions. It also contains many support routines which are called in the code generated by the compiler. These routines aid in integer arithmetic, operating system support, etc.

When the linker sees that a call to *printf* was made, it pulls the function from the library and combines it with the "hello, world" program. The link command would look like this:

```
ln hello.o c.lib
```

When *hello.c* was compiled, calls were made to some invisible support functions in the library. So linking without the standard library will cause some unfamiliar symbols to be undefined. All programs will need to be linked with *c.lib*.

The Linking Process

Since the standard library contains only a limited number of general purpose functions, all but the most trivial programs are certain to call user-defined functions. It is up to the linker to connect a function call with the definition of the function somewhere in the code.

In the example given below, the linker will find two function calls in file 1. The reference to *func1* is "resolved" when the definition of *func1* is found in the same file. The following command

```
ln file1.o c.lib
```

will cause an error indicating that *func2* is an undefined symbol. The reason is that the definition of *func2* is in another file, namely *file2.o*. The linkage has to include this file in order to be successful:

```
ln file1.o file2.o c.lib
```

file 1

```
main()
{
  func1();
  func2();
}

func1()
{
  return;
}
```

file 2

```
func2()
{
return;
}
```

Libraries

A library is a collection of object files put together by a librarian. Libraries intended for use with *ln* must be built with the Manx librarian, *lb*. This utility is described in the Utility Programs chapter.

All the object files specified to the linker will be "pulled into" the linkage; they are automatically included in the final executable file. However, when a library is encountered, it is searched. Only those modules in the library which satisfy a previous function call are pulled in.

For Example

Consider the "hello, world" example. Having looked at the module, *hello.o*, the linker has built a list of undefined symbols. This list includes all the global symbols that have been referenced but not defined. Global variables and all function names are considered to be global symbols.

The list of undefined's for *hello.o* includes the symbol *printf*. When the linker reaches the standard library, this is one of the symbols it

will be looking for. It will discover that *printf* is defined in a library module whose name also happens to be *printf*. (There is not any necessary relation between the name of a library module and the functions defined within it.)

The linker pulls in the *printf* module in order to resolve the reference to the *printf* function.

Files are examined in the order in which they are specified on the command line. So the following linkages are equivalent:

```
ln hello.o

ln c.lib hello.o
```

Since no symbols are undefined when the linker searches *c.lib* in the second line, no modules are pulled in. It is good practice to leave all libraries at the end of the command line, with the standard library last of all.

The Order of Library Modules

For the same reason, the order of the modules within a library is significant. The linker searches a library once, from beginning to end. If a module is pulled in at any point, and that module introduces a new undefined symbol, then that symbol is added to the running list of undefined's. The linker will not search the library twice to resolve any references which remain unresolved. A common error lies in the following situation:

module of program	*references (function calls)*
main.o	getinput, do_calc
input.o	gets
calc.o	put_value
output.o	printf

Suppose we build a library to hold the last three modules of this program. Then our link step will look like this:

```
ln main.o proglib.lib c.lib
```

But it is important that *proglib.lib* is built in the right order. Let's assume that *main()* calls two functions, *getinput()* and *do_calc()*. *getinput()* is defined in the module *input.o*. It in turn calls the standard library function *gets()*. *do_calc()* is in *calc.o* and calls *put_value()*. *put_value()* is in *output.o* and calls *printf()*.

What happens at link time if *proglib.lib* is built as follows?

```
proglib.lib:    input.o
                output.o
                calc.o
```

After *main.o*, the linker has *getinput* and *do_calc* undefined (as well as some other support functions in *c.lib*). Then it begins the search of

proglib.lib. It looks at the library module, *input,* first. Since that module defines *getinput,* that symbol is taken off the list of undefined's. But *gets* is added to it.

The symbols *do_calc* and *gets* are undefined when the linker examines the module, *output.* Since neither of these symbols are defined there, that module is ignored. In the next module, *calc,* the reference to *do_calc* is resolved but *put_value* is a new undefined symbol.

The linker still has *gets* and *put_value* undefined. It then moves on to *c.lib,* where *gets* is resolved. But the call to *put_value* is never satisfied. The error from the linker will look like this:

Undefined symbol: put_value_

This means that the module defining *put_value* was not pulled into the linkage. The reason, as we saw, was that *put_value* was not an undefined symbol when the *output* module was passed over. This problem would not occur with the library built this way:

proglib.lib:	input.o
	calc.o
	output.o

The standard libraries were put together with much care so that this kind of problem would not arise.

Occasionally it becomes difficult or impossible to build a library so that all references are resolved. In the example, the problem could be solved with the following command:

ln main.o proglib.lib proglib.lib c.lib

The second time through *proglib.lib,* the linker will pull in the module *output.* The reason this is not the most satisfactory solution is that the linker has to search the library twice; this will lengthen the time needed to link.

2. Using the Linker

The general form of a linkage is as follows:

ln [-options] file1.o [file2.o etc] [lib1.lib etc]

The linker combines object modules produced by the Manx assembler into an executable program. It can search libraries of object modules for functions needed to complete the linkage; including just the needed modules in the executable program. The linker makes just a single pass through a library, so that only forward references within a library will be resolved.

The executable file

The linker can create both *.exe* and *.com* files for PCDOS and MSDOS. It creates *.cmd* files for CP/M-86.

The name of the executable output file can be selected using the -O linker option. If this option isn't used, the linker will derive the name of the output file from that of the first object file listed on the command line, by changing its extension to *.exe* on MSDOS and PCDOS, and to *.cmd* on CP/M-86. In the default case, the executable file will be located in the same area as the first object file. (an "area" is a directory on a drive, on MSDOS and PCDOS, and is a user area on a drive, on CP/M-86). For example,

ln prog.o c.lib

will produce the disk file *prog.exe,* on MSDOS and PCDOS, and the file *prog.cmd* on CP/M-86. The standard library, *c.lib,* will have to be included in most linkages.

A different output file can be specified with the -O option, as in the following command:

ln -o program.com mod1.o mod2.o c.lib

This command also shows how several individual modules can be linked together. A "module", in this sense, is a section of a program containing a limited number of functions, usually related. These modules are compiled and assembled separately and linked together to produce an executable file.

Libraries

Several libraries of object modules are provided with Aztec C86. The most frequently-used of these are *c.lib*, which contains the non-floating point functions and which use the 'small code' and 'small data' memory model; and *m.lib*, which contains the floating point functions, which perform the operations using software routines, and which use the 'small code' and 'small data' memory model. Other libraries are provided with some versions of Aztec C86; for their description, see the Libraries section of the Technical Information chapter.

All programs must be linked with one of the versions of *c.lib*. In addition to containing all the non-floating point functions described in the Functions chapter, it contains internal functions which are called by compiler-generated code, such as functions to perform long arithmetic.

Programs that perform floating point operations must be linked with one of the versions of *m.lib*, in addition to a version of *c.lib*. The floating point library must be specified on the linker command line before *c.lib*.

Libraries of user modules can also be searched by the linker. These are created with the Manx *lb* program, and must be listed on the linker command line before the Manx libraries.

For example, the following links the module *program.o*, searching the libraries *mylib.lib*, *new.lib*, *m.lib*, and *c.lib* for needed modules:

```
ln program.o mylib.lib new.lib m.lib c.lib
```

Each of the libraries will be searched once in the order in which they appear on the command line.

Libraries can be conveniently specified using the *-L* option. For example, the following command is equivalent to the following:

```
ln -o program.o -lmylib -lnew -lm -lc
```

For more information, see the description of the *-L* option in the Options section of this chapter.

3. Linker Options

3.1 Summary of options

3.1.1 General Purpose Options

-O file Write executable code to the file named *file.*

-Lname Search the library *name.lib* for needed modules.

-F file Read command arguments from *file.*

-T Generate a symbol table file.

-M Don't issue warning messages.

-N Don't abort if there are undefined symbols.

-S size Tell DOS not to load the program unless at least *size* bytes is available for its stack and heap. *size* is a hex value.

-X size Tell DOS to allocate memory to the program so that the program doesn't have more than *size* paragraphs (16-byte blocks) for its stack and heap. *size* is a hex value. Only valid *.exe* programs running on DOS 2.0 or later.

-V Be verbose.

3.1.2 Options for Segment Address Specification

-B addr When linking a DOS *.com* file, set the program's base address to the hex value *addr.*

-C addr When linking an *.exe* program that will be burned into ROM, set the starting paragraph number of the program's code segments to the hex value *addr.*

When linking a *.com* program, set the starting offset of the program's code segment from the beginning of the physical segment containing the program to the hex value *addr.*

-D addr When linking an *.exe* program that will be burned into ROM, set the starting paragraph number of the program's data segments to the hex value *addr.*

When linking a DOS *.com* file, set the starting offset of the program's initialized data segment to the hex value *addr.*

-U addr When linking a DOS *.com* file, set the starting offset of the program's uninitialized data segment to the hex value *addr.*

3.1.3 Options for Overlay Usage

-R Create a symbol table to be used when linking overlays.

+C size Reserve *size* bytes at end of the program's code segment (the overlay's code segment is loaded here). *size* is a hex value.

+D size Reserve *size* bytes at end of the program's initialized and uninitialized data segments (the overlay's data is loaded here). *size* is a hex value.

3.2 Detailed description of the options

3.2.1 General Purpose Options:

The -O option

The *-O* option can be used to specify the name of the file to which the linker is to write the executable program. The name of this file is in the parameter that follows the *-O*. For example, the following command writes the executable program to the file *prog.com*:

```
ln -o prog.com prog.o c.lib
```

If this option isn't used, the linker derives the name of the executable file from that of the first input file, by changing its extension to *.exe* on DOS and to *.cmd* on CP/M-86.

The linker decides what type of executable program to create, based on the extension of the file to which it is written. Thus, if you're creating a program that will run on CP/M-86, the extension of the executable file must be *.cmd*. And if you're creating a DOS program, its extension must be *.exe* or *.com*.

The -L option

The *-L* option provides a convenient means of specifying to the linker a library that it should search, when the extension of the library is *.lib*.

On DOS, the name of the library is derived by concatenating the value of the environment variable *CLIB*, the letters that immediately follow the *-L* option, and the string *.lib*. For example, with the libraries *subs.lib*, *io.lib*, *m.lib*, and *c.lib* in a directory specified by *CLIB*, you can link the module *prog.o*, and have the linker search the libraries for needed modules by entering

```
ln prog.o -lsubs -lio -lm -lc
```

CLIB is set using the DOS *set* command. For example, the first command that follows sets *CLIB* when the libraries are in the root directory on the *c:* drive; the second sets it when they are in the directory *\cc\libs* on the default drive, and the third sets it when they are in the directory *libs* on the *d:* drive:

```
set CLIB=c:
set CLIB=\cc\libs\
set CLIB=d:\libs\
```

Note the terminating backslash on the *CLIB* variable when the libraries are not in a root directory. This is required since the linker simply prepends the value of the *CLIB* variable to the *-L* string.

On CP/M-86, the linker derives the name of the file containing a library that is specified in a *-L* option by appending *.lib* to the string that immediately follows the *-L*.

The -F option

-F file causes the linker to merge the contents of the given file with the command line arguments. For example, the following command causes the linker to create an executable program in the file *myprog.exe* (on DOS) or *myprog.cmd* (on CP/M-86). The linker includes the modules *myprog.o*, *mod1.o*, and *mod2.o* in the program, and searches the libraries *mylib.lib* and *c.lib* for needed modules.

```
ln myprog.o -f argfil c.lib
```

where the file *argfil*, contains the following:

```
mod1.o mod2.o
mylib.lib
```

The linker arguments in *argfile* can be separated by tabs, spaces, or newlines.

There are several uses for the *-F* option. The most obvious is to supply the names of modules that are frequently linked together. Since all the modules named are automatically pulled into the linkage, the linker does not spend any time in searching, as with a library. Furthermore, any linker option except *-F* can be given in a *-F* file. *-F* can appear on the command line more than once, and in any order. The arguments are processed in the order in which they are read, as always.

The -T option

The *-T* option creates a disk file which contains a symbol table for the linkage. This file is just a text file which lists each symbol with a hexadecimal address. This address is either the entry point for a function or the location in memory of a data item. A perusal of this file will indicate which functions were actually included in the program.

The symbol table file will have the same name as that of the file containing the executable program, with extension changed to *.sym*. This file can be used in conjunction with the Manx *db* debugger or with the Digital Research debugger, SID86.

There are several special symbols which will appear in the table. They are defined in the Program Organization section of the Technical Information chapter.

The -M option

The linker issues the message "multiply defined symbol" when it finds a symbol that is defined with the assembly language directives *global* or *public* in more than one module. The *-M* option causes the linker to suppress this message unless the symbol is defined in more than one *public* directive.

To maintain compatibility with previous versions of Aztec C, the linker will generate code for a variable that is defined in multiple *global* statements and in at most one *public* statement, and also issue the "multiply defined symbol" message. Thus, if you use the *global* and *public* directives in this way, and don't want to get this message, use the *-M* option to suppress them.

The definition of a symbol in more than one *public* directive is never valid, so the *-M* option doesn't suppress messages in this case.

For more information, see the discussion on global symbols in the Programmer Information sections of the Compiler and Assembler chapters.

The -N option

Normally, the linker halts without generating an executable program if there are undefined symbols; that is, symbols that are defined in one module using the assembly language *extrn* directive but that aren't defined in another module using a *global* or *public* directive.

The *-N* option causes the linker to go ahead and generate an executable program anyway.

The -S option

The *-S size* option can be used when linking programs that will run on PCDOS or MSDOS, to tell DOS to load the program only if there is enough memory for the program to have a stack and heap whose combined size is at least *size* bytes, where *size* is a hex value.

If this option isn't specified, the size defaults to 4K bytes.

This option is provided for compatibility with earlier versions of Aztec C, and is not useful now: the global variables *__STKSIZ* and *__HEAPSIZ* define the sizes of these areas; if the startup routine is not able to give the program the requested space for its stack and heap, it will halt the program.

The -X option

The *-X size* option can be used when linking *.exe* programs that will run on PCDOS or MSDOS 2.0 or later, to tell DOS to allocate memory to the program such that it has at most *size* paragraphs (16-byte blocks) available for its stack and heap. *size* is a hex value.

If this option isn't specified, the size defaults to a huge value.

When a DOS program starts, the startup routine will allocate as much memory to the program as it needs and can use, and frees the rest of memory. If the program's stack is below its heap, then the *-X* option is not useful at all, because the size of the program's heap will grow automatically to satisfy program requests for dynamically-allocated buffers.

If the program's stack is above its heap, the *-X* option may be used to create a program whose allocated memory is less than the maximum allowed amount.

For more information, see the Program Organization section of the Technical Information chapter.

The -V option

The *-V* option causes the linker to send a progress report of the linkage to the screen as each input file is processed. This is useful in tracking down undefined symbols and other errors which may occur while linking.

3.2.2 Options for segment address specification

The linker organizes a program into three areas: code, initialized data, and uninitialized data areas. the following paragraphs discuss the positioning of these areas using the linker's *-C*, *-D*, *-U*, and *-B* options.

For more information on a program's areas, see the Program Organization section of the Technical Information chapter.

3.2.2.1 Segment specification for .exe programs

An *.exe* program that is created by the linker has some fields containing long pointers whose values must be adjusted when the starting addresses of the program's segments are known. Normally, DOS determines these addresses and adjusts the long pointer fields when it loads the program, using information that the linker sets in the beginning of the *.exe* file.

If the program is to be burned into ROM, the linker itself must adjust the long pointer fields. For it to do this, you must tell it the starting addresses of the program's code and data areas, using the *-C codebgn* and *-D databgn* options. *codebgn* and *databgn* are the starting paragraph numbers, in hex, of the program's code and data areas, respectively.

For example, if the program *prog.exe* is to be burned into ROM, and its code and data are to start at paragraphs 0xf000 and 0, respectively, then the command to link it could be

```
ln -C f000 -D 0 -o prog.exe srom.o prog.o -lc
```

For more information on generating ROMable code, see the Technical Information chapter.

3.2.2.2 Segment specification for .com programs

When you create a *.com* program, the program will occupy a single physical segment and won't contain any fields that need adjustment when the program is loaded. You normally won't need to specify the location of the program's areas within the physical segment, but if you do, the offset of the start of the logical code, initialized data, and

uninitialized data segments from the beginning of the physical segment containing them can be specified by the -C, -D, and -U linker options, respectively. A fourth linker option, -B, will set the "base address" of the program. These options are followed by the desired offset, in hex.

By default, the base address is at 0x100, the logical code segment starts at 0x103, the initialized data follows the code, and the uninitialized data follows the initialized data.

A *.com* file contains a memory image of the program, from its base address through the end of its code or initialized data segments (whichever is higher). This image is loaded into its physical segment, with the first byte in the file loaded at the offset specified by the base address.

When the program is to be loaded by DOS, the base address must be 0x100; the DOS loader simply loads the contents of the *.com* file into the program's physical segment, with the first byte in the file loaded at offset 0x100, and transfers control to 0x100.

The program is expected to begin execution at its base address. Most programs have a startup routine, which performs initialization activities and then calls the program's *main* function. This startup routine is usually somewhere in the middle of the program, so at the base address the linker will normally set a near jump instruction to the startup routine.

You can explicitly specify that a label in a module is the beginning of a startup routine by placing the label in the operand field of the module's assembly language *end* directive. For example, the *sbegin* module in *c.lib* contains the function *$begin*. This label is declared in a *public* directive and also in the module's *end* directive. When a C module is compiled, the compiler always generates a reference to *$begin*; thus, when the program is linked, *ln* will include the *sbegin* module from *c.lib* and place a jump to it in the first byte of the program's *.com* file (ie, at its base address).

If the linker doesn't find a startup routine when it links a program, it won't set the jump instruction at the program's base address. In this case, if you don't specify a starting offset for the program's code segment, it will begin right at the base address.

For example, the following command sets the base address of *prog.com* to 0x500:

```
ln -b 500 -o prog.com prog.o -lc
```

Because none of the other segment selection options were used in this example, the program's code will begin at offset 0x503, followed by its initialized data, followed by its uninitialized data.

In the next example, the program's base address is set to 0x200, the offset of its code, initialized data, and uninitialized data segments to

0x500, 0x1000, and 0x3000, respectively:

```
ln -b 200 -c 500 -d 1000 -u 3000 prog.o -lc
```

3.2.3 Options for Overlay Usage

The *-R* option causes the linker to generate a file containing the symbol table. It's used when linking a program which calls overlays.

The name of the symbol table file is derived from that of the executable file by changing the extension to *.rsm*. The file is placed in the same area as the executable file.

The *+C* and *+D* options effectively increase the size of the code and data segments of the linked program. For example,

```
ln +c 3000 +d 1000 prog.o -lc
```

will reserve 0x3000 bytes in the code segment and 0x1000 bytes in the data segment for overlays. See the Technical Information chapter for more details.

4. Linker Error Messages

4.1 Summary of Error Messages

4.1.1 Command line errors:

1. unknown option '<bad option letter>'
2. too few arguments in command line.
3. No input given!
4. Cannot have nested -f options.
5. too few arguments in -f file: <filename>
6. multiple <origin> declarations, last one used.

4.1.2 I/O errors:

1. can't open <filename>, err=<error number>
2. Cannot open -f file: <filename>
3. I/O error (<error number>) reading/writing output file
4. Cannot write output file
5. Cannot create output file: <filename>
6. Cannot create symbol table output
7. Cannot create overlay symbol table output

4.1.3 Corrupted object files:

1. object file is bad!
2. invalid operator in evaluate <hex value>
3. library format is invalid!
4. Cannot read module from <input> on pass2
5. can't find symbol, <symbol name>, on pass two
6. <filename> is not a rel file!

4.1.4 Errors in use of Memory:

1. Insufficient memory!
2. Too many symbols!
3. -C or -D value less than base address
4. Code and data regions overlap

4.1.5 Errors arising from source code:

1. Undefined symbol: <symbol name>

2. <symbol name> multiply defined
3. pass1(<hex value>) and pass2(<hex value>) values differ:
4. symbol type differs on pass two: <symbol name>
5. Attempt to Initialize Data in Root
6. undefined COMMON <symbol name>

4.2 Description of Linker Error Messages

When invoked, the linker processes the arguments given it, performs the linkage requested, and generates an executable output file on the disk. The first line to appear on the screen is a banner which indicates that the linker has been loaded and is running. If everything goes well, the base address message will follow and the linker will finish. The linker may encounter an error while it is running, in which case it will send a message to the screen.

Errors may be reported at a variety of points during the linking process. *ln* does its work in two stages, known as pass 1 and pass 2. The base address message is printed at the end of pass 1, so any errors occurring after that have been detected during pass 2 of the linker.

Following is a list of the messages which the linker will generate in response to an error. The messages are grouped according to the source of the errors which cause them. Elements which are variable are enclosed by angled brackets: <>.

4.2.1 Command line errors:

1. unknown option '<bad option letter>'

An option letter has been specified which the linker does not recognize. Only the letter will be ignored; everything else on the command line has been preserved, and the linker will try to execute what it has interpreted. See the Options section of this chapter for a list of options which are supported.

2. too few arguments in command line.

Several of the linker options have an associated value or name, such as -B 2000. If a needed value is missing, the linker will give this message and die.

3. No input given!

The linker will quit immediately if not given any input to process.

4. Cannot have nested -f options.

A file which is given as a -f argument can contain any option letter except -f itself. However, more than one -f is allowed on a command line.

5. too few arguments in -f file: <filename>

An option letter specified in the file, "filename," requires a value or name to follow it. If an option appears at the end of the file, its associated value may not appear back on the command line.

6. multiple <origin> declarations, last one used.

The message will specify that one of the segment address selection options, -C, -D, or -U, was specified more than once in the command

line:

The linker will use the last value specified for a segment address.

4.2.2 I/O errors:

1. can't open <filename>, err=<errno>

If any file in the command line cannot be opened, this message will be sent to the screen, specifying the filename and the current value of *errno*.

2. Cannot open -f file: <filename>

A file given with the -f option cannot be opened.

3. I/O error (<errno>) reading/writing output file

An error reading or writing the output file probably means there is no more disk space available. In particular, a block of the output file was written to disk and then could not be read back. The current value of *errno* is given in these messages.

4. Cannot write output file

The description of the previous message applies to this one, too.

5. Cannot create output file: <filename>

This message usually indicates that all available directory space on the disk has been exhausted.

6. Cannot create symbol table output

The -T option was given in the command line, but the file containing the linkage symbol table cannot be written to disk. It is possible that there is no more space on the disk.

7. Cannot create overlay symbol table output

Occurs when using the -R option. The file containing the overlay symbol table cannot be written to disk.

4.2.3 Corrupted object files:

1. object file is bad!

This is the most explicit indication that an object file in the linkage has been corrupted. The solution is simply to recompile and assemble the source file.

2. invalid operator in evaluate <hex value>

This is really the same as the previous error message. Unless you have changed the object code by hand, the file has been corrupted.

3. library format is invalid!

A library in the linkage has been corrupted.

4. Cannot read module from <input> on pass2

Indicates that a module has been corrupted between pass 1 and pass 2. On a multiuser system, it is possible that another user changed the file while the linker was running. Otherwise, the error was probably due to a hardware failure.

5. can't find symbol, <symbol name>, on pass two

Same as the previous message.

6. <filename> is not a rel file!

A file given to the linker does not contain relocatable object code which *ln* can process. For instance, a source file may have been included in the link.

4.2.4 Errors in use of memory:

1. Insufficient memory!

The linkage process needs memory space for *ln*, global and local symbol tables, and approximately 5K for buffers. Just as with compilation, most memory use is devoted to the program software and symbol tables. Since *ln* is not especially large, only an extremely complicated linkage might run out of memory.

2. Too many symbols!

This is another way of saying that not enough memory was available for the symbol tables needed for the linkage.

3. -C or -D value less than base address

It is not possible for the starting address of the code or data to be less than the base address of the program, which is specified by the option, -B.

4. Code and Data Regions Overlap

By default, data resides above the code area in memory. The starting addresses of both areas must be spaced far enough apart to accommodate all the code. If the -D option is used to begin the data area in the middle of the code, this error message will be put out.

4.2.5 Errors arising from source code:

1. Undefined symbol: <symbol name>

A global symbol name has remained undefined. This is commonly a function which has been referenced in the source code but not included anywhere in the link.

2. <symbol name> multiply defined

A global symbol has been defined more than once. For instance, if two functions are accidentally given the same name, this message will be generated.

3. pass1(<hex value>) and pass2(<hex value>) values differ:

This message may be generated during pass 2 following a 'multiply defined' message in pass 1.

4. symbol type differs on pass two: <symbol name>

Same as the previous message.

5. Attempt to Initialize Data in Root

On the source code level, this means that a global symbol was defined in the root of an overlay and then initialized in an overlay module. For example,

```
root:                overlay:
int i;               int i = 3;
```

The problem arises because the initialization is performed by the linker, but the variable to be initialized is in an entirely different file.

The situation which follows is valid because the assignment statement is evaluated at run time:

```
root:                overlay:
int i;               int i;
                     function()
                     {
                        i = 3;
                     }
```

6. undefined COMMON <symbol name>

This error now occurs only in reference to the user's own assembly language routines. It is generated by a COMMON block of size zero.

UTILITY PROGRAMS

Chapter Contents

Utility Programs ..... util
arcv (Source dearchiver) 4
cnm (Object file utility) 5
crc (File verificator) 9
hex86 (ROM Hex Generator) 10
lb (Object module librarian) 11
ls (list directory contents 22
obd (Object file utility) 25
obj (MSDOS/PCDOS Object code generator) 26
ord (Object library generation utility) 27
prof (Execution profiler) 28
sqz (Object file utility) 29
term (terminal emulator for IBM PC) 30

Utility Programs

This chapter describes utility programs that are provided with Aztec C86. For a description of other utility programs, which are provided in some Aztec C86 packages, see the Debugger Utilities chapter and the Unitools chapter.

NAME

arcv - source dearchiver

SYNOPSIS

arcv arcfile [dest-area]

DESCRIPTION

arcv extracts the source from the archive file *arcfile* and places the results in separate files. You can't create archive files yourself; you can just unpack those that are provided with some versions of Aztec C86.

The optional parameter *dest-area* defines the area in which the files are created, where on DOS and 'area' is a directory on a drive and on CP/M-86 it's a user area on a drive. *arcv* generates the name of a file it wants to create by prepending the *dest-area* parameter to the file name as recorded in the archive. Thus, if *dest-area* isn't specified, the files will be created on the current area, which on DOS is the current directory on the default drive, and on CP/M-86 is the current user area on the default drive.

For example, the file *stdio.arc* contains the source for all the standard i/o files. To create these files in the current area, enter:

```
arcv stdio.arc
```

On DOS, the first of the following two commands will create the files on the root directory on the *b:* drive, and the second on the *\src* directory on the *c:* drive:

```
arcv stdio.arc b:
arcv stdio.arc c:\src\
```

Because *arcv* prepends the *dest-area* parameter to a file name, the terminating backslash is required when *dest-area* contains a directory name.

On CP/M-86, the first of the following two commands will create the files on the current user area on the b: drive, and the second on user area 4 on the *c:* drive:

```
arcv stdio.arc b:
arcv stdio.arc 4/c:
```

NAME

cnm - display object file info

SYNOPSIS

cnm [-sol] file [file ...]

DESCRIPTION

cnm displays the size and symbols of its object file arguments. The files can be object modules created by the Manx assembler, libraries of object modules created by the Manx librarian *lb*, and 'rsm' files created by the Manx linker during the linking of an overlay root.

For example, the following displays the size and symbols for the object module *sub1.o*, the library *c.lib*, and the rsm file *root.rsm*:

```
cnm sub1.o c.lib root.rsm
```

By default, the information is sent to the console. It can be redirected to a file or device in the normal way. For example, the following three commands send information about *sub1.o* to the display, the file *dispfile*, and the printer, respectively:

```
cnm sub1.o
cnm sub1.o > dispfile
cnm >lst: sub1.o
```

A filename can optionally specify multiple files, using the "wildcard" characters *?* and *. These have their standard meanings: *?* matches a single character; * matches zero or more characters. For example

*.o	Specifies all files with extent '.o'
a??.lib	Specifies all files whose filename has three characters, the first of which is 'a', and whose extent is '.lib'

The first line listed by *cnm* for an object module has the following format:

```
file (module): code: cc  data: dd  udata: uu  total: tt (0xhh)
```

where

* *file* is the name of the file containing the module,
* *module* is the name of the module; if the module is unnamed, this field and its surrounding parentheses aren't printed;
* *cc* is the number of bytes in the module's code segment, in decimal;
* *dd* is the number of bytes in the module's initialized data segment, in decimal;
* *uu* is the number of bytes in the module's uninitialized data segment, in decimal;
* *tt* is the total number of bytes in the module's three segments,

in decimal;

* *hh* is the total number of bytes in the module's three segments, in hexadecimal.

If *cnm* displays information about more than one module, it displays four totals just before it finishes, listing the sum of the sizes of the modules' code segments, initialized data segments, and uninitialized data segments, and the sum of the sizes of all segments of all modules. Each sum is in decimal; the total of all segments is also given in hexadecimal.

The *-s* option tells *cnm* to display just the sizes of the object modules. If this option isn't specified, *cnm* also displays information about each named symbol in the object modules.

When *cnm* displays information about the modules' named symbols, the *-l* option tells *cnm* to display each symbol's information on a separate line and to display all of the characters in a symbol's name; if this option isn't used, *cnm* displays the information about several symbols on a line and only displays the first eight characters of a symbol's name.

The *-o* option tells *cnm* to prefix each line generated for an object module with the name of the file containing the module and the module name in parentheses (if the module is named). If this option isn't specified, this information is listed just once for each module: prefixed to the first line generated for the module.

The *-o* option is useful when using *cnm* in combination with *grep*. For example, the following commands will display all information about the module *perror* in the library *c.lib*:

```
cnm -o c.lib >tmp
grep perror tmp
```

cnm displays information about an module's 'named' symbols; that is, about the symbols that begin with something other than a dollar sign followed by a digit. For example, the symbol *quad* is named, so information about it would be displayed; the symbol *$0123* is unnamed, so information about it would not be displayed.

For each named symbol in a module, *cnm* displays its name, a two-character code specifying its type, and an associated value. The value displayed depends on the type of the symbol.

If the first character of a symbol's type code is lower case, the symbol can only be accessed by the module; that is, it's local to the module. If this character is upper case, the symbol is global to the module: either the module has defined the symbol and is allowing other modules to access it or the module needs to access the symbol, which must be defined as a global or public symbol in another module. The type codes are:

ab The symbol was defined using the assembler's EQUATE directive. The value listed is the equated value of its symbol.

The compiler doesn't generate symbols of this type.

pg The symbol is in the code segment. The value is the offset of the symbol within the code segment.

The compiler generates this type symbol for function names. Static functions are local to the function, and so have type *pg*; all other functions are global, that is, callable from other programs, and hence have type *Pg*.

dt The symbol is in the initialized data segment. The value is the offset of the symbol from the start of the data segment.

The compiler generates symbols of this type for initialized variables which are declared outside any function. Static variables are local to the program and so have type *dt*; all other variables are global, that is, accessable from other programs, and hence have type *Dt*.

Cm The symbol is the name of a segment whose combine-type is COMMON. The value is the size of the segment, in bytes. *Cm* is in upper case because common block names are always global.

The compiler doesn't generate this type symbol.

rf The symbol is defined within a segment whose combine-type is COMMON. The value is the offset of the symbol from the beginning of the segment.

The compiler doesn't generate this type symbol.

ov When an overlay is being linked and that overlay itself calls another overlay, this type of symbol can appear in the rsm file for the overlay that is being linked. It indicates that the symbol is defined in the program that is going to call the overlay that is being linked.

The value is the offset of the symbol from the beginning of the physical segment that contains it.

un The symbol is used but not defined within the program. The value has no meaning.

Un symbols (that is, the u is capitalized) have been defined with the assembly language directive *extrn*.

The compiler generates *Un* symbols for functions that are called but not defined within the program, for

variables that are declared to be *extern* and that are actually used within the program, and for uninitialized, global dimensionless arrays. Variables which are declared to be *extern* but which are not used within the program aren't mentioned in the assembly language source file generated by the compiler and hence don't appear in the object file.

bs The symbol is in the uninitalized data segment. The value is the space reserved for the symbol.

The compiler generates *bs* symbols for static, uninitialized variables which are declared outside all functions and which aren't dimensionless arrays.

The assembler generates *bs* symbols for symbols defined using the *bss* assembler directive.

Gl The symbol is in the uninitialized data segment. The value is the space reserved for the symbol.

The compiler generates *Gl* symbols for non-static, uninitialized variables which are declared outside all functions and which aren't dimensionless arrays.

The assembler generates *Gl* symbols for variables declared using the *global* directive which have a non-zero size.

NAME

crc - Utility for generating the CRC for files

SYNOPSIS

crc files

DESCRIPTION

crc computes a number, called the CRC, for the specified *files*. By using the standard 'wild-card' characters, *files* can specify multiple files.

The CRC for a file is entirely dependent on the file's contents, and it is very unlikely that two files whose contents are different will have the same CRCs. Thus, *crc* can be used to determine whether a file has the expected contents.

The file *crclist* that is on the Aztec C disks lists the CRC values for each of the files on the disks. By comparing these values with those computed by your own running of *crc*, you can easily determine whether what we thought we sent you is what you got.

As an example of the usage of *crc*, the following command computes the crc of all files whose extension is *.c*:

```
crc *.c
```

NAME

hex86 - Intel Hex Generator

SYNOPSIS

hex86 [-zeos] infile [outfile]

DESCRIPTION

hex86 is used when generating ROMable programs. It converts an *.exe* file to Intel hex format, which can then be read by a ROM programmer.

For more information, see the section entitled "Generating ROMable Code" in the Technical Information chapter.

NAME

lb - object file librarian

SYNOPSIS

lb library [options] [mod1 mod2 ...]

DESCRIPTION

lb is a program that creates and manipulates libraries of object modules. The modules must be created by the Manx assembler.

This description of *lb* is divided into three sections: the first describes briefly *lb*'s arguments and options, the second *lb*'s basic features, and the third the rest of *lb*'s features.

1. The arguments to *lb*

1.1 The *library* argument

When started, *lb* acts upon a single library file. The first argument to *lb* (*library*, in the synopsis) is the name of this file. The filename extension for *library* is optional; if not specified, it's assumed to be *.lib*.

1.2 The *options* argument

There are two types of *options* argument: function code options, and qualifier options. These options will be summarized in the following paragraphs, and then described in detail below.

1.2.1 Function code options

When *lb* is started, it performs one function on the specified library, as defined by the *options* argument. The functions that *lb* can perform, and their corresponding option codes, are:

function	*code*
create a library	(no code)
add modules to a library	-a, -i, -b
list library modules	-t
move modules within a library	-m
replace modules	-r
delete modules	-d
extract modules	-x
ensure module uniqueness	-u
define module extension	-e
help	-h

In the synopsis, the *options* argument is surrounded by square brackets. This indicates that the argument is optional; if a code isn't specified, *lb* assumes that a library is to be created.

1.2.2 Qualifier options

In addition to a function code, the *options* argument can optionally specify a qualifier, that modifies *lb*'s behavior as it is performing the requested function. The qualifiers and their codes are:

verbose	-v
silent	-s

The qualifier can be included in the same argument as the function code, or as a separate argument. For example, to cause *lb* to append modules to a library, and be silent when doing it, any of the following option arguments could be specified:

```
-as
-sa
-a -s
-s -a
```

1.3 The *mod* arguments

The arguments *mod1*, *mod2*, etc. are the names of the object modules, or the files containing these modules, that *lb* is to use. For some functions, *lb* requires an object module name, and for others it requires the name of a file containing an object module. In the latter case, the file's extension is optional; if not specified, the *lb* that's supplied with native Aztec C systems assumes that it's *.o*, and the *lb* that's supplied with cross development versions of Aztec C assumes that the extension is *.r*. You can explicitly define the default module extension using the *-e* option.

1.4 Reading arguments from another file

lb has a special argument, *-f filename*, that causes it to read command line arguments from the specified file. When done, it continues reading arguments from the command line. Arguments can be read from more than one file, but the file specified in a *-f filename* argument can't itself contain a *-f filename* argument.

2. Basic features of *lb*

In this section we want to describe the basic features of *lb*. With this knowledge in hand, you can start using *lb*, and then read about the rest of the features of *lb* at your leisure.

The basic things you need to know about *lb*, and which thus are described in this section, are:

* How to create a library
* How to list the names of modules in a library
* How modules get their names

* Order of modules in a library
* Getting *lb* arguments from a file

Thus, with the information presented in this section you can create libraries and get a list of the modules in libraries. The third section of this description shows you how to modify selected modules within a library.

2.1 Creating a Library

A library is created by starting *lb* with a command line that specifies the name of the library file to be created and the names of the files whose object modules are to be copied into the library. It doesn't contain a function code, and it's this absence of a function code that tells *lb* that it is to create a library.

For example, the following command creates the library *exmpl.lib*, copying into it the object modules that are in the files *obj1.o* and *obj2.o*:

```
lb exmpl.lib obj1.o obj2.o
```

Making use of *lb*'s assumptions about file names for which no extension is specified, the following command is equivalent to the above command:

```
lb exmpl obj1 obj2
```

An object module file from which modules are read into a new library can itself be a library created by *lb*. In this case, all the modules in the input library are copied into the new library.

2.1.1 The temporary library

When *lb* creates a library or modifies an existing library, it first creates a new library with a temporary name. If the function was successfully performed, *lb* erases the file having the same name as the specified library, and then renames the new library, giving it the name of the specified library. Thus, *lb* makes sure it can create a library before erasing an existing one.

Note that there must be room on the disk for both the old library and the new.

2.2 Getting the table of contents for a library

To list the names of the modules in a library, use *lb*'s *-t* option. For example, the following command lists the modules that are in *exmpl.lib*:

```
lb exmpl -t
```

The list will include some ***DIR*** entries. These identify blocks within the library that contain control information. They are created and deleted automatically as needed, and cannot be changed by you.

2.3 How modules get their names

When a module is copied into a library from a file containing a single object module (that is, from an object module generated by the Manx assembler), the name of the module within the library is derived from the name of the input file by deleting the input file's volume, path, and extension components.

For example, in the example given above, the names of the object modules in *exmpl.lib* are *obj1* and *obj2*.

An input file can itself be a library. In this case, a module's name in the new library is the same as its name in the input library.

2.4 Order in a library

The order of modules in a library is important, since the linker makes only a single pass through a library when it is searching for modules. For a discussion of this, see the tutorial section of the Linker chapter.

When *lb* creates a library, it places modules in the library in the order in which it reads them. Thus, in the example given above, the modules will be in the library in the following order:

```
obj1 obj2
```

As another example, suppose that the library *oldlib.lib* contains the following modules, in the order specified:

```
sub1   sub2   sub3
```

If the library *newlib.lib* is created with the command

```
lb newlib mod1 oldlib.lib mod2 mod3
```

the contents of the newly-created *newlib.lib* will be:

```
mod1   sub1   sub2   sub3   mod2   mod3
```

The *ord* utility program can be used to create a library whose modules are optimally sorted. For information, see its description later in this chapter.

2.5 Getting *lb* arguments from a file

For libraries containing many modules, it is frequently inconvenient, if not impossible, to enter all the arguments to *lb* on a single command line. In this case, *lb*'s *-f filename* feature can be of use: when *lb* finds this option, it opens the specified file and starts reading command arguments from it. After finishing the file, it continues to scan the command line.

For example, suppose the file *build* contains the line

```
exmpl obj1 obj2
```

Then entering the command

lb -f build

causes *lb* to get its arguments from the file *build*, which causes *lb* to create the library *exmpl.lib* containing *obj1* and *obj2*.

Arguments in a -f file can be separated by any sequence of whitespace characters ('whitespace' being blanks, tabs, and newlines). Thus, arguments in a -f file can be on separate lines, if desired.

The *lb* command line can contain multiple *-f* arguments, allowing *lb* arguments to be read from several files. For example, if some of the object modules that are to be placed in *exmpl.lib* are defined in *arith.inc*, *input.inc*, and *output.inc*, then the following command could be used to create *exmpl.lib*:

lb exmpl -f arith.inc -f input.inc -f output.inc

A -f file can contain any valid *lb* argument, except for another -f. That is, -f files can't be nested.

3. Advanced *lb* features

In this section we describe the rest of the functions that *lb* can perform. These primarily involve manipulating selected modules within a library.

3.1 Adding modules to a library

lb allows you to add modules to an existing library. The modules can be added before or after a specified module in the library or can be added to the beginning or end of the library.

The options that select *lb*'s add function are:

option	*function*
-b target	add modules before the module *target*
-i target	same as *-b target*
-a target	add modules after the module *target*
-b+	add modules to the beginning of the library
-i+	same as *-b+*
-a+	add modules to the end of the library

In an *lb* command that selects the *add* function, the names of the files containing modules to be added follows the add option code (and the target module name, when appropriate). A file can contain a single module or a library of modules.

Modules are added in the order that they are specified. If a library is to be added, its modules are added in the order they occur in the input library.

3.1.1 Adding modules before an existing module

As an example of the addition of modules before a selected module, suppose that the library *exmpl.lib* contains the modules

obj1 obj2 obj3

The command

lb exmpl -i obj2 mod1 mod2

adds the modules in the files *mod1.o* and *mod2.o* to *exmpl.lib*, placing them before the module *obj2*. The resultant *exmpl.lib* looking like this:

obj1 mod1 mod2 obj2 obj3

Note that in the *lb* command we didn't need to specify the extension of either the file containing the library to which modules were to be added or the extension of the files containing the modules to be added. *lb* assumed that the extension of the file containing the target library was *.lib*, and that the extension of the other files was *.o*.

As an example of the addition of one library to another, suppose that the library *mylib.lib* contains the modules

mod1 mod2 mod3

and that the library *exmpl.lib* contains

obj1 obj2 obj3

Then the command

lb -b obj2 mylib.lib

adds the modules in *mylib.lib* to *exmpl.lib*, resulting in *exmpl.lib* containing

obj1 mod1 mod2 mod3 obj2 obj3

Note that in this example, we had to specify the extension of the input file *mylib.lib*. If we hadn't included it, *lb* would have assumed that the file was named *mylib.o*.

3.1.2 Adding modules after an existing module

As an example of adding modules after a specified module, the command

lb exmpl -a obj1 mod1 mod2

will insert *mod1* and *mod2* after *obj1* in the library *exmpl.lib*. If *exmpl.lib* originally contained

obj1 obj2 obj3

then after the addition, it contains

obj1 mod1 mod2 obj2 obj3

3.1.3 Adding modules at the beginning or end of a library

The options *-b+* and *-a+* tell *lb* to add the modules whose names follow the option to the beginning or end of a library, respectively. Unlike the *-i* and *-a* options, these options aren't followed by the name of an existing module in the library.

For example, given the library *exmpl.lib* containing

obj1 obj2

the following command will add the modules *mod1* and *mod2* to the beginning of *exmpl.lib*:

lb exmpl -i+ mod1 mod2

resulting in *exmpl.lib* containing

mod1 mod2 obj1 obj2

The following command will add the same modules to the end of the library:

lb exmpl -a+ mod1 mod2

resulting in *exmpl.lib* containing

obj1 obj2 mod1 mod2

3.2 Moving modules within a library

Modules which already exist in a library can be easily moved about, using the *move* option, *-m*.

As with the options for adding modules to an existing library, there are several forms of *move* functions:

option	*meaning*
-mb target	move modules before the module *target*
-ma target	move modules after the module *target*
-mb+	move modules to the beginning of the library
-ma+	move modules to the end of the library

In the *lb* command, the names of the modules to be moved follows the 'move' option code.

The modules are moved in the order in which they are found in the original library, not in the order in which they are listed in the *lb* command.

3.2.1 Moving modules before an existing module

As an example of the movement of modules to a position before an existing module in a library, suppose that the library *exmpl.lib* contains

obj1 obj2 obj3 obj4 obj5 obj6

The following command moves *obj3* before *obj2*:

lb exmpl -mb obj2 obj3

putting the modules in the order:

obj1 obj3 obj2 obj4 obj5 obj6

And, given the library in the original order again, the following command moves *obj6*, *obj2*, and *obj1* before *obj3*:

lb exmpl -mb obj3 obj6 obj2 obj1

putting the library in the order:

obj1 obj2 obj6 obj3 obj4 obj5

As an example of the movement of modules to a position after an existing module, suppose that the library *exmpl.lib* is back in its original order. Then the command

lb exmpl -ma obj4 obj3 obj2

moves *obj3* and *obj2* after *obj4*, resulting in the library

obj1 obj4 obj2 obj3 obj5 obj6

3.2.2 Moving modules to the beginning or end of a library

The options for moving modules to the beginning or end of a library are *-mb+* and *-ma+*, respectively.

For example, given the library *exmpl.lib* with contents

obj1 obj2 obj3 obj4 obj5 obj6

the following command will move *obj3* and *obj5* to the beginning of the library:

lb exmpl -mb+ obj5 obj3

resulting in *exmpl.lib* having the order

obj3 obj5 obj1 obj2 obj4 obj6

And the following command will move *obj2* to the end of the library:

lb exmpl -ma+ obj2

3.3 Deleting Modules

Modules can be deleted from a library using *lb*'s *-d* option. The command for deletion has the form

lb libname -d mod1 mod2 ...

where *mod1*, *mod2*, ... are the names of the modules to be deleted.

For example, suppose that *exmpl.lib* contains

obj1 obj2 obj3 obj4 obj5 obj6

The following command deletes *obj3* and *obj5* from this library:

lb exmpl -d obj3 obj5

3.4 Replacing Modules

The *lb* option 'replace' is used to replace one module in a library with one or more other modules.

The 'replace' option has the form *-r target*, where *target* is the name of the module being replaced. In a command that uses the 'replace' option, the names of the files whose modules are to replace the target module follow the 'replace' option and its associated target module. Such a file can contain a single module or a library of modules.

Thus, an *lb* command to replace a module has the form:

lb library -r target mod1 mod2 ...

For example, suppose that the library *exmpl.lib* looks like this:

obj1 obj2 obj3 obj4

Then to replace obj3 with the modules in the files *mod1.o* and *mod2.o*, the following command could be used:

lb exmpl -r obj3 mod1 mod2

resulting in *exmpl.lib* containing

obj1 obj2 mod1 mod2 obj4

3.5 Uniqueness

lb allows libraries to be created containing duplicate modules, where one module is a duplicate of another if it has the same name.

The option *-u* causes *lb* to delete duplicate modules in a library, resulting in a library in which each module name is unique. In particular, the *-u* option causes *lb* to scan through a library, looking at module names. Any modules found that are duplicates of previous modules are deleted.

For example, suppose that the library *exmpl.lib* contains the following:

obj1 obj2 obj3 obj1 obj3

The command

lb exmpl -u

will delete the second copies of the modules *obj1* and *obj2*, leaving the library looking like this:

```
obj1    obj2    obj3
```

3.6 Extracting modules from a Library

The *lb* option *-x* extracts modules from a library and puts them in separate files, without modifying the library.

The names of the modules to be extracted follows the *-x* option. If no modules are specified, all modules in the library are extracted.

When a module is extracted, it's written to a new file; the file has same name as the module and extension *.o*.

For example, given the library *exmpl.lib* containing the modules

```
obj1 obj2 obj3
```

The command

```
lb exmpl -x
```

extracts all modules from the library, writing *obj1* to *obj1.o*, *obj2* to *obj2.o*, and *obj3* to *obj3.o*.

And the command

```
lb exmpl -x obj2
```

extracts just *obj2* from the library.

3.7 The 'verbose' option

The 'verbose' option, *-v*, causes *lb* to be verbose; that is, to tell you what it's doing.

This option can be specified as part of another option, or all by itself. For example, the following command creates a library in a chatty manner:

```
lb exmpl -v mod1 mod2 mod3
```

And the following equivalent commands cause *lb* to remove some modules and to be verbose:

```
lb exmpl -dv mod1 mod2
lb exmpl -d -v mod1 mod2
```

3.8 The 'silence' option

The 'silence' option, *-s*, tells *lb* not to display its signon message.

This option is especially useful when redirecting the output of a list command to a disk file, as described below.

3.9 Rebuilding a library

The following commands provide a convenient way to rebuild a library:

```
lb exmpl -st > tfil
lb exmpl -f tfil
```

The first command writes the names of the modules in *exmpl.lib* to the file *tfil*. The second command then rebuilds the library, using as arguments the listing generated by the first command.

The *-s* option to the first command prevents *lb* from sending information to *tfil* that would foul up the second command. The names sent to *tfil* include entries for the directory blocks, ***DIR***, but these are ignored by *lb*.

3.10 Defining the default module extension.

Specification of the extension of an object module file is optional; the *lb* that comes with native development versions of Aztec C assumes that the extension is *.o*, and the *lb* that comes with cross development versions of Aztec C assumes that it's *.r*. You can explicitly define the default extension using the *-e* option. This option has the form

```
-e .ext
```

For example, the following command creates a library; the extension of the input object module files is *.i*.

```
lb my.lib -e .i mod1 mod2 mod3
```

3.11 Help

The *-h* option is provided for brief lapses of memory, and will generate a summary of *lb* functions and options.

NAME

ls - list directory contents

SYNOPSIS

ls [-options] [name1 name2 ...]

DESCRIPTION

ls displays information about the files and directories *name1*, *name2*, ... If no names are specified, *ls* displays information about all the files and directories in the current directory on the default drive. For example, the following command displays information about the files *sub1.o* and *sub1.c* in the default drive's current directory, and the directory *d:\include*:

```
ls sub1.o sub1.c d:\include
```

A name can optionally specify multiple files, using the "wildcard characters" * and *?*. These have their standard meaning: * matches one or more characters, and ? matches a single character. For example, the following command displayes information about all files that have extension *.c* and that are in the directory *c:\src*:

```
ls c:\src\*.c
```

Wildcard characters can be used only in file names, and not in directory or drive names. A wildcard character won't match the period that separates a file or directory name and its extension. Thus, the command *ls* * will list just the files and directories that don't have extensions.

ls sends the information to its standard output. This information thus by default is sent to the console, but can be redirected to a file or other device in the normal way. For example, the first of the following commands displays on the console information about files that have extension *.o* and that are in the current directory. The second and third commands send information about the same files to the file *info.obj* and to the printer *prn*, respectively:

```
ls *.o
ls *.o >info.obj
ls *.o >prn
```

ls by default displays information in 'short form', listing just the names of the specified files and directories. You can also specify the *-l* option to cause *ls* to display information in 'long form', listing lots of information.

Even when long form is specified, *ls* will only list the name of specified directories. To list the contents of a directory, you must specify the files in the directory, using wildcard characters. For example, to see the contents of the directory *\work*, you would say

```
ls \work\*.*
```

When *ls* sends information in short form to the console, the names are in columns on the screen, with a dash preceding directory names. When the information is sent to a file or other device, the names are listed one per line, and a directory name isn't by default preceded by a dash.

ls sorts the list it's going to display. By default, the list is sorted alphabetically; you can also specify options to cause *ls* to sort based on other the list such as 'last modified' time and file size, and, for a given criteria, to sort in the reverse of the normal order. default drives.

All options to *ls* must be specified in one parameter to *ls*. This parameter begins with a dash and comes before the file and directory names. *ls* supports the following options:

-l	List in long form. For a description of the 'long form' information, see below.
-b	When listing in long form, list the number of 512-byte blocks that a file uses, in addition to the other information.
-a	List all files, including those whose first character is a period (such as . and ..).
-p	When listing in short form, precede directory names with a dash.
-t	Sort by 'last-modified' time.
-s	Sort by file size.
-r	Reverse the order of the sort. For example, when sorting alphabetically, list names beginning with 'z' first and those beginning with 'a' last.

When displaying in long form, the line on which information is displayed for a file or directory has the following form:

```
dshr  bytesize (blksize)  date  name
```

The first four letters define attributes of the file or directory. If it doesn't have an attribute, the letter is replaced with a dash. The meaning of the letters are:

d	Directory
s	System file
h	Hidden file
r	Read-only file

The other fields in a long form listing have the following meaning:

bytesize	Number of bytes in a file. This field is zero for a directory.

blksize Number of 512-byte blocks that the file uses. This field is zero for a directory, and is only listed when the *-b* option is specified.

date Date and time at which the file was last modified. If the file was modified within the last six months, this field lists the month, day, and time of modification; otherwise, it lists the month, day, and year of modification.

name The name of the file or directory.

NAME

obd - list object code

SYNOPSIS

obd <objfile>

DESCRIPTION

obd lists the loader items in an object file. It has a single parameter, which is the name of the object file.

NAME

obj -- convert object modules from Aztec to Microsoft format

SYNOPSIS

obj [options] infile [outfile]

DESCRIPTION

obj converts object modules from Aztec to Microsoft format. The resultant files can then be linked using the Microsoft linker with other Microsoft object files to produce an executable program.

For more information, see the section entitled "Using the Microsoft Linker" in the Technical Information chapter.

Options

obj supports the following options:

-u	Don't strip trailing underscores from names.
-s	Truncate external names to eight characters.

NAME

ord - sort object module list

SYNOPSIS

ord [-v] [infile [outfile]]

DESCRIPTION

ord sorts a list of object file names. A library of the object modules that is generated from the sorted list by the object module librarian, *lb*, will have a minimum number of 'backward references'; that is, global symbols that are defined in one module and referenced in a later module.

Since the specification of a library to the linker causes it to search the library just once, a library having no backward references need be specified just once when linking a program, and a library having backward references may need to be specified multiple times.

infile is the name of a file containing an unordered list of file names. These files contain the object modules that are to be put into a library. If *infile* isn't specified, this list is read from *ord*'s standard input. The file names can be separated by space, tab, or newline characters.

outfile is the name of the file to which the sorted list is written. If it's not specified, the list is written to *ord*'s standard output. *outfile* can only be specified if *infile* is also specified.

The *-v* option causes *ord* to be verbose, sending messages to its standard error device as it proceeds.

NAME

prof - execution profiler report program

SYNOPSIS

prof -s symfile [-m monfile] [-[ant]] [-[xo]] [-[zh]]

DESCRIPTION

prof processes a monitor file produced by the *monitor* function, and produces a report on the execution of the monitored program. For each function in the range specified in *monitor*, *prof* counts the number of ticks encountered in that function and determines the percentage of program run time spent in the function.

Options

-s The -s argument is the name of the symbol table file for the program generated by the linker -t option. This argument must be present.

-m The -m option allows the user to specify the *monitor* output file to be processed. If this option is not present, *prof* assumes the file is named mon.out (the name always used by *monitor*) and is on the current directory.

The -t, -a, and -n options determine the sorting of lines in the report.

-t Sort by percentage of time spent in function, greatest to least (This option is the default).

-a Sort by address of function.

-n Sort alphabetically by function name.

The -o and -x options cause prof to display the addresses of the functions in the report along with their names.

-o Specify function addresses in octal.

-x Specify function addresses in hexadecimal.

-z The -z option causes all symbols in the range specified in the call to monitor to be displayed, regardless of whether any ticks were encountered in these functions. The default is to supress listing any unencountered functions.

-h The -h option causes prof to suppress printing its normal header in the report. This is useful if the information is to undergo further processing.

NAME

sqz - squeeze an object library

SYNOPSIS

sqz file [outfile]

DESCRIPTION

sqz compresses an object module that was created by the Manx assembler.

The first parameter is the name of the file containing the module to be compressed. The second parameter, which is optional, is the name of the file to which the compressed module will be written.

If the output file is specified, the original file isn't modified or erased.

If the output file isn't specified, *sqz* creates the compressed module in a file having a temporary name, erases the original file, and renames the output file to the name of the original file. The temporary name is derived from the input file name by changing it's extent to *.sqz*.

If the output file isn't specified and an error occurs during the creation of the compressed module the original file isn't erased or modified.

NAME

term - Terminal Emulator

SYNOPSIS

term [baud]

DESCRIPTION

term is a terminal emulation program that allows an IBM PC operator to talk to another computer. To the other system, the IBM PC will appear to be a terminal that supports some of the special features of the ADM-3A terminal.

term reads characters from the keyboard and writes them to the serial interface whose base address is 0x3f8. It also reads characters from this interface and writes them to the console. This address is normally associated with the PC-DOS device *com1*.

The optional parameter *baud* defines the baud rate of the i/o ports. If not specified, it's assumed to be 9600 baud.

The source for *term* is in the archive *term.arc*. Files can be extracted from this archive using the Manx utility program *arcv*.

LIBRARY FUNCTIONS OVERVIEW: 8086 INFORMATION

Library Functions Overview: 8086 Information

The *Library Functions Overview* chapter presented overview information that is independent of the system on which your programs run. This chapter presents overview information about the library functions that is specific to programs that run on an 8086 under PCDOS, MSDOS, or CP/M-86.

The sections in this appendix and in the *Overview of Library Functions* chapter are numbered. The information discussed in a section of this appendix relates to the section in the *Overview of Library Functions* chapter that has the same number.

1. Overview of I/O: 8086 Information

For systems using PCDOS and MSDOS, the operating system places a limit on the maximum number of devices and files simultaneously open for standard and unbuffered i/o: this limit is defined by the operating system's configuration option named FILES. The default value for FILES is 10.

The Aztec C i/o routines impose a further restriction, limiting the number of files and devices that can be simultaneously open for standard i/o to eleven, regardless of the value of FILES.

For systems using CP/M-86, a maximum of eleven files and devices, including the standard i/o devices, can be open at once for both standard and unbuffered i/o. When this limit is reached, an open file or device must be closed before another can be opened.

1.1 Pre-opened devices and command line arguments

For programs running on an 8086, whether under PCDOS, MSDOS, or CP/M-86, a null pointer is the first item in the array that is pointed at by the second argument of the of the program's *main* function. That is, if the *main* function begins

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

then *argv[0]* is a null pointer.

1.2 File I/O

1.2.1 Sequential I/O

On PCDOS/MSDOS, data can always be correctly appended to a file, since PCDOS/MSDOS keep track of the exact number of bytes that have been written to the file.

On CP/M-86 it isn't always possible to correctly append data to a file, since this system doesn't keep track of the exact number of bytes that have been written to a file; see below for details.

1.2.2 Random I/O

On PCDOS/MSDOS, positioning of a file using *fseek* or *lseek* is always correctly done, since these systems keep track of the exact number of bytes that have written to the file.

On CP/M-86, a file can always be correctly positioned relative to its beginning and current position. But positioning relative to its end can't always be correctly done, since this system doesn't keep track of the exact number of bytes that have been written to the file. This is discussed in the following paragraphs.

Finding the end of a file on CP/M-86

UNIX keeps track of the last character written to a file. Since the Aztec I/O functions attempt to make a file look like a UNIX file to a program, when a program requests that a file be positioned relative to its end (that is, relative to the last character which was written to it), the Aztec C routines must try to locate the last character which was written to it. This can always be done if the operating system on which Aztec C is running also keeps track of the last character written to a file.

PCDOS and MSDOS do this, and so positioning relative to the end of a file on these systems is always correctly done.

However, CP/M-86 only keeps track of the last record written to a file, and not the last character. Because of this, it is not always possible for the Aztec C i/o functions to determine the last character written to the file, when the program in which they are contained is running on CP/M-86. And because of this, it is not always possible for a program running on CP/M-86 to correctly position a file relative to its end.

When a program running on CP/M-86 requests positioning of a file relative to its end, the Aztec i/o functions try to find the last character written to the file. They always succeed if the file contains only text; for files containing arbitrary data, they may not succeed.

To locate the last valid character in a file on CP/M-86, the Aztec routines use the following fact: when a file is created on these systems using Aztec C, the last record in the file is padded at the end with the special character which denotes the end of a text file. For CP/M-86, the special character is control-z. If the program exactly filled the last record, it won't have any padding.

When a program requests that a file be positioned relative to its end, the Aztec C i/o routines search the file's last record; end of file is declared to be located at the position following the last non-end-of-file character.

For files of text, this algorithm always correctly determines the last character in the file, so appending to text files is always correctly done.

For other files, this algorithm will still correctly determine the last valid character in the file...most of the time. However, if the last valid characters in the file are end-of-file characters, the file will be incorrectly positioned.

1.2.3 Opening Files

1.2.3.1 Opening files on PCDOS and MSDOS

When opening a file on systems running PCDOS or MSDOS, the filename has the standard DOS 2.x format; that is, it consists of an optional drive identifier, an optional directory path, and a filename. The drive defaults to the default drive and the directory to the current directory.

1.2.3.2 Opening files on CP/M-86

On CP/M-86, the character string which specifies the file to be opened has the following fields, which must be in the order listed: (1) a user number followed by a forward slash, (2) a drive identifier followed by a colon, (3) the filename, (4) a period followed by an extension. Only the third field is mandatory. If a user number isn't specified, the file is assumed to be on the current user. If the drive isn't specified, the file is assumed to be on the default drive.

For example, the following are valid file names:

file.ext	file.ext is on default drive, current user
b:file.ext	file.ext is on b: drive, current user
15/file.ext	file.ext is on default drive, user 15
12/c:file.ext	file.ext is on c: drive, user 12

A program can have files located in several different user areas open at once.

There are several functions which may be useful to programs which need to access files in various user areas: *getusr*, which returns the current user number; *setusr*, which sets the current user number; and *rstusr*, which resets the current user number. See the USER section in

the 8086 Functions chapter for more details.

1.3 Device I/O

On PCDOS/MSDOS, a program accesses devices using their standard PCDOS/MSDOS names.

On CP/M-86, a program accesses devices using the following names:

Device	*name*
keyboard	con:
display	con:
printer	prn:
"	lst:
RS232 in	rdr:
RS232 out	pun:

2. Overview of Standard I/O: 8086 Information

2.5 Buffering

On PCDOS, MSDOS, and CP/M-86, the size of a buffer used for standard I/O is 1024 bytes.

4. Console I/O Overview: 8086 Information

4.2 Character-oriented Input

4.2.1 Character-oriented Input on CP/M-86

On CP/M-86, a program issuing a read request to the console when it is in character- oriented input mode will wait until at least one character has been typed.

If the console is in RAW mode or in CBREAK mode with ECHO turned off, an unbuffered read request for more than a single character may return before all requested characters have been typed. That is, if the operator doesn't type the characters fast enough, the read operation will time-out and return whatever characters have been entered up to that point. For example, if a program issues the call

```
read(0, buf, 80);
```

to read 80 characters into *buf* from the console, with the console in RAW or CBREAK without ECHO modes, the read will return at least one character, but may return fewer than 80 characters, if the operator doesn't type fast enough.

If the console is in CBREAK mode with ECHO enabled, a read request to the console always returns the requested number of characters; that is, the operator can take his or her own sweet time entering characters.

4.2.2 Character-oriented Input on PCDOS and MSDOS

A read request to the console will always return the requested number of characters.

4.4 The *sgtty* fields

4.4.1 The *sg_flags* field

On PCDOS and MSDOS, some bits in the *sg_flags* field have meaning to MSDOS and PCDOS, in addition to the ones we have described. Thus, on MSDOS and PCDOS systems, a program that wants to change the console options must fetch the current options using the TIOCGETP mode of *ioctl*, modify as desired just the bits that we have defined in this chapter, and then call *ioctl* to set the new options.

When a program terminates, the console stays in the mode set by the program. So programs which change the console options from their default settings should set them back before terminating.

8086 FUNCTIONS

Chapter Contents

8086 Functions lib86
Index 5
The functions 7

8086 Functions

This chapter describes functions which are available only to programs which are running on 8086- and 8088-based systems and which use as an operating system MSDOS, PCDOS, or CP/M-86.

As with the System Independent Functions chapter, this chapter is divided into sections, each of which describes a group of related functions.

Some of the functions in this section will only run on a specific operating system. The header to a section defines the systems on which the section's functions will run, as does the index which follows this introduction. The codes defining the systems on which functions run are:

DOS	Function runs on any version of MSDOS or PCDOS;
DOS2x	MSDOS or PCDOS, version 2.x or later;
DOS11	MSDOS or PCDOS, version 1.1;
PCDOS	PCDOS, any version;
CP/M-86	CP/M-86, any version.

As with the system independent functions, the header to a section has a parenthesised letter that specifies the library containing the section's functions. The codes and their related libraries are:

C	c.lib;
S	s.lib;
G	g.lib.

c.lib contains only functions for the system on which the your Aztec C86 programs run. Thus, a parenthesized 'c' in a section's title doesn't always mean that the section's functions are included in your *c.lib*. The functions are in *c.lib* only if they are available on all 8086-based systems or on your specific system. For example, if you have the PCDOS version of Aztec C86, your *c.lib* includes, in addition to system-independent functions, functions which will run on any 8086-based system and functions which run on PCDOS/MSDOS, version 2.x. It doesn't contain functions which run on CP/M-86 or on MSDOS/PCDOS, version 1.1.

Some Aztec C86 packages contain libraries that can be used in place of *c.lib*, to support different memory models or to generate code that will run in different environments. For more information, see the release document.

For *Apprentice C*, the library functions are all in the run-time system, and not in libraries.

Index to 8086 Functions

This section lists the 8086-specific functions that are provided with Aztec C86. The list is sorted alphabetically by function name. For each function it gives the function's name, the title of the section in which the function is described, a phrase describing the function's purpose, and a parenthesised code that defines the systems on which the function is provided. The codes are defined at the beginning of this chapter, with the exception of 'all', which of course means that the function is provided for all systems.

function *page* *description*

abstoptr ..... LONGPTR ..... absolute addr to seg:off ptr (DOS2x)
access ..... ACCESS ..... determine file accessibility (all)
asctime ..... TIME ..... convert data & time to ASCII (DOS)
assert ..... ASSERT ..... verify program assertion (all)
bdos ..... DOS ..... issue DOS int 21 function call (DOS)
bdos ..... BDOS ..... issue CPM86 BDOS call (CPM86)
bdosx ..... BDOSX ..... issue bdos call with a far pointer (DOS2x)
brk ..... BREAK ..... set heap pointer (all)
chdir ..... DIRECTORY ..... change current directory (DOS2x)
chmod ..... CHMOD ..... set attributes of file (DOS2x)
circle ..... CIRCLE ..... draw a circle (PCDOS)
clock ..... CLOCK ..... get time (DOS)
color ..... COLOR ..... set color (PCDOS)
__csread ..... CSREAD ..... read into code segment (all)
ctime ..... TIME ..... convert binary data & time to ASCII (DOS)
dos ..... DOS ..... issue DOS int 21 function call (DOS)
dostime ..... TIME ..... get data & time (DOS)
dosx ..... BDOSX ..... issue DOS int 21 call with far ptr (DOS2x)
dup ..... DUP ..... open second file descriptor for file (DOS2x)
execl, etc ..... EXEC ... jump to another program (DOS2x & CPM86)
exit ..... EXIT ..... terminate program (all)
__exit ..... EXIT ..... terminate program (all)
farcall ..... FARCALL ..... issue far call (all)
fcbinit ..... FCBINIT ..... initialize an FCB (all)
fdup ..... DUP ..... open second file descriptor for file (DOS2x)
fexecl, fexecv .. FEXEC ..... call another program (DOS2x)
ftime ..... FILETIME ..... get or set file's date & time (DOS2x)
getcwd ..... DIRECTORY get name of current directory (DOS2x)
getenv ..... GETENV get value of environment variable (DOS2x)
getusr ..... USER ..... get current user number (CPM86)
gmtime ..... TIME ..... convert date & time (DOS2x)

ground ..... COLOR ..... set background color (PCDOS)
inportb, etc ..... PORT ..... read from a port (all)
__int__sp ..... MONITOR ..... set monitoring clock speed (PCDOS)
line, lineto ..... LINE ..... draw a line (PCDOS)
localtime ..... TIME ..... convert date & time (DOS2x)
memccpy, etc .. MEMORY ..... memory operations (all)
mktemp ..... MKTEMP ..... make name for temporary file (all)
mode ..... MODE ..... set screen mode (PCDOS)
monitor ..... MONITOR ..... profiling function (PCDOS)
movblock ..... MOVBLOCK ..... move block of memory (all)
mkdir ..... DIRECTORY ..... make a directory (DOS2x)
outportb, etc .... PORT ..... write to a port (all)
palette ..... COLOR ..... set palette (PCDOS)
peekb, peekw ... PEEK ..... get memory byte or word (all)
perror, etc ..... PERROR ..... write error message (all)
point ..... POINT ..... plot a point (PCDOS)
pokeb, pokew .. PEEK ..... set memory byte or word (all)
__ptradd, ... LONGPTR ..... long pointer arithmetic (DOS2x)
__ptrdiff, ... LONGPTR ..... long pointer arithmetic (DOS2x)
ptrtoabs ..... LONGPTR ..... seg:off ptr to absolute addr (DOS2x)
rmdir ..... DIRECTORY ..... remove a directory (DOS2x)
rstusr ..... USER .. reset user number to previous value (CPM86)
rsvstk ..... BREAK ..... set heap-stack boundary (all)
sbrk ..... BREAK ..... set heap pointer (all)
scdir ..... SCDIR ..... Scan directory (DOS2x)
scr__curs, etc ... SCREEN ..... access console via ROM BIOS (PCDOS)
signal ..... SIGNAL ..... define how to handle a signal (DOS2x)
segread ..... SEGREAD ..... get contents of segment registers (all)
setusr ..... USER ..... set current user number (CPM86)
stat ..... CHMOD ..... get file attributes (DOS2x)
sysint ..... FARCALL ..... execute int instruction (all)
system ..... SYSTEM ..... call program or batch file (DOS2x)
time ..... TIME ..... get date & time (DOS2x)
tmpfile ..... TMPFILE ..... create & open temporary file (all)
tmpnam ..... TMPNAM ..... make name for temporary file (all)
utime ..... FILETIME ..... get or set file's date & time (DOS2x)
wait ..... FEXEC get rtn code from fexec-ed program (DOS2x)

NAME

access - determine accessibility of a file or directory

SYNOPSIS

int access (filename, mode)
char *filename;
int mode;

DESCRIPTION

access determines whether a file or directory can be accessed in the way that the calling function wants to access it. It can also be used to just test for the existence of a file or directory.

filename points to the name of the file or directory; this name optionally contains the drive and path of directories that must be passed through to get to the file or directory. If the drive component isn't specified, the file or directory is assumed to reside on the default drive. If the path component isn't specified, the file or directory is assumed to reside in the current directory on the specified drive.

mode is an *int* that specifies the type of access desired:

mode	*meaning*
4	read
2	write
1	execute (if a file) or search (if a directory)
0	check existence of the file or directory.

If the existence of the file or directory is being checked (ie, mode=0), *access* returns 0 if the file exists and -1 if it doesn't. In the latter case, *access* also sets the symbolic value ENOENT in the global integer *errno.*

When *access* is called to determine if a file can be accessed in a certain way (ie, *mode* isn't 0), *access* returns 0 if the file can be accessed in the desired manner; otherwise, it returns -1 and sets a code in the global integer *errno* that defines why the access is not permitted.

When asked, *access* says that a directory can be read or written; this means that a program can create and delete files on the directory, not that it can directly read and write the directory itself.

The symbolic values that *access* may set in *errno* when it's called with a non-zero *mode* parameter are:

errno	*meaning*
ENOTDIR	A component of the path prefix is not a directory.

ENOENT	The file or directory doesn't exist.
EACCES	The file or directory can't be accessed in the desired manner.

SEE ALSO

The "Errors" section of the Library Overview chapter discusses *errno*.

NAME

assert - verify program assertion

SYNOPSIS

#include <assert.h>

assert (expr)
int expr;

DESCRIPTION

assert is useful for putting diagnostic messages in a program. When executed, it will determine whether the expression *expr* is true or false. If false, it prints the message

Assertion failed: *expr*, file *fff*, line *lnnn*

where *fff* is the name of the source file and *nnn* is the line number of the *assert* statement.

To prevent assertion statements from being compiled in a program, compile the program with the option *-DNDEBUG*, or place the statement *#define NDEBUG* ahead of the statement *#include <assert.h>*.

NAME

bdos

SYNOPSIS

bdos(func, dx) /* CP/M-86 version */

DESCRIPTION

bdos issues CP/M-86 bdos call number *func*, with register DX set to *dx*.

It returns as its value the contents of AL, as set by CP/M-86.

NAME

bdosx, dosx -- perform a bdos call with a far pointer

SYNOPSIS

```
int bdosx(func, dsdxval, cxval)
int func;
int cxval;
int *dsdxval;

int dosx(func, bxval, cxval, dsdxval, dival, sival)
int func;
int bxval, cxval, sival, dival;
int *dsdxval;
```

DESCRIPTION

Many MSDOS/PCDOS system calls require an argument that is pointed to by the register pair ds-dx. This is no problem in small data models as the entire data segment is addressable from the default value of ds. In large data models, however, pointers always contain a segment portion and are not necessarily addressable with the default ds.

The *bdosx* and *dosx* functions provide equivalent functionality to bdos and dos calls, respectively, when such pointers are required in large data memory models.

NAME

sbrk, brk, rsvstk

SYNOPSIS

brk(ptr)
void *ptr;

void *sbrk(size)

rsvstk(size)

DESCRIPTION

sbrk and *brk* provide an elementary means of allocating and deallocating space from the heap. More sophisticated buffer management schemes can be built using these functions; for example, the standard functions *malloc*, *free*, etc call *sbrk* to get heap space, which they then manage for the calling functions.

sbrk increments a pointer, called the 'heap pointer', by *size* bytes, and, if successful, returns the value that the pointer had on entry. Initially, the heap pointer points to the base of the heap. *size* is a signed *int*; if it is negative, the heap pointer is decremented by the specified amount and the value that it had on entry is returned. Thus, you must be careful when calling *sbrk*: if you try to pass it a value greater than 32K, *sbrk* will interpret it as a negative number, and decrement the heap pointer instead of incrementing it.

brk sets the heap pointer to *ptr*, and returns 0 if successful.

For programs whose stack is below the heap, the size of the heap, and the amount of space allocated to the program, will expand or contract automatically as necessary, such that the resultant memory allocated to the program contains the smallest integral number of 1K-byte blocks needed to just contain the entire program and the location referenced by the updated heap pointer. For example, if *brk* is called to set the heap pointer either above the top of the space currently allocated to the heap or below the location currently pointed at by the heap pointer, *brk* computes the number of 1K-byte blocks that the program needs to just contain the location pointed at by the *brk* parameter, and then issues a DOS *setblock* call to allocate that number of blocks to the program.

For programs whose stack is above the heap, the heap cannot automatically grow. If a *sbrk* or *brk* call is made that would result in the heap pointer passing beyond the end of the heap, an error code is returned. For these programs, the function *rsvstk* can be called to set the heap-stack boundary *size* bytes below the current top of the stack, thus changing the amount of space allocated to the heap and stack.

SEE ALSO

The functions *malloc*, *free*, etc, implement a dynamic buffer-allocation scheme using the *sbrk* function. See the Dynamic Buffer Allocation section of the Library Functions Overviews chapter for more information.

The standard i/o functions usually call *malloc* and *free* to allocate and release buffers for use by i/o streams. This is discussed in the Standard I/O section of the Library Functions Overviews.

Your program can safely mix calls to the *malloc* functions, standard i/o calls, and calls to *sbrk* and *brk*, as long as the your calls to *sbrk* and *brk* don't decrement the heap pointer. Mixing *sbrk* and *brk* calls that decrement the heap pointer with calls to the *malloc* functions and/or the standard i/o functions is dangerous and probably shouldn't be done by normal programs.

For more information on the heap and its relationship to the other areas of a program, see the Program Organization section of the Technical Information chapter.

ERRORS

sbrk and *brk* return -1 if an error occurs, after setting the global integer *errno* to the symbolic value ENOMEM.

NAME

chmod, stat

SYNOPSIS

```
chmod(name, attr)        /* DOS 2.x functions */
char *name;

stat(name, buf)
char *name, *buf;
```

DESCRIPTION

chmod sets the attribute byte of file *name* to *attr.*

stat returns the attribute byte, date and time, and size of the file *name.* This information is returned in *buf*, which has the following format:

```
struct stat {
    char st_attr;
    long st_mtime;
    long st_size;
}
```

This structure, and the meaning of the bits in the attribute and time fields are defined in the header file *stat.h*, and in the TIME section.

name can optionally specify the drive on which the file is located and the path to it.

ERRORS

chmod and *stat* return -1 if they fail, after setting a code in the global integer *errno.* The Errors section of the Library Overview chapter describes these codes.

NAME

circle, set_asp

SYNOPSIS

```
void circle (x, y, r)    /* IBM PC-only function */
int x, y, r;

void set_asp (x_asp, y_asp)  /* IBM PC-only function */
int x_asp, y_asp;
```

DESCRIPTION

circle draws a circle with center (x,y) and radius r.

The aspect of the circle is determined by the global variables, *_xaspect* and *_yaspect*. These values are set automatically when the *mode* function is called, and can also be altered directly or by calling *set_asp*.

SEE ALSO

color, line, mode, point

NAME

clock - get time

SYNOPSIS

#include <time.h>

clock_t clock()

DESCRIPTION

clock is used to determine the time interval between events that occur within a 48-hour period.

clock usually returns the number of hundredths of seconds that have elapsed since the beginning of the day. If a new day has begun since the last call to *clock*, *clock* instead returns the number of hundredths of seconds that have elapsed since the beginning of the previous day. The returned value can be divided by the macro *CLK_TCK*, which is defined in *time.h*, to determine the number of seconds that have elapsed.

clock_t, which is also defined in *time.h*, is a *long*, allowing intervals between calls to *clock* to be determined by simply subtracting the values returned by *clock*.

NAME

color, palette, ground

SYNOPSIS

```
void color (c)     /* IBM PC-only function */
int c;

void palette (c)   /* IBM PC-only function */
int c;

void ground (c)    /* IBM PC-only function */
int c;
```

DESCRIPTION

color defines the color that will be used when a point is plotted by *point*. This information is stored in the global variable, *__color*. The argument is interpreted as follows:

'w', 'W', 'y', 'Y', 3	- white or yellow
'm', 'M', 'r', 'R', 2	- magenta or red
'c', 'C', 'g', 'G', 1	- cyan or green
anything else	- background

The colors that are used depend upon the current color configuration.

palette sets the current palette with int 16. The palette can be either "cyan, magenta, white" or "green, red, yellow", chosen by a non-zero argument and a zero argument, respectively.

ground sets the background color with int 16. The argument can be valued 0 - 15. The background color will be set as follows:

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

SEE ALSO

mode, circle, line, point

NAME

_csread

SYNOPSIS

_csread(fd, addr, len)
char *addr;

DESCRIPTION

_csread is equivalent to the function *read*, except it reads data into the code segment rather than the data segment.

fd is the file descriptor associated with the file or device to be read;

addr is the offset within the code segment into which data is to be read;

len is the number of bytes to be read.

_csread returns the number of bytes read. 0 means that the end of the input has been reached.

ERRORS

_csread returns -1 if an error occurs, after setting a code in the global integer *errno*. The codes are defined in the Errors section of the Library Overview chapter.

NAME

mkdir, rmdir, chdir, getcwd

SYNOPSIS

```
mkdir(name)          /* DOS 2.x functions */
char *name;

rmdir(name)
char *name;

chdir(name)
char *name;

char *getcwd(buf, size)
char *buf;
```

DESCRIPTION

These functions perform directory-related activities.

mkdir, *rmdir*, and *chdir* create a directory, remove a directory, and change the current directory, respectively. For them, the argument *name* is a character string specifying the drive on which the directory is located and the path to the directory. They return 0 if no error occurred.

getcwd returns as its value a pointer to a character string specifying the name of the current working directory on the default drive. If the first parameter to *getcwd*, that is, *buf*, is non-null, the name is placed in *buf*. Otherwise, *getcwd* calls *malloc* to dynamically allocate a buffer of *size* bytes into which the name is placed.

ERRORS

If *mkdir*, *rmdir*, or *chdir* fails, it sets an error code in the global integer *errno* and returns -1 as its value. These codes are defined in the Errors section of the Library Overview chapter.

If *getcwd* fails, it returns 0, after freeing the dynamically allocated buffer, if any.

NAME

dos, bdos

SYNOPSIS

dos(func, bx, cx, dx, di, si) /* DOS 2.x Function */

bdos(func, dx, cx) /* DOS 1.1 & 2.x Function */

DESCRIPTION

These functions issue a DOS functions call, using interrupt 21h.

func is the number of the function. The number can be in the low or high order byte of *func*; in either case, it will be loaded into register AH.

The other arguments are loaded into registers before the int 21h is performed: for example, the argument *cx* is loaded into register CX.

For the *dos* function, if int 21h returns with the carry flag reset, *dos* returns as its value the contents of register AX, as set by int 21h. Otherwise, the contents of AX are set in the global integer *errno* and *dos* returns -1 as its value.

The *bdos* function always returns the contents of AL, as set by int 21h, as its value.

NAME

dup, fdup

SYNOPSIS

dup(oldfd) **/* DOS 2.x functions */**

fdup(oldfd, newfd)

DESCRIPTION

These functions perform a second opening, for unbuffered i/o, of a file or device, returning as their value a file descriptor. The file or device can then be accessed by either the original file descriptor or the returned file descriptor.

The parameter *oldfd* is the original file descriptor.

dup and *fdup* differ in that *dup* selects the second file descriptor, while the caller passes *fdup* the second file descriptor.

For *fdup*, if the second file descriptor is already associated with an open file or device, it will be closed before being reopened.

ERRORS

If *dup* or *fdup* fails, it sets an error code in the global integer *errno* and returns -1 as its value. These codes are defined in the Errors section of the Library Overview chapter.

NAME

execl, execv, execlp, execvp

SYNOPSIS

execl(name, arg0, arg1, arg2, ..., argn, 0)
char *name, *arg0, *arg1, *arg2, ...;

execv(name, argv)
char *name, *argv[];

execlp(name, arg0, arg1, arg2, ..., argn, 0)
char *name, *arg0, *arg1, *arg2, ...;

execvp(name, argv)
char *name, *argv[];

DESCRIPTION

These functions load, and transfer control to, another program. The called program is loaded on top of the calling program; thus, if the exec function succeeds, it doesn't return to the caller.

Both programs which have been created using the Aztec software and those that haven't can be called.

On DOS, Both *.com* and *.exe* programs can be called.

The following paragraphs will first describe the parameters to the exec functions, then describe the differences between the functions, and finally discuss other features of the functions.

Parameters

name is the name of the file containing the program to be loaded. On DOS, *name* can specify the drive on which the file is located, and the path to the file. On CP/M-86, *name* can specify the drive on which the file is located.

The exec functions can pass arguments to the called program. *execl* and *execlp* build a command line by concatenating the strings pointed at by *arg1*, *arg2*, and so on. If a C program is being called, its *main* function will see *arg1* as *argv[1]*, *arg2* as *argv[2]*, and so on. *arg0*, which on UNIX is normally the name of the program being called, isn't part of the constructed command line; also, *argv[0]* of a called C program is always a pointer to a null string. Even though *arg0* isn't passed to a program when using Aztec C86, it must still be specified, even if it's just a pointer to a null string. We recommend that you set it to a pointer to the name of the called program, for UNIX compatibility.

execv and *execvp* build a command line by concatenating the strings pointed at by *argv[1]*, *argv[2]*, and so on. The *argv* array must be have a null pointer as its last entry. If a C program is

being called, its *main* function will see the calling function's *argv[i]* as its *argv[i]*. *argv[0]*, which on UNIX is normally the name of the program being called, isn't part of the constructed command line; also, *argv[0]* of a called C program is always a pointer to a null string. Even though *argv[0]* isn't passed to a program when using Aztec C86, it must still be specified, even if it's just a pointer to a null string. We recommend that you set it to a pointer to the name of the called program, for UNIX compatibility.

The Functions

execl and *execv* load a program from the specified file: *execl* is useful when a fixed number of arguments are being passed to a program. *execv* is useful for programs which are passed a variable number of arguments.

On DOS, *execlp* and *execvp* search a list of directories for the program to be loaded. The first directory searched is the current directory on the default drive. If the program isn't there, the directories specified in the PATH environment variable are searched. For these two functions, if the filename doesn't have an extension, the exec functions will first append *.com* to the name and search for a file having this extended name. If this search fails, *.exe* will be appended to the name and the search will begin again.

On CP/M-86, *execl* and *execlp* are identical, as are *execv* and *execvp*.

DOS Information

If an exec function fails, for example because the file doesn't exist, it will return -1 as its value.

Files opened for unbuffered i/o in the calling program will also be open in the called program, and will have the same file descriptors.

Files opened for standard i/o in the calling program won't be open for standard i/o in the called program, although they will be open for unbuffered i/o. Thus, before a program activates another using an exec function, it should cause the buffered data for files opened for standard i/o to be written to disk, using either the *fclose* or *fflush* functions.

The standard input, standard output, and standard error devices are open in the called program to the same devices or files as in the calling program. For the reasons discussed above, care is needed when either the calling or called program accesses these logical devices using standard i/o calls.

The environment of the called program is the same as that of the calling program.

CP/M-86 Information

On CP/M-86, files open in the calling program aren't open in the called program. Thus, the calling program should close all files before issuing an exec function.

SEE ALSO

The FEXEC functions also load and execute another program; they differ from the EXEC functions in that they return to the caller.

The *system* function will execute a DOS command, whether built-in, batch file, or executable file.

NAME

exit, _exit

SYNOPSIS

exit(code)

_exit(code)

DESCRIPTION

These functions cause a program to terminate and control to return to the operating system.

For programs which are activated by batch or submit files, *code* can be used to control the continuation of them following the termination of the program. On MSDOS and PCDOS, the batch file can directly examine *code*, and act accordingly.

On CP/M-86, if *code* is non-zero and if the A: drive is logged in, the file A:$$$.SUB will be erased, thus preventing the continuation of an active submit file.

exit and *_exit* differ in that *exit* closes all files opened for standard i/o, while *_exit* doesn't.

NAME

farcall, sysint

SYNOPSIS

farcall(ip, cs, inregs, outregs)
struct regs *inregs, *outregs;

sysint(sint, inregs, outregs)
struct regs *inregs, *outregs;

DESCRIPTION

farcall issues a far call to the function located at *cs:ip.*

sysint issues interrupt number *sint.*

Before the far call or interrupt is issued, the registers are loaded from the block of memory pointed at by *inregs*; after the far call or interrupt returns, the registers are stored in the block of memory pointed at by *outregs.*

inregs and *outregs* can point to the same block of memory.

farcall returns as its value the contents that were in the processor's status register on return from the far call function.

sysint returns as its value the state of the flags as set by the interrupt routine.

The blocks pointed at by *inregs* and *outregs* have the following format:

```
struct regs {
    int AX;
    int BX;
    int CX;
    int DX;
    int SI;
    int DI;
    int DS;
    int ES;
```

NAME

fcbinit

SYNOPSIS

fcbinit(name, fcb)
char *name, *fcb;

DESCRIPTION

fcbinit initializes an fcb pointed at by *fcb* with the filename pointed at by *name.*

The following comments apply to the *name* parameter:

* The drive identifier in the file name is optional; if not present, the file is assumed to be on the default drive;
* The extension in the file name is optional; if not present, it's assumed to be all blanks;
* The characters ? and * can appear in the file name; in the latter case, then it and all remaining characters in the filename or extension are set to ?;
* On DOS, the name can't specify a path, since the fcb will be used for DOS 1.1-compatible i/o, which doesn't support paths;
* On CP/M-86, the name can specify a user number. See the I/O overview section for the format of a name.

On DOS, *fcbinit* can return the following values:

0	If no errors occurred;
1	If ? or * was in the file name;
255	If the drive identifier was invalid;
-1	If the file name was invalid.

On CP/M-86, it can return:

the user number,	if specified in the file name;
255	if the user number wasn't specified.

NAME

fexecl, fexecv

SYNOPSIS

fexecl(name, arg0, arg1, arg2,..., argn, 0) /* DOS 2.x Functions */
char *name, *arg0, *arg1, *arg2, ..., *argn;

fexecv(name, argv)
char *name, *argv[];

wait()

DESCRIPTION

The fexec functions load and call another program. The calling program is suspended while the called program is executing; the fexec function returns when the called program terminates.

wait returns as its value the return code from the fexec-executed program.

The parameters

name specifies the name of the file from which the program is to be loaded, and optionally, the drive on which it's located and the path to it.

The fexec functions can pass arguments to the called program. *fexecl* builds a command line by concatenating the strings pointed at by *arg1*, *arg2*, and so on. If a C program is being called, its *main* function will see *arg1* as *argv[1]*, *arg2* as *argv[2]*, and so on. *arg0*, which on UNIX is normally the name of the program being called, isn't part of the constructed command line; also, *argv[0]* of a called C program is always set to a pointer to a null string. Even though *arg0* isn't passed to a program when using Aztec C86, it must still be specified, even if it's just a pointer to a null string. We recommend that you set it to a pointer to the name of the called program, for UNIX compatibility.

fexecv builds a command line by concatenating the strings pointed at by *argv[1]*, *argv[2]*, and so on. The *argv* array must have a null pointer as its last entry. If a C program is being called, its *main* function will see the calling function's *argv[i]* as its *argv[i]*. *argv[0]*, which on UNIX is normally the name of the program being called, isn't part of the constructed command line; also, *argv[0]* of a called C program is always a pointer to a null string. Even though *argv[0]* isn't passed to a program when using Aztec C86, it must still be specified, even if it's just a pointer to a null string. We recommend that you set it to a pointer to the name of the called program, for UNIX compatibility.

The Functions

fexecl is useful when a fixed number of arguments must be passed, and *fexev* when a variable number of arguments must be passed.

Other Information

The fexec functions load the called program after the calling program in memory.

Files opened for unbuffered i/o in the calling program will also be open in the called program, and will have the same file descriptors.

Files opened for standard i/o in the calling program won't be open for standard i/o in the called program, although they will be open for unbuffered i/o. Thus, before a program activates another using an fexec function, it should cause the buffered data for files opened for standard i/o to be written to disk, using either the *fclose* or *fflush* functions.

The standard input, standard output, and standard error devices are open in the called program to the same devices or files as in the calling program. For the reasons discussed above, care is needed when either the calling or called program accesses these logical devices using standard i/o calls.

The environment of the called program is the same as that of the calling program.

SEE ALSO

The EXEC functions also load programs. Since they overlay the calling program, they allow a larger program to be loaded. They never return to the caller.

The *system* function can execute a DOS built-in command, batch command, or executable file.

ERRORS

If an fexec function fails, it returns -1 as its value after setting a code in the global integer *errno*. These codes are defined in the Errors section of the Library Overview chapter.

NAME

ftime, utime

SYNOPSIS

```
long ftime(0, fd)                    /* get */

long ftime(1, fd, newtime)           /* set */
long newtime;

struct utimebuf {
   long actime;  /* not used on DOS */
   long modtime; /* modification date and time */
};

utime(name, timeptr)
char *name; struct utimebuf *timeptr;
```

DESCRIPTION

These functions are used to get and set a file's date and time, which are stored in the directory entry for the file.

ftime can be used when the file is already open; the *fd* argument is the unbuffered i/o file descriptor associated with the file.

ftime can both get and set a file's date and time:

If the first argument to *ftime* is 0, *ftime* returns as its value the date and time for the file.

If the first argument to *ftime* is non-zero, *ftime* sets the file's date and time to the value in *newtime*. It returns 0 if no errors occurred.

utime can be used to set the date and time for a file, which can be either open or closed. *name* points to a character string specifying the name of the file, the drive it's on, and the path to it.

If *timeptr* is not a null pointer *utime* will set the file's date and time to the the value of the *modtime* field in the structure pointed at by *timeptr*. If *timeptr* is a null pointer, *utime* will set the file's date and time to the current date and time.

utime returns 0 if no errors occurred.

The *long* time and date fields that are passed to, and returned by, these functions have the following format (bit 0 is the least significant bit in the field, bit 31 the most significant):

bits	*meaning*
0-4	seconds/2
5-10	minutes
11-15	hours
16-20	day of month
21-24	month (0=Jan,...)
25-31	year since 1980

ERRORS

If an error occurs, *ftime* and *utime* return -1, after setting a code in the global integer *errno*. These codes are defined in the Errors section of the Library Overview chapter.

NAME

getenv

SYNOPSIS

```
char *getenv(name)          /* DOS 2.x function */
char *name;
```

DESCRIPTION

getenv returns a pointer to the character string associated with the environment variable *name*, or 0 if the variable isn't in the environment.

The character string is in a static buffer and will be overwritten when the next call is made to *getenv*.

NAME

line, lineto

SYNOPSIS

void line (x1, y1, x2, y2) /* IBM PC-only function */
int x1, y1, x2, y2;

void lineto (x2, y2) /* IBM PC-only function */
int x2, y2;

DESCRIPTION

line draws a line from (x1, y1) to (x2, y2).

lineto draws a line from the last point drawn to or plotted, to (x2, y2).

lineto assumes that the coordinates of the point at which it is to start are stored in the global integers *__oldx* and *__oldy*. These fields are set by:

* *line* and *lineto*, to the coordinates of the last point plotted in these fields;
* The point-plotting function, *point*, to the coordinates of the plotted point;
* Direct assignment from the user's program.

SEE ALSO

circle, color, mode, point

NAME

ptrtoabs & abstoptr - long pointer conversion functions

SYNOPSIS

```
long ptrtoabs(lptr)
void * lptr; /* lptr is a long pointer */

void *abstoptr(laddr)
long laddr;

void * _ptradd(lptr, val)
void *lptr; long val;

long _ptrdiff(lptr1, lptr2)
void *lptr1, *lptr2;
```

DESCRIPTION

ptrtoabs takes a long pointer *lptr* that is in segment/offset form and returns as its value the absolute address of the referenced location.

abstoptr takes the absolute address of a location and returns as its value a long pointer to the location, in segment/offset form. The segment is selected so that the offset is between 0 and 15.

_ptradd takes a long value *val* and a long pointer *lptr*, and returns as its value the sum of the two.

_ptrdiff takes two long pointers, *lptr1 lptr2*, that are in segment/offset form, and returns as its value their difference.

SEE ALSO

The compiler chapter discusses long pointers and absolute addresses and demonstrates how these functions can be used to access arrays that contain more than 64K bytes.

NAME

memccpy, memchr, memcmp, memset - memory operations

SYNOPSIS

```
#include <memory.h>

char *memccpy (dst, src, c, n)
char *dst, *src;
int c, n;

char *memcpy (dst, src, n)
char *dst, src;
int n;

char *memset (buf, c, n)
char *buf;
int c,n;

char *memchr (buf, c, n)
char *buf;
int c,n;

int memcmp (buf1, buf2, n)
char *buf1, *buf2;
int n;
```

DESCRIPTION

These functions operate efficiently on areas of memory, copying characters from one area to another, searching an area for a character, and setting the bytes in an area to a specified value. Unlike the string functions, which are described in the STRING section, these functions don't automatically stop when they find a null character.

memccpy copies characters from the memory area *src* into *dst*, stopping after the first occurrence of the character *c* or after *n* characters have been moved, whichever comes first. If *c* was found and moved, *memccpy* returns a pointer to the byte following *c* in *dst*; otherwise, it returns a NULL pointer.

memcpy copies *n* bytes from memory area *src* to *dst*, and returns *dst* as its value.

memccpy and *memcpy* always copy from the first byte in *src* to the last. Because of this, if the two areas overlap, the *dst* area may not be an exact copy of the *src*.

memset sets the first *n* bytes in memory area *buf* to *c*, and returns *buf* as its value.

memchr searches the first *n* bytes of the memory area *buf* for the character *c*. If it finds the character, it returns as its value a pointer to it; otherwise, it returns a NULL pointer.

memcmp compares the first *n* bytes of the memory areas pointed at by *buf1* and *buf2*. It returns an integer that is less than, equal to, or greater than 0, depending on whether *buf1* is lexicographically less than, equal to, or greater than *buf2*.

The file *memory.h* declares the types of these functions.

SEE ALSO

The non-UNIX functions *movmem* and *setmem* are equivalent to *memcpy* and *memset*, respectively, except that their parameters are in a different order.

Functions for operating on null-terminated strings are described in the STRING section of the system-independent function section.

NAME

mktemp - make a unique file name

SYNOPSIS

```
char *
mktemp (template)
char *template;
```

DESCRIPTION

mktemp replaces the character string pointed at by *template* with the name of a non-existent file, and returns as its value a pointer to the string.

The string pointed at by *template* should look like a file name whose last few characters are *X*s with an optional imbedded period.

mktemp replaces the *X*s with a letter followed by the least significant digits of the starting address of its program's data segment. The letter will be between 'A' and 'Z', and will be chosen such that the resulting character string isn't the name of an existing file.

DIAGNOSTICS

For a given character string, *mktemp* will try to convert the string into one of 26 file names. If all of these files exist, *mktemp* will replace the first character pointed at by *template* with a null character.

SEE ALSO

tmpfile, tmpnam

EXAMPLES

The following program calls *mktemp* to get a character string that it can use as a file name. If the program's data segment begins at the decimal address 123456, then the generated name will be one of the strings *abcA23.456*, *abcB23.456*, ..., *abcZ23.456*. If all these strings are the names of existing files, *mktemp* will replace the first character of the string passed to it, *a* in this case, with 0.

```
#include <stdio.h>
main()
{
    char *fname, *mktemp();
    FILE *fp, fopen();
    fname=mktemp("abcXXX.XXX")==0)
    if (!*fname){
        printf("mktemp failed");
        exit(1);
    } else
        fp=fopen(fname, "w");
    ...
}
```

NAME

mode - set screen mode

SYNOPSIS

int mode (c) /* IBM PC-only function */
int c;

DESCRIPTION

This function, which can only be used on an IBM PC, sets the screen mode by issuing an interrupt hex 10. The parameter is interpreted as follows:

0 -	40 x 25 b/w
1 -	40 x 25 color
2 -	80 x 25 b/w
'l', 'L', or 3 -	80 x 25 color
'm', 'M', or 4 -	320 x 200 color
5 -	320 x 200 b/w
'h', 'H', 6 -	640 x 200 b/w

An argument out of range will cause the function to return a -1 and take no action. Otherwise, the following global variables are set:

__plotf - to determine the plotting function for *point*
__xaspect, __yaspect - to determine curvature of circles
__max__x - limit on horizontal coordinate

SEE ALSO

point, circle, line, color

DIAGNOSTICS

mode will return -1 when its argument is out of range.

EXAMPLE

To enter high res mode, the following calls are equivalent:

```
mode ('H');
mode (6);
```

NAME

monitor, int_sp - profiling functions

SYNOPSIS

```
    /* functions for IBM PC & compatibles only */
int monitor(lowpc, highpc, buffer, size, numcalls)
int (*lowpc)();
int (*highpc)();
short *buffer;
int size;
int numcalls;

int _int_sp(speed)
int speed;
```

DESCRIPTION

monitor is a function which sets up the IBM-PC to perform runtime analysis of where the user program is spending its execution time. This is accomplished by trapping the IBM-PC clock interrupt and recording a tick if the current execution address at the clock interrupt is in the address range being analyzed.

Once the analysis is complete the tick summary is written to a file called mon.out which can be used as input to the *prof* utility to produce a report of runtime activity. *monitor* is called once with non-zero arguments to initiate analysis and once with all zero arguments to terminate analysis.

Since the IBM-PC clock is set normally to such a poor time granularity (~18.2 interrupts/sec), a special routine *_int_sp* is provided to set the monitoring clock speed to a higher rate. Rates permitted are from 18 (the default rate) to 120 interrupts per second. 60 interrupts per second provides a reasonable time granularity. *monitor* will restore the default speed when it is called to write out the mon.out file. In addition, any exit or Cntrl-C exit from the program will be trapped and will reset the default speed.

Non-IBM-PC users may be able to produce a working version of *monitor* for their systems by customizing *clk.asm*, which is in dos20.arc (Commercial package only). Otherwise, *monitor* should not be used on anything but guaranteed IBM-PC compatibles.

These functions have only been tested on IBM-PC, XT, and AT processors.

EXAMPLE

The simplest way to describe the use of monitor is through an example.

Suppose there is a program *foo.c* for which analysis is desired. At the start of the main routine of *foo.c*, place the following code:

```
#ifdef MONITOR
/* _Corg & _Cend are always the first & last addresses */
/* in the program: */
int _Corg();
int _Cend();
#define MONSIZE 2000   /* use fine granularity */
/* buffer for monitor to gather ticks: */
short monbuf[MONSIZE];
#endif
```

Then comes *main* for *foo.c* with the calls to *monitor.*

```
/* ... Global declarations ... */
main()
{
   /* ... local declarations ... */
   /* ... first executable statement of program ... */
#ifdef MONITOR
                        /* set clock for 60 interrupts per second:
   int_sp(60);
   /* start monitoring  the program: */
   monitor(_Corg, _Cend, monbuf, MONSIZE, 0);
#endif
  /* ... main body of program ... */
   /* ... last statement of program ... */
#ifdef MONITOR
   /* turn off monitoring and  write out mon.out file: */
   monitor(0,0,0,0,0);
#endif
}
```

In this example, all of the functions of the program are being monitored, since the monitor routine's arguments *lowpc* and *highpc* are set to the erigin and end of the user program, respectively. These arguments control how much of the program will be examined. If the arguments were main and printf, respectively, all of the functions physically before main and after (and including) printf would be ignored in the analysis. These arguments to monitor MUST be declared as functions prior to the monitor call.

The remaining arguments to monitor *monbuf* and MONSIZE, provide the system with an area to store tick counts prior to

writing the mon.out file. It is important to make the buffer provided as large as possible, since the number of locations in this array determines the space granularity of the analysis.

For example, if the code for the program is 5000 bytes long and the entire program is being analyzed, a buffer size of 1000 would mean that every five bytes of the program would have its own bucket for collecting ticks, while a buffer size of 100 would mean that every 50 bytes of program would have its own bucket. Functions shorter than 50 bytes might not show up at all in the prof report for such a program, even though they were heavily executed.

The last argument, *numcalls*, represents an as yet unimplemented facility in *monitor*, and is provided for UNIX(tm) compatibility only.

Linking a program with monitor calls

When linking a program containing monitor calls, the user should be careful to use the *-t option* which produces a symbol table for the program, as this is needed for running the *prof* utility which produces the report.

NAME

movblock

SYNOPSIS

movblock(from_off, from_seg, to_off, to_seg, len)

DESCRIPTION

movblock moves a block of data from one area of memory to another.

from_seg and *from_off* define the beginning of the source block: *from_seg* is the paragraph number of the segment in which it's located and *from_off* is the offset in bytes of the block from the beginning of this segment.

to_seg and *to_off* define the beginning of the destination block: *to_seg* is the paragraph number of the segment in which it's located, and *to_off* is the offset in bytes of the block from the beginning of this segment;

len is the number of bytes to move.

movblock always copies from the beginning of the source block to the end.

NAME

peekw, peekb, pokew, pokeb - examine & modify memory

SYNOPSIS

```
/* peeks & pokes taking a long pointer: */

peekw(lptr)
void *lptr; /* lptr is a long pointer */

peekb(lptr)
void *lptr;

pokew(lptr, val)
void *lptr;

pokeb(lptr, val)
void *lptr;

/* peeks & pokes when the segment & offset components */
/* are passed as separate arguments: */

peekw(offset, segment)

peekb(offset, segment)

pokew(offset, segment, val)

pokeb(offset, segment, val)
```

DESCRIPTION

These functions get and set either one or two bytes located anywhere in memory.

When a program uses long pointers, the entire pointer to the location to be accessed can be passed as a single argument. In the synopsis, this is shown as *lptr*.

When a program uses short pointers, the pointer to the location to be accessed must be passed as two arguments. A program using long pointers can also pass the location address as two arguments, if desired. In the synopsis, these two arguments are shown as *segment* and *offset*, where *segment* is the paragraph number of the segment in which the target field is located and *offset* is the offset in bytes of the target field from the beginning of this segment.

peekw returns as its value the word (that is, two bytes) at the target location.

peekb returns as its value the byte at the target location.

pokew sets the word at the target location to *val*.

pokeb sets the byte at the target location to *val*.

NAME

perror, errno, sys_errlist, sys_nerr - system error messages

SYNOPSIS

```
int perror (s)
char *s;

#include <errno.h>

extern int errno;

extern char *sys_errlist[];

extern int sys_nerr;
```

DESCRIPTION

When a library function detects an error, it will generally set an error code, which is a positive integer, in the global integer *errno* and return an appropriate, function-dependent value.

sys_errlist is an array of pointers to character strings, each of which is a message corresponding to an *errno* error code. That is, when an error occurs, *errno* can be used as an index into *sys_errlist* to get a message corresponding to the error. The messages don't contain a newline character.

The maximum value that can be placed in *errno*, and the total number of entries in *sys_errlist*, is in the global integer *sys_nerr*.

The *extern* declarations of *errno*, *sys_errlist*, and *sys_nerr* are in *errno.h*.

When an error occurs, *perror* can be called to write a message describing the error on the standard error device. The message consists of the following:

* *s*, the string pointed at by the argument to *perror*,
* a colon and a blank,
* the *sys_errlist* message corresponding to the current value of *errno*,
* a newline character.

perror returns 0 if *errno* contains a valid value; otherwise it returns -1 without printing a message.

SEE ALSO

Error Overview (O)

NAME

point

SYNOPSIS

```
int point (x, y)        /* IBM PC-only Function */
int x, y;
```

DESCRIPTION

This function plots a point on the medium or high resolution screen. The origin, (0,0), is in the lower left hand corner of the screen.

A second function is also provided in source form. It performs a system interrupt to plot the point. This is included for compatibility reasons.

Either function will set *_oldx* and *_oldy* to the values, x and y, respectively.

SEE ALSO

circle, color, line, mode

NAME

inportb, inportw, outportb, outportw

SYNOPSIS

inportb(port)

inportw(port)

outportb(port, val)

outportw(port, val)

DESCRIPTION

These functions transfer data to and from the i/o device whose address is *port.*

inportb inputs a single byte, which it returns as its value.

inportw inputs a word (two bytes), which it returns as its value.

outportb outputs the least significant byte of *val.*

outportw outputs both bytes of *val.*

NAME

scdir -- return the name of the next file matching pattern

SYNOPSIS

```
char * scdir(pat)
char *pat;
```

DESCRIPTION

scdir is a function which permits the user to perform wild card expansion on file name patterns using native MSDOS/PCDOS facilities.

When *scdir* is called with a pattern, it returns a pointer to a static area containing the null terminated name of the next file which matches the pattern or zero if no more files match the pattern. Since the area containing the name is statically allocated, the name will be overwritten by subsequent calls to *scdir.*

EXAMPLE

```
main()
{
        char *sav[100];
        register char  *pat;
        register int i;

        /* find all c files in current dir */

        pat = "*.c";
        i = 0;

        while ((ptr = scdir(pat)) && i < 100) {

        sav[i] = malloc(strlen(ptr)+1);
        strcpy(sav[i++], ptr);

        }
        /* rest of program */
}
```

NAME

screen manipulation functions:
scr_clear, scr_home, scr_curs, scr_loc, scr_eol,
scr_eos, scr_linsert, scr_ldelete, scr_cinsert,
scr_cdelete, scr_invers, scr_echo, scr_putc,
scr_getc, scr_poll, scr_call
scr_printf, scr_puts, scr_setatr, scr_getatr, scr_resatr

SYNOPSIS

```
scr_setatr(background,forground,intensity,blink)

scr_getatr()

scr_resatr(attribute)

scr_invers(flg)
int flg;
scr_clear()      /* IBM PC-only functions */

scr_home()

scr_curs(lin, col)
int lin, col;

scr_loc(lin, col)
int *lin, *col;

scr_eol()

scr_eos()

scr_linsert()

scr_ldelete()

scr_cinsert()

scr_cdelete()

scr_echo(flg)
int flg;

scr_putc(c)
int c;

scr_getc()

scr_poll()

scr_puts(str)
char *str;

scr_printf(fmt, args ...)
char *fmt;

scr_call(ax, bx, cx, dx)
int ax, bx, cx, dx;
```

DESCRIPTION

These functions can only be used on IBM PC systems. They access the console and keyboard by making calls directly to the ROM BIOS. There are functions to clear the screen, position the cursor, insert and delete characters and lines, enable and disable inverse video mode, enable and disable echo mode, get and put characters, and issue the video i/o int 10 assembly language call.

These functions are designed to be used on a console that is in 80x25 mode, although some of them may work in other modes.

Setting character attributes

When writing a character to the console, many of these functions set the attributes of the character to the value contained in the least significant byte in the global integer *_attrib*. The following functions access *_attrib*:

scr_setatr() sets the _attrib variable to the given foreground, background colors and for blinking and intensity. There are 8 colors defined in *color.h*. The background may use any one of those colors. The foreground may also have the intensity set, so that there are 16 colors for foreground.

scr_setatr() returns the packed attribute, which can be used in *scr_resatr()*.

scr_getatr() returns the current attribute setting, which can also be used in *scr_resatr()*.

scr_resatr() returns the old attribute setting, and installs the new attribute specified.

scr_invers sets *_attrib* to 0x7 or 0x70, depending on whether the *flg* parameter to *scr_invers* is zero or non-zero, respectively. An attribute of 7 causes a character to be black on a white background, while 0x70 causes it to be white on a black background. For both values, the characters will be visible, non-blinking, and normal intensity.

Other screen functions

scr_clear sets each character on the screen to a blank, and sets the attributes of each character to the value contained in *_attrib*.

scr_home homes the cursor to the upper left hand corner of the screen.

scr_curs moves the cursor to the line and column specified by the *lin* and *col* parameters, respectively. With the console in 80x25 mode, there are 25 lines, each containing 80 columns or

characters; line numbers range between 0 and 24, inclusive; column numbers range between 0 and 79, inclusive; and the line and column number of the top left corner of the screen are both 0.

scr_loc places the line and column number at which the cursor is located in *lin* and *col*, respectively.

scr_eol erases the line at which the cursor is located, from the current cursor position to the end of the line. It does this by writing a blank character to each position that is to be erased, and setting its attribute to the value contained in *_attrib*.

scr_eos erases the screen from the current cursor position to the end of the screen. It does this by writing a blank character to each position that is to be erased, and setting its attribute to the value contained in *_attrib*.

scr_linsert inserts a line of blank characters at the cursor location, moving the lines below the cursor down one line. The attributes of the characters on the inserted line are set to the value contained in *_attrib*.

scr_ldelete deletes the line at the cursor location, moving the lines below the cursor up one line and placing a line of blank characters at the bottom of the screen. The attributes of the characters on the inserted line are set to the value contained in *_attrib*.

scr_cinsert inserts a blank character at the cursor location, shifting right one character the characters in the line which are on the right of the cursor. The attributes of the position at which the blank was written are set to the value contained in *_attrib*.

scr_cdelete deletes the character at the cursor location, shifting left one character the characters in the line which are on the right of the cursor. A blank character is written to the last position on the line (column number 79), and its attributes are set to the value contained in *_attrib*.

scr_putc writes *c*, the character that is passed to it, to the current cursor location, setting the attributes of the position to the value contained in *_attrib*.

scr_echo sets the global integer *_echo* to the value specified by the *flg* parameter to *scr_echo*. This variable controls the echoing of characters by *scr_getc*, as defined below.

scr_getc reads a character from the keyboard, waiting if a key hasn't been depressed, and echoes it to the screen if the global integer *_echo* is non-zero. *scr_getc* returns one of the following values:

* For normal characters, its ASCII value (a number between decimal 0 and 127).
* For special characters, a number between 128 and 255.
* For control-break, -2.

Special characters are those for which the ROM BIOS returns an extended code consisting of a null character followed by a second code. For these characters, *scr_getc* returns as its value the second code returned by the BIOS, with 0x80 OR-ed in. As an example, for the function key F1 the ROM BIOS returns an extended function whose second code is 0x3b; so for this key *scr_getc* returns 0xbb. See the "Keyboard Encoding and Usage" section of the IBM Technical Reference manual for a complete list of the keys for which the ROM BIOS returns an extended code.

scr_poll is used to determine if a key has been typed, without waiting if not, without removing the key from the BIOS input buffer, and without echoing the character to the console. *scr_poll* returns -1 if no character is available; otherwise it returns values that have the same meanings as *scr_getc*.

Since *scr_poll* doesn't remove a character from the input buffer, a program that has determined, by calling *scr_poll*, that a character is available should then call *scr_getc* to actually read the character and remove it from the input buffer.

scr_printf() and *scr_puts()* work exactly the same as their standard C counterparts. The difference is that these routines can use the 3 attribute setting routines to allow for color.

scr_call issues the ROM BIOS video i/o call, int 10, after loading the registers AX, BX, CX, and DX with *scr_call*'s parameters *ax*, *bx*, *cx*, and *dx*. It returns as its value the contents of AX.

EXAMPLES

Here is an example that uses *scr_setatr* to change the color that *scr_printf* and *scr_puts* will use next. The default color used by *scr_printf* and *scr_puts* will be the last color that was set. If no previous color has been set then it will default to white on black.

Colors are defined in the file *color.h*.

```
scr_setatr(CYAN,GREEN,LOW,NO_BLINK);

scr_printf("hello world\n");
```

This will print "hello world" with a cyan background, green low intensity forground, and non blinking.

NAME

segread

SYNOPSIS

segread(ptr)
unsigned ptr[];

DESCRIPTION

segread returns the values of the segment registers in the 8-byte field pointed at by *ptr*.

The *ptr* array is organized as follows:

ptr element	*segment register*
0	CS
1	SS
2	DS
3	ES

NAME

signal - define how to handle a signal

SYNOPSIS

```
#include <signal.h>
void (*signal (sig, func))()
int sig;
int (*func)();
```

DESCRIPTION

signal specifies that the signal whose number is *sig* is to be handled as defined by *func*. A 'signal' is a special, asyncronous event such as an operator-initiated interrupt or an arithmetic fault.

Signals and the *sig* parameter

The following list defines the symbolic values that *sig* can have, and the signal associated with each value. These values are defined in *signal.h*. Although a program can specify any of these values for *sig*, the only one that currently has any effect is *SIGINT*.

SIGINT	Operator-initiated interrupt (ie, operator typed control-break or control-C on a PC).
SIGABRT	Abnormal termination.
SIGFPE	Erroneous arithmetic operation, such as a divide by zero or an operation resulting in an overflow.
SIGILL	Invalid function image.
SIGSEGV	Invalid access to a data object.
SIGTERM	Termination request.

Signal processing and the *func* parameter

func defines the action to be performed on receipt of the specified signal. It can be one of three values: *SIG_DFL*, *SIG_IGN*, or a function address.

If the *func* for a signal is *SIG_DFL*, the program will be terminated by the operating system, without execution of the normal Aztec C *exit* code. This could result in a loss of information in files opened for standard output. The *func* for all of a program's signals is *SIG_DFL* until the program calls *signal* and specifies otherwise.

If the *func* for a signal is *SIG_IGN*, the signal will be ignored.

Any other value for *func* is assumed to be the address of a function. In this case, when the specified signal occurs, the function will be called, passing the signal's number as the function's only argument. Before the function is called, the value of *func* for the received signal will be set to *SIG_DFL*.

The function associated with a signal can terminate its program, if desired, by calling *exit* or *longjmp*. It can also return to the program at the point of interruption by issuing a *return* statement.

When a program activates another program, by calling one of the *exec*, *fexec*, or *system* functions, the *func* for the signals of the called program are initially set to *SIG_DFL*, regardless of their setting in the calling program. If the called program returns to the calling program, the *func* for the signals in the calling program resume the value that they had just prior to the activation of the called program.

Return values from *signal*

If a requested change is accepted, *signal* returns the value that the specified signal's *func* had on entry to *signal*. If the change is rejected, *signal* returns the value *SIG_ERR* and the global int *errno* is set to indicate the error. Currently, the only cause for rejection is an invalid signal number, causing *errno* to be set to *EINVAL*.

EXAMPLES

The following program calls *signal*, so that upon receipt of an operator-initiated interrupt (on the PC, this means that control-C or control-BREAK is typed) the program can shut itself down in an orderly fashion, closing opened files, deleting temporary files, and so on.

```
#include <signal.h>
main()
{
    signal(SIGINT, shutdown);
    ...    /* normal program execution */
}

shutdown(sig)
int sig;
{
    printf("received signal %d\n", sig);
    ... /* termination code */
    exit(1);
}
```

NAME

system

SYNOPSIS

```
system(cmd)          /* DOS 2.x function */
char *cmd;
```

DESCRIPTION

system causes the command processor to execute the command pointed at by *cmd*. The command can be a DOS built-in command, batch command, or transient command. When the command terminates, *system* returns to the caller.

system first searches the environment for the variable COMSPEC; if found, the command is executed by the command processor file specified by this variable. If an alternate command processor isn't specified, the standard DOS command processor, in the file *command.com*, executes the command.

Files opened for unbuffered i/o in the calling program will also be open in the command processor and the called program, and will have the same file descriptors.

Files opened for standard i/o in the calling program won't be open for standard i/o in the called program, although they will be open for unbuffered i/o. Thus, before a program activates another using *system*, it should cause the buffered data for files opened for standard i/o to be written to disk, using either the *fclose* or *fflush* functions.

The standard input, standard output, and standard error devices are open in the called program to the same devices or files as in the calling program. For the reasons discussed above, care is needed when either the calling or called program accesses these logical devices using standard i/o calls.

The environment of the command processor and the called program is the same as that of the called program.

SEE ALSO

exec, fexec

ERRORS

If *system* fails, it returns -1 as its value, and in some cases may set a code in the global integer *errno*. These codes are described in the Errors section of the Library Overview chapter.

NAME

time, dostime, ctime, localtime, gmtime, asctime

SYNOPSIS

```
long time(tloc)          /* DOS functions */
long *tloc;

dostime(buf)
struct tm *buf;

char *ctime(clock)
long *clock;

#include "time.h"

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;
```

DESCRIPTION

time and *dostime* return the date and time, which they get from the operating system. The other functions convert the date and time, which are passed as arguments, to another format.

time returns the current date and time packed into a long int. If its argument *tloc* is non-null, the return value is also stored in the field pointed at by the argument. The format of the value returned by *time* is described below.

dostime returns the current date and time in the buffer pointed at by its argument, *buf*. The format of this buffer is described below.

ctime, *localtime*, and *gmtime* convert a date and time pointed at by their argument, which is in a format such as returned by *time*, to another format:

ctime converts the time to a 26-character ASCII string of the form

Mon Apr 30 10:04:52 1984\n\0

localtime and *gmtime* unpack the date and time into a structure and return a pointer to it. The structure, named *tm*, is described below and defined in the header file *time.h*.

asctime converts a date and time pointed at by its argument, which is in a structure such as returned by *localtime* and *gmtime*, to a 26-character ASCII string in the

same form as returned by *ctime*.

The long int returned by *time* and passed to *ctime*, *localtime*, and *gmtime* has the following form (bit 0 is the least significant bit in the field, bit 31 the most significant):

bits	*meaning*
0-4	seconds/2
5-10	minutes
11-15	hours
16-20	day of month
21-24	month (0=Jan,...)
25-31	year since 1980

The long int fields used by the functions described in the FILETIME section also have the above format.

The structure returned by *dostime*, *localtime* and *gmtime*, and passed to *asctime*, has the following format:

```
struct tm {
    short tm_sec;  /* seconds */
    short tm_min; /* minutes */
    short tm_hour;/* hours */
    short tm_mday;/* day of the month */
    short tm_mon; /* month */
    short tm_year; /* year since 1900 */
    short tm_wday;/* day of the week (0 = Sunday */
    short tm_yday;/* day of year */
    short tm_isdst; /* not used */
    short tm_hsec;  /* hundredths of seconds */
}
```

NAME

tmpfile - create a temporary file

SYNOPSIS

```
#include <stdio.h>
FILE *
tmpfile ()
```

DESCRIPTION

tmpfile creates a temporary file and opens it for standard i/o in update (w+) mode. *tmpfile* returns as its value the file's FILE pointer.

When the temporary file is closed, either because the program explicitly closes it or because the program terminates, the temporary file will automatically be deleted.

SEE ALSO

tmpnam, mktemp

NAME

tmpnam - create a name for a temporary file

SYNOPSIS

char *tmpnam (s)
char *s;

DESCRIPTION

tmpnam creates a character string that can be used as the name of a temporary file and returns as its value a pointer to the string. The generated string is not the name of an existing file.

s optionally points to an area into which the name will be generated. This must contain at least *L_tmpnam* bytes, where *L_tmpnam* is a constant defined in *stdio.h.*

s can also be a NULL pointer. In this case, the name will be generated in an internal array. The contents of this array are destroyed each time *tmpnam* is called with a NULL argument.

The generated name is prefixed with the string that is associated with the symbol *P_tmpnam*; this symbol is defined in *stdio.h.* In the distribution version of *stdio.h*, *P_tmpnam* is a null string; this results in the generated name specifying a file that will be located in the 'current area'. The location of this area is system dependent: on PC-DOS/MS-DOS 2.x, it's the current directory on the default drive; on CP/M-86, it's the current user area on the default drive.

SEE ALSO

tmpfile, mktemp

NAME

getusr, setusr, rstusr

SYNOPSIS

```
getusr()                    /* CP/M-86 functions */

setusr(user)

rstusr()
```

DESCRIPTION

getusr returns the current user number as its value.

setusr sets the user number to *user*. The user number which was active on entry to *setusr* is saved for subsequent use by *rstusr*.

rstusr resets the user number to the value which was saved during the last call to *setusr*.

TECHNICAL INFORMATION

Chapter Contents

Technical Information ..... tech
1. Program Organization 4
   1.1 The program areas 5
   1.2 Factors affecting Program Organization 7
   1.3 Symbols related to Program Organization 13
   1.4 Startup routine Termination Codes 14
2. Overlay Support 15
   2.1 Introduction to Overlays 15
   2.2 Programmer Information 19
3. Libraries 25
4. Cross Development 26
5. Using the PCDOS/MSDOS Linker 27
6. Assembly Language Functions 30
   6.1 Conventions for C-callable Functions 30
   6.2 Assembly Language Macros 33
   6.3 Embedded Assembler Source 39
7. Generating ROMable code 41
   7.1 Features of ROMable Programs 41
   7.2 Special ROM-related Programs 42
   7.3 The Procedure 42
   7.4 Description of hex86 43

Technical Information

This chapter discusses technical topics, and topics that couldn't be conveniently discussed elsewhere.

It's divided into the following sections:

1. *Program Organization.* Discusses the factors that affect the memory organization of a program.
2. *Overlays.* Describes overlays: what they are, and how they are used.
3. *Libraries.* Discusses the object module libraries that are provided with Aztec C86.
4. *Cross Development.* Discusses the development of CP/M-86 programs with the PC-DOS/MS-DOS version of Aztec C86, and the development of MS-DOS/PC-DOS programs with the CP/M-86 version of Aztec C86.
5. *Using the MS-DOS/PC-DOS Linker.* Describes how to use the MS-DOS/PC-DOS linker *link* to create an executable program from object modules that have been generated by the Aztec C86 compiler and assembler.
6. *Mixing Assembler and C Routines.* Describes how to interface assembly language routines with C routines.
7. *Generating ROMable code.*

1. Program Organization

An executable program is organized into several areas. In previous chapters we peripherally discussed these areas, while discussing other topics. In this section we want to focus on these areas.

Some of the information in this section just paraphrases previously-presented information, and some is brand new. New information related to programs that run on DOS 2.0 or later, which you should be on the lookout for as you read this section, includes a discussion of the global variables *__STKLOW*, *__STKSIZ*, and *__HEAPSIZ*, which in many, but not all, cases determine whether a program's stack area is below its heap or vice versa, the size of the stack area, and the initial size of the heap.

Another new topic related to programs that run on DOS 2.0 or later concerns the total amount of memory allocated to the program: when a program's stack is below its heap, the program is initially allocated just enough memory to contain all its areas; the heap, and with it the total space allocated to the program, will automatically grow and contract as necessary to satisfy the program's needs. When the stack is above the heap, the total size of the program is fixed and won't change as the program is running.

An executable program is organized into the following areas:

* *code area*, containing the program's executable code;
* *overlay code area*, into which code for the program's overlays are loaded;
* *initialized data area*;
* *uninitialized data area*;
* *overlay data area*, into which data for the program's overlays are loaded;
* *stack area*, containing the program's stack;
* *heap*, from which buffers are dynamically allocated.

There are several factors that determine the size of these areas and their position in memory. These are:

* The operating system that the program runs on.
* Whether the program uses the large code or small code memory model.
* Whether the program is linked with a version of *c.lib* that uses the 'large data' or 'small data' memory model.
* On PC- and MS-DOS, version 2.0 or later, whether the program is an *.exe* or a *.com* file.
* The value of the int *__STKLOW*, which in most cases determines whether the stack is below the heap or vice versa.
* The value of the unsigned int *__STKSIZ*, which defines the size of the stack area.
* The value of the unsigned int *__HEAPSIZ*, which defines the

size of the heap area in some cases.

* The value specified in the linker's -X option when the program was linked, in some cases.

Programs that run on PC-DOS or MS-DOS, version 2.0 or later, have the most control over the size and placement of program areas. Programs that run on PC-DOS or MS-DOS, version 1.1, or on CP/M-86 have limited control; these programs are discussed at the end of this section.

The following paragraphs first generally discuss the different areas of a program. Then follows a discussion of the areas of a program when using specific combinations of factors.

1.1 The Program Areas

1.1.1 The Code Area

The use of large or small code memory model by a program affects the maximum size of the program's memory-resident, executable code: a 'large code' program can have unlimited code, all of which must be memory resident; a 'small code' program can have at most 64K bytes of executable code. The code memory model used by a program doesn't affect the size or relative placement of the program's other areas, and so won't be discussed much in the following paragraphs.

For either 'large code' or 'small code' programs, the linker sets the size of the code area for the program so that it's just big enough to hold the program's executable code.

Only *.exe* programs on DOS 2.0 or later can use a large memory model. DOS *.com* programs, DOS 1.1 programs, and CP/M-86 programs must use the 'small code' and 'small data' memory model.

1.1.2 The Overlay Code Area

An overlay's code is placed in the overlay code area. You set the size of the overlay code area when you link the program, using the linker's +C option.

To use overlays, a program must use the 'small code' and 'small data' memory model options. A PCDOS/MDSOS program must be in an *.exe* file, and not in a *.com* file.

1.1.3 The Initialized Data, Uninitialized Data, and Overlay Data Areas

The linker sets the size of the initialized and uninitialized data areas so that they're just big enough to hold the data that goes in these areas. You set the size of the overlay data area for a program when you link it, using the linker's +D option.

The maximum size of these areas depends on several factors, which we will discuss later.

1.1.4 The Stack and Heap Areas

The stack and heap areas of a program are always adjacent. However, for programs that run on DOS 2.0 or later, you can usually control whether the stack is above the heap or vice versa. Programs that are linked with a version of *c.lib* that uses the 'large data' memory model always has the stack below the heap.

By default, programs that are linked with a version of *c.lib* that uses the 'small data' memory model have the stack above the heap, but you can override this, forcing the stack to be placed below the heap.

When a program's stack is below its heap, the program will initially be allocated just the minimum amount of space necessary to hold the program. The heap has an initial size, but it, and the total space allocated to the program, will expand and contract as necessary to satisfy the program's requests for dynamically allocated buffers.

Thus, the advantage of a program whose stack is below its heap is that the program will occupy just the amount of space it actually needs. The disadvantage is that the area used by the stack is a fixed size, allowing the possibility that the stack may overflow its area and overwrite the program's global data.

When a program's stack is above its heap, the program will be allocated a fixed amount of memory, with its code and data located at the bottom of the area, its stack area at the top, and its heap in between.

The advantage to locating a program's stack above its heap is that the program can dynamically change the boundary between its stack and heap (by calling the function *rsvstk*). The disadvantage is that the program will typically use more or less memory than it actually needs.

1.1.4.1 The _STKLOW, _STKSIZ, and _HEAPSIZ variables

For 'small data' programs running on DOS 2.0 or later, the global int *_STKLOW* defines whether the program's stack is below or above its heap: 1 for stack below heap, 0 for stack above heap.

For programs running on DOS 2.0 or later, the unsigned int *_STKSIZ* defines the number of paragraphs (16-byte blocks) in the program's stack area.

Similarly, the unsigned int *_HEAPSIZ* defines the number of paragraphs that are initially in the program's heap area. For programs whose stack is above the heap, its stack is at the top of its allocated memory, its code and data are at the bottom, and the heap is allocated all space in between; thus, in this case, *_HEAPSIZ* has no effect.

These three variables are defined in the module *stklow* within the various versions of *c.lib*. The source for this module is in the file *stksiz.c*. The default settings for these variables are:

```
int __STKLOW = 0;
int __STKSIZ = 4096/16;  /* (in paragraphs) */
int __HEAPSIZ = 4096/16; /* (in paragraphs) */
```

This results in a program whose stack is above its heap ('small data' programs only), and whose stack and heap each contain 4K bytes.

To change the default values of any of these variables, you should modify, compile, and assemble *stksiz.c*. Then either replace the *stksiz* module in the version of *c.lib* that you are using with the resultant module, or link the modified *stksiz* into your programs before *c.lib* is searched by the linker.

If you want to override any of these variables, you must redefine ALL of them. If you only redefine one of them, you will get a multiply defined message from the linker.

1.2 Program organization for different combinations of factors

Program organization is affected by several factors. We have discussed the factors above; now we want to make the discussion more concrete by explicitly defining the organization of programs that use specific combinations of factors.

Program organization for the following combinations of factors for programs that will run on DOS 2.0 or later are discussed:

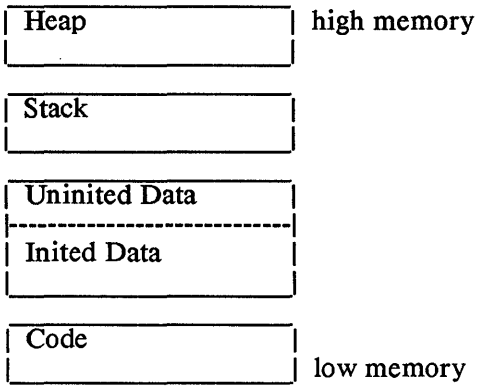
* large data, *.exe*;
* small data, *.exe*, stack below heap;
* small data, *.exe*, stack above heap;
* small data, *.com*, stack below heap;
* small data, *.com*, stack above heap;

The term 'small data' means that the program is linked with a version of *c.lib* that uses the 'small data' memory model.

Then follows a discussion of program organization for programs that run on DOS version 1.1 and/or CP/M-86.

1.2.1 Large data, *.exe* programs

A program that runs on DOS, v 2.0 or later, uses the 'large data' memory model, and is in a file having extension *.exe* has the following organization:

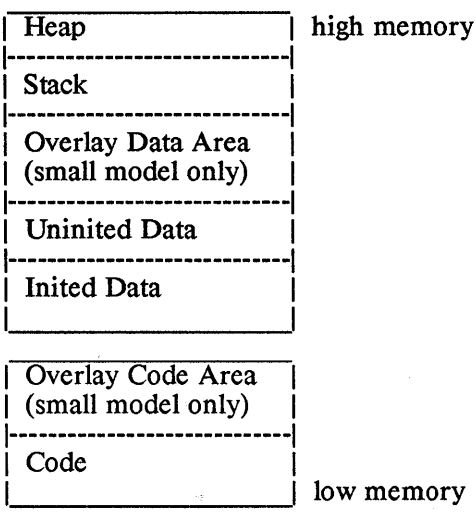


Programs of this type have the following additional features:

* It can use either the 'large code' or 'small code' memory model;
* 64K bytes is the maximum total size of the block of memory containing the initialized data, uninitialized data, and overlay code areas;
* *__STKSIZ* defines the size of the stack; the maximum size of the stack is 64K bytes.
* *__HEAPSIZ* defines the number of paragraphs initially allocated to the heap. The heap, and with it the total amount of memory allocated to the program, will automatically grow as necessary. The maximum size of the heap is limited only by the amount of available memory.
* The stack area is always below the heap, regardless of the setting of *__STKLOW*.
* It cannot use overlays.

1.2.2 'Small data', *.exe*, 'stack below heap' programs

A program that uses the 'small data' memory model (that is, that is linked with a version of *c.lib* that uses the 'small data' memory model), is contained in an *.exe* file, has its stack below its heap, and runs on DOS version 2.0 or later has the following organization:

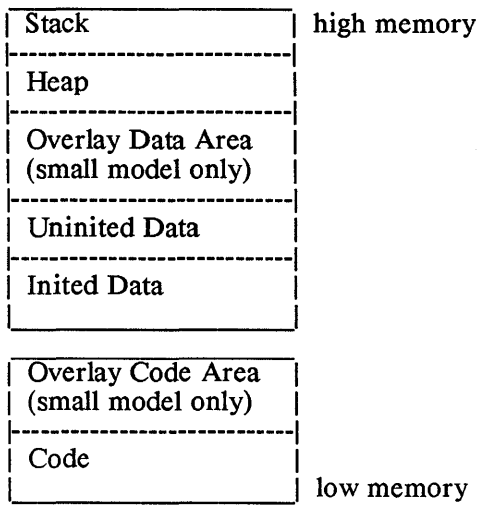


This type of program has the following additional features:

* It can use either the 'large code' or 'small code' memory model;
* Its non-code areas are in a single block of memory, the maximum size of which is 64K bytes.
* *__STKSIZ* defines the number of paragraphs in the program's stack area; there's no exact limit on the stack size, other than the limit on the block containing it and the other areas.
* *__HEAPSIZ* defines the number of paragraphs that are initially in the program's heap. The size of the heap, and with it the amount of space allocated to the program, will expand automatically as necessary to meet the program's needs. There isn't an exact limit on the size of the heap, other than the limit on the block containing it and the other areas.
* The variable *__STKLOW* must be non-zero, of course, to force the stack to be set below the heap.

1.2.3 'Small data', *.exe*, 'stack above heap' programs

A program that uses the 'small data' memory model (ie, that is linked with a 'small data' version of *c.lib*), is contained in an *.exe* file, has its stack above its heap, and runs on DOS version 2.0 or later, has the following organization:

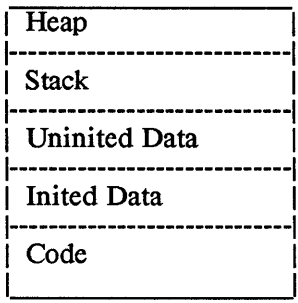


This type of program has the following additional features:

* It can use either the 'large code' or 'small code' memory model;
* Its non-code areas are in a single block of memory, the maximum size of which is 64K bytes. As mentioned above, DOS will attempt to allocate 64K bytes to the block, unless you request a smaller block using the linker's -X option.
* The variable *__STKLOW* must be zero, of course, to force the stack to be set above the heap.
* *__STKSIZ* defines the initial size of the program's stack area.
* As mentioned above, within the block of memory that contains the heap, the heap will be allocated all the space that isn't used by the other areas.
* The total size of the program is set when the program is loaded and can't be changed. The boundary between the heap and stack can be dynamically changed by the program, by calling the *rsvstk* function.

1.2.4 'small data', *.com*, 'stack below heap' programs

A program that uses the 'small data' memory model, is contained in a *.com* file, has its stack below its heap, and runs on DOS, v2.0 or later, has the following organization:

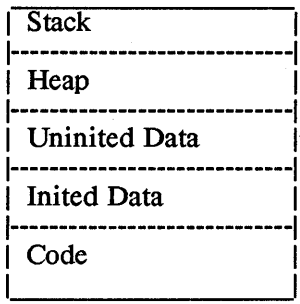


This type of program has the following additional features:

* It must use the 'small code' memory model.
* The total size of the block of memory occupied by the program can't exceed 64K bytes. The program is initially allocated just enough memory to hold all of its areas.
* The size of the program's stack is defined by *_STKSIZ*. There isn't an exact limit on the size of the stack, other than the limit on the size of the block containing the program.
* The initial number of paragraphs in the program's heap are defined by *_HEAPSIZ*. The heap will expand automatically as necessary when requests for dynamically-allocated buffers are made until the program reaches its 64K byte limit.
* It cannot use overlays.

1.2.5 'Small data', *.com*, 'stack above heap' programs

A program that uses the 'small data' memory model, that is in a *.com* file, whose stack is above its heap, and that runs on DOS, v2.0 or later, has the following organization:

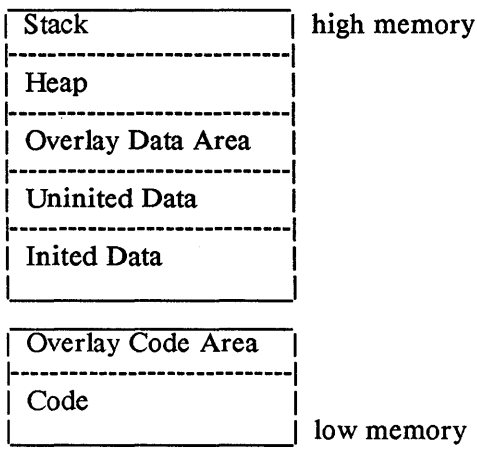


This type of program has the following additional features:

* The program must use the 'small code' memory model.
* The maximum size of the block of memory containing the program's areas is 64K bytes. DOS will allocate as much memory as possible to the program, up to the 64K byte limit. The linker's -X option has no effect on *.com* files.
* Initially, the size of the stack area is set as specified by *_STKSIZ*.
* Initially, the heap is given that part of memory allocated to the program that isn't used by the program's other areas.
* The program can dynamically change the boundary between the stack and heap by calling the *rsvstk* function.
* It cannot use overlays.

1.2.6 Programs on DOS 1.1 and CP/M-86

A DOS *.exe* or CP/M-86 *.cmd* program looks like this:

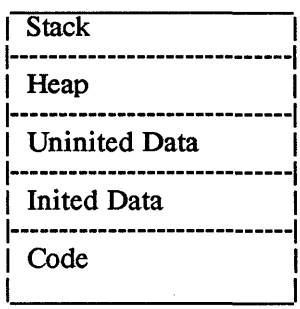


This type of program has the following additional features:

* The program must use the 'small code' and 'small data' memory models.
* The maximum size of the block containing the program's code areas is 64K bytes, as is the maximum size of the block containing the program's non-code areas. The operating system allocate as much space as possible to the program, up to the 64K-byte limit. On CP/M-86, but not on DOS 1.1, you can use the linker's -X option to request a smaller limit.
* The initial size of the program's stack area is 2 K bytes.
* Within the area allocated to the program's non-code areas, the heap will be given all space not used by the program's other areas.
* The program can dynamically change the boundary between

the stack and heap areas by calling the *rsvstk* function.

A DOS *.com* file looks like this:



This type of program has the following additional features:

* It must use the 'small code' and 'small data' memory models.
* The maximum size of the block containing the program is 64K bytes. DOS will allocate as much space as possible to the program, up to the 64K-byte limit. The linker's -X option has no effect on *.com* programs, or on DOS 1.1 programs.
* The initial size of the program's stack area is 2 K bytes.
* The heap will be given all space not used by the program's other areas.
* The program can dynamically change the boundary between the stack and heap areas by calling the *rsvstk* function.
* It cannot use overlays.

1.3 Symbols related to Program Organization

The following global symbols are related to program organization. The symbols are given in the form that an assembly language program would use to access them.

__Corg__ Name of the beginning of the program's code.

__Cend__ Name of the first byte beyond the program's executable code.

__Dorg__ Name of the beginning of the program's initialized data.

__Dend__ Name of the first byte beyond the program's initialized data.

__Uorg__ Name of the beginning of the program's uninitialized data.

__Uend__ Name of the first byte beyond the program's uninitialized data.

__mbot__ Name of a field containing a pointer to the beginning of the program's heap.

$MEMRY Name of a field containing a pointer to the next byte to be allocated from the heap.

__mtop__ Name of a field containing a pointer to the first byte beyond the program's heap.

__sbot__ Name of a field containing a pointer to the bottom of the program's stack area (small data model only).

__dsval__ Name of a word containing the paragraph number at which the program's data begins.

__csval__ Name of a word containing the paragraph number at which the program's code begins.

__PSP__ Name of a word containing the paragraph number at which the program's Program Segment Prefix begins.

__STKSIZ__ Name of a word containing the number of paragraphs in the program's stack area.

__HEAPSIZ__ Name of a word containing the initial number of paragraphs in the program's heap area.

__STKLOW__ Name of a word specifying whether the stack is to be below the heap (1) or vice versa (0).

The pointer fields contain short pointers for 'small data' programs and long pointers for 'large data' programs.

A C module can access all the above symbols, except for $MEMRY, by removing the appended underscore from the symbol name.

1.4 Startup routine termination codes

There are a few instances when the startup routine will not be able to properly start its program. In these cases, it will terminate with a non-zero return code. The return codes and their meanings are:

code	*meaning*
254	The startup routine first attempts to allocate exactly as much memory to its program as is needed, by issuing a *setblock* DOS call. If DOS rejects this call, the startup routine then tries to allocate just the amount that DOS says is available. If this second call fails, the startup routine exits with return code 254.
253	If an attempt is made to start a *.com* program that has been linked with the 'large data' startup routine *lbegin*, the startup routine exits with return code 253.

2. Overlay Support

In order to allow users to run programs which are larger than the limited memory size of a microcomputer, Manx provides overlay support. This feature allows a user to divide a program into several segments. One of the segments, called the root segment, is always in memory. The other segments, called overlays, reside on disk and are only brought into memory when requested by the root segment. There are only two areas of memory into which the overlays are loaded: one for the executable code part of an overlay, and one for its data.

If an overlay is in memory when the root requests that another be loaded, the newly specified overlay overlays the first, that is, replaces it in memory.

The Manx linker allows overlays to be "nested"; that is, an overlay at one level can call another overlay nested one level deeper. However, an overlay cannot call an overlay which is at the same level.

This description is divided into two sections. The first presents basic information about creating overlays, and the second presents more information.

2.1 Introduction to overlays

What is an Overlay?

An overlay is one or more sections of executable code that run in the same area of memory. The advantage of an overlay, therefore, is that it allows the user to run programs of unlimited size in a machine which has a limited memory capacity.

How do I Call an Overlay From a Program

The following is the format for calling an overlay:

ovloader(overlay name,p1, p2, p3...)

ovloader's first parameter must be the name of the overlay file. The parameters *p1*, *p2*, and *p3* are passed directly to the overlay. The overlay is loaded from a file whose name is overlay name and whose extent is *.ovr*.

ovloader returns as its value the value which was returned by the overlay.

How do I make a function an overlay?

The function, *ovloader*, loads the overlay and then passes control to *ovbgn*, a function which is linked with every overlay. *ovbgn* in turn calls *ovmain*. *ovmain* must be the name of your function which takes control when the overlay is loaded. This function can then call any other function which is in memory.

So other than the naming of *ovmain*, the overlay does not have to know that it is an overlay. *ovmain* executes and returns just like any other function.

What Files are Created on the Disk?

.exe or .cmd
: The file which contains the root has the extent *.exe* on MSDOS or PCDOS, and *.cmd* on CP/M-86. A DOS *.com* program cannot have overlays.

.ovr
: There is one file for each overlay, the extent of which is *.ovr*.

.rsm
: There is a file containing the relocatable symbol table with the extent *.rsm* for the root and for each overlay that invokes another overlay.

What about Overlaying Large Model Programs?

A program that calls overlays must use the 'small code' memory model. Individual modules in the root can be compiled to use long pointers to data objects, but the program must be linked with a 'small data' version of *c.lib*, and none of the overlay modules can be compiled to use long pointers to data objects.

Sample Run:

step	*command*
1)	ln +c 1020 +d a00 -r myroot.o ovloader.o -lm -lc
2)	ln mysub1.o myroot.rsm ovbgn.o -lm -lc
3)	ln mysub2.o myroot.rsm ovbgn.o -lm -lc

In this example, there are three modules which comprise the program, namely, *myroot.o*, *mysub1.o* and *mysub2.o*. The first step serves two purposes, to create the executable file and to generate a file, *myroot.rsm*. On MSDOS and PCDOS, the executable file is named *myroot.exe*; on CP/M-86, it's named *myroot.cmd*.

This second file is called the relocatable symbol table. It contains information about the contents of the root module which is needed when an overlay is linked.

The *+C* and *+D* options are explained below. The *-R* option specifies that the following module is a root. An *.rsm* file for that module will be created.

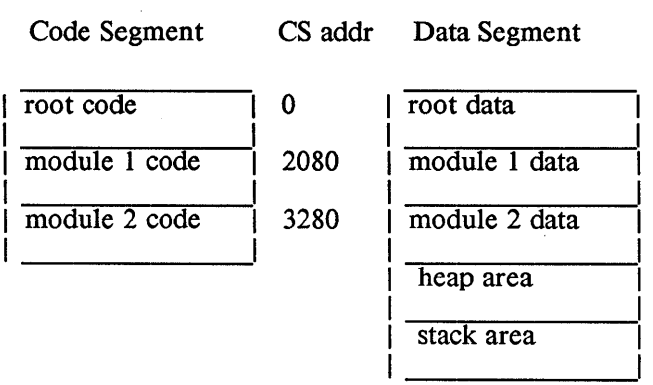
The module *ovloader.o* contains the routine which loads the overlay into memory.

The second step links the first overlay, *mysub1.o*. The *.rsm* file for the root must be included in the linkage. The module *ovbgn.o* is the startup routine which calls *ovmain*.

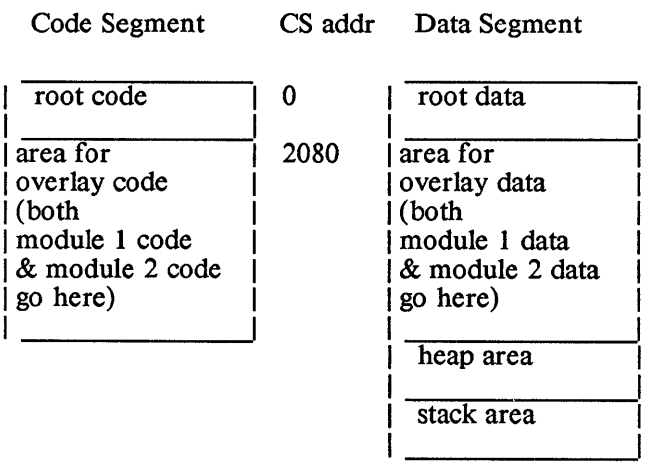
Step three performs the linkage of the second overlay in a manner identical to the first.

Figure 1 shows a program, run as a single module, that can be logically divided into three segments. Figure 2 shows the same program run as an overlay. In figure 2, module 1 and module 2 occupy the same memory locations. A possible flow of control would be for the base routine to call module 1, module 1 then returns to the root and the root calls module 2, module 2 returns to the root and the root calls module 1 again. Then module 1 returns to the root the root exits to the operating system.

Notice that all overlay segments must return to their caller and that overlays at the same level cannot directly invoke each other.



Picture of a non-overlayed program
Figure 1



Layout of the Program in Figure 1 as an Overlay
Figure 2

2.2 Programmer Information

The root loads an overlay by calling the Manx-supplied function *ovloader*, which must reside in the root. The call has the form

ovloader(ovlyname, p1, p2, ...)

where *ovlyname* is a pointer to a character string identifying the overlay name, and *p1*, *p2*, ... are parameters that are to be passed to the overlay as its first, second, ... parameters.

ovloader derives the name of the file containing the overlay by appending *.ovr* to the string pointed at by *ovlyname*.

On DOS 2.0 or later, there are two versions of *ovloader*; the one in the file *ovldpath.o* will search for the overlay in the directories defined by the *PATH* environment variable; that is, it will search the same directories that are searched by DOS for a command program. The one in the file *ovld.o* will look in just the current directory on the default drive.

On DOS 1.1 and CP/M-86, *ovloader* looks for an overlay in just the current area on the default drive, where the area is the current directory on DOS and is the current user area on CP/M-86.

When the overlay is loaded, control passes to the Manx- supplied function *ovbgn*, which must be linked with the overlay. In turn, *ovbgn* transfers control to the function in the overlay whose name is *ovmain*. *ovmain* receives the arguments passed to *ovloader*.

When *ovmain* completes its processing, it simply returns. Control then passes back to the root at the instruction in the user's program following the one that called *ovloader*. The value returned by *ovloader* is the value which was returned by *ovmain*.

Overlays can be nested; that is, an overlay can call a second overlay, provided that they are not at the same nesting level. Also, an overlay can access any global functions and variables which are defined in the calling segment.

The +C and +D options

When the root module is linked (with the -r option), the linker has to reserve some space into which the overlay can be loaded. This is done using the +C and +D linker options.

The +C option reserves space in the root's physical code segment. An overlay's code is loaded into this space.

The +D option reserves space in the root's physical data segment. An overlay's data is loaded into this space.

If overlays are nested, a called overlay is located in memory immediately following the calling overlay. The amount of space reserved for the overlays must be enough to hold the longest 'thread'

of overlays.

Creating a root and overlays

To create a root and one or more overlays, the Manx linker must be run several times. Each execution creates one program (root or overlay) and places it in a separate disk file. The first execution must create the root. This execution also creates a file containing a symbol table, which must be specified during the subsequent executions of the linker which create the overlays.

On DOS, the extension of the file containing the root must be *.exe.*

When creating a program (root or overlay) which calls an overlay, the option -R must be specified; this causes the linker to generate a symbol table for use in linking the called overlay. The table is in a file whose filename is the same as that of the first file specified in the command line and whose extent is *.rsm.*

When creating an overlay, the .rsm file which was generated during the linking of the calling program must be specified. This causes the linker to create an overlay in the file whose filename is the same as that of the first file specified in the command line and whose extent is *.ovr.*

If an overlay is both called by a root or overlay, and itself calls another overlay, the command line to the linker must specify both the *.rsm* file of the calling program and the '-R' option.

Two examples follow. The first demonstrates overlay usage when overlays are not 'nested'. The second demonstrates nested overlays.

Example 1

In this example, the root segment, which consists of the function *main* and any neccesary run-time library routines, behaves as follows:

1. It calls the overlay *ovly1*, passing it a pointer to the string "first message".
2. It prints the integer value returned to it by *ovly1*;
3. It calls the overlay *ovly2,* passing it a pointer to the string "second message";
4. It prints the integer value returned to it by *ovly2.*

The overlay segment *ovly1* consists of the function *ovly1*, the Manx function *ovbgn*, and any neccesary run-time library routines. It prints the message "in ovly1" plus whatever character string was passed to it by *main.*

The overlay segment *ovly2* consists of the function *ovly2*, the function *ovbgn*, and any neccesary run-time library routines. It prints the message "in ovly2", plus whatever character string was passed to it

by *main.*

Here then is the *main* function:

```
main()
{
    int a;
    a = ovloader("ovly1","first message");
    printf("in main. ovly1 returned %d\n", a);
    a = ovloader("ovly2","second message");
    printf("in main. ovly2 returned %d\n",a);
}
```

Here is ovly1:

```
ovmain(a)
char *a;
{
    printf("in ovly1. %s\n",a);
    return 1;
}
```

Here is ovly2:

```
ovmain(a)
char *a;
{
    printf("in ovly2. %s\n",a);
    return 2;
}
```

The following commands link the root (which is in the file *root.c*) and the overlays:

```
ln -R +C 4000 +D 1000 root.o ovloader.o -lc
ln ovly1.o ovbgn.o root.rsm -lc
ln ovly2.o ovbgn.o root.rsm -lc
```

The command to link the root reserves 0x4000 bytes for the overlay's code and 0x1000 bytes for it's data. Techniques for determining this value are discussed below.

When the segments are generated and the root activated, the following messages appear on the console:

```
in ovly1. first message.
in main. ovly1 returned 1.
in ovly2. second message.
in main. ovly2 returned 2.
```

Example 2: nested overlays

In this example, there are three segments: a root segment, *root*, and two overlays segments, *ovly1* and *ovly2*. *root* calls *ovly1*, which calls

ovly2. *ovly2* just returns.

Here is the root:

```
main()
{
   ovloader("ovly1","in ovly1");
}
```

Here is ovly1:

```
ovmain(a)
char * a;
{
   printf("%s\n",a);
   ovloader("ovly2", "in ovly2");
}
```

Here is ovly2:

```
ovmain(a)
char *a;
{
   printf("%s\n",a);
}
```

The following commands link the root and the two overlays:

```
ln -R root.o ovloader.o -lc
ln -R ovly1.o ovbgn.o root.rsm -lc
ln ovly2.o ovbgn.o ovly1.rsm -lc
```

When executed, the following messages appear on the console:

```
in ovly1
in ovly2
```

Determining the size of the overlay area

When you link the root module, you will have to know how much memory to reserve for the overlay, that is, you will have to know how large the overlay is. But since the overlays haven't been linked yet, how can you know how much space is needed for overlays?

The easiest way is to guess. That is, estimate the size and go ahead and link the root and the overlays, keeping track of the size of the code and data for the overlays as reported by the linker.

After all overlays have been linked, the size of the area need for overlays is the size of the largest overlay (if overlays aren't nested) or the size of the longest 'thread' of overlays (if they are nested). You can then go back and relink the root, if necessary, with this value. You won't have to relink any overlays, since the +C and +D options don't affect the position of the overlays in memory.

Error messages from ovloader

If an error occurs while loading an overlay, ovloader will print a message of the form

Error %d loading overlay %s

where %d is a number defining the error and %s is the name of the overlay. The error codes and their meanings are:

10	Can't open overlay file
20	Can't read overlay header record
30	Invalid header record
40	Overlay data overlaps with heap
50	Error reading overlay
60	Overlay code or data overlaps caller's code or data

Possible Problems

A possible source of difficulty in using overlays concerns initialized data. In the following program module, a global variable is initialized:

```
int i = 3;

function()
{
   return;
}
```

The initialization of "i" is performed by the linker, rather than at run time. In the same program, the following module is allowed:

```
int i;

main()
{
   function();
}
```

The global variables in each module refer to the same integer, "i". At link time, this variable is set to the value 3. Although this works when the two modules are linked together, a problem arises when the first module is linked as an overlay:

ln func.o ovbgn.o main.rsm -lc

From the *.rsm* file, the linker knows that *int i* has been declared in *main.o*, the root. But it tries to initialize *i* from the statement in the *func.o* module. This attempt fails because the variable *i* is part of *main.o*, a module which is not included in the linkage.

An attempt to initialize, in an overlay, a variable which has been declared in the root will produce an error:

attempt to initialize data in root

The simple solution is to change the statement, *int i = 3*, to the following:

```
int i;
i = 3;
```

This assignment will be performed at run time, so that the linker does not try to perform an initialization.

3. Aztec C86 libraries

The standard library that is provided with Aztec C86 is *c.lib*. Your package may have several versions of this library: for example, some that support different memory models, and that allow you to create programs that will run on other systems. For a description of the versions that are provided with your package, see the release document. And for more information on cross development, see the section of the same name in this chapter.

There are several different math libraries available with Aztec C86: one uses the 8087; another provides software emulation of the 8087. Another, the 'sensing library', uses the 8087 if it is available on the system on which the linked program runs, and otherwise uses software emulation.

There are several different versions of each math library, which support different memory models.

Not all Aztec C86 packages provide all the math libraries. For a description of those that are provided with your package, see the release document.

4. Cross Development

With the PRO extensions to Aztec C86, programs can be created that will run on other 8086-based systems: with the MSDOS/PCDOS version, programs can be created to run on MSDOS/PCDOS, version 1.1, and on CP/M-86. With the CP/M-86 version, programs can be created to run on any version of MSDOS or PCDOS.

To do this, a program is linked in the normal manner, with the following exceptions:

* Another library is used instead of *c.lib*. The library used depends on the target system:

  dos20.lib Program will run on MSDOS/PCDOS, version 2.x;
  dos11.lib Program will run on MSDOS/PCDOS, version 1.1;
  cpm86.lib Program will run on CP/M-86.

* The extension of the executable file generated by the linker must be:

  .exe or .com
  If the program is to run on MSDOS or PCDOS, any version;
  .cmd If the program is to run on CP/M-86.

Programs that are to run on CP/M-86 or DOS 1.1 must use the 'small code' and 'small data' memory models. Programs that are to run on DOS 2.0 or later can use any memory model.

For programs which perform floating point, one of the math libraries is used, just as if the program was being linked to run on the host system.

5. Using the PC-DOS/MS-DOS Linker

You can use the PC-DOS/MS-DOS linker instead of the Aztec linker to link object modules that have been generated by the Aztec C compiler and assembler and that are to run on DOS, version 2.0 or later. To do this, you must (1) convert the object modules from Aztec to PC-DOS/MS-DOS format using the Aztec utility program *obj*, and (2) include the special startup routine *crt0.obj* as the first module in the program.

You can also link modules that have been converted from Aztec format with modules that are in PC-DOS/MS-DOS format and that have been created using other Manufacturer's compilers and assemblers. For this to work, all the modules must use the same conventions regarding the calling of functions, register usage, and so on.

A program generated using the PC-DOS/MS-DOS linker and the Aztec compiler and assembler can use any of the features available to programs linked with the Aztec linker, except for overlays.

A simple example

The following commands create an executable version of the C source program that's in the file *exmpl.c*:

```
cc exmpl.c
obj exmpl.o
link crt0+exmpl,exmpl,,mc
```

cc compiles and assembles the C source, leaving the Aztec format object module in the file *exmpl.o*. *obj* converts the object module to PC-DOS/MS-DOS format, placing the result in the file *exmpl.obj*. *link* links the object module together with the startup routine *crt0.obj* and the library *mc.lib*, writing the executable program to *exmpl.exe*.

The library *mc.lib* with which the *exmpl* program was linked is a PC-DOS/MS-DOS-format version of the Aztec-format library *c.lib*. It was generated by feeding *c.lib* through *obj*, as follows:

```
obj c.lib mc.lib
```

If the *exmpl* program performed floating point, the command for linking the program would have been:

```
link crt0+exmpl,exmpl,,mm+mc
```

The first two commands would not have to be changed. In this command, *mm* refers to the library *mm.lib*, which is a PC-DOS/MS-DOS-format version of the Aztec-format library *m.lib*. It was generated by the command

```
obj m.lib mm.lib
```

Using *obj*

obj converts the object modules that are in a specified file from Aztec to PC-DOS/MS-DOS format, writing the result to another file. *obj* is started with a command of the form

```
obj infile [outfile]
```

infile is the name of the file containing the Aztec-format object modules. *infile* can contain a single object module (created by the Aztec assembler, *as*) or a library of object modules (created by the Aztec librarian, *lb*).

The optional parameter *outfile* is the name of the file to which the PC-DOS/MS-DOS object modules will be written. If this parameter is not specified, the output will be sent to a file whose name is derived from that of the input file name, by changing the extension to *.obj*. For example,

```
obj subr.o
```

reads Aztec-format object modules from the file *subr.o* and writes PC-DOS/MS-DOS-format object modules to the file *subr.obj*.

Converting the libraries

As we demonstrated in the above examples, when the PC-DOS/MS-DOS linker is used to link converted versions of modules that were generated by the Aztec C compiler and assembler, PC-DOS/MS-DOS-format versions of the Aztec libraries must be included in the linkage. These versions of the Aztec libraries are generated using *obj*.

Global variables

When you use the MS-DOS/PC-DOS linker to link multiple modules that have been generated with the Aztec compiler and assembler, the standard C rule regarding global variables must be followed. According to this rule, a global variable must be declared *extern* in all but one of the modules that reference the variable.

For example, if modules *a*, *b*, and *c* all reference the global variable *gvar*, then *gvar* could be declared as *int gvar* in *a*, and as *extern int gvar* in *b* and *c*.

The Aztec linker supports both this rule and the modified version of the rule that was supported by earlier versions of Aztec C. This modified rule allows several modules to declare the same variable, with the *extern* keyword being optional in all of them, requiring only that at most one module's declaration specify an initial value for the variable. If you have programs whose modules follow the modified rule regarding global variables, and you want to now link the programs using the MS-DOS/PC-DOS linker, you can use the compiler's *+U* option. When a module is compiled with this option, the module's

uninitialized global variables are made into *externs*.

For example, you can place all uninitialized global variables in one header file and then have modules that need these global variables include this file. Then compile with the +U option all files but one that include this header file.

The reason that the Aztec linker is able to support the modified version of the C rule on global variables is that it supports the *global* assembly language directive. The PCDOS/MSDOS linker doesn't support the *global* directive, so when *obj* converts an object module from Aztec to PCDOS/MSDOS format, it must convert each *global* directive into directives that the PCDOS/MSDOS linker understands. These directives are a *public* and a storage-reservation directive.

For more information on global variables, see the Programmer Information sections of the Compiler and Assembler chapters.

6. Assembly-Language Functions

This section discusses assembly-language functions that can be called by, and themselves call, C-language functions. It first discusses the conventions that such functions must follow, then describes a set of assembly-language macros that facilitate the writing of such functions, and finally discusses the in-line placement of assembly-language statements within C-language functions.

6.1 Conventions for C-callable, assembly-language functions

A C-callable, assembly-language function must obey the following conventions:

6.1.1 Executable code

The executable code must be in a segment named *codeseg*.

6.1.2 Global variables

A C module's global variables are in either the uninitialized data segment or in the initialized data segment, named *dataseg*.

An assembly language module can create an uninitialized variable that can be accessed by a C function, using the *global* directive. For example, the following code creates the global variable *var__*, reserves 8 bytes of storage for it, and sets the type of *var__* to *word*. *var__* can be accessed as an *int* array by a C function.

```
global var__:word,8
```

A C function that wants to access *var__* could have the following declaration:

```
extern int var[];
```

An assembly language module can create an initialized variable that can be accessed by a C function using the *public* directive and the *db*, *dw*, or *dd* directive; the variable must be in the *dataseg* segment. For example, the following code creates the *public* variable *ptr__* that initially contains a short pointer to the symbol *str*, and that can be accessed as a *char* pointer by a C function:

```
dataseg segment
public  ptr__
ptr__   dw      str
dataseg ends
```

To access *ptr__*, a C function could use the following declaration:

```
extern char *ptr;
```

An assembly language module can access global initialized or uninitialized variables that are created in C modules by defining the variables using an *extrn* directive that is in the *dataseg* segment. For example, suppose a C module creates a global, uninitialized *int* named

count and a global, initialized *int* named *total* using the statement:

```
int count, total=1;
```

An assembly language module can access these variables by using the following directives:

```
dataseg segment
        extrn   count_:word, total_:word
dataseg ends
```

The above discussion assumes that the C modules and the assembly language modules follow the standard rule in the C language regarding external variables. This rule requires a global variable to be defined without the *extern* keyword in exactly one module, and with that keyword in all other modules. Aztec C also supports a relaxed version of this rule. For information on this, see the discussion on External variables in the Programmer Information sections of the Compiler and Assembler chapters, and the description of the *-D* option in the Linker chapter.

6.1.3 'Small code' and 'Large code' programs

All modules that are linked together into a program must use the same code model; that is, they must either all use the 'small code' memory model, or all use the 'large code' memory model.

An assembly language module will use 'large code' if it contains the *largecode* directive, and will use 'small code' otherwise.

The entry point to an assembly language function can be defined using the *proc* directive. In this case, the operand to *proc* that defines whether the function is a near or far proc is optional; if not specified, the assembler sets the type to *near* if the module uses 'small code', and to *far* if it uses 'large code'.

The entry point can also be defined using the *label* directive. In this case, the program must explicitly specify the type of the entry point, whether near or far.

When the 'small code' memory model is being used, an entry point can be defined in the label field of an instruction.

Finally, an entry point can be defined using the macros that are provided with Aztec C; this is described below.

6.1.4 'Small data' and 'large data' programs

The modules in a program can use different data memory models. The only restriction is that a pointer to a data object can be passed between two functions only if they use the same data memory model.

An assembly language module doesn't do anything special to declare that it uses 'small data' or 'large data': it just goes ahead and uses short or long pointers to data objects.

A long pointer is represented within a four byte field in segment:offset form, with the segment paragraph number in the most significant word of the field and the offset from the beginning of the segment in the least significant word.

6.1.5 Names of external functions and variables

The C compiler translates the name of a function or variable to assembly language by truncating the name to 31 characters and then appending an underscore character. Thus, assembly language modules that are to be accessible from C-language modules or that are to access C modules must obey this convention.

For example, the following C language module calls the function *bmp*, which simply adds 10 to the global *int count*. A C-language module refers to this function as *bmp*, and an assembly-language module refers to it as *bmp__*.

```
int count;
main()
{
        bmp();
}
```

An assembly language version of *bmp__* could be:

```
dataseg segment
        extrn count_:word
dataseg ends
codeseg segment
        public  bmp_
bmp_    proc
        add     count_,10
        ret
bmp_    endp
codeseg ends
```

6.1.6 Function calls and returns

The assembly language code generated by the compiler for a C language call to another function pushes the arguments onto the stack, in the reverse order in which they were specified in the call's argument list, and then calls the function.

An assembly language function returns to a C function caller by issuing a *ret* instruction, leaving the caller's arguments on the stack. The caller then removes the arguments from the stack.

An *int* or short pointer is returned by an assembly language function in register AX; a *long* or long pointer is returned in registers AX and DX, with the most significant word in DX. Floating point values are returned in internal memory locations, and are not discussed here.

For example, consider the following assembly language function, *sub_*, that takes two *int* arguments that are passed to it on the stack, subtracts them, and returns the difference as the function value. A C-language function will refer to this function using the name *sub*.

```
codeseg segment         para
        public  sub_
sub_    proc
        mov     bx,sp
        mov     ax,[bx+2]       ;get first argument
        sub     ax,[bx+4]       ;subtract second argument
        ret
sub_    endp
codeseg ends
```

This function is coded in such a way that it can only be used in a 'small code' program. We'll recode it below, using the Aztec C macros, so that it can be used by either a 'large code' or 'small code' program without requiring recoding.

The following C function calls *sub* to subtract *b* from *a*, and stores the difference in *c*:

```
main()
{
        int a,b,c;
        ...
        c = sub(a,b);
}
```

6.1.7 Register usage

An assembly language function that is called by a C function must preserve the segment registers and registers BP, SP, SI, and DI.

6.2 Assembly-language Macros

Aztec C provides a set of assembly language macros in the file *lmacros.h*, which can be used by assembly language modules that are to be called by C language functions. These macros simplify the task of writing such a module, allowing it to be accessed by programs that use different memory models.

To gain access to these macros, an assembly language module must contain the directive

```
include lmacros.h
```

at the beginning of the module. The assembler will search for this file as defined in the Assembler chapter.

In addition, the symbol *MODEL* should be defined, either with an *equ* directive contained within the module or with the *-D* option when the module is assembled. *MODEL* must be defined before *lmacros.h* is

included. The least significant bit of *MODEL* defines the code model used by the module and the program containing it: 0 for 'small code' and 1 for 'large code'. The next bit defines the data model used by the module: 0 for 'small data' and 1 for 'large data'; As usual, 'small data' and 'large data' mean that the module uses short or long pointers to data objects, respectively (at least when receiving data pointers from, or passing data pointers to, another module).

For example, the following commands assemble the file *prog.asm*, with the resulting code using different memory models:

command	*memory model*
as -DMODEL=0 prog	*small code, small data*
as -DMODEL=1 prog	*large code, small data*
as -DMODEL=2 prog	*small code, large data*
as -DMODEL=3 prog	*large code, large data*

If a program uses the 'large code' memory model, the macros automatically create two symbols: FARPROC, assigning it the value 1, and FPTRSIZE, assigning it the value 4. FARPROC's existence specifies that the function uses the 'large code' memory model. FPTRSIZE defines the number of bytes in a pointer to a function.

If the program uses the 'small code' memory model, the macros create FPTRSIZE and assign it the value 2, specifying that function pointers are two bytes long.

If the program uses the 'large data' memory model, the macros create the symbol LONGPTR, assigning it the value 1. Here are the macros:

The PROCDEF macro

procdef pname [,<arglist>]

The *procdef* macro defines the C-callable function named *pname_*, which a C module will refer to as *pname*, by issuing a *proc* directive for *pname_*. The type of the proc, ie near or far, will be set according to the memory model used by the program.

The macro automatically appends an underscore to *pname*; thus, *pname* looks just like it does in a C-language statement.

The optional parameter *<arglist>* defines the arguments that are passed to the function. It consists of a list of comma-separated items, each of which defines one argument, with the entire list being surrounded by angle brackets, <>. An item consists of a pair of comma-separated values, and is itself surrounded by angle brackets. The first value in an argument's item is the name by which the module can access the argument on the stack, and the second value defines the type of the argument.

The allowed codes for an argument's type value, and the types that the macro sets for the argument are:

code	*type*
byte	byte
word	word
dword	dword
cdouble	qword
ptr	data pointer
fptr	function pointer

The actual type of a data pointer depends on the memory model used by the module: for 'small data' the type of a *ptr* argument is *word*, and for 'large data' it's *dword*. Similarly, the type of a function pointer will be *word* or *dword*, depending on whether the module uses 'small code' or 'large code'.

For example, the assembly language function *sub* that was defined above could be recoded to use the macros as:

```
include lmacros.h
procdef  sub, <<arg1,word>,<arg2,word>>
mov      ax,arg1 ;get first argument
sub      ax,arg2 ;subtract second argument
pret             ;return from sub (macro defined below)
pend     sub     ;this macro is defined below
finish           ;as is this one
```

The use of these macros allows the program to refer to the function arguments by name instead of by their location on the stack; the recoded *sub* module refers to its first argument as *arg1* instead of 2[bx], and refers to its second argument as *arg2* instead of 4[bx]. In addition, the program doesn't have to be modified in order for it to be used by a 'large code' program, whereas the non-macroized version will have to be modified.

If *arglist* is specified, the macro also pushes BP onto the stack and sets a new value in BP; it is this known value in BP that allows a function to refer to an argument on the stack by name instead of by its position on the stack. When BP has been automatically saved on entry to a function, the macro that is used to exit the function also automatically restores it.

The PEND Macro

pend pname

The *pend* macro defines the end of the *pname* function, which was previously defined using the *procdef* macro.

The INTERNAL Macro

internal pname

The *internal* macro defines the function *pname*, which is globally-accessible but which can't be called from C functions. It simply issues the directives

```
        public  pname
pname   proc
```

The ENTRDEF Macro

entrdef pname [,<arglist>]

The *entrdef* macro defines a secondary, C-callable entry point *pname_*, which a C module will refer to using the name *pname*, by issuing a *public* directive for *pname_*.

pname should be specified just as it will be called by a C function, as the macro automatically appends an underscore to it.

As with the *procdef* macro, *arglist* is an optional list of items that define the arguments that will be passed to *pname*. See the *procdef* macro for a discussion of *arglist*.

As with *procdef*, *entrdef* automatically saves BP if an *arglist* is specified, sets a new value in it that allows programs to access arguments on the stack by name, and causes it to be restored on exit from the function.

The INTRDEF Macro

intrdef pname

The *intrdef* macro defines the secondary entry point *pname*, which is globally-accessible but which can't be called by C functions.

The PRET Macro

pret

The *pret* macro is used to return from a function whose entry point was defined by one of the above macros. It automatically restores BP, if it was saved by the entry macro, and then returns.

The LDPTR Macro

ldptr offreg, src [,segreg]

The *ldptr* macro loads a pointer to the data object that is in the memory location referenced by *src*.

The offset component of the object's address is loaded into the register specified by *offreg*. If the program uses the 'small data' memory model, this is all that the macro does.

If the module uses the 'large data' memory model, the segment component of the object's address is also loaded into a segment register. This register can be explicitly specified using the *segreg* argument. Otherwise, DS is used if *offreg* is SI, and ES is used if *offreg* is DI.

For example, the *sub* function defined above can be modified to take pointers to the *int* values that are to be subtracted:

```
include lmacros.h
procdef sub, <<arg1ptr, ptr>,<arg2ptr, ptr>>
pushds                  ;push ds if using large data
ldptr   di, arg1ptr,ds  ;get ptr to arg1 in ds:di
mov     ax,[di]
ldptr   di, arg2ptr,ds  ;get ptr to arg2 in ds:di
sub     ax,[di]
popds                   ;pop ds if it was saved
pret
pend    sub
```

This version of *sub* can be used without modification, regardless of the memory model used by the program.

The RETPTRM Macro

retptrm src

The *retptrm* macro can be used by a function that returns as its value a pointer to a data object. It loads a data object pointer from the memory location referenced by *src* into the register or registers in which pointers are returned. The function using the macro can then simply return to the caller.

For example, the following function, *func*, performs some operations and then returns the data pointer that is in the argument *memptr* to the caller. This module can be used by a program using any type of memory model, without modification.

```
procdef func, <<memptr, ptr>>
        ...
retptrm memptr
pret
pend    func
```

The RETPTRR Macro

retptrr offreg [,segreg]

The *retptrm* macro can be used by a function that returns as its

value a pointer to a data object. It loads the offset component of the pointer into AX from the register specified by *offreg*. If the module uses the 'large data' memory model it also loads the segment component of the pointer into DX from the register specified by *segreg*. The function using *retptrr* can then simply return, and the pointer will be returned to the caller.

The RETNULL Macro

retnull

The *retnull* macro can be used by a function that returns as its value a null pointer to a data object. It loads 0 as the offset component of the pointer into AX. If the module uses the 'large data' memory model, it also loads 0 as the segment component of the pointer into DX. The function using *retnull* can then simply return, and the pointer will be returned to the caller.

The PUSHDS and POPDS Macros

pushds
and
popds

The *pushds* and *popds* macros push and pop the DS segment register, respectively, only if the program uses the 'large data' memory model.

The FINISH Macro

finish

The *finish* macro issues the directive

codeseg ends

6.2.1 Example

Here is the code for the *rindex* function, which uses the Aztec C macros:

```
;Copyright (C) 1983 by Manx Software Systems
        include lmacros.h
        procdef rindex, <<string,ptr>,<chr,byte>>
        pushf
        cld
        push    di
ifndef LONGPTR
        mov     di,ds
        mov     es,di
endif
        ldptr   di,string,es
        mov     dx,di           ;save for later
        sub     ax,ax
        mov     cx,7fffH
repne   scasb
        mov     cx,di
        sub     cx,dx           ;compute length of string
        dec     di              ;backup to null byte
        mov     al,chr          ;get byte to look for
        std                     ;now go backwards
repne   scasb
        mov     ax,di
        je      found
        retnull
        pop     di
        popf
        pret
found:
        retptrr ax,es
        inc     ax
        pop     di
        popf
        pret
        pend    rindex
        finish
        end
```

6.3 Embedded Assembler Source

Assembly language statements can be embedded in a "C" program between an *#asm* and an *#endasm* statement. The pound sign (#) must stand in column one of the line, and the letters must be lower case.

Embedded assembler code must preserve the contents of the BP, SP, SI, DI registers, and the segment registers. It should make no assumptions about the contents of the registers, since the code that the compiler currently generates for C statements may change in the future.

Normally, the compiler keeps track of the contents of registers, in order to avoid having to generate code that unneccesarily reloads a value that is already in a register. When the compiler encounters embedded assembly language code, it forgets the contents of registers. Thus, embedding assembly language code in a C module may actually make the module less efficient.

In general, it is safest to contain assembly code in a separate, assembly-language, module rather than embedding it in C source.

7. Generating ROMable code

Programs created using the Aztec C86 software can be burned into ROM. The standard Aztec development software automatically places a program's executable code, initialized data, and uninitialized data in separate segments.

This discussion is divided into several sections: the first describes the features of ROMable programs; the second describes the special programs provided with Aztec C for creating ROMable code; the third demonstrates how to create a ROMable program; and the last describes *hex86*, a utility program that generates Intel hex code for a program.

7.1 Features of ROMable programs

Programs created with the Aztec software that are intended to be burned into ROM have the following features:

* The program can use any memory model;
* The program's stack is located in a 2K byte area within the program's uninitialized data area.
* Global and static variables can be pre-initialized. This means that the declaration of a variable outside of a function can define the value that the variable is to have when the program begins executing.
* Uninitialized global and static variables are automatically cleared when the program begins executing.
* Optionally, the program can automatically gain control on system reset or power-up.
* The program can be contained in any number of ROMs. Optionally, the program's even- and odd-numbered bytes can be contained in separate ROMs.
* The program's code and data can start at locations that you define using the linker's *-C* and *-D* options.
* You can specify the location within the first ROM at which the program is to begin.

7.2 Special ROM-related programs

Aztec C86 includes the following programs that are of use in the development of ROMable programs:

* Special startup routines, one of which must be used by ROM-based programs instead of the standard startup routine. The files containing object module versions of the startup routines are:
    + *srom.o*, for programs that use the 'small code' and 'small data' memory model;

+ *lrom.o*, for 'large code', 'large data' programs;
+ *lcrom.o*, for 'large code', 'small data' programs;
+ *ldrom.o*, for 'small code', 'large data' programs.

These object modules are generated from the assembler source file *lrom.asm*. When this file is assembled, the symbol MODEL, which can be defined using the assembler's -D option, defines the memory model:

-D option	*Memory Model*
none	Small code, small data
-DMODEL=1	Large code, small data
-DMODEL=2	Small code, large data
-DMODEL=3	Large code, large data

* *hex86*, a utility program that converts an executable program generated by the Aztec linker to Intel loader format. Many machines that burn programs into ROM require that the program be input to the machine in this format.

In order to allow a program to have preinitialized global and static data, *hex86* causes the ROM to contain a copy of the program's initialized data segment following the program's code. When the program starts, the startup routines copies the initialized data from ROM to RAM.

7.3 The procedure for generating ROMable code

To create a ROMable program, compile and assemble it in the usual way; link it, following the rules described below; generate Intel hex code for the program, by feeding it through *hex86*; and then feed the hex code into the ROM burner.

The following rules must be followed when linking the program:

* the starting paragraph numbers of the program's code and data must be specified using the linker's *-C* and *-D* options.
* The file to which the linker sends the linked program must have extension *.exe*.
* A special startup routine, which corresponds to the program's memory model, must be explicitly included in the program.

For example, the following commands generate a ROMable, 'small code', 'small data' program that contains the three user-written modules *main*, *in*, and *out*, the *srom.o* startup routine, and any needed modules from *c.lib*. It uses a 2K-byte ROM that begins at paragraph number 0xff80. Its data begins at paragraph 0x40. The startup routine in *srom.o* will gain control of the processor on system reset or power-up.

hex86 places a far jump instruction at location 0xffff0 to the beginning of the startup routine that is in *srom.o*. When the processor

is reset, or on power-up, this startup routine will copy the initialized data from ROM into RAM, clear the uninitialized data area in RAM, set up the segment registers, stack pointer, and several other fields, and branch to the program's *main* function.

Here now are the commands:

```
cc main
cc in
cc out
ln -C ff80 -D 40 -o main.exe main.o in.o out.o srom.o -lc
hex86 main.exe
```

The Intel hex code for the program, generated by *hex86*, is in the file *main.hex* and can be fed into a ROM burner.

7.4 Description of *hex86*

hex86 converts a program's executable code and a copy of its initialized data to Intel hex code, which is suitable for input to a ROM programmer.

Burning programs into multiple ROMs

By default, *hex86* generates Intel hex code for one ROM chip, which will contain all the program's code and a copy of its initialized data. It can optionally generate Intel hex code for several ROMs, placing the code for each ROM in a separate file. It can also optionally place the Intel hex code for the program's even- and odd-numbered bytes in separate files; for this, *hex86* must be run twice: once to generate the files for the ROMs that contain the even-numbered bytes, and once for the ROMs that contain the odd-numbered bytes.

Code and initialized data

The copy of the program's initialized data will follow the program's executable code in ROM. By default, the executable code will begin at the beginning of the first ROM. Using the *-B* option, you can have the code begin at a specified offset within this ROM.

Using the linker

A program that is converted by *hex86* is constrained as follows: it must have been linked with the Aztec linker; the extension of the file containing it must be *.exe*; when it was linked its starting code and data addresses must have been specified using the linker's *-C* and *-D* options. The program can use any memory model and can be any size.

Reset code

In addition to translating the code that's in a disk file into Intel hex code, *hex86* can generate system reset code; that is, code that, when burned into ROM, will begin at the location to which the processor will transfer control on system reset (0xffff:0). This code consists of

(1) optional hex code that you specify when you start *hex86*, (2) optionally followed by a jump instruction to the startup routine that is in the main body of the program. The hex code is specified using the *-j* option, while the jump to the startup routine is suppressed by specifying the *-z* option.

The ability to place special hex code at the reset address is useful for 80186-based systems, for which system reset causes all except the last 1K of memory to be disabled. If your program is larger than 1K, the special code at the reset address must enable memory before jumping to the program.

The ability to suppress the placing of a jump instruction at the system reset address is useful for code that won't reside at the top of the system's address space; it's also useful for programs that want complete control over the program's reset code (via the *-j* option).

In order for *hex86* to know where in the ROM to place the reset code, it must know the size of each ROM and the total size of all the ROMs. It assumes that each ROM is 2K bytes and that the total size is 2K (ie, that there is just one ROM). These assumptions can be overriden using the *-P* and *-S* options.

hex86 operating instructions

hex86 is started with a command of the form:

hex86 [-options] infile

infile is the name of the *.exe* file containing the linked program. It can't contain any relocation records; thus, if *hex86* terminates with the message "input file contains relocation records", it usually means that you didn't specify the *-C* and *-D* options when linking the program.

hex86 derives the output file names from input file name, by changing its extension as follows:

* If none of the *-P*, *-E* and *-O* options are specified, the output file's extension is *.hex*.
* If the *-P* option is specified, but the *-E* and *-O* options aren't, *hex86* will generate multiple files, one for each ROM. The extension of the output files are *.h00*, *.h01*, ...*h99*, with the number indicating its code's position in the memory space.
* If the *-E* or *-O* option is specified, but the *-P* option isn't, the output extension is *.hxe* or *.hxo*, respectively.
* If the *-E* and *-P* options are specified, *hex86* generates multiple files containing the program's even numbered bytes, with one file for each ROM. The output file names have extension *.e00*, *.e01*, ...*e99*. The number of a file's extension indicates the position of its code in the memory space.
* If the *-O* and *-P* options are specified, *hex86* generates multiple files containing the program's odd-numbered bytes,

with one file for each ROM. The output file names have extension *.o00*, *.o01*, ...*o99*. The number of a file's extension indicates the position of its code in the memory space.

hex86 supports the following options:

The -J option

Causes *hex86* to place special code at the system reset address.

The hex values for the code immediately follow the *-j* option, with no intervening spaces.

When *-j* is specified and *-z* isn't, *hex86* places a jump to the program's startup routine immediately after the *-j* code.

hex86 will display an error message if the total amount of reset code (*-j* code and jump instruction) exceeds the 16 byte maximum.

The -Z option

Suppresses the creation of the jump instruction to the program's startup routine. When *-z* isn't specified, this jump instruction is placed at the system reset address immediately following the optional *-j* code. You should use the *-z* option if the ROM will not occupy the top of the processor's memory space or if the ROM code is not to receive control on system startup. If you use the -Z option, it is your responsibility to place the proper branch at the reset address.

The -E and -O options

The -E and -O options cause *hex86* to produce output files containing only the program's bytes that have even- or odd-numbered addresses, respectively. Only one of these options can be used with a particular invocation of *hex86*. These options are useful when programming 8-bit ROMs for use on an 8086 (which has a 16-bit data bus).

The -S option

This option specifies the total size of the ROMs used by the program, in kilobytes. If this option isn't specified, the total size is assumed to be 2K bytes. If the -E or -O options are specified, the size given should still be the total size of the ROMs. For example, if two 2Kb ROMs are being programmed with odd and even bytes of the input data, the invocation of *hex86* for the even byte file should be

```
hex86 -e -s4 infile.exe
```

It is crucial that this option be specified correctly as it is used by the hex utility to calculate both the address of the system reset branch location and the address in high physical memory of the

ROM code segment.

The -P option

The *-P* option is used when a program is too big to fit in one ROM. It defines the size of each ROM, and has the format

-pn

where *n* is the size of each ROM, in Kbytes. When *-P* is used, the *-S* option must be too. For example, if 1K ROMs are being used, and the program needs three of them, and separate ROMs aren't being used for odd and even bytes, the command to start *hex86* would be:

hex86 -s3 -p1 infile.exe

The -B option

The *-B* option defines the offset within the first ROM at which the program is to begin. It has the form

-bx

where *x* is the offset in hex. For example, suppose the program *rupt* contains code that initializes the interrupt vectors, which begin at 0x400, and is followed by the interrupt handlers. The command to *hex86*, when an 8K ROM is used, might be

hex86 -b400 -z -s8 rupt.exe

The *-z* option is used since the startup vector need not be initialized.

UNITOOLS

Chapter Contents

Unitools ..... unitools
diff (Source File Comparator) 6
grep (Pattern Matcher) 10
make (Program Maintainance Utility) 16
1. The Basics 16
1.1 What MAKE does 17
1.2 The makefile 17
1.3 Rules 19
1.3.1 MAKE's use of rules 20
1.3.2 An Example 20
1.3.3 Interaction or rules and dependency entries 21
2. Advanced Features 21
2.1 Dependent files 21
2.2 Macros 22
2.2.1 Using Macros 22
2.2.2 Defining macros in a makefile 22
2.2.3 Defining macros in a command line 23
2.2.4 Macros used by built-in rules 23
2.2.5 Special macros 23
2.3 Rules 24
2.3.1 Rule definition 24
2.3.2 Built-in rules 25
2.4 Commands 26
2.4.1 Allowed commands 26
2.4.2 Logging commands and aborting MAKE 26
2.4.3 Long command lines 26
2.5 Makefile syntax 27
2.5.1 Comments 27
2.5.2 Line continuation 27
2.6 Starting MAKE 28
2.6.1 The command line 28
2.6.2 MAKE's standard output 29
2.7 Executing commands 29
2.8 Differences between the Manx and UNIX MAKEs 29
3. Examples 30
3.1 Example 1 30
3.2 Example 2 31
Z - the text editor 34
1. Getting Started 37
1.1 Creating a new file 37
1.2 Editing an existing file 40

2. More commands 45
2.1 Introduction 46
2.2. Paging and scrolling 48
2.3. Searching for strings 49
2.3.1 The other string search commands 49
2.3.2 Regular expressions 49
2.3.3 Disabling extended pattern matching 50
2.4. Local moves 52
2.4.1 Moving around on the screen 52
2.4.2 Moving within a line 52
2.4.3 Word movements 53
2.4.4 Moves within C programs 53
2.4.5 Marking and returning 54
2.4.6 Adjusting the screen 55
2.5. Making changes 56
2.5.1 Small changes 56
2.5.2 Operators for deleting and changing text 56
2.5.3 Deleting and changing lines 57
2.5.4 Moving blocks of text 57
2.5.5 Duplicating blocks of text 58
2.5.6 Named buffers 59
2.5.7 Moving text between files 60
2.5.8 Shifting text 60
2.5.9 Undoing and redoing changes 60
2.6. Inserting text 61
2.6.1 Additional commands 61
2.6.2 Insert mode commands 61
2.7. Macros 63
2.7.1 Immediate macro definition 63
2.7.2 Examples 63
2.7.3 Indirect macro definition 64
2.7.4 Re-executing macros 65
2.8 The Ex-like commands 67
2.8.1 Addresses in Ex commands 67
2.8.2 The 'substitutute' command 68
2.8.3 The '&' (repeat last substitution) command 69
2.9. Starting and stopping Z 70
2.10. Accessing files 73
2.10.1 File names 73
2.10.2 Writing files 73
2.10.3 Reading files 74
2.10.4 Editing another file 74
2.10.5 File lists 76
2.10.6 Tags 76
2.10.7 The CTAGS utility 77
2.11. Executing system commands 79
2.12. Options 80
2.13. Z vs. Vi 81

2.14. System dependent features 82
2.14.1 IBM PC features 82
3. Command Summary 85

Unitools

This chapter describes the Aztec C utility programs *z*, *make*, *grep*, and *diff*. *z* is similar to the UNIX text editor *vi*; the others are similar to the UNIX programs of the same names.

NAME

diff - Source file comparison utility

SYNOPSIS

diff [-b] file1 file2

DESCRIPTION

diff is a program, similar to the UNIX program of the same name, that determines the differences between two files containing text. *file1* and *file2* are the names of the files to be compared.

1. The -b option

The -b option causes *diff* to ignore trailing blanks (spaces and tabs) and to consider strings of blanks to be identical. If this option isn't specified, *diff* considers two lines to be the same only if they match *exactly.*

For example, if file1 contains the the line

^abc$

(^ and $ stand for "the beginning of the line" and "the end of the line", respectively, and aren't actually in the file) and if file2 contains the line

^abc $

then *diff* would consider the two lines to be the same or different, depending on whether or not it was started with the -b option.

And *diff* would consider the lines

^a b c$

and

^a b c$

to be the same or different, depending on whether or not it was started with the -b option.

diff will never consider blanks to match a null string, regardless of whether -b was used or not. So *diff* will never consider the lines

^abc$

and

^a bc$

to be the same.

2. The conversion list

diff writes, to its standard output, a "conversion list" that describes the changes that need to be made to *file1* to convert it into *file2*. The list is organized into a sequence of items, each of which describes one operation that must be performed on *file1*.

2.1 Conversion items

There are three types of operations that can be specified in a conversion list item:

* adding lines to *file1* from *file2*;
* deleting lines from *file1*;
* replacing (changing) *file1* lines with *file2* lines.

A conversion list item consists of a command line, followed by the lines in the two files that are affected by the item's operation.

2.1.1 The command line

An item's command line contains a letter describing the operation to be performed: 'a' for adding lines, 'd' for deleting lines, and 'c' for changing lines.

Preceding and following the letter are the numbers of the lines in *file1* and *file2*, respectively, that are affected by the command. If a range of lines in a file are affected, just the beginning and ending line numbers are listed, separated by a comma.

For example, the following command line says to add line 3 of *file2* after line 5 of *file1*:

5a3

and the next command line says to add lines 8,9, and 10 of *file2* after line 16 of *file1*:

16a8,10

The next command line says to delete lines 100 through 150 from *file1*, and that the last line in *file2* that matched a *file1* line was number 75:

100,150d75

The following command says to replace (change) line 32 in *file1* with line 33 in *file2*:

32c33

and the next command says to replace lines 453 through 500 in *file1* with lines 490 through 499 in *file2*:

453,500c490,499

2.1.2 The affected lines

As mentioned above, the lines affected by a conversion item's operation are listed after the item's command line. The affected lines from *file1* are listed first, flagged with a preceding '<'. Then come the affected lines from *file2*, flagged with a preceding '>'. The *file1* and *file2* lines are separated by the line

```
---
```

For example, the following conversion item says to add line 6 of *file2* after line 4 of *file1*. Line 6 of *file2* is "for (i=1; i<10;++i)":

```
4a6
> for (i=1; i<10;++i)
```

Since no lines from *file1* are affected by an 'add' conversion item, only the *file2* lines that will be added to *file1* are listed, and the separator line "---" is omitted.

The following conversion item says to delete lines 100 and 101 from *file1*, and that the last *file2* line that matched a *file1* line was numbered 110. The deleted lines were "int a;" and "double b;". Only the deleted lines are listed, and the separator line "---" is omitted:

```
100,101d110
< int a;
< double b;
```

The following conversion item says to replace lines 53 through 56 in *file1* with lines 60 and 61 in *file2*. Lines 53 through 56 in *file1* are "if (a=b){", " d = a;", " a++;", and "}". Lines 60 and 61 of *file2* are "if (a==b)" and "d = a++;".

```
53,55c60,61
< if (a=b){
<   d = a;
<   a++;
< }
---
> if (a==b)
>   d = a++;
```

3. Differences between the UNIX and Manx versions of *diff*

The Manx and UNIX versions of *diff* are actually most similar when the latter program is invoked with the -h option. As with the UNIX *diff* when used with the -h option, the Manx *diff* works best when changed stretches are short and well separated, and works with files of unlimited length.

Unlike the UNIX *diff*, the Manx *diff* doesn't support the options e, f, or h.

Unlike the UNIX *diff*, the Manx version requires that both operands to *diff* be actual files. Because of this, the Manx version of *diff* doesn't support the features of the UNIX version which allows one operand to be a directory name, (to specify a file in that directory having the same name as the other operand), and which allows one operand to be '-' (to specify *diff*'s standard input instead of a file).

NAME

grep - pattern-matching program

SYNOPSIS

grep [-cflnv] pattern [files]

DESCRIPTION

grep is a program, similar to the UNIX program of the same name, that searches files for lines containing a pattern. By default, such lines are written to *grep*'s standard output.

1. Input files

The **files** parameter is a list of files to be searched. If no files are specified, *grep* searches its standard input. Each file name can specify a single file to be searched. A name can also specify a class of files to be searched, using the special characters '*' and '?'. The character '*' matches any string of characters in a file name, and '?' matches any single character. For example,

grep int main.c sub1.c sub2.c

searches *main.c, sub1.c,* and *sub2.c* for the string *int.* The command

grep int *.c

searches all files whose extension is *.c* for the string *int.* The command

grep int a*.txt b*.doc

searches for the string *int* in each file whose (1) extension is *.txt* and *first* character is *a* and whose (2) extension is *.doc* and first character is *b.* The command

grep int sub?.c

searches for the string *int* in each file whose filename contains four characters, the first three being *sub,* and whose extension is *.c.*

2. Options

The following options are supported:

v	Print all lines that don't match the pattern.
c	Print just the name of each file and the number of matching lines that it contained.
l	Print the names of just the files that contain matching lines.
n	Precede each matching line that's printed by its relative line number within the file that contains it.
f	A character in the pattern will match both its upper and lower case equivalent.

3. Patterns

A pattern consists of a limited form of regular expression. It describes a set of character strings, any of whose members are said to be matched by the regular expression.

Some patterns match just a single character; others, which match strings, can be constructed from those that match single characters. In the following paragraphs, we'll first describe the patterns that match a single character, and then describe patterns that match strings of characters.

3.1 Matching single characters

The patterns that match a single character are these:

* An ordinary character (that is, one other than the special characters described below) matches itself.

* A period (.) is a pattern that matches any character except newline.

* A non-empty string of characters enclosed in square brackets, [], matches any one character in that string. For example, the pattern

  [ad9@]

  matches any one of the characters *a, d, 9,* or @.

  If, however, the string begins with the caret character (^), the regular expression matches any character except the other enclosed characters and newline. The '^' has this special meaning only if it is the first character of the string. For example, the pattern

  [^ad9@]

  matches any single character *except a, d, 9,* or @.

  The minus character ,-, can be used to indicate a range of consecutive ASCII characters. For example, [0-9] is equivalent to [0123456789].

* A backslash (\) followed by a special character matches the special character itself. The special characters are:

  ., *, [, and \, which are always special, except when they appear in square brackets, [].

  ^ (caret), which is special when it is at the beginning of an entire regular expression (as discussed in 3.4) and when it immediately follows the left of a pair of square brackets.

  $, which is special at the end of an entire regular

expression (discussed in 3.4).

3.2 Matching character strings

Patterns can be concatenated. In this case, the resulting pattern matches strings whose substrings match each of the concatenated patterns. For example, the pattern

```
abc
```

matches the string *abc*. This pattern is built from the patterns *a*, *b*, and *c*. The pattern

```
a.c
```

matches strings containing three characters, whose first and last characters are *a* and *c*, respectively, such as

```
abc
a@c
axc
```

3.3 Matching repeating characters

A pattern can be built by appending an asterick (*) to a pattern that matches a single character. The resulting pattern matches zero or more occurrences of the single-character pattern For example, the pattern

```
a*
```

matches any line containing zero or more *a* characters. And the pattern

```
sub[1-4]*end
```

matches lines containing strings such as

```
subend
sub132132end
```

3.4 Matching strings that begin or end lines

An entire pattern may be constrained to match only character strings that occur at the beginning or the end of a line, by beginning or ending the pattern with the character '^' or '$', respectively. For example, the pattern

```
^main
```

matches the line that begins

```
main
```

but not one that begins

```
the main ...
```

The pattern

line$

matches the line ending in

... the end of the line

but not the line ending in

a hard-hit line drive.

4. Examples

4.1 Simple string matching

The following command will search the files *file1.txt* and *file2.txt* and print the lines containing the word *heretofore*:

grep heretofore file1.txt file2.txt

If you aren't interested in the specific lines of these files, but just want to know the names of the files containing the word *heretofore*, you could enter

grep -l heretofore file1.txt file2.txt

The above two examples ignore lines in which *heretofore* contains capital letters, such as when it begins a sentence. The following command will cover this situation:

grep -lf heretofore file1.txt file2.txt

grep processes all options at once, so multiple options must be specified in one dash parameter. For example, the command

grep -l -f heretofore file1.txt file2.txt

won't work.

4.2 The special character '.'

Suppose you want to find all lines in the file prog.c that contain a four-character string whose first and last characters are 'm' and 'n', respectively, and whose other characters you don't care about. The command

grep m..n prog.c

will do the trick, since the special character '.' matches any single character.

4.3 The backslash character

There are occasions when you want to find the character '.' in a file, and don't want grep to consider it to be special. In this case, you can use the backslash character, '\', to turn off the special meaning of the next character.

For example, suppose you want to find all lines containing

.PP

Entering

grep .PP prog.doc

isn't adequate, because it will find lines such as

THE APPLICATION OF

since the '.' matches the letter 'A'. But if you enter

grep \.PP prog.doc

grep will print just what you want.

The backslash character can be used to turn off the special meaning of any special character. For example,

grep \\n prog.c

finds all lines in prog.c containing the string '\n'.

4.4 The dollar sign and the caret ($ and ^)

Suppose you want to find the number of the line on which the definition of the function *add* occurs in the file *arith.c.* Entering

grep -n add arith.c

isn't good, because it will print lines in which *add* is called in addition to the line you're interested in. Assuming that you begin all function definitions at the beginning of a line, you could enter

grep ^add arith.c

to accomplish your purpose.

The character '$' is a companion to '^', and stands for 'the end of the line'. So if you want to find all lines in *file.doc* that end in the string *time,* you could enter

grep time$ file.doc

And the following will find all lines that contain just *.PP:*

grep ^\.PP$

4.5 Using brackets

Suppose that you want to find all lines in the file *file.doc* that begin with a digit. The command

grep ^[0123456789] file.doc

will do just that. This command can be abbreviated as

grep ^[0-9] file.doc

And if you wanted to print all lines that don't begin with a digit, you could enter

```
grep ^[^0-9] file.doc
```

4.6 Repeated characters

Suppose you want to find all lines in the file *prog.c* that contain strings whose first character is 'e' and whose last character is 'z'. The command

```
grep e.*z prog.c
```

will do that. The 'e' matches an 'e', the '.*' matches zero or more arbitrary characters, and the 'z' matches a 'z'.

5. Differences between the Manx and UNIX versions of grep

The Manx and UNIX versions of *grep* differ in the options they accept and the patterns they match.

5.1 Option differences

* The option -f is supported only by the Manx *grep.*
* The options -b and -s are supported only by the UNIX *grep.*

5.2 Pattern differences

Basically, the patterns accepted by the Manx *grep* are a subset of those accepted by the UNIX *grep.*

* The Manx *grep* doesn't allow a regular expression to be surrounded by '\(' and '\)'.
* The Manx *grep* doesn't accept the construct '\{m\}'.
* The Manx *grep* doesn't allow a right bracket, ']', to be specified within brackets.
* Quoted strings can't be passed to the Manx *grep.* For example, the Manx *grep* won't accept

```
grep 'this is a fine kettle of fish' file.doc
```

NAME

make - Program maintenance utility

SYNOPSIS

make [-n] [-f makefile] [-a] [name1 name2 ...]

DESCRIPTION

make is a program, similar to the UNIX program of the same name, whose primary function is to create, and keep up-to-date, files that are created from other files, such as programs, libraries, and archives.

When told to make a file, *make* first ensures that the files from which the target file is created are up-to-date or current, recreating just the ones that aren't. Then, if the target file is not current, *make* creates it.

Inter-file dependencies and the commands which must be executed to create files are specified in a file called the 'makefile', which you must write.

make has a rule-processing capability, which allows it to infer, without being explicitly told, the files on which a file depends and the commands which must be executed to create a file. Some rules are built into *make*; you can define others within the makefile.

A rule tells *make* something like this:

> "a target file having extension '.x' depends on the file having the same basic name and extension '.y'. To create such a target file, apply the commands ...".

Rules simplify the task of writing a makefile: a file's dependency information and command sequences need be explicitly specified in a makefile only if this information can't be inferred by the application of a rule.

make has a macro capability. A character string can be associated with a macro name; when the macro name is invoked in the makefile, it's replaced by its string.

Preview

The rest of this description of *make* is divided into the following sections:

1. The basics
2. Advanced features
3. Examples

1. The basics

In this section we want to present the basic features of *make*, with which you'll be able to start using *make*. Section 2 describes the other

features of *make*.

Before you can begin using *make*, you must know what *make* does, how to create a simple makefile that contains dependency entries, how to take advantage of *make*'s rule-processing capability, and, finally, how to tell *make* to make a file. Each of these topics is discussed in the following paragraphs.

1.1 What make does

The main function of *make* is to make a target file "current", where a file is considered "current" if the files on which it depends are current and if it was modified more recently than its prerequisite files. To make a file current, *make* makes the prerequisite files current; then, if the target file is not current, *make* executes the commands associated with the file, which usually recreates the file.

As you can see, *make* is inherently recursive: making a file current involves making each of its prerequisite files current; making these files current involves making each of their prerequisite files current; and so on.

make is very efficient: it only creates or recreates files that aren't current. If a file on which a target file depends is current, *make* leaves it alone. If the target file itself is current, *make* will announce the fact and halt without modifying the target.

> **It is important to have the time and date set for *make* to behave properly, since *make* uses the 'last modified' times that are recorded in files' directory entries to decide if a target file is not current.**

1.2 The makefile

When *make* starts, one of the first things it does is to read a file, which you must write, called the 'makefile'. This file contains dependency entries defining inter-file dependencies and the commands that must be executed to make a file current. It also contains rule definitions and macro definitions.

In the following paragraphs, we want to just describe dependency entries. In section 2 we discuss the somewhat more advanced topics of rule and macro definition.

A dependency entry in a makefile defines one or more target files, the files on which the targets depend, and the operating system commands that are to be executed when any of the targets is not current. The first line of the entry specifies the target files and the files on which they depend; the line begins with the target file names, followed by a colon, followed by one or more spaces or tabs, followed by the names of the prerequisite files. It's important to place spaces or tabs after the colon that separates target and dependent files; on systems that allow colons in file names, this allows *make* to distinguish

between the two uses of the colon character.

The commands are on the following lines of the dependency information entry. The first character of a command line must be a tab; *make* assumes that the command lines end with the last line not beginning with a tab.

For example,consider the following dependency entry:

```
prog.com: prog.o sub1.o sub2.o
        ln -o prog.com prog.o sub1.o sub2.o -lc
```

This entry says that the file *prog.com* depends on the files *prog.o*, *sub1.o*, and *sub2.o*. It also says that if *prog.com* is not current, *make* should execute the *ln* command. *make* considers *prog.com* to be current if it exists and if it has been modified more recently than *prog.o*, *sub1.o*, and *sub2.o*.

The above entry describes only the dependence of *prog.com* on *prog.o*, *sub1.o*, and *sub2.o*. It doesn't define the files on which the '.o' files depend. For that, we need either additional dependency entries in the makefile or a rule that can be applied to create '.o' files from '.c' files.

For now, we'll add dependency entries in the makefile for *prog.o*, *sub1.o*, and *sub2.o*, which will define the files on which the object modules depend and the commands to be executed when an object module is not current. In section 1.3 we'll then modify the makefile to make use of *make*'s built-in rule for creating a '.o' file from a '.c' file.

Suppose that the '.o' files are created from the C source files *prog.c*, *sub1.c*, and *sub2.c*; that *sub1.c* and *sub2.c* contain a statement to include the file *defs.h* and that *prog.c* doesn't contain any #include statements. Then the following long-winded makefile could be used to explicitly define all the information needed to make *prog.com*

```
prog.com: prog.o sub1.o sub2.o
        ln -o prog.com prog.o sub1.o sub2.o -lc

prog.o: prog.c
        cc prog.c

sub1.o: sub1.c defs.h
        cc sub1.c

sub2.o: sub2.c defs.h
        cc sub2.c
```

This makefile contains four dependency entries: for *prog.com*, *prog.o*, *sub1.o*, and *sub2.o*. Each entry defines the files on which its target file depends and the commands to be executed when its target isn't current. The order of the dependency entries in the makefile is not important.

We can use this makefile to make any of the four target files defined in it. If none of the target files exists, then entering

```
make prog.com
```

will cause *make* to compile and assemble all three object modules from their C source files, and then create *prog.com* by linking the object modules together.

Suppose that you create *prog.com* and then modify *sub1.c*. Then telling *make* to make *prog.com* will cause *make* to compile and assemble just *sub1.c*, and then recreate *prog.com*.

If you then modify *defs.h*, and then tell *make* to make *prog.com*, *make* will compile and assemble *sub1.c* and *sub2.c*, and then recreate *prog.com*.

You can tell *make* to make any file defined as a target in a dependency entry. Thus, if you want to make *sub2.o* current, you could enter

```
make sub2.o
```

A makefile can contain dependency entries for unrelated files. For example, the following dependency entries can be added to the above makefile:

```
hello.exe: hello.o
        ln hello.o -lc

hello.o: hello.c
        cc hello.c
```

With these dependency entries, you can tell *make* to make *hello.exe* and *hello.o*, in addition to *prog.com* and its object files.

1.3 Rules

You can see that the makefile describing a program built from many .o files would be huge if it had to explicitly state that each .o file depends on its .c source file and is made current by compiling its source file.

This is where rules are useful. When a rule can be applied to a file that *make* has been told to make or that is a direct or indirect prerequisite of it, the rule allows *make* to infer, without being explicitly told, the name of a file on which the target file depends and/or the commands that must be executed to make it current. This in turn allows makefiles to be very compact, just specifying information that *make* can't infer by the application of a rule.

Some rules are built into *make*; you can define others in a makefile. In the rest of this section, we're going to describe the properties of rules and how you write makefiles that make use of *make*'s built-in rule for creating a .o file from a .c file. For more information on rules,

including a complete list of built-in rules and how to define rules in a makefile, see section 2.2.

1.3.1 make's use of rules

A rule specifies a target extension, source extension, and sequence of commands. Given a file that *make* wants to make, it searches the rules known to it for one that meets the following conditions:

* The rule's target extension is the same as the file's extension;
* A file exists that has the same basic name as the file *make* is working on and that has the rule's source extension.

If a rule is found that meets these conditions, *make* applies the first such rule to the file it's working on, as follows:

* The file having the source extension is defined to be a prerequisite of the file with the target extension;
* If the file having the target extension doesn't have a command sequence associated with it, the rule's commands are defined to be the ones that will make the file current.

One rule built into *make*, for converting .c files into .o files, says

> "a file having extension '.o' depends on the file having the same basic name, with extension '.c'. To make current such a .o file, execute the command
>
> cc x.c
>
> where 'x' is the name of the file"

Another built-in rule exists for converting .asm files into .o files, using the Manx assembler.

1.3.2 An example

The .c to .o rule allows us to abbreviate the long-winded makefile given in section 1.2 as follows:

```
prog.com: prog.o sub1.o sub2.o
        ln -o prog.com prog.o sub1.o sub2.o -lc

sub1.o sub2.o: defs.h
```

In this abbreviated makefile, a dependency entry for *prog.o* isn't needed; using the built-in '.c to .o' rule, *make* infers that the *prog.o* depends on *prog.c* and that the command *cc prog.c* will make *prog.o* current.

The abbreviated makefile says that both *sub1.o* and *sub2.o* depend on *defs.h*. It doesn't say that they also depend on *sub1.c* and *sub2.c*, respectively, or that the compiler must be run to make them current; *make* infers this information from the .c to .o rule. The only information given in the dependency entry is that which *make* couldn't

infer by itself: that the two object files depend on *defs.h*.

1.3.3 Interaction of rules and dependency entries

As we showed in the above example, a rule allows you to leave some dependency information unspecified in a makefile. The *prog.o* entry in the long-winded makefile of section 1.2 was not needed, since its information could be inferred by the .c to .o rule. And the dependence of *sub1.o* and *sub2.o* on their respective C source files, and the commands needed to create the object files was also not needed, since the information could be inferred from the .c to .o rule.

There are occasions when you don't want a rule to be applied; in this case, information specified in a dependency entry will override that which would be inferred from a rule. For example, the following dependency entry in a makefile

```
add.o:
    cc -DFLOAT add.c
```

will cause *add.o* to be compiled using the specified command rather than the command specified by the .c to .o rule. *make* still infers the dependence of *add.o* on *add.c*, using the .c to .o rule, however.

2. Advanced features

In the last section we presented the basic features of *make*, with which you can start using *make*. In this section, we present the rest of *make*'s features.

2.1 Dependent Files

A dependent file can be in a different volume or directory than its target file, with the following provisos.

If the file name contains a colon (for example, because the file name defines the volume on which the file is located), the colon must be followed by characters other than spaces or tabs, so that *make* can distinguish between this use of the colon character and its use as a separator between the target and dependent files in a dependency line. This shouldn't be a problem, since most systems don't allow file names to contain spaces or tabs.

All references to a file must use the same name. For example, if a file is referred to in one place using the name

```
/root/src/foo.c
```

then all references to the file must use this exact same name.

On PCDOS and MSDOS, note that the following names may refer to different files:

a:dir/sub/foo.c
a:/dir/sub/foo.c.

For the first name, the search for *foo.c* begins with the current directory on the *a:* drive; for the second, the search begins with the root directory on the *a:* drive.

2.2 Macros

make has a simple macro capability that allows character strings to be associated with a macro name and to be represented in the makefile by the name. In the following paragraphs, we're first going to describe how to use macros within a makefile, then how they are defined, and finally some special features of macros.

2.2.1 Using macros

Within a makefile, a macro is invoked by preceding its name with a dollar sign; macro names longer than one character must be parenthesized. For example, the following are valid macro invocations:

```
$(CFLAGS)
$2
$(X)
$X
```

The last two invocations are identical.

When *make* encounters a macro invocation in a dependency line or command line of a makefile, it replaces it with the character string associated with the macro. For example, suppose that the macro OBJECTS is associated with the string *a.o b.o c.o d.o*. Then the dependency entries:

```
prog.exe: prog.o a.o b.o c.o d.o
        ln prog.o a.o b.o c.o d.o

a.o b.o c.o d.o: defs.h
```

within a makefile could be abbreviated as:

```
prog.exe: prog.o $(OBJECTS)
    ln prog.o $(OBJECTS)

$(OBJECTS): defs.h
```

There are three special macros: $$, $*, and $@. $$ represents the dollar sign. The other two are discussed below.

2.2.2 Defining macros in a makefile

A macro is defined in a makefile by a line consisting of the macro name, followed by the character '=', followed by the character string to be associated with the macro.

For example, the macro OBJECTS, used above, could be defined in the makefile by the line

```
OBJECTS = a.o b.o c.o d.o
```

A makefile can contain any number of macro definition entries. A macro definition must appear in the makefile before the lines in which it is used.

2.2.3 Defining macros in a command line

A macro can be defined in the command line that starts *make*. The syntax for a command line definition has the following form:

```
mac=str
```

where *mac* is the name of the macro, and *str* is its value.

str cannot contain spaces or tabs.

For example, the following command assigns the value *-DFLOAT* to the macro *CFLAGS*:

```
make CFLAGS=-DFLOAT
```

The assignment of a value to a macro in a command line overrides an assignment in a makefile statement.

2.2.4 Macros used by built-in rules

make has two macros, CFLAGS and AFLAGS, that are used by the built-in rules. These macros by default are assigned the null string. This can be overriden by a macro definition entry in the makefile.

For example, the following would cause CFLAGS to be assigned the string "-T":

```
CFLAGS = -T
```

These macros are discussed below in the description of built-in rules.

2.2.5 Special macros

Before issuing any command, two special macros are set: $@ is assigned the full name of the target file to be made, and $* is the name of the target file, without its extension. Unlike other macros, these can only be used in command lines, not in dependency lines.

For example, suppose that the files *x.c*, *y.c*, and *z.c* need to be compiled using the option "-DFLOAT". The following dependency entry could be used:

```
x.o y.o z.o:
    cc -DFLOAT $*.c
```

When *make* decides that *x.o* needs to be recreated from *x.c*, it will assign $* the string "x", and the command

```
cc -DFLOAT x.c
```

will be executed. Similarly, when *y.o* or *z.o* is made, the command *cc -DFLOAT y.c* or *cc -DFLOAT z.c* will be executed.

The special macros can also be used in command lines associated with rules. In fact, the $@ macro is primarily used by rules. We'll discuss this more in the description of rules, below.

2.3 Rules

In section 1, we presented the basic features of rules: what they are and how they are used. We also noted that rules could be defined in the makefile and that some rules are built into *make*. In the following paragraphs, we describe how rules are defined in a makefile and list the built-in rules.

2.3.1 Rule definition

A rule consists of a source extension, target extension, and command list. In a makefile, an entry defining a rule consists of a line defining the two extensions, followed by lines containing the commands.

The line defining the extensions consists of the source extension, immediately followed by the target extension, followed by a colon.

All command lines associated with a rule must begin with a tab character. The first line following the extension line that doesn't begin with a tab terminates the commands for the rule.

For example, the following rule defines how to create a file having extension *.rel* from one having extension *.c*:

```
.c.rel:
    cc -o $@ $*.c
```

The first line declares that the rule's source and target extension are *.c* and *.rel*, respectively.

The second line, which must begin with a tab, is the command to be executed when a *.rel* file is to be created using the rule.

Note the existence of the special macros $@ and $* in the command line. Before the command is executed to create a *.rel* target file using the rule, the macro $@ is replaced by the full name of the target file, and the macro $* by the name of the target, less its extension.

Thus, if *make* decides that the file *x.rel* needs to be created using this rule, it will issue the command

```
cc -o x.rel x.c
```

If a rule defined in a makefile has the same source and target extensions as a built-in rule, the commands associated with the

makefile version of the rule replace those of the built-in version. For example, the built-in rule for creating a *.o* file from a *.c* file looks like this:

```
.c.o:
    cc $(CFLAGS) $*.c
```

If you want the rule to generate an assembly language listing, include the following rule in your makefile:

```
.c.o:
    cc $(CFLAGS) -a $*.c
    as -ZAP -l $*.asm
```

2.3.2 Built-in rules

The following rules are built into *make*. The order of the rules is important, since *make* searches the list beginning with the first one, and applies the first applicable rule that it finds.

```
.c.o:
    cc $(CFLAGS) -o $@ $*.c


.c.obj:
    cc $(CFLAGS) $*.c
    obj $*.o $@

.asm.obj:
    as $(AFLAGS) $*.asm
    obj $*.o $@

.asm.o:
    as $(AFLAGS) -o $@ $*.asm


.a86.o:
    as $(AFLAGS) -o $@ $*.a86
```

The two macros CFLAGS and AFLAGS that are used in the built-in rules are built into *make*, having the null character string as their values. To have *make* use other options when applying one of the built-in rules, you can define the macro in the makefile.

For example, if you want the options -T and -DDEBUG to be used when *make* applies the *.c.o* rule, you can include the line

```
CFLAGS = -T -DDEBUG
```

in the makefile. Another way to accomplish the same result is to redefine the *.c.o* rule in the makefile; this, however, would use more lines in the makefile than the macro redefinition.

2.4 Commands

In this section we want to discuss the execution of operating system commands by *make*.

2.4.1 Allowed commands

A command line in a dependency entry or rule within a makefile can specify any command that you can enter at the keyboard. This includes batch commands, commands built into the operating system, and commands that cause a program to be loaded and executed from a disk file.

2.4.2 Logging commands and aborting make

Normally, before *make* executes a command, it writes the command to its standard output device; and when the command terminates, *make* halts if the command's return code was non-zero. Either or both of these actions can be suppressed for a command, by preceding the command in the makefile with a special character:

@	Tells *make* not to log the command;
-	Tells *make* to ignore the command's return code.

For example, consider the following dependency entry in a makefile:

```
prog.exe: a.o b.o c.o d.o
    ln -o prog.exe a.o b.o c.o d.o -lc
    @echo all done
```

When the *echo* command is executed, the command itself won't be logged to the console.

2.4.3 Long command lines

Makefile commands that start a Manx program, such as *cc*, *as*, or *ln*, or that start a program created with *cc*, *as*, *ln*, and *c.lib*, can specify a command line containing up to 2048 characters.

For example, if a program depends on fifty modules, you could associate them with the macro OBJECTS in the makefile, and also include the dependency entry

```
prog.exe: $(OBJECTS)
    ln -o prog.exe $(OBJECTS) -lc
```

This will result in a very long command line being passed to *ln*.

In the next section we will describe how OBJECTS could be defined.

For the execution of other commands, the command line can contain at most 127 characters.

2.5 Makefile syntax

We've already presented most of the syntax of a makefile; that is, how to define rules, macros, and dependencies. In this section we want to present two features of the makefile syntax not presented elsewhere: comments and line continuation.

2.5.1 Comments

make assumes that any line in a makefile whose first character is '#' is a comment, and ignores it. For example:

```
#
# the following rule generates an 8080 object module
# from a C source file:
#
.c.o80:
    cc80 -o cc.tmp $*.c
    as80 -ZAP -o $*.o80 cc.tmp
```

2.5.2 Line continuation

Many of the items in a makefile must be on a single line: a macro definition, the file dependency information in a dependency entry, and a command that *make* is to execute must each be on a single line.

You can tell *make* that several makefile lines should be considered to be a single line by terminating each of the lines, except the last, with the backslash character, '\'. When *make* sees this, it replaces the current line's backslash and newline, and the next line's leading blanks and tabs by a single blank, thus effectively joining the lines together.

The maximum length of a makefile line after joining continued lines is 2048 characters.

For example, the following macro definition equates OBJ to a string consisting of all the specified object module names.

```
OBJ = printf.o fprintf.o format.o\
    scanf.o fscanf.o scan.o\
    getchar.o getc.o
```

As another example, the following dependency entry defines the dependence of *driver.lib* on several object modules, and specifies the command for making *driver.lib*:

```
driver.lib: driver.o printer.o \
    in.o \
    out.o
        lb driver.lib driver.o\
          printer.o \
          in.o out.o
```

This second example could have been more cleanly expressed using a macro:

```
DRIVOBJ= driver.o printer.o\
    in.o out.o

driver.lib: $(DRIVOBJ)
    lb driver.lib $(DRIVOBJ)
```

This was done to show that dependency lines and command lines can be continued, too.

2.6 Starting make

You've already seen how *make* is told to make a single file. Entering

```
make filename
```

makes the file named *filename*, which must be described by a dependency entry in the makefile. And entering

```
make
```

makes the first file listed as a target file in the first dependency entry in the makefile.

In both of these cases, *make* assumes the makefile is named 'makefile' and that it's in the current directory on the default drive.

In this section we want to describe the other features available when starting *make*.

2.6.1 The command line

The complete syntax of the command line that starts *make* is:

```
make [-n] [-f makefile] [-a] [-dmacro=str] [file1] [file2] ...
```

Square brackets indicate that the enclosed parameter is optional.

The parameters *file1*, *file2* ... are the names of the files to be made. Each file must be described in a dependency entry in the makefile. They are made in the order listed on the command line.

The other command line parameters are options, and can be entered in upper or lower case. Their meanings are:

-n	Suppresses command execution. *make* logs the commands it would execute to its standard output device, but doesn't execute them.
-f makefile	Specifies the name of the makefile
-a	Forces *make* to make all files upon which the specified target files directly or indirectly depend, and to make the target files, even those that it considers current.
-dMACRO=str	

Creates a macro named MACRO, and assigns *str* as its value.

2.6.2 make's standard output

make logs commands and error messages to its standard output device. This can be redirected in the standard way. For example, to make the first target file in the first dependency entry and log messages to the file *out*, enter

```
make >out
```

The standard input and output devices of programs started by *make* are set as they are for *make* itself, unless one or both of them are explicitly redirected in the command that starts the program.

2.7 Executing commands

When *make* decides that a command needs to be executed, it executes it immediately, and waits for the command to finish. It activates a command whose code is contained in a disk file by issuing an *fexec* function call. It activates DOS built-in commands and batch commands by calling the *system* function, which causes a new copy of the command processor to be loaded. Thus, to use *make*, your system must have enough memory for DOS, *make*, and whatever programs are loaded by *make* to be in memory simultaneously.

2.8 Differences between the Manx and UNIX 'make' programs

The Manx *make* supports a subset of the features of the UNIX *make*. The following comments present features of the UNIX *make* that aren't supported by the Manx *make*.

* The UNIX *make* will let you make a file that isn't defined as a target in a makefile dependency entry, so long as a rule can be applied to create it. The Manx *make* doesn't allow this. For example, if you want to create the file *hello.o* from the file *hello.c* you could say, on UNIX

  ```
  make hello.o
  ```

  even if *hello.o* wasn't defined to be a target in a makefile dependency entry. With the Manx *make*, you would have to have a dependency entry in a makefile that defines *hello.o* as a target.

* The UNIX *make* supports the following options, which aren't supported by the Manx *make*:

  p, i, k, s, r, b, e, m, t, d, q

  The Manx *make* supports the option '-a', which isn't supported by the UNIX *make*.

* The special names .DEFAULT, .PRECIOUS, .SILENT, and .IGNORE are supported only by the UNIX *make*.

* Only the UNIX *make* allows the makefile to be read from *make*'s standard input.
* Only the UNIX *make* supports the special macros $<, $?, and $%, and allows an upper case D or F to be appended to the special macros, which thus modifies the meaning of the macro.
* Only the UNIX *make* requires that the suffixes for additional rules be defined in a .SUFFIXES statement.
* Only the UNIX *make* allows macros to be defined on the command line that activates *make*.
* Only the UNIX *make* allows a target to depend on a member of a library or archive.

3. Examples

3.1 First example

This example shows a makefile for making several programs. Note the entry for *arc*. This doesn't result in the generation of a file called *arc*; it's just used so that we can generate *arcv* and *mkarcv* by entering *make arc*.

```
#
# rules:
#
.c.o80:
        cc80 -DTINY -o $@ $*.c
#
# macros:
#
OBJ=make.o parse.o scandir.o dumptree.o rules.o command.o
#
# dependency entry for making make:
#

make.com: $(OBJ) cntlc.o envcopy.o
        ln -o make.com $(OBJ) envcopy.o cntlc.o -lc
#
# dependency entries for making arcv & mkarcv:
#

arc: mkarcv.com arcv.com
        @echo done

mkarcv.com: mkarcv.o
        ln -o mkarcv.com mkarcv.o -lc
arcv.com : arcv.o
        ln -o arcv.com  arcv.o -lc
#
# dependency entries for making CP/M-80 versions of arcv & mkarcv:
#
mkarcv80.com: mkarcv.o80
        ln80 -o mkarcv80.com mkarcv.o80 -lt -lc
arcv80.com: arcv.o80
        ln80 -o arcv80.com arcv.o80 -lt -lc

$(OBJ): libc.h make.h
```

3.2 Second example

This example uses *make* to make a library, my.lib. Three directories are involved: the directory *libc* and two of its subdirectories, *sys* and *misc*. The C and assembly language source files are in the two subdirectories. There are makefiles in each of the three directories, and this example makes use of all of them. With the current directory being *libc*, you enter

```
make my.lib
```

This starts *make*, which reads the makefile in the libc directory. *make* will change the current directory to *sys* and then start another *make* program.

This second *make* compiles and assembles all the source files in the *sys* directory, using the makefile that's in the *sys* directory.

When the 'sys' *make* finishes, the 'libc' *make* regains control, and then starts yet another *make*, which compiles and assembles all the source files in the *misc* subdirectory, using the makefile that's in the *misc* directory.

When the 'misc' *make* is done, the 'libc' *make* regains control and builds *my.lib*. You can then remove the object files in the subdirectories by entering

```
make clean
```

3.2.1 The makefile in the 'libc' directory

```
my.lib: sys.mk misc.mk
        del  my.lib
        lb my.lib -f my.bld
        @echo my.lib done

sys.mk:
        cd sys
        make
        cd ..

misc.mk:
        cd misc
        make
        cd ..

clean:
        cd sys
        make clean
        cd ..
        cd misc
        make clean
        cd ..
```

3.2.2 Makefile for the 'sys' directory

```
REL=asctime.o bdos.o begin.o chmod.o croot.o csread.o ctime.o \
    dostime.o dup.o exec.o execl.o execlp.o execv.o execvp.o \
    fexec.o fexecl.o fexecv.o ftime.o getcwd.o getenv.o  \
    isatty.o localtim.o mkdir.o open.o stat.o system.o time.o\
    utime.o wait.o dioctl.o ttyio.o access.o syserr.o

COPT=
HEADER=../header

.c.o:
        cc $(COPT) -I$(HEADER) $*.c -o $@
        sqz $@
.asm.o:
        as $*.asm -o $@
        sqz $@

all: $(REL)
        @echo sys done

clean:
        del  *.o
```

3.2.3 Makefile for the 'misc' directory

```
REL=atoi.o atol.o calloc.o ctype.o format.o malloc.o qsort.o \
        sprintf.o sscanf.o fformat.o fscan.o

COPT=
HEADER=../header

.c.o:
        cc $(COPT) -I$(HEADER) $*.c -o $@
        sqz $@
.asm.o:
        as $*.asm -o $@
        sqz $@

all: $(REL)
        @echo misc done

fformat.o: format.c
        cc -I$(HEADER) -DFLOAT format.c -o fformat.o

fscan.o: scan.c
        cc -I$(HEADER) -DFLOAT scan.c -o fscan.o

clean:
        del  *.o
```

NAME

z - A text editor

SYNOPSIS

z [files]

DESCRIPTION

Z is a text editor which is especially useful for creating source programs in the C programming language. It has the following features:

* It's very similar to the Unix editor *vi*: if you know *vi*, you know *Z*.
* It's a full-screen editor: the screen acts as a window into the file being edited.
* Z has a wealth of commands, and commands are specified with just a few keystrokes, allowing editing to be performed quickly and efficiently. The simple and natural way of entering commands and the mnemonic assignment of commands to keys makes the commands easy to remember and use.
* Z has commands for the following:
  + Bringing different sections of a file into view;
  + Inserting text;
  + Making changes to text;
  + Rearranging text by moving blocks of text around and by inserting text from other files;
  + Accessing files;
  + Searching for character strings and "regular expressions".
* Z has several commands which are useful for editing C programs: there are commands for finding matching parentheses, square brackets, and curly braces; for finding the beginning of the next or preceding function; and for finding the next or preceding blank line.
* Most commands can be easily executed repeatedly.
* Sequences of commands, called macros, can be defined and executed one or more times.
* Changes are made to an in-memory copy of a file; the file itself isn't changed until a command is explicitly given;
* Z has a feature which is useful when editing a large number of related files: the operator can request that a file containing a certain function be edited; Z will automatically find the file and prepare it for editing.

Requirements

Z runs on several systems, including

* IBM PC, running PCDOS version 2.0 or later
* 8086-based systems running CP/M-86 and using an ADM-3A or LSI terminal;
* The Macintosh
* The Amiga
* TRS-80, model 4, using TRSDOS

For 8086-based systems, *Z* requires at least 128KB of memory and allows you to edit programs containing up to 58 K bytes of text.

For 8080- and Z80-based systems, *Z* requires 64 KB of memory and allows you edit programs containing up to 11 K bytes of text.

Components

The *Z* package contains two programs:

Z, the text editor;

ctags, a utility for creating a file which relates tags to C source files.

Preview

The remainder of this description of Z is divided into the following sections:

getting started , which describes how to quickly start using *Z*;

commands and features , which presents an overview of the features and commands of *Z*;

summary , which summarizes the *Z* commands.

1. GETTING STARTED

Z is a very powerful tool for creating and editing C source programs, but its wealth of commands and options can be overwhelming to someone not familiar with it. The purpose of this chapter is to get you using *Z* as quickly as possible, by presenting a small subset of the *Z* commands, with which programs can be created and edited. Then, with the ability to create and edit programs, you can continue reading the rest of this manual at your leisure to learn about the other features and commands of *Z*.

This section is divided into two subsections: the first describes how to create a new C program, and the second how to edit an existing program.

1.1 Creating a new program

Z is activated by entering a command of the form:

```
z hello.c
```

where *hello.c* is the name of the file to be edited. Since we're creating a new program, the file doesn't exist yet, so *Z* says so by displaying a message on its status line (which may be either the first or last line of the display, depending on the system on which Z is running). On systems that use the first display line for status information, the screen then looks like this:

```
"hello.c" no such file or directory
~
~
...
~
~
```

with the cursor on the left-hand column of the second line. On systems that use the last display line for status information, the screen looks like this:

```
~
~
...
~
hello.c doesn't exist
```

with the cursor on the left-hand column of the first line.

Z is now waiting for you to enter a command.

The screen

As mentioned above, *Z* uses the one line of the display for displaying information and for echoing the characters of some commands which are entered. On the Macintosh, the last line is the status line; on other systems, the first line is the status line.

The rest of the lines on the screen are used to display text of the file being edited.

The tilde characters on the screen lines are *Z*'s way of saying that the end of the file has been reached: these characters are not actually in the file.

Modes of Z

Z has two modes: command and insert, which allow you to enter commands and to insert text, respectively.

In this section, we'll spend most of our time in insert mode, using commands only to enter insert mode and to exit *Z*. When we get to the next section, in which we edit a file, we'll discuss more commands.

Insert mode

With *Z* in insert mode, characters that you type are entered into a memory-resident buffer; the characters don't appear in the file until you exit insert mode and explicitly issue a command which causes *Z* to write the buffer to the file.

Z has several commands for entering insert mode; the one we want to use is *i*, which allows text to be entered before the cursor. So type *i*. Notice that *Z* doesn't echo this command on the screen; it only does that for a few commands. Notice also that we are in command mode, as evidenced by the message

<insert mode>

on the right-hand side of the status line.

Now you can enter a program, just as you would on a typewriter. Notice that the cursor is positioned where the next character will be entered. Try entering the "hello world" program:

```
main()
{
   printf("hello, world\n");
}
```

When you hit the <return> key after entering the printf line, the cursor was left positioned on the next line of the screen underneath the first non-white space character of the preceding line. This feature, which is known as "autoindent", is useful when creating C programs, encouraging statements within a compound statement to be indented and lined up. Autoindent can be disabled and enabled, and we'll show you how later.

We want the closing curly brace of the main function to be on the first column of the line, not indented. So type the backspace key to get back to the first column, and then type the '}' key.

The backspace key can also be used to backspace over characters that you incorrectly type.

When you're done inserting the program, hit the escape key to exit insert mode and return to command mode. The key used as the escape key varies from system to system. On the IBM PC, the key labeled ESC is the escape key. On the TRS-80, models III and 4, the key labeled BREAK is the escape key. And on the Macintosh, the backquote key, `, is the escape key.

Exiting Z

To write the program you've just entered from the text buffer to the disk file *hello.c* and then exit Z, type *ZZ*.

Occasionally you may want to exit Z without writing the text you've entered to a file; in this case, type

:q!

followed by a carriage return, CR.

1.2 Editing an Existing File

In this section we're going to present a few commands which will allow you to make changes to an existing file.

Starting and stopping Z

You get in and out of *Z* when editing an existing file just as you do when creating a new file. To start *Z*, enter

```
z hello.c
```

where hello.c is the name of the file to be edited. And to stop *Z* and save the changes you've made, put *Z* in command mode and enter:

```
ZZ
```

Z knows if you made changes to the original text or not; if you did, it saves the original file by changing the extension of its name to *.bak* and then writes the modified text to a new file having the specified name. If a *.bak* file with that name already exists, it will be deleted before the rename occurs.

If you didn't make any changes, the *ZZ* command causes *Z* to halt without changing any disk files.

The command *:q!* will cause *Z* to halt without writing anything to the file being edited.

Going back to the startup of *Z*, *Z* reads the specified file into the text buffer, displays the first screenful of the file's text, displays the file's statistics (name, number of lines, number of characters) on the status line, positions the cursor at the first character of the first line, and enters command mode, waiting for you to enter a command.

The cursor

Before describing the commands for viewing and changing the text in *Z*'s memory-resident buffer, we need to discuss the cursor.

In *Z*, the character position in the text which is pointed at by the cursor acts as a reference point: most commands perform an action relative to that position. For example, the *i* command, described in the last section, allows you to enter text before the cursor. And the *x* command, to be discussed, deletes the character at which the cursor is located.

So we will be describing two types of commands in this section: those that move the cursor around in the text, thus bringing different sections of text into view, and those that modify text in the vicinity of the cursor.

Moving around in the text: scrolling

The text you created in section 1, for the "hello, world" program, easily fit on a single screen. But most text files are too large to be

viewed all at once, so we need commands to bring different sections into view.

Two such commands are the "scroll" commands: "scroll down", represented by the character control-D, and "scroll up", represented by control-U. That is, to execute the "scroll down" command, you hold down the control key and then depress the 'D' key.

The key used as the control key differs from system to system. On the IBM PC, it's the key labeled 'Ctrl'. On the Macintosh, it's the cloverleaf key (the key next to the 'Option' key that has the unusual symbol).

In the rest of this manual, we will refer to control characters using notation of the form ^D rather than control-D, for brevity. Thus, the "scroll up" and "scroll down" commands are represented as ^U and ^D, respectively.

A scroll command moves the screen up or down in the file, bringing another half-screen's worth of text into view. It's as if the text was on a reel of tape and the screen is a viewer: scrolling down moves the viewer down the reel, and scrolling up moves the viewer up the reel.

When scrolling, the cursor will be left on the same position within the text after the scroll as before, if that position is still within view. Otherwise, the cursor is moved to a line in the text which was newly brought into view.

Moving around in the text: the 'Go' command

Scrolling is one way to move around in the text, but it's slow. If we have a large text file, to which we want to append text, it would take a long time and many scroll commands to reach the end.

The *go* command, *g*, is one way to move rapidly to the point of interest in the text: entering *g* by itself will move the cursor to the end of the text and, if necessary, redraw the screen with the text which precedes it.

The *g* command can also be preceded by the number of the line of interest; in this case, the cursor is moved to the beginning of that line. So to move back to the first line of text, enter:

```
1g
```

The *g* command can be used to move to any line within the text, but since you usually don't know the numbers of the lines, the *g* command is mainly used to move to the beginning and end of the text.

Moving around in the text: string searching

So, scrolling allows us to take a casual stroll through text, and the *g* command to move rapidly to the beginning and end of the file. What

we need is a command to rapidly move to a specific point in the middle of the text.

The "string search" command, /, is such a command. When you enter /, followed by the string of interest, followed by a carriage return, *Z* searches forward in the text from the cursor position, looking for the string If *Z* reaches the end of the text without finding the string, it will "wrap around", and continue searching from the beginning of the text.

If the string is found, the cursor is positioned at its first character and, if necessary, the screen is redrawn with its surrounding text.

If the string isn't found, a message saying so is displayed on the status line of the screen and the cursor isn't moved.

While the "string search" command and its string are being entered, the characters are displayed on the status line, and normal editing operations can be used, such as backspacing over mistyped characters.

Z remembers the last string searched for. To repeat the search, enter the "find next string" command, *n*.

Finely tuned moves

With the commands presented up to now, you can move to the area of interest in the text. The next few paragraphs present commands which move the cursor from somewhere within the area of interest to a specific character position, from which changes will be made.

Some commands for this, from the many available in *Z*, are:

- and *CR* (carriage return)
  Move the cursor up and down one line, respectively, to the first non-whitespace character on the line;

space and backspace
  Move the cursor right and left, respectively, on the line on which the cursor is located.

These commands can be preceded by a number, which cause the command to be performed the specified number of times. For example,

```
3-
```

moves the cursor up three lines, and

```
5<space>
```

moves the cursor right five characters. Note that <space> represents the space bar.

Deleting text

You now have a repetoire of commands which allow you to move the cursor fairly quickly to any location in a text file. We're ready to

move on to a few commands for modifying the text.

Two such commands, for deleting text, are "delete character", *x*, and "delete line", *dd*:

x	Deletes the character under the cursor;
dd	Deletes the entire line on which the cursor is located.

Each of these commands can be preceded by a number, causing the command to be repeated the specified number of times. For example,

```
2x
```

deletes two characters, and

```
3dd
```

deletes three lines.

More insert commands

You already know one command for inserting text: *i*, which allows text to be inserted before the cursor. We need a few more insert commands:

a	Enters insert mode such that text is inserted following the cursor;
o	Creates a blank line below the current line (ie, the line on which the cursor is located), moves the cursor to the new line, and enters insert mode;
O	Same as *o*, but the new line is above the current line.

Summary

With the set of commands presented in this chapter, you can edit any text file. You should continue reading this manual, to learn more about *Z*, while you use the basic command set for performing your editing chores.

You'll find that *Z* has many more capabilities, which allow you to perform functions more quickly, with fewer keystrokes, than with the basic command set, and which allow you to perform functions which you can't perform with the basic command set.

The commands in the basic set are listed on the next page.

Starting and stopping Z

Z filename	Start Z, and prepare 'filename' for editing
ZZ	Stop Z, and write modified text to the edit file
:q!	Stop Z, without writing anything to the edit file

scrolling

^D	Move down half a screenful
^U	Move up half a screenful

Moving the cursor

g	Move the cursor to the end of the text, or to a specific line
/str	Search for the character string "str" and move the cursor to it
n	Search again, using the same string
-	Move cursor up a line
CR	Move cursor down a line
space	Move cursor right one character
backspace	Move cursor left one character

Inserting text

i	Insert before cursor
a	Insert after cursor
o	Insert new line below current line
O	Insert new line above current line

Deleting text

x	Delete character under cursor
dd	Delete line on which cursor is located

2. More commands

In this section we're going to describe the rest of the features and commands of Z, building and expanding on the information presented in the previous chapter. The section is organized into subsections; some describe a group of related commands, some a particular feature, and some how to perform a specific function with Z.

2.1 Introduction

Before getting into the *Z* commands, we want to discuss in more detail the way that *Z* displays information on the screen and the way that commands are entered.

2.1.1 The screen

We've already discussed the basic details on *Z*'s use of the screen. There's just a few more things to discuss: the display of unprintable characters and the display of lines which don't fit on the screen.

2.1.1.1 Displaying unprintable characters

A file edited by *Z* can contain any character whose ASCII value in decimal is less than 128, including unprintable characters, such as SOH, LF, and ESC. *Z* displays unprintable characters as two characters; the first is ^, and the second is the character whose ASCII value equals that of the character itself plus 0x40. For example, the unprintable character SOH is displayed as the pair of characters ^*A*, since the ASCII value of SOH is 1, and 1 plus 0x40 is 0x41, which is the ASCII value for the character 'A'.

2.1.1.2 Displaying lines that don't fit on the screen

In the previous chapter we said that lines beyond the end of the file are displayed with the character ~ in the first column of the line on the screen. When you see the ~ character in the leftmost column of a line on the screen, this usually signifies that this line of the display doesn't contain a line of text. Lines which don't fit on the screen are displayed by *Z* in a similar manner, as you'll soon see.

Z allows lines to be entered which are longer than a screen line. Normally, *Z* simply displays such lines on several screen lines. In some cases, however, the entire line won't fit on the screen. For example, if the cursor is positioned at the beginning of the file, it may not be possible to display the text of an entire big line at the bottom of the screen. In this case, *Z* displays an @ character in the first column of the screen lines on which the text would be displayed.

Thus, when you see the @ character in the leftmost column of a line on the screen, this usually signifies that the text which would have appeared on this line of the screen was too big, and not that the @ character is in the text.

2.1.2 Commands

When most commands are entered, *Z* doesn't echo the characters on the screen. For some commands, however, it does. In this latter category are the commands whose first character begins with *:* and with the string search commands.

For these commands, the characters are displayed on the screen's status line, and can be backspaced over and reentered, if necessary.

Also, *Z* doesn't act on such commands until you type the carriage return key, CR.

2.1.3 Special Keys

There are two keys that have special meaning for *Z*: the escape key, which is used to exit insert mode, and the control key, which is used in conjunction with another key to generate control characters. The actual keys used for these functions varies from system to system, as mentioned in the previous chapter.

The escape key is ESC on the IBM PC. On the TRS-80, models III and 4, it's the BREAK key. And on the Macintosh, it's the backquote key, `.

The control key is 'Ctrl' on the IBM PC. On theMacintosh it's the key next to the 'Option' key that has the cloverleaf symbol.

On the Macintosh, there are times when you want to generate a backquote, and not escape. For example, backquote is a cursor motion command to *Z*. To generate backquote, hold down the control key (the key next to the option key), and then type backquote.

2.2 Paging and Scrolling

In the last chapter we described commands for scrolling through text, ^*U* and ^*D*. Another pair of commands allow you to page, instead of scroll, through text. They are ^*B* and ^*F*, which page backwards and forwards, respectively.

A page command brings the previous or next screenful of text into view by redrawing the screen with the new text. Whereas scrolling was described as a viewer moving over a reel of tape, paging can be described as the turning of pages of a book.

Paging moves you through text more quickly than scrolling does. However, since paging redraws the screen all at once, while scrolling changes it gradually, it's often more difficult to keep a sense of continuity when paging than when scrolling. As an aid to continuity when paging, two lines of text which were previously in view are still in view after paging.

In the discussion of scrolling in the last chapter, we neglected to mention that the scroll commands can be preceded by a value specifying the number of lines to be scrolled up or down. If a number isn't specified, the last scroll value entered is used; if a scroll value was never entered, it defaults to half a screen's worth of lines. Separate values are maintained for scrolling up and for scrolling down.

The scrolling and paging commands necessarily move the cursor within the text, but they can't be used to home the cursor to an exact position at which changes are to be made. For this, you'll have to use commands described in subsequent sections.

2.3 Searching for strings

In the previous chapter, we described the string search command, /, which causes Z to scan forward, looking for the string. In this section, we describe the rest of the searching capabilities of Z. First, the rest of the string searching commands are described; then, the capability of Z to match patterns called "regular expressions", of which specific character strings are a special case, is described.

2.3.1 The other string-searching commands

The other string-searching commands are:

? Behaves like /, but Z finds the previous occurence of the string rather than the next;

n Repeats the last string-search command;

N Repeats the last string-search command, but in the opposite direction;

:se ws=0 and :se ws=1
Turns the wrap scan option off or on, respectively.

When Z reaches the end or beginning of text without finding the string of interest, it normally "wraps around" to the opposite end of the text and continues the search. It does this because by default the "wrap scan" option is on. This option can be disabled by entering the "set option" command:

:se ws=0

thus causing the search to end when it reaches the end of text. The option can be reenabled by entering:

:se ws=1

Note that for this colon command, as for all colon commands, carriage return must be typed before the command is executed.

2.3.2 Regular expressions

The string you tell Z to search for is actually a "regular expression". A regular expression is a pattern which is matched to character strings. The pattern can define a specific sequence of characters which comprise the string; in this case, only that specific string matches the pattern. The pattern can also contain special characters which match a class of characters; in this case, the pattern can match any of a number of character strings.

For example, one such special construct is square brackets surrounding a character string; this matches any character in the enclosed string. So the regular expression

ab[xyz]cd

matches the strings

abxcd
abycd
abzcd

Another special character is *, which matches any number of occurences of the preceding pattern. For example, the regular expression

ab*c

matches many strings, including

ac
abc
abbc

and so on. And the pattern

ab[xyz]*cd

matches many strings, including:

abcd
abxcd
abxycd
abzzxcd

and so on.

The complete list of special characters and constructs that can be included in regular expressions is:

^	When the first character of a pattern, it matches the beginning of the line
$	When the last character of a pattern, it matches the end of the line;
.	Matches any single character;
<	Matches the beginning of a word;
>	Matches the end of a word;
[str]	Matches any single character in the enclosed string;
[^str]	Matches any single character *not* in the enclosed string;
[x-y]	Matches any character between x and y;
*	Matches any number of occurences of the preceding pattern.

2.3.3 Disabling extended pattern matching

The "magic" option enables and disables the extended pattern matching capability. To turn off this option, enter:

:se ma=0

And to turn it on, enter:

:se ma=1

By default, extended pattern matching is disabled.

With the magic option off, only the characters ^ and $ are special in patterns.

2.4 Local Moves

In this section we're going to present more commands for moving the cursor fairly short distances; up or down a few lines, along the line on which it's located, and so on. We've already presented several, namely CR (carriage return), space, and backspace; but there are many more, reflecting the importance of finely-tuned, quickly-executed movements to the user.

2.4.1 Moving around on the screen:

Here are some commands for moving the cursor short distances:

h	Moves to the left one character;
j	Moves down one line, leaving the cursor in the same column;
k	Moves up one line, leaving the cursor in the same column;
l	Moves right one cursor;

The keys ^H, LF, ^K, and ^L are synonyms for h,j,k, and l, respectively.

These commands can be preceded by a number, which specifies the number of times the command is to be repeated.

Z has commands for moving the cursor to the top, middle, and bottom of the screen; they are H, M, and L, respectively. The cursor is positioned at the beginning of the line to which it's moved.

Remember the - command, which moved the cursor up a line, to the first non-whitespace character? As you might expect, + will move the cursor down a line, to the first non-whitespace character. + is thus equivalent to CR, the command presented in the last chapter.

2.4.2 Moving within a line

We've already presented several commands for moving the cursor around within the line on which it's located:

h, ^H, backspace	Left one character;
l, ^L, space	Right one character;

Here are a few more:

^	Moves the cursor to the first non-whitespace character on the line;
0	Moves to the first character on the line;
$	Moves to the last character on the line;

A few commands fetch another character from the keyboard, search for that character, beginning at the current cursor location, and leave the cursor near the character:

f	Scan forward, looking for the character, and leave the

cursor on it;

t Same as *f*, but leave the cursor on the character preceding the found character;

F Same as f, but scan backwards;

T Same as t, but scan backwards.

; Repeat the last f, t, F, or T command;

, Repeat the last f, t, F, or T command in the opposite direction.

Finally, the command | moves the cursor to the column whose number precedes the command. For example, the following command moves the cursor to column 56 on the current line:

```
56|
```

2.4.3 Word movements

Z has several commands for moving the cursor to the beginning or end of a word which is near the cursor:

w Moves to the beginning of the next word;

b Moves to the beginning of the previous word;

e Moves to the end of the current word.

For the preceding commands, a "word" is defined in the normal way: a string of alphabetical and numerical characters surrounded by whitespace or punctuation. There is a variant of each of these commands, differing only in the definition of a "word": they think that a word is any string of non-whitespace characters surrounded by whitespace. The variant of each of these commands is identified by the same letter, but in upper case instead of lower:

W Moves to the beginning of the next big word;

B Moves to the beginning of the previous big word;

E To the end of the current big word.

Each of these commands can be preceded by a number, specifying the number of times the command is to be repeated. For example,

```
5w
```

moves forward five words.

The word movement commands will cross line boundaries, if necessary, to find the word they're looking for.

2.4.4 Moves within C programs

Z has several commands for moving the cursor within C programs:

]] and [[Move to the opening curly brace, {, of the next or previous function, respectively;

% Move to the parenthesis, square bracket, or curly bracket which matches the one on which the cursor is currently located;

{ and } Move to the preceding or next blank line.

The [[and]] commands assume that the opening and closing curly braces for a function are in the first column of a line, and that all other curly braces are indented.

As an example of the '%' command, given the statement:

```
while (((a = getchar()) != EOF) && (c != 'a'))
```

with the cursor on the parenthesis immediately following the 'while', the % command will move the cursor to the last closing parenthesis on the line.

2.4.5 Marking and returning

Z has commands which allow you to set markers in the text and later return to a marker. Twenty six markers are available, identified by the alphabetical letters.

Unlike the other commands described in this section, these commands are not limited to moves within the current area of the cursor - they can move the cursor anywhere within the text.

A marker is set at the current cursor location using the command

*m*x

where x is the letter with which you want to mark the location.

There are two commands for returning to a marked position:

`x Moves the cursor to the location marked with the letter 'x';

'x Moves the cursor to the first non-whitespace character on the line containing the 'x' marker.

Remember, to generate backquote on the Macintosh, hold down the control key and then type backquote.

Occasionally, you may accidently move the cursor far from the desired position. There are two single quote commands for returning you to the area from which you moved:

`` Returns the cursor to its exact starting point;

'' Returns the cursor to the first non-whitespace character on the line from which the cursor was moved.

For example, if the cursor is on the line:

```
if (a >= 'm' && a <= 'z')
```

at the character '<', then following a command which moves the cursor far away, the command `` will return the cursor to the '<' character, and the command '' will return it to the beginning of the word 'if'.

2.4.6 Adjusting the screen

The *z* command is used to redraw the screen, with a certain line at the top, middle, or bottom of the screen.

To use it, place the cursor on the desired line, then enter the *Z* command, followed by one of these characters:

CR	To place the line at the top of the screen;
.	To place it in the middle of the screen;
-	To place it at the bottom.

The *z* command isn't a true cursor motion command, because the cursor is in the same position in the text after the command as before.

2.5 Making changes

That concludes the presentation of cursor movement commands. The next several sections describe commands for making changes to the text.

2.5.1 Small changes

In this section we present commands for making small changes. We've already presented two such commands in the previous chapter:

x	Which deletes the character at which the cursor is located;
dd	Which deletes the line at which the cursor is located.

The other commands are:

X	Delete the character which precedes the cursor; can be preceded by a count of the number of characters to be deleted;
D	Delete the rest of the line, starting at the cursor position;
rx	Replace the character at the cursor with 'x';
R	Start overlaying characters, beginning at the cursor. Type the escape key to terminate the command. (Remember, this key differs from system to system);
s	Delete the character at the cursor and enter insert mode; when preceded by a number, that number of characters are deleted before entering insert mode;
S	Delete the line at the cursor and enter insert mode; when preceded by a count, that number of lines are deleted before entering insert mode;
C	Delete the rest of the line, beginning at the cursor, and enter insert mode;
J	Join the line on which the cursor is positioned with the following line; when preceded by a count, that many lines are joined.

2.5.2 Operators for deleting and changing text

Z has a small number of commands, called 'operators', for modifying text. They all have the same form, consisting of a single letter command, optionally preceded by a count and always followed by a cursor motion command. The count specifies the number of times the command is to be executed. The command affects the text from the current cursor position to the destination of the cursor motion command, if the starting and ending position of the cursor are on the same line. If these positions are on different lines, the command affects all lines between and including the lines which contain the starting position and ending positions.

In this section, we're going to describe the operators for deleting and changing text, *d* and *c*:

d	Deletes text as defined by the cursor motion command;
c	Same as *d*, but *Z* enters insert mode following the deletion.

For example,

dw	Deletes text from the current cursor location to the beginning of the next word;
3dw	Deletes text from the cursor to the beginning of the third word;
d3w	Same as '3dw';
db	Deletes text from the current to the beginning of the previous word;
d`a	Deletes text from the cursor to the marker 'a', if the marker and the starting cursor position are on the same line. Otherwise, deletes lines from that on which the cursor is located through that on which the marker is located; On the Macintosh, generate backquote by holding down the control key and then typing the backquote key.
d/var	Deletes text either from the cursor to the string "var" or between the lines at which the cursor is currently located and that on which the string is located.
d$	Deletes the rest of the characters on the line, and hence is equivalent to *D*.

2.5.3 Deleting and changing lines

In the last chapter, we presented a command for deleting lines: *dd*. As you can see now, this is a special form of the *d* command, because the character following the first *d* is not a cursor motion command.

For all the operator commands, typing the command character twice will affect whole lines. Thus, typing *cc* will clear the line on which the cursor is located and enter insert mode. Preceding *cc* with a number will compress the specified number of lines to a single blank line and enter insert mode on that line.

2.5.4 Moving blocks of text

When text is deleted using the *d* or *c* command, it's moved to a buffer called the "unnamed buffer". (There are other buffers available, which have names. More about them later).

Data in the unnamed buffer can be copied into the main text buffer using one of the "put" commands:

p	Copies the unnamed buffer into the main text buffer, after the cursor;

P Same as *p*, but the text is placed before the cursor.

Thus, the delete and put commands together provide a convenient way to move blocks of text within a file.

The contents of the unnamed buffer is very volatile: when any command is issued that modifies the text, the text which was modified is placed in the unnamed buffer. This is done so that the modification can be 'undone', if necessary, using one of the 'undo' commands. For example, if you delete a character using the *x* command, the deleted character is placed in the unnamed buffer, replacing whatever was in there. So you have to be careful when moving text via the unnamed buffer: if you delete text into the unnamed buffer, expecting to put it back somewhere, then issue another command which modifies the text before issuing the put command, the deleted text is no longer in the unnamed buffer.

As you'll see, the named buffers can also be used to move blocks of text, and their contents are not volatile.

2.5.5 Duplicating blocks of text: the 'yank' operator

The 'yank' operator, *y*, copies text into the unnamed buffer without first deleting it from the main text buffer. When used with the 'put' command, it thus provides a convenient way for duplicating a block of text.

Since *y* is an operator, it has the same form as the other operators: an optional count, followed by the *y* command, followed by a cursor motion command. The command yanks the text from the cursor position to the destination of the cursor motion command, if the starting and ending positions are on the same line. If they are on separate lines, a whole number of lines are yanked, from that on which the cursor is currently located through that to which the cursor would be moved by the cursor motion command. The text is yanked into the unnamed buffer.

For example,

yw Copies text from the cursor to the next word into the unnamed buffer;

y3w Copies text from the cursor to the beginning of the third word;

3yw Same as 'y3w';

y'a Copies text from the cursor location to the marker 'a' into the unnamed buffer, if the two positions are on the same line. Otherwise, copies entire lines between and including those containing the two positions.

As a special case, the command *yy* will yank a specified number of whole lines. The command *Y* is a synonym for *yy*. For example,

yy Yanks the line at which the cursor is located;

3Y Yanks three lines, beginning with the one on which the cursor is located.

2.5.6 Named buffers

In addition to the unnamed buffers, *Z* has twenty six named buffers, each identified by a letter of the alphabet, which can used for rearranging text. Text can be deleted or yanked into a named buffer and put from it back into the main text buffer.

The advantage of these buffers over the unnamed buffer in rearranging text is that their contents are not volatile: when you put something in a named buffer, it stays there, and won't be overwritten unexpectedly. Also, as you'll see, the named buffers can be used to move text from one file to another.

To yank text into a named buffer, use the yank operator, preceded by a double quote and the buffer name, and followed by a cursor motion command. For example, the following will yank three words into the 'a' buffer:

"ay3w

and the following yanks four lines into the 'b' buffer, beginning with the line on which the cursor is located:

"b4yy

Text is deleted into a named buffer in the same way: the delete command is used, preceded by a double quote and the buffer name. For example, to delete characters from the cursor to the 'a' marker into the 'h' buffer:

"hd`a

The preceding command, when the source and destination cursor positions are on separate lines, will delete a number of whole lines into the 'h' buffer, from that on which the cursor is initially located through that containing the destination position.

As you remember, on the Macintosh, the backquote key is interpreted as the escape key. To generate a real backquote for use in the preceding example, you must hold down the control key (the key with the strange symbol next to the option key) and then type backquote.

To delete ten lines into the 'c' buffer:

"c10dd

Text in a named buffer is put back into the main text using the 'put' commands *p* and *P*, preceded by a double quote and the buffer name. For example:

ap Puts text from the 'a' buffer, after the cursor;

zP Puts text from the 'z' buffer, before the cursor.

2.5.7 Moving text between files

The named buffers are conveniently used to move text from one file to another. First yank or delete text from one file into a named buffer; then switch and begin editing the target file, using the *:e* command:

:e filename

(More on this later). Then move the cursor to the desired position; then put text from the named buffer.

This technique only works when using named buffers, not with the unnamed buffer. When switching to a new file, the unnamed buffer is cleared, but the named buffers are not.

2.5.8 Shifting text

The last two operator commands to introduce are the 'shift' operators, < and >, which are used to shift text left and right a tabwidth, respectively.

For example,

>/str

shifts right one tab width the lines from that on which the cursor is located through that containing the string "str".

Following the standard operator syntax, repeating the shift operator twice affects a number of whole lines:

5<< Shifts five lines left;
>> Shifts one line right.

2.5.9 Undoing and redoing changes

Z remembers the last change you made, and has a command, *u*, which undoes it, restoring the text to its original state.

Z also remembers all the changes which were made to the last line which was modified. Another 'undo' command, *U*, undoes all changes made to that line.

Finally, the period command, '.', reexecutes the last command that modified text.

2.6 Inserting text

We've already presented most of the commands for entering insert mode:

a	Append after cursor;
i	Insert before cursor;
o	Open new line below cursor;
O	Open new line above cursor;
C	Delete to end of line, then enter insert mode;
s	Delete characters, then enter insert mode;
S	Delete lines, then enter insert mode;

In this section we want to present the remaining few commands for entering insert mode, and present some other features of insert mode.

2.6.1 Additional insert commands

The other commands for entering insert mode are:

A	Append characters at the end of the line on which the cursor is located. This is equivalent to *$a*;
I	Insert before the first non-whitespace character on the current line. This is equivalent to *^i*.

2.6.2 Insert mode commands

Some editing can be done on text entered during insert mode, using the following control characters:

backspace	Delete the last character entered;
^H	Same as 'backspace' character;
^D	Same as "backspace';
^X	Erase to beginning of insert on current line;
^V	Enter next character into text without attempting to interpret it.

^V is used to enter non-printing characters into the text. For example, to enter the character 'control-A' into the text, type

^V^A

That is, hold down the control key, then type the 'V' key, then the 'A' key, then release the control key. As mentioned earlier, non-printing characters are displayed as two characters: '^' followed by a character whose ASCII code equals that of the non- printing character plus 0x40.

2.6.3 Autoindent

The *Z* 'autoindent' option is useful when entering C programs. When you are in insert mode and type the 'carriage return' key, with the autoindent option enabled, the cursor will be automatically indented on the new line to the same column on which the first non-whitespace character appeared on the previous line. This feature is useful for editing C programs because it encourages statements which

are part of the same compound statement to be indented the same amount, thus making the program more readable.

Z autoindents a line by inserting tab and space characters at the beginning of a new line. If you don't want to be indented that much, you can backspace over these automatically inserted tabs and spaces until you reach the desired degree of indentation.

The autoindent option can be selectively enabled and disabled using the 'set options' command:

:se ai=0 to disable autoindent
:se ai=1 to enable autoindent

When *Z* is activated, autoindent is enabled.

2.7 Macros

Z allows you to define a sequence of commands, called a 'macro', and then execute the macro one or more times.

When a macro is defined to *Z*, it's placed in a special buffer, called the macro buffer, and then executed once. There are two ways to define a macro to *Z*: immediately and indirectly.

2.7.1 Immediate macro definition

An 'immediate' macro definition is initiated by typing the characters

```
:>
```

Z responds by clearing the status line, displaying these characters on the line, and waiting for you to enter the sequence of commands.

As you enter the commands, *Z* displays them on the status line and enters them immediately into the macro buffer; that's why it's called 'immediate macro definition'.

If you make a mistake while entering commands, you can simply backspace and enter the correct characters.

To terminate the definition, type the carriage return key. *Z* will then execute the sequence of commands in the macro buffer. The contents of this buffer are not altered by executing the macro, so you can reexecute the macro without reentering it, as described below.

2.7.2 Some examples

The following macro advances the cursor one line, and deletes the first word on the new line:

```
+dw
```

contains two commands: +, which advances the cursor, and *dw*, which deletes the word beneath the cursor.

The next macro moves the cursor to the previous line and deletes the last character on the line:

```
-$x
```

It contains three commands: -, which moves the cursor to the previous line; *$*, which moves the cursor to the last character on that line; and *x*, which deletes the character beneath the cursor.

You can also insert text using a macro. You enter insert mode using one of the normal insert commands. The characters which follow the insert command on the macro line, up to a terminating escape character, are then inserted into the text. The escape character causes *Z* to return to command mode and continue executing commands in the macro which follow the insert command.

Remember, the key used as the escape character differs from system to system. See section 1 of this chapter for details.

For example, the following macro advances the cursor to the next line, deletes the second word on the line, inserts the character string "and furthermore", and deletes the last word on the line:

```
+wdwiand furthermore<ESC>$bdw
```

The last macro contains the following commands:

+	Advances the cursor to the next line;
w	Moves the cursor to the second word on the line;
dw	Deletes the word beneath the cursor;
iand furthermore<ESC>	Inserts the text "and furthermore". <ESC> stands for the escape key;
$	Moves the cursor to the last character on the line;
b	Moves the cursor to the beginning of the last word on the line;
dw	Deletes that word.

Z also allows you to search for a string from within a macro. Enter in the macro the 'string search' command (for example, /), followed by the string, followed by the ESC character. For example, the following macro moves the cursor to the word "Ralph" and deletes it:

```
/Ralph<ESC>dw
```

It contains the commands

/Ralph<ESC>	Moves the cursor to "Ralph". <ESC> stands for the escape key;
dw	Deletes "Ralph".

The following macro finds "Ralph" and replaces it with "Sarah":

```
/Ralph<ESC>cwSarah<ESC>
```

It contains the commands:

/Ralph<ESC>	Moves the cursor to "Ralph";
cwSarah<ESC>	Changes "Ralph" to "Sarah".

2.7.3 Indirect macro definition

The other way of defining a macro is to yank a line containing a sequence of commands from the main text buffer into a named buffer and then have *Z* move the contents of the named buffer to the macro buffer.

Commands for indirect macro definition are:

@x	Causes *Z* to move the contents of the 'x' buffer to the macro buffer and then execute it once;
xv	A synonym for '@x'.

Indirect macro definition of macros has several advantages over immediate definition: for one, if a macro defined immediately is incorrect, you have to reenter the entire macro. With an indirectly defined macro, you can edit the macro definition in the main text buffer and then move it back to the macro buffer.

Another advantage is that you can store several macros in the named buffers and easily reexecute a macro, without having to reenter it. With immediate definition, when a new macro is defined, the previously defined macro is lost, and must be reentered to be reexecuted.

One difference between entering macros immediately and via the text buffer and named buffer concerns the method for specifying the end of a search string and for exiting insert mode. With immediate definition, you do this by typing the ESC key directly. For indirect definition, in which the macro is first entered into the main text buffer, typing the ESC key would cause *Z* to exit insert mode, not to enter the ESC key into to text of the macro. In this case, you enter the ESC key by first typing control-V, then ESC. This causes *Z* to enter the ESC character into the text of the macro and remain in insert mode.

2.7.4 Re-executing macros

Once a macro is defined and is in the macro buffer, it can be re-executed by typing one of the commands:

@@
v

Preceding the command with a count will cause the macro to be executed the specified number of times.

2.7.5 Wrapping around during macro execution

While executing a macro, *Z* may reach the beginning or end of the text, and want to continue beyond that point. This is especially true when reexecuting macros. The 'macro wrap' option, *wm*, specifies whether *Z* should terminate the macro execution at that point, or continue at the opposite end of the text.

This option is enabled and disabled using the 'set options' command:

:se wm=0	To disable macro wrapping;
:se wm=1	To enable it.

When *Z* starts, this option is enabled.

2.8 The Ex-like commands

The 'substitute' and 'repeat last substitution' commands are part of a set of commands that are being added to the Z editor and that are similar to commands in the UNIX Ex editor. In this section we will first generally describe the syntax of these commands, then the 'substitute' command, and finally the 'repeat last substitution' command.

The Ex-like commands consist of a leading colon, followed by zero, one, or two addresses identifying the lines to be affected by the command, followed by a single-letter command, followed by command parameters, and terminated by a carriage return. Most commands have a default set of lines that they affect, thus frequently allowing you to enter commands without explicitly specifying a range.

These commands support regular expressions, as defined in the Z documentation, for identifying addresses and strings to be searched for.

2.8.1 Addresses in Ex commands

An address can be one of the following:

* A period, ., addresses the current line; that is, the line on which the cursor is located.
* The character $ addresses the last line in the edit buffer.
* A decimal number *n* addresses the *n*-th line in the edit buffer.
* 'x addreses the line marked with the mark name *x*. Lines are marked with the *m* command.
* A regular expression surrounded by slashes (/) addresses the first line containing a string that matches the regular expression. The search begins with the line following the current line and continues towards the end of the edit buffer. If a line isn't found when the end of the buffer is reached, and if Z's *ws* option is set to 1 (ie, by the *:se ws=1* command) the search continues at the beginning of the buffer, stopping when the current line is reached.
* A regular expression surrounded by question marks (?) also addresses the first line containing a string that matches the regular expression. But in this case, the search begins with the line preceding the current line in the edit buffer and continues towards the beginning of the buffer. If a line isn't found when the beginning of the buffer is reached, and if Z's *ws* option is set to 1 (ie, by the *:se ws=1* command) the search continues at the end of the buffer, stopping when the current line is reached.
* An address followed by a plus or a minus sign, which in turn is followed by a decimal number *n* addresses the *n*-th line

following or preceding the line identified by the address.

When two addresses are entered to define the range of lines affected by a command, the addresses are usually separated by a comma. They can also be separated by a semicolon; in this latter case, the current line is set to the line defined by the first address, and then the line corresponding to the second address is located.

When no value is specified for the first address in an address range, it's assumed to be the current line or the first line in the buffer, depending on whether the second address was preceded with a comma or a semicolon. When no value is specified for the second address in an address range, it's assumed to be the last line in the buffer. Thus, if neither the beginning nor the ending address of a range is specified, the range consists of either all the lines in the buffer or the lines from the current through the last line in the buffer, depending on whether comma or semicolon is used to separate the unspecified addresses.

2.8.2 The 'substitute' command

The 'substitute' command has the following form:

:[range]s/pat/rep/[options]

where square brackets surround a parameter to indicate that the parameter is optional.

Z searches the lines specified by *range* for strings that match the regular expression *pat*, replacing them with the *rep* string. If *range* isn't specified, just the current line is searched. When the command is completed, the cursor is left on the character following the last replaced string.

Normally, Z automatically replaces a string that matches *pat*. Specifying *c* as an *option* causes Z instead to pause when it finds a matching string, ask if you want the string to be replaced, and make the replacement only if you give your permission.

Normally, Z will replace only the first *pat*-matching string on a line. Specifying *g* as an option causes Z instead to replace all matching strings on a line; in this case, after Z replaces a string on a line, it continues searching for more strings on the line at the character following the replaced string.

An ampersand (&) in the replacement string *rep* is replaced by the string that matched *pat*. The special meaning of *&* can be suppressed by preceding it with a backslash, \.

A replacement string consisting of just the percent character (%) is replaced in the current substitution by the replacement string that was used in the last substitution. The special meaning of % can be suppressed by preceding it with a backslash, \.

2.8.2.1 Examples

:s/abc/def/

Search the line on which the cursor is located for the string *abc*; if found, replace it with the string *def*.

:1,$s/ab*c/xyz/

Search all lines in the edit buffer for strings that begin with *a*, end in *c*, and have zero or more *b*'s in between; replace such strings with *xyz*. On any given line, only the first occurrence of a string that matches the pattern is replaced.

:/{/;/}/s/for/while/c

Find the first line following the current line that contains a {; then find the first line following this line that contains a }. In the lines between and including these lines, search for the string *for*; for each such string, ask if it should be replaced; if yes, replace it with *while*.

2.8.3 The '&' (repeat last substitution) command

The & command has the form

:[range]&

where brackets indicate that the parameters are optional.

The & command causes the last 'substitute' command to be executed again, using the same search pattern, replacement string, and options as were used in the previous command. The command searches the lines that are specified in the & command's *range*; if *range* isn't specified, the substitution is performed on just the current line.

2.9 Starting and stopping Z

You already know how to start and stop *Z*, from the previous chapter. In this section we present more information related to the starting and stopping of *Z*.

2.9.1 Starting Z

In the previous chapter, we said that *Z* was started by specifying the name of the file to be edited on the command line:

```
Z filename
```

Z can also be started without specifying a file name or by specifying a list of files to be edited.

2.9.1.1 Starting Z without a filename

When *Z* is started without a filename being specified, you will normally tell *Z* the name of the file to be edited, once it's active, using the *:e* command:

```
:e filename
```

It isn't absolutely necessary for *Z* to know the name of the file you're editing: *Z* will allow you to create and modify text in the text buffer without knowing the name of the file to which you intend to write the text. But then you'll have to explicitly tell *Z* to write the text, using the command

```
:w filename
```

Z can't automatically write the text, since it doesn't know which file you're editing.

2.9.1.2 Starting Z with a list of files

Z can be started and passed a list of names of files to be edited, as follows:

```
Z file1 file2 ...
```

Z will remember the list, and make the first file in the list the 'edit file'; that is, read the file into the main text buffer and allow it to be edited.

Z has a command, *:n*, which will make the next file in the list the edit file, after writing the contents of the text buffer back to the current edit file.

File lists are discussed in more detail below.

2.9.1.3 The options file

Z has several options for controlling its operation in different situations. You've already met most of them, including the 'autoindent', 'macro wrap', and other options. The complete list of

options will be presented later. In this section, we want to present another feature of *Z* related to options; the ability to set options automatically, when *Z* is started.

When *Z* starts, it will read options from the file named 'z.opt', if it exists. *Z* looks for the file in different places on different systems.

On PCDOS and on the Macintosh, the environment variable ZOPT defines the name of the options file. If this variable doesn't exist, or if the file isn't found there, *Z* then looks for the file *z.opt* on the current directory on the default drive.

Each line in the options file defines the value of one option, with a statement of the form

```
opt=val
```

where 'opt' is the name of the option, and 'val' is its value. For example, the following sets the 'tab width' option to 8 characters:

```
ts=8
```

2.9.1.4 Setting options for a file

When *Z* makes a file the 'edit file' by reading it into the edit buffer, the file itself can specify the options to be in effect during its edit session. This feature is most useful in editing files which have different tab settings.

A file specifies option values by including strings of the form

```
:opt=val
```

in the first ten lines of the file. For example, the following line could be used near the front of a C program, causing a tab width of 8 characters to be used:

```
/* :ts=8   */
```

When *Z* starts editing a file, the tab width is set back to the default

value, 4 characters, before the file is scanned for option settings.

2.9.2 Stopping Z

In the preceding chapter we presented the following commands for stopping *Z*:

ZZ	If the file's text in the edit buffer has been modified, the text is written to the file, after changing the extension of the original file to ".bak".
:q!	Stops *Z* without writing the text to the file.

Two other commands for exiting *Z* are:

:wq	Which is the name as ZZ, except that the text in the main text buffer is always written to the file, even if no changes have been made;
:q	Which conditionally stops *Z*. If no changes were made to the file's text, *Z* stops; otherwise, it displays a message and remains active.

2.10 Accessing files

Z has other commands for accessing files besides ZZ and :wq, and we're going to discuss them in this chapter.

Z usually knows the name of the file you are editing, and in the sections that follow we will call this the 'edit file'. *Z* makes use of this knowledge, allowing you to write to the edit file without specifying it by name. For example, the ZZ command writes text to the edit file without requiring you to enter the name of the file.

Some commands allow you to access files without redefining *Z*'s idea of the edit file. The commands described in the next two subsections fall into this category.

Other commands cause *Z* to terminate editing of one file and begin editing another; this new file becomes the edit file. The commands described in the other sections of this section are of this type.

2.10.1 File names

In the *Z* commands that require a file name, the name is usually entered using the standard system conventions. However, some characters are special to *Z*:

#	Refers to the last edit file;
%	Refers to the current edit file;
\	Causes the next character to be used in the filename and not be interpreted.

To enter a file name which contains these characters, precede the special character with the character '\'. For example, on PCDOS, to edit the file

```
a:\subs\hello.c
```

use the command

```
:e \\subs\\hello.c
```

On PCDOS, the '/' character can also be used as a separator between directories and between a directory and file name. Thus, the above command could also be entered as:

```
:e /subs/hello.c
```

2.10.2 Writing files

The command *:w* writes the contents of the main text buffer to a file, without redefining the identity of the current edit file. It has the following forms:

:w	Write to the current edit file;
:w	Write to the specified file;
:w!	Same as ':w filename', but the file is overwritten if it exists.

As with all colon commands, carriage return must be typed to cause *Z* to execute the command.

When entered without a filename, *:w* creates a new file having the name of the current edit file and writes the contents of the edit buffer to it. This form of the :w command is commonly used to periodically save text during a long edit session, to guard against system failures.

The option *bk* tells *Z* whether it should save the original edit file before creating a new one. If *bk* is 1 the original will be saved, and if 0 it won't. *Z* saves the original file by changing its name to *.bak*. An existing *.bak* file will be erased before the rename occurs. For details on setting options, see the Options section.

When a filename is entered with the :w command, the text is written to that file, if it doesn't already exist. If it does, nothing is written, and *Z* displays a message on the status line; in this case you must use the *:w!* form of the command to overwrite the file.

The *:w!* command unconditionally writes the text to the specified file, after truncating the file, if it exists, so that nothing is in it. Unlike the *:w* command which doesn't specify a file name, the :w! command doesn't save the original file as a ".bak" file.

2.10.3 Reading files

The command

:r filename

merges one file with a file being edited, without redefining the identity of the edit file.

It reads the contents of the specified file into the main text buffer, inserting the new text following the line on which the cursor is located. It doesn't alter text which is already in the edit buffer.

2.10.4 Editing another file

The following commands cause *Z* to stop editing one file and begin editing another, which thus becomes the 'edit file':

:e	Edit the specified file;
:e!	Edit the file, discarding changes to the current edit file;
:e	Reload the current edit file;
:e!	Reload the current edit file, discarding changes;
:e	Re-edit the previous edit file;
^^	Synonym for ':e #'. (the command is 'control-^').

Z begins editing another file by erasing the contents of the main text buffer and the unnamed buffer, resetting the tab width to four characters, redrawing the display with the first screenful of lines from the file, and setting the cursor at the first character in the text.

When switching to a new edit file, *Z* doesn't change the contents of the named buffers. Thus, these buffers can be used to hold text which is to be moved from one file to another and to contain commonly used macros.

The command

:e filename

causes the specified file to conditionally become the edit file. The condition is that changes must not have been made to the text of the current edit file since it was last written to disk. If this condition is met, then the switch is made; otherwise, *Z* displays a message on the status line and nothing is changed: the identity of the edit file is the same, the contents of the edit buffer are not modified, and the options are not changed.

If *Z* doesn't let you switch edit files when you enter

:e filename

and you want to save the changes to the current edit file, enter the sequence:

:w
:e filename

You can unconditionally cause *Z* to begin editing a new file by entering:

:e! filename

In this case, *Z* doesn't care whether or not you made changes to the current edit file since it was last written to disk; it begins editing the new file without changing the previous edit file.

Sometimes the text in the edit file may get hopelessly scrambled, and you want to get a fresh copy of the edit file contents. The command

:e!

specified without a file name will do just that.

Z not only remembers the name of the current edit file you're editing; it remembers the name of the last file you edited as well. *Z* allows you to refer to this name using the character '#' in :e commands, thus providing a quick means to re-edit the previous edit file:

:e #

causes the previous edit file to conditionally become the current edit file, and

:e! #

causes it to unconditionally become the edit file.

The command ^^ (that is, control-^) is a synonym for ':e #'.

Z also remembers the position at which the cursor was located in the previous edit file, and when you begin re-editing this file it sets the cursor back to this position.

2.10.5 File lists

Z's 'file list' feature is convenient to use when you have several files to edit: you pass *Z* a list of the files and begin editing the first one. When you're finished with one file, a command switches to the next file in the list, after automatically saving the changes to the current edit file. An option to the command prevents *Z* from saving changes, and another command "rewinds" the file list so that you're back editing the first file in the list again.

There's two ways to pass the list of files to be edited to *Z*: as parameters to the command that starts *Z*, and as parameters to the ':n' command. In each case, *Z* remembers the list and makes the first file in the list the 'edit file'. For example,

Z file1 file2 file3

starts *Z* and defines the list of files file1, file2, and file3. *Z* makes file1 the edit file; that is, prepares it for editing by reading it into the edit buffer and displaying its first lines.

When *Z* is active, the command

:n file4 file5 file 6

defines a new list of files: file4 file5 and file6. *Z* makes file4 the edit file.

When used without a files list, the ':n' command switches from one file in the list to the next:

:n	Writes the text in the edit buffer to the current edit file before switching;
:n!	Switches without writing anything to the current edit file.

The ':rew' command "rewinds" the file list; that is, makes the first file in the list the edit file. This command behaves like the ':n' command, in that it by default writes changes to the current edit file before rewinding; and when an exclamation mark is appended to the comand, the rewind occurs without writing to the current edit file.

2.10.6 Tags

Z has a feature useful for editing large C programs which contain many functions distributed over several files. With the aid of a cross-

reference file relating 'tags', that is, function names, to the files containing them, you simply tell *Z* the name of the function that you want to edit and *Z* makes the file containing it the edit file by reading it into the edit buffer and positioning the cursor to the function.

The following commands specify the tag of the function to be edited:

:ta tag	Position to the function named 'tag' in the appropriate file, if the current edit file is up to date;
:ta! tag	Same as ':ta tag', but the switch to the new file occurs even if the current edit file isn't up to date.

When using the ':ta' command, the current edit file is considered 'up to date' if the text in the edit buffer hasn't been modified since it was last written to the file. When used without the trailing '!', the ':ta' command won't switch edit files if the current edit file isn't up to date; it'll just display a message on the status line. You can then either write the text in the edit buffer to the file and re-enter the ':ta' command, or immediately enter the ':ta!' command, to switch edit files anyway.

The command

```
^]
```

that is, control-], is convenient when, while editing or viewing one function, you want to edit or examine a function which it calls. You just set the cursor to the name of the called function and enter '^]'; *Z* will make the file containing the called function the edit file, and position the cursor to this function.

For example, while examining the file *crtdvr.c*, you may come across a call to the function *pcdvr*, and want to take a look at it. By positioning the cursor at the beginning of the word 'pcdvr' and typing '^]', *Z* will make the file containing *pcdvr* the edit file and leave the cursor positioned at this function.

2.10.7 The CTAGS utility

The utility program *ctags* creates the cross reference file, *tags*, which relates function names to the file containing them.

ctags is activated by a command of the form

```
ctags file1 file2 ...
```

where file1, ..., are names of files whose functions are to be placed in the cross reference file. A file name can specify a group of files using the character '*'. For example:

```
*.c
```

specifies all files whose extension is ".c", and

```
f*.c
```

specifies all files whose first character is 'f' and whose extension is ".c".

ctags considers a character string in a file it is scanning to be a function name, for inclusion in the cross reference file if it's a valid C name which begins on the first column of a line and which is terminated by an open parenthesis character. Thus, the function which begins

```
FILE *
fopen(...
```

would be included in the cross reference, but the function which begins

```
FILE * fopen(...
```

wouldn't.

ctags creates the cross reference file, *tags*, in the current directory on the default drive.

When a tags command is given, *Z* searches for this file in locations which differ from system to system. On PCDOS, it searches for the file in the current directory on the default drive.

2.11 Executing system commands

On PCDOS, *Z* has two commands which allow you to execute system commands while *Z* is active and then return to *Z*:

:!cmd	Executes the system command 'cmd';
:!!	Re-executes the last command.

For example,

```
:!dir *.c
```

executes the system command 'dir *.c' and returns to *Z*.

2.12 Options

Z has has several options under user control which define how *Z* behaves in certain situations. Most of these options have been discussed peripherally in previous sections, when appropriate. In this section we want to focus on the options.

Each option is identified by a code. The options and their codes are:

ai — The 'auto-indent' option. When this option is enabled and you begin inserting text on a new line, *Z* automatically indents the line by inserting tabs and spaces so that the first character you type will be located in the same column as the first non-whitespace character on the previous line. By default, this option is enabled.

eb — The 'error bells' option. When this option is enabled, *Z* will beep when you make a mistake. By default, this option is enabled.

ma — The 'magic' option. When this option is enabled, regular expessions used in string searches can include extended pattern matching characters. Otherwise, only the characters '^' and '$' are special and the extended pattern matching constructs are gotten by preceding them with ". By default, this option is disabled.

ts — The 'tab set' option. Specifies the number of characters between tab settings. By default, the tab width is four characters.

wm — The 'wrap on macro' option. When this option is enabled, and a macro being executed reaches the end of the buffer, the macro will wrap around to the beginning of the buffer and continue. By default, this option is enabled.

ws — The 'wrap on search' option. When this option is enabled, and a search for a string reaches the end of the buffer without finding the string, the search continues at the opposite end of the buffer. By default, this option is enabled.

bk — This option defines whether Z, when a *:w* command is entered to write the edit buffer to the current edit file, should save the original edit file before creating a new one.

An option is enabled by setting it to 1, and disabled by setting it to 0.

2.13 Z vs. Vi

Z is very similar to the UNIX editor VI:

* Both are full-screen editors, display text in the same way, and reserve one line of the display for messages;
* They have the same two modes: command and insert;
* Z supports most of the Vi commands. The Z commands are activated by the same keystrokes and perform the same functions as their Vi counterparts.

Z and Vi differ in the following ways:

* In Z, the buffer in which text is edited is entirely within RAM memory; in Vi, the buffer is both in memory and on disk. Because of this, Z is restricted in the size of program that can be edited, but Vi is not;
* A single copy of Vi can be configured to use any type terminal. A single copy of Z is pre-configured to use just one terminal;
* Vi has an underlying editor, *ex*, whose commands can be executed while Vi is active. Z doesn't have an underlying editor. However, Z does support some *ex* commands directly; these are the commands whose first character is ':'. (Vi interprets the ':' as a request to execute the *ex* command which is entered after the ':');
* Vi has commands and options useful for editing documents and for editing LISP programs, but Z doesn't;
* With Vi, you can create a shell and suspend Vi while executing commands from within the new shell. With some Vis, you can also suspend Vi while executing commands from the shell that activated Vi. Z doesn't support either of these features, although it will allow you to suspend Z while executing a single system command;
* Vi saves the last nine deleted blocks of text, and has commands with which it can recover them, if necessary. Z lets you recover the last deleted block;
* With Vi, operator commands can affect exactly the characters between the starting and ending cursor positions, even when the positions are on different lines. It has variations of these commands which allow whole lines to be affected, between and including the lines containing the two positions.

  In Z, operator commands in which the starting and ending cursor positions are on different lines always affect whole lines, between and including the lines containing the two positions.

2.14 System-dependent features

2.14.1 IBM PC Features

The following features are supported by the PCDOS version of the Z text editor:

* Two versions of Z are supplied.
* The PC's cursor-motion keys move the cursor.
* The function keys cause macros to be executed.

These features are discussed in the following paragraphs.

2.14.1.1 Two versions of Z

Two versions of the editor supplied, *z* and *pcz*, which differ only in the speed with which they access the screen.

Both editors require an IBM PC or an IBM PC look-alike. *pcz* performs direct screen I/O and is much faster at drawing the screen than *z*. It runs only on systems with very close compatibility with an IBM PC.

z is slower than *pcz* and requires ROM BIOS compatibility with the IBM PC.

2.14.1.2 Key substitutions

When you type certain special keys on a PC keyboard, keys that normally don't have any meaning to Z, Z substitutes for these keys characters that do have meaning to Z. The following table lists these special keys and the characters that are substituted for them. As shown in the table, for each special key up to three possible substitutions can be made: the *Normal* column indicates the substitutions when neither the shift nor the control key is being held down when a key is typed; the *Shift* and *Control* columns define the substitutions that are made when a special key is typed while the shift or control key is being held down, respectively.

Typed key	Substituted characters		
	Normal	Shift	Control
Home	^	1g	z\r
Up Arrow	k	H	
PgUp	^U	^B	[[
Left Arrow	h	B	b
5 (keypad)		%	
End	$	G	z-
Down Arrow	j	L	
PgDn	^D	^F	]]
Ins	i	o	
Del	x	D	
- (keypad)	-	-	
+ (keypad)	+	+	

In this table, \r stands for "carriage return"; and ^ stands for "control key".

If you type one of the special keys while in insert mode, except when holding down the shift key, Z will return to edit mode and then execute the command that corresponds to the substituted characters. In the special case, that is, typing a special key with shift depressed while in edit mode, the appropriate character is entered into the edit buffer.

You can enable and disable the substitutions that are normally made when a special key is typed with the shift key depressed, by setting the option *xt* to 1 or 0, respectively (ie, by entering *:se xt=1* or *:se xt=0*).

2.14.1.3 Function key macros

With the PC version of Z, macros containing up to 19 characters can be associated with function keys and then executed when the function key is typed. Up to four macros can be associated with a given function key: the macro that's executed depends on whether the shift, control, alternate, or none of these is depressed when the function key is typed.

You can execute a function key macro while in insert mode; in this case, Z will return to edit mode and then execute the macro.

To define the macro that's associated with a function key, enter a command of the form

```
:se xn=macro
```

where *n* is the number of the function key. *x* is *s*, *c*, *a*, or *f*, depending on whether the macro is to be executed when the function key is typed in conjunction with the shift, control, or alternate key, or with none of them, respectively.

For a list of the macros that are associated with the function keys, type *:se all.*

For your convenience, Z, when it starts, associates some commonly-used commands with the function keys. This association is listed in the following table. You can of course redefine the function key-macro association as described above. In this table, \ *r* stands for the carriage return character.

Function key	Associated Macros			
	Normal	Shift	Control	Alternate
F1	:x\r	:q!\r	:W\r	"a
F2	:!	:!!	:!dir	'a
F3	:>	@@		"b
F4	:se	:se ts=	:se ma=	'b
F5	:rew\r	:rew!\r		"c
F6	:ta	:ta!	^]	'c
F7	:e #\r	:e! #\r	:f\r	"d
F8	:e	:e!	:e\r	'd
F9	:n\r	:n!\r	:n	:se xt=0\r
F10	:w\r	:w!	:w	:se xt=1\r

3. Command Summary

Starting Z

z name	edit file name
z name1 name2	edit file name1, rest via :n

The Display

~ lines	lines past end of file
@ lines	lines that don't fit on screen
^x	control characters
tabs	expand to spaces, cursor on last

Options

ai=1/0	auto-indent on/off
eb=1/0	error bells on/off
ma=0/1	magic off/on
ts=val	tab width (4)
wm=1/0	wrap on search when executing macro
ws=1/0	wrap on search scan
bk=1/0	save original file as *.bak*

Adjusting the Screen

^F	forward screenful
^B	backward screenful
^D	scroll down half screen
^U	scroll up half screen
zCR	redraw, current line at top
z-	redraw, current line at bottom
z.	redraw, current line at center

Positioning within File

g	go to line (default is end of file)
G	go to line (default is end of file)
/pat	move cursor to pat searching forwards
?pat	move cursor to pat searching backwards
n	repeat last / or ?
N	repeat last / or ? in reverse direction
]]	next "^{"
[[	previous "^{"
%	find matching (), {}, or [].

Marking and Returning

“	previous context
”	first non-white at previous context
mx	mark position with letter 'x'
‘x	to mark 'x'
'x	first non-white at mark 'x'

Line Positioning

H	top of screen
M	middle of screen
L	bottom of screen
+	next line, first non-white
CR	next line, first non-white
-	previous line, first non-white
LF	next line, same column
j	next line, same column
^K	previous line, same column
k	previous line, same column

Character Positioning

0	beginning of line
^	first non-white at beginning of line
$	end of line
space	forward a character
^L	forward a character
l	forward a character
^H	backwards a character
h	backwards a character
fx	find character 'x' forward
Fx	find character 'x' backwards
tx	position before character 'x' forward
Tx	position before character 'x' backwards
;	repeat last f, F, t or T
,	repeat last f, F, t or T in reverse direction
|	move to specified column number

Words and Paragraphs

w	word forward
W	blank delimited word forward
b	back word
B	back blank delimited word
e	end of word
E	end of blank delimited word
}	to next blank line
{	to previous blank line

Insert and Replace

a	append after cursor
A	append at end of line
i	insert before cursor
I	insert before first non-blank in line
o	open line below current line
O	open line above current line
rx	replace single character with 'x'
R	replace characters

Corrections During Insert

^H	erase last character
^D	erase last character
^X	erase to beginning of insert on current line
^V	insert following character directly

Operators

d	delete
c	delete and insert
<	left shift
>	right shift
y	yank

Miscellaneous Operations

D	delete rest of line
C	change rest of line
s	substitute characters
S	substitute lines
J	join lines
x	delete characters starting at cursor
X	delete characters before cursor
Y	yank lines

Yank and Put

p	put after current
P	put before current
"xp	put from buffer 'x'
"xy	yank to buffer 'x'
"xd	delete to buffer 'x'

Undo and Redo

u	undo last change
U	restore current line
	repeat last change command

Macros

@x	execute macro in buffer 'x'
"xv	execute macro in buffer 'x'
@@	repeat last macro
v	repeat last macro

Colon Commands

:e name	edit file name
:e	reedit last file
:e! name	edit file name, discarding changes
:e!	reedit last file, discarding changes
:e #	edit alternate file
^^	edit alternate file
:e! #	edit alternate file, discarding changes
:r name	read file "name" into current file
:w	write back to file being edited
:wq	write back to file and quit
:w name	write to file "name" if does not exist
:w! name	write to file "name", delete if exists
:q	quit
:q!	quit, discarding changes
:x	quit, saving file if modified
ZZ	quit, saving file if modified
:f	show current file and line
^G	show current file and line
:n	edit next file in list
:n!	edit next file in list, discarding changes
:n arg1 arg2	specify new list
:rew	point back to beginning of list
:rew!	point back to beginning, discarding changes
:ta tag	position to tag in appropriate file
^]	same as :ta using word at cursor
:ta! tag	position to tag, discarding changes
:!cmd	execute cmd, then return (PCDOS only)
:!!	re-execute last cmd (PCDOS only)
:>macro	specify and execute immediate macro
:set opt1=val opt2=val ...	set editor options
:se opt1=val opt2=val ...	set editor options
:set all	display current option settings
:[range]s/pat/rep/[options]	substitute *rep* for *pat* in *range*
:[range]&	repeat last substitute command

SOURCE LEVEL DEBUGGER

Chapter Contents

Source Level Debugger ..... sdb
1. Overview 5
1.1 Basic Commands 5
1.2 Names 6
1.2.1 Code and Data Symbols 6
1.2.2 Operator Usage of Names 6
1.3 Loading programs and symbols 6
1.4 Breakpoints 7
1.5 Memory-change breakpoints 8
1.6 Separate screens for programs and *sdb* 8
1.7 Trace mode 9
1.8 Backtracing 9
1.9 Macros 9
1.10 Displaying source files 9
1.11 Other features 10
2. Using SDB 11
2.1 Starting SDB 11
2.2 Commands 11
2.2.1 Definitions 11
2.3 Command descriptions 13
2.3.1 The BREAKPOINT (b) commands 13
2.3.2 The DISPLAY (d) commands 16
2.3.3 The 'Find source string' (/) command 20
2.3.4 The FRAME (f) commands 21
2.3.5 The GO (g) commands 21
2.3.6 The INPUT (i) commands 22
2.3.7 The LOAD (l) commands 23
2.3.8 The MODIFY MEMORY (m) commands 24
2.3.9 The OUTPUT (o) commands 25
2.3.10 The PRINT (p) command 25
2.3.11 The QUIT (q) command 32
2.3.12 The REGISTER (r) command 33
2.3.13 The SINGLE STEP (s) commands 33
2.3.14 The UNASSEMBLE (u) commands 34
2.3.15 The VARIABLE (v) command 34
2.3.16 The MACRO (x) commands 35
2.3.17 The EXPRESSION commands 35
2.3.18 The 'Redirect command input' (<) commands 36
2.3.19 The HELP (?) command 36
2.3.20 The 'Change Mode' (z) command 37
3. Command Summary 38

Source Level Debugger

This chapter describes the source level debugger utility *sdb* that is provided with some versions of Aztec C86. For a description of the assembly language debugger, *db*, see the Assembly Language Debugger chapter. For a description of other utility programs that are provided in some Aztec C86 packages, see the Utilities and Unitools chapters.

NAME

sdb - source level debugger

SYNOPSIS

sdb [options] [progfile] [arg1 arg2 ...]

DESCRIPTION

sdb is used to debug programs which have been created using the Aztec C compiler, assembler, and linker.

sdb has all the standard features of a source language debugger. Some of these features are:

* allows the user to debug at the C source level and at the assembly language level,
* allows the user to reference memory locations using full C expressions, (including function calls and conditionals using the variable names from the source program)
* keeps track of the types and locations of the variables so you don't have to,
* allows the display and modification of variables using their C source names,
* allows the creation of macros that can be saved and reused in the next *sdb* session,
* allows the user to display structures and arrays and their members using their C source names.

In addition, *sdb* has features specifically tailored to its use with Aztec C, such as the ability to list the name and parameters of the currently executing function, and the function that called it, and so on, back to the initial function. Another special feature is the ability to display, on entry and exit from each function, the function's parameters and return value.

Requirements

A minimum of 256 K bytes of RAM memory is recommended for use with *sdb*. The debugger itself uses about 128K.

sdb can only be used on 8086, 8018x, 8028x, and 8088-based systems running MSDOS or PCDOS, version 2.0 or later.

Preview

The remainder of this description of *sdb* is in three sections: *overview*, which describes *sdb* features in more detail and introduces the commands; *usage*, which describes in full detail how to use *sdb*; and a *command summary*.

1. Overview

sdb commands consist of one or two characters, the first of which identifies the command category. If there's only one command in the category, then the command has just this one letter; otherwise, the command has a second letter which identifies the specific operation to be performed.

1.1 Basic commands

sdb has two types of commands for examining memory: display and print, whose first characters are *d* and *p*, respectively. The 'display' commands *db* and *dw* simply display hexadecimal bytes and words.

The 'print' command, *p*, is more powerful, allowing you to print variables, arrays, and structures by name, without you having to worry about data types. For example, you can tell *sdb* to print a structure whose name is symbol by typing *p symbol*, and *sdb* will figure out the data types and print the structure and it's members in the proper format. You can also display strings that are pointed to by a structure member using the *p* command.

The 'evaluate' command, *e*, allows you to perform general C expression evaluation, including calls to C functions, assignment, pre- and post- increment and decrement, casts, and conditionals.

The 'register' command, *r*, displays and modifies the 8086 registers.

The 'frame' command, *f*, allows you to walk up and down the call frame.

The 'memory modify' commands, *m*, modify memory.

The *u* commands 'unassemble' code; that is, display it symbolically, in a form similar to its appearance in an assembly language source file.

The *s*, *t*, and *g* commands cause the user's program to be executed. *s* and *t* commands "single step" the user's program; that is, execute a specified number of instructions in the user's program and then return control to *sdb*. *g* commands transfer control of the processor unconditionally to the user's program. In this case, *sdb* regains control when the user's program terminates, when an error occurs (such as division by zero), or when a "breakpoint" is taken. Breakpoints are discussed below.

? is the *help* command: it causes *sdb* to display a summary of all *sdb* commands. For some command categories, you can get information about the commands in a category by typing the first letter of the category's commands followed by a *?*. For example, typing *m?* gets you information about the memory modification commands (all of whose first letter is *m*).

1.2 Names

sdb allows memory locations to be referenced by name as well as by location. It learns a program's variable names by reading the file containing the program's symbol table. The linker generates a symbol table file for a program in response to the -g option. Ways of causing *sdb* to read the symbol table are described below.

sdb allows global symbols, automatic variables, static variables, arrays and structures to be accessed by name.

The operator can also define names to *sdb* using the *v* command, and the 'clear symbols' command, *cs*, will remove symbols from the memory-resident symbol table.

1.2.1 Code and Data symbols.

sdb classifies symbols as being either code or data symbols; that is, as referring to a location in a physical code segment or the physical data segment. All symbols in the program's symbol table file which occur between the special linker symbols __Corg__ and __Cend__ are considered to be code symbols, and all others are data symbols. This classification of symbols frees the operator from having to specify the physical segment in which a symbol is located, when using commands which reference the location; the debugger knows which segment it's in. The classification is also of use when unassembling memory.

There are two commands for viewing the symbols which are known to *sdb*: *dc* and *dd*, which display code and data symbols, respectively.

1.2.2 Operator usage of names.

When a C source program is compiled, all global symbols are truncated to a maximum of 31 characters and are then appended with an underscore character.

To refer to symbols which, in a C source file contain less than 31 characters, the inclusion of the appended underscore character is optional: if it's specified, *sdb* will search for just that symbol in it's symbol table. Otherwise, it will first search for the specified symbol; if the search fails, it will then append an underscore to the name and search again.

To refer to symbols which contain 32 or more characters, only the first 31 are significant; *sdb* ignores all other characters in the name.

1.3 Loading programs and symbols

A program and its symbols can be loaded into memory when *sdb* is started; in this case, the command line defines the program to be loaded. The *sdb* 'load program' command, *lp*, can also be used. If *sdb* is started with a program name specified on the command line, the *lp* command can be used to reload the same program for another debugging session. If *sdb* is started without specifying a program

name, the *lp* command must be issued with the desired program name. Only one user program can be in memory at once.

When told to load a program, *sdb* automatically tries to load the program's symbol table, too; it assumes the symbol table file has the same name as the program file, with the extension changed to *.dbg*.

When the *lp* command finds a symbol table file, it clears the symbol table of all symbols, other than those defined with *v* commands, before loading the new symbols.

When a program exits, it must be reloaded with the *lp* command before execution can begin again.

A memory map can be obtained using the *dm* command.

1.4 Breakpoints

Before transferring control of the processor to a user's program in response to a *g* command, *sdb* can set "breakpoints" at specified locations in the code. When the user's program reaches a breakpoint, *sdb* regains control.

A breakpoint has a 'skip count' associated with it, which allows a breakpoint to be passed several times before actually taking the breakpoint and returning control to *sdb* and the user. When a breakpoint is reached, *sdb* is always activated; it increments a counter associated with the breakpoint. When the counter's value is greater than the breakpoint's skip count, the breakpoint is taken; that is, *sdb* retains control of the processor. Otherwise, *sdb* returns control of the processor to the user's program after the breakpoint. By default, a breakpoint's skip count is 0; thus, each time the breakpoint is reached, it's taken.

A breakpoint can also have a sequence of *sdb* commands associated with it. When a breakpoint is taken, these commands will be executed before *sdb* allows the operator to enter commands. For example, if you just want to examine a variable each time a certain location in the code is reached and then have the program continue execution, you could define a breakpoint at the location, and specify a list of commands to do just that: the first command in the sequence would be a *d* command to display memory, and the second would be a *g* command to continue execution of the program.

There are two ways to define breakpoints: with the *g* command, and with special breakpoint commands, whose first letter is *b*.

The breakpoint commands manipulate a table of breakpoints: there are commands for entering breakpoints into the table, displaying the entries, reseting their counters, and removing them from the table.

There's a difference between a breakpoint defined in a *g* command and those in the breakpoint table: the *g* command breakpoint is

temporary, while a breakpoint table is more permanent (it exists until removed from the table). Before transferring control to the user's program in response to a *g* command, *sdb* sets all breakpoints that are in the breakpoint table and that are specified in the *g* command itself. When a breakpoint is taken, *sdb* removes all breakpoints from the code and forgets all about the *g* command breakpoint. The breakpoint table breakpoints, however, are still in the table and will be set back in memory when control is again returned to the user's program.

sdb remembers the skip counter associated with a breakpoint which is in the breakpoint table: when it sets breakpoints in memory, the count for such a breakpoint is set to its remembered value (that is, its value in the table); and when a breakpoint is taken, the accumulated count for the breakpoints in memory are saved in the breakpoint table.

1.5 Memory-change breakpoints

The breakpoints described above are taken when a program reaches a specified point in the code. A second type of breakpoint, called a memory-change breakpoint, is taken when a specified memory location is changed from or set to a particular value.

With a memory-change breakpoint set, *sdb* will detect either the function or the instruction which modifies the specified memory location, depending on whether the user's program was activated using a *g* command or is being single-stepped using an *s* command, respectively.

When the user's program is activated with a *g* command and a memory-change breakpoint is set, *sdb* will examine the specified memory location on entry to, and exit from, each function. It will take a breakpoint, that is, interrupt execution of the program and return control to the operator, when the contents of the memory location meets the specified condition.

When an *s* command is used to single-step a program and a memory-change breakpoint is set, *sdb* will examine the specified memory location after each instruction is executed, and take a breakpoint when appropriate.

The *bm* command is used to set and remove memory-change breakpoints.

1.6 Separate screens for programs and *sdb*

When used on an IBM PC or an equivalent, *sdb* can optionally maintain separate screens for itself and for a program that is being debugged. With this feature enabled, a program-generated screen is displayed while a program is executing, and a screen of operator-*sdb* interactions is displayed while *sdb* is executing.

This feature is implemented as follows: when a program encounters a breakpoint, *sdb* saves the contents of the screen and displays the

debug screen; similarly, when *sdb* continues a program, it saves the debug screen and restores the program screen.

To enable this feature, specify the *-w* option when you start *sdb*. This option must precede the name of the program that is to debugged.

Two *sdb* commands are related to separate program/debug screens:

* The *w* command causes *sdb* to toggle between displaying the debug and program screen.
* The *W* command disables screen saving and restoring.

1.7 Trace mode

sdb supports two 'trace modes', which displays information whenever a function is entered or exited, or when a source line is passed.

With the first mode enabled, on entry to, and exit from, a function, the function name, its arguments, and return values are displayed.

The commands *bt* and *bT* affect trace mode: *bt* enables and disables call trace mode, and *bT* enables and disables source line trace mode.

1.8 Backtracing

When *sdb* regains control from an executing program (for example, because a breakpoint was taken), it has the ability to display information on how the program got to its current location: the *ds* and *dS* commands will display information about the currently executing function, and the function which called it, and so on, back to the Manx function *Croot*, which called the user's function *main*.

ds displays, for each function, its name, arguments which were passed to it, and the address to which it will return. *dS* displays the function's automatic variables as well.

1.9 Macros

sdb allows the user to define and execute 'macros'; that is, a sequence of *sdb* commands.

Macros are written to a file which is saved so that the macros can be reused in a subsequent session. The file name for the saved macros is derived by taking the program name and appending a *.mac*.

The *sdb* command *x* is used both to define and execute a macro.

1.10 Displaying source files

Since *sdb* is a source level debugger, it allows the user to display source files, thus providing a convenient means to examine the source of a program being debugged.

Only a single source file can be examined at a time. The 'display source lines' command, *df*, and the 'context' command, *c*, can be used

to display its lines.

The 'find string' command, /, will find a character string in the source file.

1.11 Other features

Some other features of *sdb* which haven't yet been discussed are:

* The 'input' and 'output' commands, *i* and *o*, will transfer data to and from an i/o port;
* The 'redirect stdin' command, <, causes *sdb* to read commands from a specified device or file and then continue reading commands from the console. The <> command allows redirection of both input and output. The 'log' commands, > and >>, allow the logging of either all I/O or commands only to a separate file.
* The 'evaluate expression' commands, = and *e*, do just that.
* The 'help' command, *?*, lists commands.

2. Using SDB

2.1 Starting SDB

sdb is started with a command of the form:

sdb [option] [progfile] [arg1 arg2 ...]

where the optional parameter [option] is any of the following:

-sPATH	where PATH is a string to be prefixed to source file names upon file opening
-w	use separate windows for debugger and program
-a	start debugger in assembly mode - default is C source mode

and the optional parameter *[progfile]* is the name of a file containing a program to be debugged, and the optional parameters *arg1*, *arg2*, ..., are character strings to be passed to the program.

The extension on the program filename is optional; if not specified, a search is first made for a file with the extension *.exe*. If that search fails, another is made with the extension *.com*.

sdb looks for a period, '.', to decide whether the program file name contains an extension. Thus, to specify a program which doesn't have an extension, include a period at the end of the file name.

If the program file name specifies a drive or directory, *sdb* searches for the program file in just that particular area. Otherwise, it searches the current directory on the default drive.

The "arg" parameters are passed to the program using the *argv* parameter of the program's *main* function: *arg1* is pointed at by *argv[1]*, *arg2* by *argv[2]*, and so on. *argv[0]* always contains zero.

2.2 Commands

This section describes in detail the *sdb* commands. It first defines some terms that are used in the command descriptions. These terms are *expr*, *term*, *addr*, *range*, and *cmdlist*.

2.2.1 Definitions

2.2.1.1 The Definition of EXPR

An EXPR is any valid C expression.

For example, an EXPR can consist of a single term, a series of terms separated by operators, the use of registers by their standard names, or 16 bit values representing memory locations of the form segment:offset.

Here are some examples of EXPR:

```
si
x + 2.0
(i == j) ? 2 : 3
ds:0x4127
sin(y)
array[j]
```

2.2.1.2 The Definition of ADDR

An ADDR is the name of a C variable, an address constant or a C expression that yields an address. Examples of an ADDR are as follows:

linkmain.c.19	address of the code corresponding to line 19 of linkmain.c
.19	same as above if current file is linkmain.c
main	address of main
cs:0x1532	address formed by using the current code and an offset of 0x1532
token[j]	address of jth element of array token

2.2.1.3 The Definition of RANGE

A RANGE defines a block of memory. It has one of the following forms:

```
ADDR,CNT
ADDR to ADDR  (the word 'to' must be included)
ADDR
,CNT
```

The form ADDR,CNT specifies the starting address, ADDR, and a number, CNT. CNT is interpreted differently by different commands. For example, the 'disassemble code' command, *u*, will display CNT lines, while the 'display bytes' command, *db*, will display CNT bytes.

The form ADDR to ADDR specifies the starting and ending addresses of the range.

A full range need not be explicitly specified, because *db* remembers the last-used range and will set unspecified RANGE parameters from the remembered values:

* When a RANGE is specified which consists of a single ADDR, the last used CNT is used.
* When a RANGE is specified which consists of ',CNT', the next consecutive address is used, and the remembered count is changed to the new value.
* When nothing is specified as the RANGE, the next consecutive address is used as the starting ADDR, and the CNT is set to the remembered value.

2.2.1.4 The Definition of CMDLIST

A CMDLIST is a list of commands. It consists of a sequence of commands or macros separated by semicolons:

COMMAND [;COMMAND ...]

If a macro is in a CMDLIST, it must be the last command in the list.

2.3 Command descriptions

The following descriptions of debugger commands uses terms and concepts which were presented in the preceding sections.

The commands are listed alphabetically. For an index, see the command summary which follows the descriptions.

2.3.1 The Breakpoint Commands

bm - Set Memory-Change Breakpoint

Syntax:

bm
bm ADDR == [VAL]
bm ADDR != [VAL]

Description:

This command is used to set and clear a memory-change breakpoint, with the parameterized version used to set breakpoints and the parameter-less version to clear them.

In the parameterized form of the commands, ADDR specifies the field to be monitored.

With the '==' form, the breakpoint will be triggered when the debugger detects that the field is equal to the specified value, VAL.

With the '!=' form, the breakpoint will be triggered when the debugger detects that the field is different from the specified value.

The VAL parameter is optional. If not specified, it defaults to the current value at the ADDR.

---

bc - Clear a single breakpoint
bC - Clear all breakpoints

Syntax:

bc ADDR
bC

Description:

These commands delete breakpoints from the breakpoint table.

bc deletes the single breakpoint specified by the address ADDR, and *bC* deletes all breakpoints from the table.

---

bd - Display breakpoints

Syntax:

bd

Description:

bd displays all entries in the breakpoint table.

For each breakpoint, the following information is displayed:

* Its address, using a symbolic name, if possible;
* The number of times it's been 'hit' without a breakpoint being taken.
* The skip count for it;
* The command list for it, if any.

For example, a *bd* display might be:

address	hits	skip	command
cs:printf__	1	2	
cs:putc__	0	0	db ds:__Cbuffs

In this example, two breakpoints are in the table. The first is at the beginning of the function *printf__*; a breakpoint will be taken for it every third time it is reached, and no command will be executed. Given its current hit count, a breakpoint will be taken the next time *printf__* is reached.

The second breakpoint is at the function *putc__*; a breakpoint will be taken each time the function is reached, and will display memory, in bytes, starting at ds:__Cbuffs.

---

br - Reset breakpoint counters

Syntax:

br [ADDR]

Description:

br resets the 'hit' counter for the specified breakpoint which is at the address, ADDR. If ADDR isn't given, the 'hit' counters for all breakpoints in the breakpoint table are reset.

---

bs - Set or modify a breakpoint

Syntax:

[#] bs ADDR [;CMDLIST]

Description:

bs enters a breakpoint into the breakpoint table, or modifies an existing entry.

The optional parameter # is the skip count for the breakpoint. If not specified, the skip count is set to 0, meaning that each time the breakpoint is reached it will be taken.

The optional parameter CMDLIST is a list of debugger commands to be executed when the breakpoint is taken.

---

bt - Toggle the call trace mode flag
bT - Toggle the source line trace mode flag

Syntax:

bt
bT

Description:

bt and *bT* toggle the call trace mode and source line trace mode flags, respectively. The state of the trace mode flag determines whether trace mode is enabled or disabled.

In call trace mode, the debugger will print the names and arguments of each call within the program as it executes. On return, the value of the function's return will be printed.

In source line trace mode, each statement of the program wll be displayed just before it is executed.

2.3.2 The Display Commands

db - Display memory in bytes
dw - Display memory in words
d - Display memory in last format

Syntax:

db [RANGE]
dw [RANGE]
d [RANGE]

Description:

The *db* and *dw* commands display successive bytes and words of memory, respectively. *d* displays memory using the last format specified; for example, if *d* is entered, and *db* was the last 'display memory' command, then *d* will display bytes, too.

The starting address of the RANGE parameter is optional; if not specified, it defaults to the ending address of the last display's RANGE, plus one.

Each line of the display begins with the segment and address, followed by a hexadecimal display of 16 bytes or 8 words, followed by an ASCII display, by bytes, of the same data. For the ASCII display, values falling outside the range 0x20 to 0x7f are displayed as a period.

If the ending address does not fall on a multiple of 16 bytes, only the number of bytes or words specified in the last line will be displayed.

---

dc - Display all code symbols

Syntax:

dc

Description:

dc lists all the code symbols in the memory-resident symbol table and all user-defined symbols.

For each symbol, its name and address are displayed.

---

dd - Display all data symbols

Syntax:

dd

Description:

dd lists all the data symbols in the memory-resident symbol table. For each symbol, its name and address are displayed.

---

df - Display source file lines

Syntax:

df [FILENAME] [RANGE]

Description:

df displays lines from the source file which was specified by the FILENAME parameter.

The RANGE parameter specifies the numbers of the lines to be displayed.

The starting line number is optional; if not specified, the display starts with the "current" line.

The current line in a source file is set by the source file commands *df* and *f*, as follows:

* When the file is first loaded with the *df* command, the first line in the file is the current line.
* When the last source file command was 'display source', *df*, the current line is the line following the last one displayed;
* When the last source file command was 'find string', *f*, the current line is the line in which the string was found.

df also sets the "F-dot" for the source file to the number of the first line displayed. The F-dot is the line referred to when the starting line number of the range in a *df* command specifies a period (.). Also, source string searches begin at the line following the F-dot line.

Each displayed line is preceded with a line number in decimal, a colon, and the line itself.

---

dg - Display global values

Syntax:

dg

Description:

For each data symbol in the debugger's symbol table, *dg* displays the type, name, and value for that symbol. If the symbol is an

array of structures, each element in the array will be printed.

---

dm - Display memory map

Syntax:

dm

Description:

dm displays a memory map of allocated memory, beginning with the debugger program itself.

An entry gives information about one allocated block of memory. It has the following form:

xxxx: owner=yyyy size=zzzzz

where *xxxx* is the paragraph number at which the block begins, *yyyy* is the Program Segment Prefix of the process that owns the block, and *zzzzz* is the number of bytes in the block, in decimal.

For example:

10cf: owner=10d0 size=85264
25a1: owner=25aa size=112
25a9: owner=25aa size=69168
368d: owner=0000 size=38688

The debugger occupies the block that begins at paragraph 0x10cf; the Program Segment Prefix for *db* begins at paragraph 0x10d0; the number of bytes in *db*'s block is 85264. The environment of the program being debugged begins at paragraph 0x25a1, and the Program Segment Prefix for the program begins at paragraph 0x25aa. The program itself occupies the block beginning at paragraph 0x25a9. Memory beginning at paragraph 0x368d is unallocated.

Note that both the block containing program and the block containing its environment are owned by the PSP of the program.

---

dn - Display 8087 State

Syntax:

dn

Description:

dn displays the state of the 8087, in the following format:

```
cccc    ssss    tttt
iilo    jjjj    oolo    oohi
st(0): top stack element
st(1): next stack element
...
st(7): last stack element
```

where

* *cccc* is the control word
* *ssss* is the status word
* *tttt* is the tag word
* *iilo* is the least significant 16 bits of the instruction pointer
* *jjjj*, most significant 4 bits are the most significant 4 bits of the instruction pointer and the least significant 11 bits are the opcode
* *oolo* is the least significant 12 bits of the operand pointer
* *oohi* has as its most significant four bits the most significant four bits of the operand pointer, and its least significant 12 bits are 0.
* *st(0)*, ... are the contents of the stack registers.

All values are in hexadecimal.

---

ds - Display Stack Backtrace
dS - Display Stack, Function Args, and Autos Backtrace

Syntax:

[#] *ds*
[#] dS

Description:

ds displays information about the current function, the function which called it, and so on in reverse order of invocation, back to Croot, the Manx function which called the user's function *main*.

The optional '#' parameter specifies the number of stack frames to be displayed.

For each function, the information consists of the function's name and the parameters passed to it.

Arguments are displayed according to their type. If no type information is available, the arguments are displayed as a series of 16-bit hex values. Argument of type long or double, when displayed as hex, will be displayed as separate words.

ds determines the number of parameters by looking at the instructions which follow the address to which the function will return.

ds assumes that the BP register points to the C stack frame for the current function, unless the current instruction is within 6 bytes of the start of the function.

dS causes the function to be displayed with the types, names, and values of each function argument. Below this, the values of all automatic variables will be displayed.

---

c - Display Source Context

Syntax:

c

Description:

This command will display 10 lines of source centered on the current line.

2.3.3 The 'Find Source String' Command

/ - find string in source file

Syntax:

/STRING

Description:

This command searches the current source file (that is, the one specified by the last *lp* command) for a specified string.

The search begins at the line following the current line.

If the string is found, the current line and the F-dot line of the source file is set to the line containing the string; otherwise, these values are unchanged.

STRING is the character string to be located, and consists of all characters following the / and preceeding the carriage return.

If the first character of the string is '^', the search will begin with the first character on a line. In this case, '^' isn't part of the search string.

The 'current line' and F-dot line for a source file are defined in the description of the *df* command.

2.3.4 The 'Frame ' Commands

fu - frame up command
fd - frame down command

Syntax:

fu
fd

Description:

Normally, you are only allowed to view variables that are visible by C rules at the point where execution stopped. That is, you cannot refer to local variables of functions that are not the "current" active function. *sdb* allows you to get around this restraint by allowing you to change what the "current" function is.

The *fu* command allows you to walk up the call frame and displays the line from which the call was issued. By changing frames, you can view all the local variables of the now "current" function. You can walk up the frame until you get to the initial function.

The *fd* commands allows you to walk down the frame displaying the function call of the previous frame.

2.3.5 The Go commands

g - Execute the program
G - Execute the program, without setting table breakpoints

Syntax:

[#]g [@ <function>] [ADDR] [;CMDLIST]
[#]G [@ <function>] [ADDR] [;CMDLIST]

Description:

The *g* commands transfer control of the processor to the user's program, at the address specified by CS:IP. The user's program then executes until it terminates, an error such as division by zero occurs, or a breakpoint is taken; control then returns to the debugger program.

The parameters to the 'g' commands allow one or two temporary breakpoints to be set in memory before the user's program is executed.

The difference between the 'g' and the 'G' command is that the 'G' command sets in memory just the breakpoints specified in the command itself, while the 'g' command also sets the breakpoints specified in the breakpoint table.

The '#' and 'ADDR' parameters define one of the temporary breakpoints that a Go command can set:

* # is the skip count for the breakpoint; it defaults to zero, meaning that the breakpoint is taken every time it's reached;
* ADDR is the address for the breakpoint;

The '@ <function>' parameter specifies that a temporary breakpoint is to be set at the return address of the specified function. If the function isn't specified, it defaults to the current function. If a function is specified, the breakpoint is set to the address to which the function will return. In this case, the breakpoint isn't set until the function is entered; thus, in programs which call the function from several different places, the breakpoint will be set at the actual address to which the function will return.

The ';CMDLIST' parameter defines a sequence of debugger commands, separated by semicolons, that the debugger is to execute once a breakpoint which is specified in the 'go' command is taken. If this parameter isn't specified, it defaults to the command list used for the last temporary breakpoint.

If *sdb* is maintaining separate screens for the program and itself, it restores the program screen before transferring control to the program. For more information on this, see the discussion of separate screens in section 1 of this *sdb* documentation.

Before setting breakpoints and transferring control to the user's program, the debugger single-steps the user's program, (that is, causes it to execute one instruction). This allows the operator to transfer control to a location in the program at which there is a breakpoint, without immediately triggering a breakpoint and re-entry to the debugger.

2.3.6 The Input Commands

ib - Input byte from port
iw - Input word from port

Syntax:

ib PORT
iw PORT

Description:

ib and *iw* input a byte or word, respectively, from the i/o port, PORT.

2.3.7 The LOAD Command

lp - Load program

Syntax:

lp
lp progfile [arg1 arg2 ...]

Description:

lp loads a program into memory. If a symbol table file can be found for the program, it will be loaded, too.

If the *lp* command is given without parameters, the last *lp* command is re-executed. The following comments describe the parameterized version of *lp*.

Loading the program

The parameter *progfile* specifies the file containing the program. The extension on the file name is optional.If not given, a search is made for the file with extension ".exe"; if this fails, another is made for the file with extension ".com". To specify a program file which doesn't have an extension, use a period after the filename.

If the program file name specifies a drive or path, the file is searched for in just that location; otherwise, it's searched for on the current directory of the default drive.

If an attempt is made to load a program when *sdb* was invoked with a program name or a previous *lp progfile* was executed will cause an error to be printed and the command ignored.

Loading the symbol table

The name of the file containing the symbol table is assumed to be the same as the program file name, with the extension changed to ".dbg".

After the program is loaded, its program segment prefix (PSP) is initialized with the arguments *arg1*, *arg2*, etc. These are available to the program as the *main* function's arguments *argv[1]*, *argv[2]*, and so on. *argv[0]* is set to 0, and *argc* is set to the number of "arg" parameters.

The U-dot (that is, the value of the period parameter associated with the *u* commands) is set to CS:IP. The D-dot (the value of the period parameter for the *d* and *m* commands) is set to DS:0.

Once a program exits, it must be reloaded with an *lp* command before it can begin again.

2.3.8 The Memory Modification Commands

mb - Modify bytes of memory
mw - Modify words of memory

Syntax:

mb ADDR EXPR1 [EXPR2 ...]
mw ADDR EXPR1 [EXPR2 ...]

Description:

db and *dw* modify bytes and words of memory, respectively.

The parameter ADDR specifies the address of the first byte or word to be modified.

The EXPR parameters are expressions, whose resulting values are set in memory, with EXPR1 set in the first byte or word specified, EXPR2 set in the next higher byte or word, and so on.

The EXPR parameters can be separated by spaces or commas.

---

mc - Compare memory

Syntax:

mc RANGE = ADDR

Description:

mc compares two blocks of memory and, for each comparison which fails, displays the corresponding segment, address, and value.

RANGE specifies one of the blocks of memory. The second begins at ADDR and has the same length as the first block.

---

mf - Fill memory

Syntax:

mf RANGE = EXPR

Description:

mf sets each byte in a block of memory to a specified value.

The RANGE parameter specifies the memory block, and EXPR an expression whose resulting value is the value to be set in the range.

---

mm - Move memory

Syntax:

mm RANGE = ADDR

Description:

mm copies one block of memory to another.

The RANGE parameter specifies the source block and ADDR the starting address of the block to be modified.

---

ms - Search memory

Syntax:

ms RANGE = EXPR1 [EXPR2 ...]

Description:

ms searches a block of memory for a sequence of bytes having specified values. For each match, the corresponding address of the start of the string is displayed.

RANGE specifies the block of memory. The EXPR parameters are expressions, each of whose resulting values is one byte of the search sequence.

2.3.9 The Output Commands

ob - Output byte to i/o port
ow - Output word to i/o port

Syntax:

ob PORT,EXPR
ow PORT,EXPR

Description:

ob and *ow* output a byte or word, respectively, to an i/o port.

PORT is the address of the port. EXPR is an expression whose resulting value is the value to be output.

2.3.10 The 'Print' Command

p - formatted print

Format:

p [@format] [ADDR]

Description:

p generates a formatted display of C variables, arrays, and structures, by converting data items in memory to a displayable form as directed by it's type or the optional conversion string *format.*

format is an optional list of format specifications, each of which defines the type of a data item and the conversion to be performed on it.

ADDR specifies the address of the first data item that *p* is to convert and display.

In the absence of the *format* string, *p* looks at the ADDR to be printed, gets it's typing information from the symbol table, builds and executes the appropriate format string. For example, to print a structure named symbol you simply enter:

```
p symbol
```

which might result in:

```
struct symboltb symbol = {
        int s_flag = 10
        char *s_name = 0xFF42
        int s_value[2] = {
                10,5
        }
}
```

If you then want to display the string pointed to by symbol.s_name, you simply type:

```
p @s *symbol.s_name
```

The @ indicates that you are overriding the default format for pointer to char and the s indicates that the format is string. The result might be:

```
"pointer"
```

If you wish to create your own format string, here is a description of the use of *format* by *p*. *p* works its way through the *format* string, converting and displaying data items in memory as requested by the format string items. When *p* reaches an item in the *format* string, it converts the data item at its 'current address' as directed by the format item. When it finishes processing a format string item, it increments its current address by the size of the data item that it just processed, so as to be ready to process the next data item as directed by the next format string item.

If ADDR is not entered, the starting address is assumed to be the print command's 'current address'. Normally, this is the address of the first byte beyond the last data item converted by the last *p*

command. However, there is a *format* item that causes *p* to remember the address contained in the current data item, and then make that the current address after it finishes processing the entire format string.

The format items have the form

[rpt][indir_flgs][size]desc_code

where

* *desc_code* is a single-letter code that defines the type of the data item and the conversion to be performed upon it. For example, the code *d* says 'take the two-byte binary value at the current address, convert it to decimal, and print it'. So if *var* is an *int*, the following command could be used to print its value in decimal:

  p @d var

  The code *x* says "take the two-byte binary value at the current address, convert it to hexadecimal, and print the result". So the hexadecimal value of *var* could be printed with the command:

  p @x var

* *indir* is a string of zero or more * and/or # characters, which are indirection indicators specifying that the value at the current data item is a pointer to a chain of zero or more pointers, the last of which points to an object whose type and requested conversion are defined by *desc_code.*

  To find the data object corresponding to a format item that has indirection indicators, *p* begins by setting its idea of the address of the data object to the current address. It then works its way from left to right through the indirection indicators; for each indicator it replaces its current idea of the data object address with the pointer that is in the field at this address. The data object address is distinct from the current address: at the end of this process, the *p* command's current address is simply incremented past the first pointer.

  A * specifies that the pointer within the field referenced by the current data object address is two bytes long. This pointer is the offset component of the new data object address from the last segment referenced.

  A # specifies that the pointer within the field referenced by the current data object address is four bytes long. The most significant word of this pointer is

the segment component of the new data object address, and the least significant word is its offset component.

For example, if the variable *cp* is a short pointer to a character string (that is, its declaration is *char *cp*), then the string pointed at by *cp* could be printed by the command

*p @*s cp*

Here we have made use of the *s desc_code*, which specifies that the data object is a character string, and that the string's characters are to be printed, with possible modifications as noted below, up to a terminating null character. After this command, the *p* command's current address is set to the byte immediately following *cp*.

As another example, if a module uses the 'large data' memory model (that is, its pointers to data objects are four bytes long), and if *cpp* is a pointer to an array of pointers to character strings (that is, the declaration of *cpp* is *char **cpp*), then the string pointed at by the first element of the array could be displayed with the command

p @##s cpp

Following this command, the *p* command's current address is set to the byte following *cpp*.

* The *rpt* parameter of a format item defines the number of times that the item is to be processed. It allows a sequence of *rpt* identical format items to be abbreviated by just one such item with a leading *rpt* count.

  For example, if *a* is an array of *floats*, then the first five items in this array could be displayed with the command

  p @5f a

  This command uses the fact that the *desc_code* to convert a four-byte floating point value at the current address to a displayable value is *f*. This command is equivalent to the command *pfffff a*. At the end of this command, the *p* command's current address is set to the address of the byte following the last displayed *float*.

* The *size* parameter of a format item defines the number of data items that are to be converted and printed. When the format item doesn't use indirection, *size* has the same effect as *rpt*; for example, in the *p5f a* command above, the 5 could be interpreted as being a

size parameter instead of a *rpt* parameter.

When the format item does use indirection, then the *size* parameter defines the number of data items to be converted and printed at the end of the indirection chain. For example, if a module using the 'small data' memory model defines *lpp* as a pointer to an array of pointers to *long*s (that is, the declaration of *lpp* is *long **ip*), then the following command would display the first four *long*s pointed at by the first element of the pointer array:

```
p @**4D lpp
```

Here we have used the *D desc_code*, which specifies that a four-byte signed binary value is to converted to decimal and printed. The following command would display the first three *long*s pointed at by the first three elements of the pointer array:

```
p @3*4D *lpp
```

To demonstrate further the difference between the *rpt* and *size* fields in a format item, consider the format items *4*d* and **4d*. The first causes the print command to take the item at the current address as a short pointer, increment the current address by two, convert to decimal and print the two-byte value referenced by the pointer, and then repeat the process three more times. At the end of the process, the current address has been advanced by eight.

The second item causes the print command to again take the item at the current address as a short pointer, increment the current address by two, and then convert to decimal and print the four successive two-byte values that begin at the address defined by the pointer. At the end of the process, the current address has been advanced by two.

As an example of the use of format strings containing several format items, consider the following code in a program that uses short data pointers:

```
struct {
    int *ip;
    float flt;
    char *cp;
} var = {&i, 3.14159, "ralph"};
int i=2;
```

The command

```
p @*d2-xf*s2-x var
```

will print

2 xxxx 3.14159 ralph

where *xxxx* is the hexadecimal address of *i* and *yyyy* is the hexadecimal address of the string.

A complete list of the *desc_codes*

We have introduced some of the *desc_codes* above. Here is a list of the basic *desc_codes*:

b	Convert to hexadecimal and print a byte.
d	Convert to decimal and print a two-byte signed binary value.
D	Convert to decimal and print a four-byte signed binary value.
f	Convert and print a four-byte *float.*
F	Convert and print an eight-byte *double.*
	Define the precision to be used in the display of floating point values; that is the number of digits to be displayed to the right of the decimal point. This number precedes the period. For more information, see below.
o	Convert to octal and print a two-byte field.
O	Convert to octal and print an eight-byte field.
x	Convert to hexadecimal and print a 2-byte field.
X	Convert to hexadecimal and print a 4-byte field.
u	Convert to decimal and print an unsigned, two-byte value.
U	Convert to decimal and print an unsigned, four-byte value.
p	Print a short, two-byte pointer in segment:offset form.
P	Print a long, four-byte pointer in segment:offset form.
c	Print a character.
C	Print a character.
s	Print a string up to a terminating null byte.
S	Print a string up to a terminating null byte.

Codes for printing floating point numbers

When a floating point number is displayed in response to an *f* or *F desc_code*, the 'precision' of the display (that is, the number of digits displayed to the right of the decimal point) is determined by the most recently-entered period *desc-code*; if a period code hasn't been entered, the precision of the display defaults to 7 digits for floats and 16 for doubles.

The period *desc_code* consists of a period preceded by the number of digits of precision. For example, the following command contains two *desc_codes*: the first sets the precision to 2 digits; the second then displays a float, listing two digits to the right of the decimal point.

p @2.F

Codes for printing characters

For the C and S *desc_codes*, each character is printed "as is", with no translations.

For the c and s codes, printable ASCII characters (that is, whose hex value is between 0x20 and 0x7f) are printed "as is". A character whose hex value is less than 0x20 is printed as two characters: ^ followed by the printable character whose hex value equals the original character's value plus 0x40. A character whose hex value is 0x80 or greater is displayed as a ' character followed by the one or two characters that would be printed for the character whose hex value equals that of the original character less 0x80. For example, 0x41, 0x1, and 0x81 would be printed as A, ^A, and '^A, respectively.

Special purpose codes

The following *desc_codes* can be used to assist in the formatting of the *p* output:

character	*output*
N or n	*Output a newline character*
R or r	*Output a blank character*
T or t	*Output a tab character*
"string"	*output "string"*

These characters can be preceded by a count specifying the number of characters or strings to be output.

Codes for setting the *p* command's current address

The next group of *desc_codes* change the *p* command's notion of the current address. They don't cause any printing.

^ Back up the current address by the size of the last data item.

- or + Back up or advance, respectively, the current address by *size* bytes, where *size* is a decimal value preceding the - code. If *size* isn't specified, it defaults to one byte.

A or a Remember the long or short pointer, respectively, that is contained in the current data object; If this pointer is not null, set the *p* command's current address to this value after the entire format string has been processed.

If the pointer is null, set the *p* command's current address to the value it had before the entire format string was processed.

The A and a *desc_codes* are useful for printing the elements of a linked list. For example, consider the following code, which defines

the structure for a symbol table item, and declares *sym_head* to be a pointer to this structure. The program that uses this structure and field will chain symbol table items together, and set a pointer to the head of the chain in *sym_head.*

```
struct symbol {
    struct symbol * sym_next;
    char *sym_name;
    unsigned sym_val;
} *sym_head;
```

The following command would display the symbol table item pointed at by *sym_head* and then set the *p* command's current address to the next symbol table item, which is pointed at by the *sym_next* field in the first item:

p @A"symbol name="#snt"value="x sym_head

After this command is entered, you can display successive symbol table items by simply entering

p

The *p* command's current address is correctly set to the next table item, and since a format string isn't specified, the *p* command will use the one that it last used.

You can print out multiple symbol table items by entering a single *p* command. To do this, place a comma and the maximum number of items to be printed after the command's starting address. The command will follow the chain, printing symbol table items until it either prints the specified number of items or it prints an item whose *sym_next* pointer is null. In the latter case, it will terminate and leave the *p* command's current address set to the address of the last symbol table item. For example, entering

p @A"symbol name="#snt"value="x sym_head,100

will print symbol table items until it either prints 100 items or it prints an item having a null *sym_next* pointer.

2.3.11 The Quit command

q - Quit the debugger

Syntax:

q

Description:

q terminates the program being debugged, restores any modified interrupt vectors, and returns control to the operating system.

2.3.12 The Register command

r - Register display

Syntax:

r
r <reg>=EXPR

Description:

r displays and modifies the registers, including the status registers, of the program being debugged.

The parameter-less version displays the registers.

The parameterized version modifies the contents of a register, with <reg> being the name of the register to be modified, and EXPR an expression whose resulting value is to be set into the register.

2.3.13 The Single Step commands

s - Single step into calls with display
S - Single step into calls without display
t - Single step over calls with display
T - Single step over calls without display

Syntax:

[#] s
[#] S
[#] t
[#] T

Description:

These commands 'single step' the user's program; that is, execute its instructions one by one.

The optional '#' parameter specifies the number of instructions to be executed; it defaults to one instruction.

The *s* and *S* commands differ in that *s* displays information after each single step, whereas *S* only displays information after the last single step. The same is true for the *t* and *T* commands.

The *s* and *t* commands differ in that the *s* commands single step into calls when encountered while the *t* command treats a function call as a single step and will step over it.

The displayed information consists of the source line of the next instruction to be executed.

When single-stepping, breakpoints aren't enabled.

2.3.14 The Unassemble commands

u - Unassemble memory, with symbols
U - Unassemble memory, without symbols

Syntax:

u RANGE
U RANGE

Description:

These commands 'disassemble' a range of memory; that is, display the assembly language instructions in the range.

Both the *u* and *U* commands make use of the symbol table during disassembly. They differ in that the *u* command will include the symbols+offset as part of the assembly language while the *U* command will print locations in hex with the sysmbol and offset for the location displayed at the far right of the line. Also, the *U* command displays, for each instruction, the hex value of each byte of the instruction, whereas the *u* command won't.

With the *u* command, the disassembly of an instruction which references memory displays the location as the symbol nearest to the location plus an offset, if possible. With the *U* command, the location is displayed as a hexadecimal value.

The RANGE parameter specifies the area of memory to be disassembled. It gives the starting address, and either the number of instructions to be disassembled, or the ending address of the area.

Both comands display the source lines and autos as comments.

2.3.15 The Variable command

v - Create a new symbol

Syntax:

v TYPE SYMBOL

Description:

The *v* command is used to create a new symbol.

TYPE is the SYMBOL's type such as int, short, char, etc. SYMBOL is the name of the symbol being created. The SYMBOL must be a unique symbol whose name is at least two characters in length. The naming of SYMBOL must follow the regular C declaration rules.

All symbols created using this command will be treated as globals.

2.3.16 The Macro commands

x - Create Macro command
X - Display Macro command

Syntax:

x macroname CMDLIST
X

Description:

The *x* command defines or executes a sequence of debugger commands, called a 'macro'. *X* lists the defined macros.

A macroname consists of a sequence of alphanumeric characters,the first of which must be an alphabetic, with a limit of 40 characters. Case is not significant.

A macro is defined by typing the letter 'x', followed by the name with which the macro is to be associated. Then follows the macro's list of debugger commands, with the commands separated by semicolons.

A macro is executed by typing 'x', followed by the name with which the macro is associated, followed by a carriage return.

The macros which have been defined can be listed using the command *X*.

2.3.17 The EXPRESSION Commands

= - Display the value of an expression in several formats
e - evaluate an expression

Syntax:

= EXPR
e EXPR

Description:

The = command displays the value of an expression.

The expression is displayed in several formats: hexadecimal, signed decimal, unsigned decimal, octal, binary, and ASCII. If a symbol table has been loaded, the closest symbol is displayed as well.

Some expressions involve a segment as well as an address. In this case, the segment is displayed at the beginning of the line, followed by a ':'.

The *e* command allows you to perform C expression evaluation, including calls to C functions, assignment, pre- and post-increment and decrement, casts, and conditionals. For example:

e c = getchar()

might result in

=10

2.3.18 The 'Redirect command input/output' Command

< - Redirect command input
<> - Redirect command input/output
> - Log all I/O
>> - Log commands only

Syntax:

< filename
<> device
> filename
>> filename

Description:

The < command causes the debugger to read and execute commands from the specified file. This command provides a convenient means for defining macros or variables.

When the end of the file is reached, the debugger returns to the console for commands.

The <> command causes the debugger to take its input from the device given.

The > command causes the debugger to log all input and output to the specified file. This feature allows you to keep a record of the commands and responses for a debugging session.

The >> command causes the debugger to log only commands to the specified file. This is useful if you wish to re-execute a sequence of commands to reproduce a problem.

2.3.19 The Help command

? - list commands

Syntax:

?

Description:

This command lists the debugger commands. For groups of related commands, the listing usually lists the first letter of the commands followed by a ?. You can get a listing of all the commands in such a group by typing the letter, the ?, and return. For example, the listing for the 'display' commands is *d?*; thus you can type *d?* followed by return to get a listing of all the 'display' commands.

2.3.20 The Change Mode command

z - change mode command

Syntax:

z

Description:

The *z* command allows you to switch from source mode to assembly mode and back again.

By default, the debugger starts in source mode unless you specify the *-a* option on the command line.

3. Command Summary

breakpoint commands

bm	set memory-change breakpoint
bc/bC	clear one/all breakpoints
bd	display the breakpoint table
br	reset the breakpoint counters
bs	set or modify a breakpoint
bt/bT	enable/disable trace mode

display commands

db/dw/d	display memory in bytes/words/last format
dc/dd	display code/data symbols
df	display source file lines
dg	display global values
dm	display memory map
dn	display 8087 status
ds/dS	display stack backtrace
c	display source context

find source string

/	find string in source file

change frame commands

fu	walk up the call frame
fd	walk down the call frame

go commands

g/G	execute user's program

port input commands

ib/iw	input byte/word from port

load commands

lp	load program

memory modification commands

mb/mw	modify bytes/words of memory
mc	compare areas of memory
mf	fill memory
mm	move memory
ms	search memory

port output commands

ob/ow	output byte/word to port

formatted print commands

p	generate formatted print

quit command

q	quit debugger

register command

r	register display

single step commands

s/S	single step with/without display, stepping into function calls
t/T	single step with/without display, stepping over function calls

unassembly commands

u/U	unassemble memory

variable command

v	create symbol

macro command

x/X	define or execute/display a command macro

expression commands

=	display value of an expression in several formats
e	evaluate an expression

redirect command input

<	redirect command input
<>	redirect command input/output
>	log all input/output
>>	log commands only

help command

?	list debugger commands

commands for screen saving and restoring

w	display saved screen (debug or program)
W	disable screen saving and restoring

change mode command

z	switch from source to assembly mode and back

ASSEMBLY LANGUAGE DEBUGGER

Chapter Contents

Assembly Language Debugger ..... db
1. Overview 5
1.1 Basic Commands 5
1.2 Names 5
1.2.1 Code and Data Symbols 6
1.2.2 Operator Usage of Names 6
1.3 Loading programs and symbols 6
1.4 Breakpoints 7
1.5 Memory-change breakpoints 8
1.6 Separate screens for programs and *db* 8
1.7 Trace mode 9
1.8 Backtracing 9
1.9 Macros 9
1.10 Displaying source files 9
1.11 Other features 9
2. Using DB 11
2.1 Starting DB 11
2.2 Commands 11
2.2.1 Definitions 11
2.3 Command descriptions 16
2.3.1 The BREAKPOINT (b) commands 16
2.3.2 The CLEAR (c) commands 19
2.3.3 The DISPLAY (d) commands 19
2.3.4 The 'Find source string' (f) command 23
2.3.5 The GO (g) commands 24
2.3.6 The INPUT (i) commands 25
2.3.7 The LOAD (l) commands 25
2.3.8 The MODIFY MEMORY (m) commands 27
2.3.9 The OUTPUT (o) commands 29
2.3.10 The PRINT (p) command 29
2.3.11 The QUIT (q) command 35
2.3.12 The REGISTER (r) command 36
2.3.13 The SINGLE STEP (s) commands 36
2.3.14 The UNASSEMBLE (u) commands 37
2.3.15 The VARIABLE (v) commands 37
2.3.16 The MACRO (x) command 38
2.3.17 The 'Display Expression' command 38
2.3.18 The 'Redirect command input' (<) command 39
2.3.19 The HELP (?) command 39
3. Command Summary 40

Assembly Language Debugger

This chapter describes the assembly language debugger, *db*, that is provided with some versions of Aztec C86. For a description of the source level debugger, *sdb*, see the Source Level Debugger chapter. For a description of other utility programs that are provided in some Aztec C86 packages, see the Utilities and Unitools chapters.

NAME

db - symbolic debugger

SYNOPSIS

db [progfile] [arg1 arg2 ...]

DESCRIPTION

db is used to debug programs which have been created using the Aztec C compiler, assembler, and linker.

db has all the standard features of an assembly language debugger. It also has features not found in all debuggers, such as the ability to reference memory locations by name as well as by address, the ability to define sequences of commands to be macros, which can then be activated by entering a single letter, and a flexible mechanism for handling breakpoints.

In addition, *db* has features specifically tailored to its use with Aztec C, such as the ability to list the name and parameters of the currently executing function, and the function that called it, and so on, back to the initial function. Another special feature is the ability to display, on entry and exit from each function, the function's parameters and return value.

Requirements

A minimum of 256 K bytes of RAM memory is recommended for use with *db*. The debugger itself uses about 96K.

db can only be used on 8086-and 8088-based systems running MSDOS or PCDOS, version 2.0 or later.

Preview

The remainder of this description of *db* is in three sections: *overview*, which describes *db* features in more detail and introduces the commands; *usage*, which describes in full detail how to use *db*; and a *command summary*.

1. Overview

db commands consist of one or two characters, the first of which identifies the command category. If there's only one command in the category, then the command has just this one letter; otherwise, the command has a second letter which identifies the specific operation to be performed.

1.1 Basic commands

db has two types of commands for examining memory: display and print, whose first characters are *d* and *p*, respectively. The 'display' commands *db* and *dw* simply display hexadecimal bytes and words.

The 'print' command, *p*, is more powerful, being able to convert a sequence of one or more possibly different types of data items to ASCII. For example, you can tell it that beginning at the location *var* are a sequence of the following items: an *int*, a *float*, and a pointer to a *char* string. The *p* command will convert the two binary items to ASCII and print them, and display the referenced character string.

The 'register' command, *r*, displays and modifies the 8086 registers.

The 'memory modify' commands, *m*, modify memory.

The *u* commands 'unassemble' code; that is, display it symbolically, in a form similar to its appearance in an assembly language source file.

The *s* and *g* commands cause the user's program to be executed. *s* commands "single step" the user's program; that is, execute a specified number of instructions in the user's program and then return control to *db*. *g* commands transfer control of the processor unconditionally to the user's program. In this case, *db* regains control when the user's program terminates, when an error occurs (such as division by zero), or when a "breakpoint" is taken. Breakpoints are discussed below.

? is the *help* command: it causes *db* to display a summary of all *db* commands. For some command categories, you can get information about the commands in a category by typing the first letter of the category's commands followed by a *?*. For example, typing *m?* gets you information about the memory modification commands (all of whose first letter is *m*).

1.2 Names

db allows memory locations to be referenced by name as well as by location. It learns a program's global names by reading the file containing the program's symbol table and placing them in a memory-resident symbol table. The linker generates a symbol table file for a program in response to the -T option. Ways of causing *db* to read the symbol table are described below.

db only allows global symbols to be accessed by name; automatic variables and static variables can't be accessed by name.

The operator can also define names to *db* using the *v* command, and the 'clear symbols' command, *cs*, will remove symbols from the memory-resident symbol table.

1.2.1 Code and Data symbols.

db classifies symbols as being either code or data symbols; that is, as referring to a location in a physical code segment or the physical data segment. All symbols in the program's symbol table file which occur between the special linker symbols __Corg__ and __Cend__ are considered to be code symbols, and all others are data symbols. This classification of symbols frees the operator from having to specify the physical segment in which a symbol is located, when using commands which reference the location; the debugger knows which segment it's in. The classification is also of use when unassembling memory.

There are two commands for viewing the symbols which are known to *db*: *dc* and *dd*, which display code and data symbols, respectively.

1.2.2 Operator usage of names.

When a C source program is compiled, all global symbols are truncated to a maximum of 31 characters and are then appended with an underscore character.

To refer to symbols which, in a C source file contain less than 31 characters, the inclusion of the appended underscore character is optional: if it's specified, *db* will search for just that symbol in it's symbol table. Otherwise, it will first search for the specified symbol; if the search fails, it will then append an underscore to the name and search again.

To refer to symbols which contain 32 or more characters, only the first 31 are significant; *db* ignores all other characters in the name.

1.3 Loading programs and symbols

A program and its symbols can be loaded into memory when *db* is started; in this case, the command line defines the program to be loaded. The *db* 'load program' command, *lp*, can also be used. When told to load a program, *db* automatically tries to load the program's symbol table, too; it assumes the symbol table file has the same name as the program file, with the extension changed to *.sym*.

When the symbol table file doesn't obey this convention, the 'load symbols' command, *ls*, can be used.

When the *lp* command finds a symbol table file, it clears the symbol table of all symbols, other than those defined with *v* commands, before loading the new symbols. The *ls* command, on the other hand, doesn't clear the symbol table first: it simply loads all the symbols. If it finds a symbol that is already in the table, it enters the new value and issues a warning message.

Only one user program can be in memory at once. If an *lp* command is entered before a currently loaded program has exited, the current program is terminated by the debugger before the new program is loaded.

When a program exits, it must be reloaded with the *lp* command before execution can begin again.

A memory map can be obtained using the *dm* command.

1.4 Breakpoints

Before transferring control of the processor to a user's program in response to a *g* command, *db* can set "breakpoints" at specified locations in the code. When the user's program reaches a breakpoint, *db* regains control.

A breakpoint has a 'skip count' associated with it, which allows a breakpoint to be passed several times before actually taking the breakpoint and returning control to *db* and the user. When a breakpoint is reached, *db* is always activated; it increments a counter associated with the breakpoint. When the counter's value is greater than the breakpoint's skip count, the breakpoint is taken; that is, *db* retains control of the processor. Otherwise, *db* returns control of the processor to the user's program after the breakpoint. By default, a breakpoint's skip count is 0; thus, each time the breakpoint is reached, it's taken.

A breakpoint can also have a sequence of *db* commands associated with it. When a breakpoint is taken, these commands will be executed before *db* allows the operator to enter commands. For example, if you just want to examine a variable each time a certain location in the code is reached and then have the program continue execution, you could define a breakpoint at the location, and specify a list of commands to do just that: the first command in the sequence would be a *d* command to display memory, and the second would be a *g* command to continue execution of the program.

There are two ways to define breakpoints: with the *g* command, and with special breakpoint commands, whose first letter is *b*.

The breakpoint commands manipulate a table of breakpoints: there are commands for entering breakpoints into the table, displaying the entries, reseting their counters, and removing them from the table.

There's a difference between a breakpoint defined in a *g* command and those in the breakpoint table: the *g* command breakpoint is temporary, while a breakpoint table is more permanent (it exists until removed from the table). Before transferring control to the user's program in response to a *g* command, *db* sets all breakpoints that are in the breakpoint table and that are specified in the *g* command itself. When a breakpoint is taken, *db* removes all breakpoints from the code

and forgets all about the *g* command breakpoint. The breakpoint table breakpoints, however, are still in the table and will be set back in memory when control is again returned to the user's program.

db remembers the skip counter associated with a breakpoint which is in the breakpoint table: when it sets breakpoints in memory, the count for such a breakpoint is set to its remembered value (that is, its value in the table); and when a breakpoint is taken, the accumulated count for the breakpoints in memory are saved in the breakpoint table.

1.5 Memory-change breakpoints

The breakpoints described above are taken when a program reaches a specified point in the code. A second type of breakpoint, called a memory-change breakpoint, is taken when a specified memory location is changed from or set to a particular value.

With a memory-change breakpoint set, *db* will detect either the function or the instruction which modifies the specified memory location, depending on whether the user's program was activated using a *g* command or is being single-stepped using an *s* command, respectively.

When the user's program is activated with a *g* command and a memory-change breakpoint is set, *db* will examine the specified memory location on entry to, and exit from, each function. It will take a breakpoint, that is, interrupt execution of the program and return control to the operator, when the contents of the memory location meets the specified condition.

When an *s* command is used to single-step a program and a memory-change breakpoint is set, *db* will examine the specified memory location after each instruction is executed, and take a breakpoint when appropriate.

The *bb* and *bw* commands are used to set and remove memory-change breakpoints.

1.6 Separate screens for programs and *db*

When used on an IBM PC or an equivalent, *db* can optionally maintain separate screens for itself and for a program that is being debugged. With this feature enabled, a program-generated screen is displayed while a program is executing, and a screen of operator-*db* interactions is displayed while *db* is executing.

This feature is implemented as follows: when a program encounters a breakpoint, *db* saves the contents of the screen and displays the debug screen; similarly, when *db* continues a program, it saves the debug screen and restores the program screen.

To enable this feature, specify the *-w* option when you start *db*. This option must precede the name of the program that is to debugged.

Two *db* commands are related to separate program/debug screens:

* The *w* command causes *db* to toggle between displaying the debug and program screen.
* The *W* command disables screen saving and restoring.

1.7 Trace mode

db supports a 'trace mode', which displays information whenever a function is entered or exited.

With this mode enabled, on entry to a function, the function name and its arguments are displayed, and, optionally, on exit from a function, its return value is displayed.

The commands *bt* and *bT* affect trace mode: *bt* enables and disables trace mode, and *bT* enables and disables the display of function exit information.

1.8 Backtracing

When *db* regains control from an executing program (for example, because a breakpoint was taken), it has the ability to display information on how the program got to its current location: the *ds* command will display information about the currently executing function, and the function which called it, and so on, back to the Manx function *Croot*, which called the user's function *main*.

ds displays, for each function, its name, arguments which were passed to it, and the address to which it will return.

1.9 Macros

db allows the user to define and excute 'macros'; that is, a sequence of *db* commands.

A macro is associated with a single alphabetical character, so up to 26 macros can be known to *db* at any time.

The *db* command *x* is used both to define and execute a macro.

1.10 Displaying source files

db allows the user to display source files, thus providing a convenient means to examine the source of a program being debugged.

Only a single source file can be examined at a time. The 'load source file' command, *lf*, defines the source file to be displayed, and the 'display source lines' command, *df*, displays its lines.

The 'find string' command, *f*, will find a character string in the source file.

1.11 Other features

Some other features of *db* which haven't yet been discussed are:

* The 'input' and 'output' commands, *i* and *o*, will transfer data to and from an i/o port;
* The 'redirect stdin' command, <, causes *db* to read commands from a specified device or file and then continue reading commands from the console;
* The 'evaluate expression' command, =, does just that.
* The 'help' command, *?*, lists commands.

2. Using DB

2.1 Starting DB

db is started with a command of the form:

db [progfile] [arg1 arg2 ...]

where the optional parameter *[progfile]* is the name of a file containg a program to be debugged, and the optional parameters *arg1*, *arg2*, ..., are character strings to be passed to the program.

The extension on the program filename is optional; if not specified, a search is first made for a file with the extension *.exe*. If that search fails, another is made with the extension *.com*.

db looks for a period, '.', to decide whether the program file name contains an extension. Thus, to specify a program which doesn't have an extension, include a period at the end of the file name.

If the program file name specifies a drive or directory, *db* searches for the program file in just that particular area. Otherwise, it searches the current directory on the default drive.

The "arg" parameters are passed to the program using the *argv* parameter of the program's *main* function: *arg1* is pointed at by *argv[1]*, *arg2* by *argv[2]*, and so on. *argv[0]* always contains zero.

2.2 Commands

This section describes in detail the *db* commands. It first defines some terms that are used in the command descriptions. These terms are *expr*, *term*, *addr*, *range*, and *cmdlist*.

2.2.1 Definitions

2.2.1.1 The Definition of EXPR

An EXPR has the following form:

TERM [binop TERM ...]

That is, an EXPR can be a single TERM or a series of TERMS separated by binary operators. The binary operators are:

+ -	addition
- -	subtraction
* -	multiplication
/ -	division
% -	modulus
& -	bitwise and
| -	bitwise inclusive or
^ -	bitwise exclusive or

All operators have the same precedence, and an unparenthesized EXPR is evaluated left to right. If you want to override the default

order of evaluation of an expression, you can parenthesize the relevant parts of the expression.

An EXPR has a 16-bit value. The operators that are applied to the TERMS out of which the EXPR is built affect just this 16-bit value.

When an EXPR refers to a memory location (that is, it is built up from an ADDR), the 16-bit value is the offset of the location from the beginning of the segment containing it. In this case, the EXPR can also specify the beginning paragraph number of the segment containing the location. For more discussion about this, see the description of ADDR below.

2.2.1.2 The Definition of TERM

A TERM always resolves to a numeric value, and can be one of the following:

REGISTER
CONSTANT
-TERM
ADDR
*ADDR
#ADDR
.
@ [function]
(EXPR)

These names are defined in the following paragraphs.

REGISTER

Registers are specified by their standard names; that is, AX, BX, and so on. The value of the TERM is the contents of the register.

CONSTANT

A CONSTANT can be a decimal, hexadecimal, or octal number, or a character.

You can explicitly define the radix of a numerical constant as follows:

* A sequence of digits preceded by '*0x*' is taken to be a hexadecimal number.
* A sequence of digits preceded by '*0o*' is taken to be an octal value.
* A sequence of digits terminated by a decimal point is taken to be a decimal value.

When the radix of a number isn't explicitly specified, *db* assumes that it uses the current default radix. By default, the default radix is

decimal. It can be changed using the *n* command, as follows:

command	*default radix*
nx	hexadecimal
no	octal
nd	decimal

A character is represented by the character, surrounded by single quotes, as in *'x'*. The value of a character constant is its ASCII value.

Certain characters, the single quote ', and the backslash \ may also be defined within the single quotes. These are identified by a leading backslash character, and are:

char	*hex value*	*db notation*
newline	0a	\n
horizontal tab	09	\t
backspace	08	\b
carriage return	0d	\r
form feed	0c	\f
backslash	5c	\
single quote	27	\'
bit pattern	ddd	\ddd

ADDR

A TERM can be an ADDR; that is, a reference to a location in memory. See the definition of ADDR, below, for more details.

***ADDR**

When a TERM consists of a * followed by an ADDR, the value of the TERM is the contents of the 16-bit field referred to by the ADDR. For example,

*VAR	The contents of the VAR field;
*BX	The contents of the 16-bit field in the data segment pointed at by BX;
*SP	The contents of the 16-bit field on the top of the stack;
*(LBL+2)	The contents of the 16-bit field referred to by LBL+2;

Because an ADDR can itself be an EXPR, the *ADDR term may require extra parentheses. For example,

```
*sp+2
```

is equivalent to *(sp+2) and not (*sp)+2. The value of the first interpretation is the contents of the second word on the stack, while the value of the second is two plus the contents of the first word on the stack.

#ADDR

The value of #ADDR is the contents of the doubleword field referenced by ADDR. The contents of the most significant word becomes the segment component of the resulting TERM, and the contents of the other word becomes the offset value of the TERM.

period(.)

The value of a TERM consisting of a period, '.', is the starting address ADDR of the last similar command. For example, if ten bytes of memory were displayed using the *db* command, as in

```
db ds:0x100,10
```

then '.' would be set to ds:0x100 for the next db or dw command. If the next db or dw command is

```
dw .
```

the same 10 bytes would be displayed as words.

The '.' has a separate value for the *u* command, for the *db*, *dw*, and *m* commands, for the *p* command, and for the *df* command. An *m* command never modifies its associated '.'.

@ [function]

The @ symbol has as its value the return address of the specified function. The function name is optional, and defaults to the current function. The main use for @ is in the *g* command.

For example,

```
g @
```

transfers control to the user's program, and sets a breakpoint at the return address of the current function.

As another example,

```
g @putc
```

transfers control to the user's program. When the function *putc* is reached, a breakpoint will be set at the address to which it will return.

2.2.1.3 The Definition of ADDR

An ADDR defines the address of a location in memory, and has the form:

```
[EXPR:]EXPR
```

The rightmost EXPR is the offset of the location from the beginning of the segment containing it.

The optional EXPR: defines the beginning paragraph number of the segment containing the location; if not specified, the paragraph

number will be determined as follows:

* If the offset uses a symbol, the paragraph number is the segment component of the symbol name;
* If the offset uses the SP or BP registers, the paragraph number is the contents of the SS register;
* If the offset uses the IP register, the paragraph number is the contents of the CS register;
* If the offset uses any other register, the paragraph number is the contents of the DS register.

For cases not covered by the above, the segment paragraph number will default to the last segment number used, or, if none, the default segment number for the particular command being executed.

Here are some examples of ADDR:

cs:ip	Address of program counter;
ip	Equivalent to cs:ip;
cs:main+10	
main+10	
ds-16:0	The beginning of the segment 256 bytes before DS;
(*sp+10):(*sp+8)	A reference to a location whose segment paragraph number and offset are on the stack;
#sp+8	Same as the above
data+*(bp+6)	

2.2.1.4 The Definition of RANGE

A RANGE defines a block of memory. It has one of the following forms:

```
ADDR,CNT
ADDR>ADDR
ADDR
,CNT
```

The form ADDR,CNT specifies the starting address, ADDR, and a number, CNT. CNT is interpreted differently by different commands. For example, the 'disassemble code' command, *u*, will display CNT lines, while the 'display bytes' command, *db*, will display CNT bytes.

The form ADDR>ADDR specifies the starting and ending addresses of the range.

A full range need not be explicitly specified, because *db* remembers the last-used range and will set unspecified RANGE parameters from the remembered values:

* When a RANGE is specified which consists of a single ADDR, the last used CNT is used.
* When a RANGE is specified which consists of ',CNT', the next consecutive address is used, and the remembered count is changed to the new value.
* When nothing is specified as the RANGE, the next consecutive address is used as the starting ADDR, and the CNT is set to the remembered value.

2.2.1.5 The Definition of CMDLIST

A CMDLIST is a list of commands. It consists of a sequence of commands or macros separated by semicolons:

COMMAND [;COMMAND ...]

If a macro is in a CMDLIST, it must be the last command in the list.

2.3 Command descriptions

The following descriptions of debugger commands uses terms and concepts which were presented in the preceding sections.

The commands are listed alphabetically. For an index, see the command summary which follows the descriptions.

2.3.1 The Breakpoint Commands

bb - Set Byte Memory-Change Breakpoint
bw - Set Word Memory-Change Breakpoint

Syntax:

bb
bw
bb ADDR == [VAL]
bb ADDR != [VAL]
bw ADDR == [VAL]
bw ADDR != [VAL]

Description:

These commands are used to set and clear a memory-change breakpoint, with the parameterized versions used to set breakpoints and the parameter-less version to clear them. The *bb* command is used to monitor a one-byte field, and the *bw* command to monitor a two-byte (word) field.

In the parameterized form of the commands, ADDR specifies the field to be monitored.

With the '==' form, the breakpoint will be triggered when the debugger detects that the field is equal to the specified value, VAL.

With the '!=' form, the breakpoint will be triggered when the debugger detects that the field is different from the specified value.

The VAL parameter is optional. If not specified, it defaults to the current value at the ADDR.

Before memory-change breakpoints can be set in a program, control must be passed to the program, using a *g* command. For example, you could load the program, then enter

```
g main
```

to start it executing and then return control to the debugger, and then set the memory-change breakpoints.

The reason for this is that when a program is loaded its data segment register, DS, doesn't point to the program's data segment. Hence, any memory-change breakpoints which are set before DS is initialized are set to incorrect addresses. The initialization of DS is done by the Manx functions which execute before control is passed to the user's *main* function, so memory-change breakpoints can safely be set once control reaches *main*.

---

bc - Clear a single breakpoint
bC - Clear all breakpoints

Syntax:

bc ADDR
bC

Description:

These commands delete breakpoints from the breakpoint table.

bc deletes the single breakpoint specified by the address ADDR, and *bC* deletes all breakpoints from the table.

---

bd - Display breakpoints

Syntax:

bd

Description:

bd displays all entries in the breakpoint table.

For each breakpoint, the following information is displayed:

* Its address, using a symbolic name, if possible;

* The number of times it's been 'hit' without a breakpoint being taken.
* The skip count for it;
* The command list for it, if any.

For example, a *bd* display might be:

address	hits	skip	command
cs:printf__	1	2	
cs:putc__	0	0	db ds:__Cbuffs

In this example, two breakpoints are in the table. The first is at the beginning of the function *printf__*; a breakpoint will be taken for it every third time it is reached, and no command will be executed. Given its current hit count, a breakpoint will be taken the next time *printf__* is reached.

The second breakpoint is at the function *putc__*; a breakpoint will be taken each time the function is reached, and will display memory, in bytes, starting at ds:__Cbuffs.

---

br - Reset breakpoint counters

Syntax:

br [ADDR]

Description:

br resets the 'hit' counter for the specified breakpoint which is at the address, ADDR. If ADDR isn't given, the 'hit' counters for all breakpoints in the breakpoint table are reset.

---

bs - Set or modify a breakpoint

Syntax:

[#] bs ADDR [;CMDLIST]

Description:

bs enters a breakpoint into the breakpoint table, or modifies an existing entry.

The optional parameter # is the skip count for the breakpoint. If not specified, the skip count is set to 0, meaning that each time the breakpoint is reached it will be taken.

The optional parameter CMDLIST is a list of debugger commands to be executed when the breakpoint is taken.

---

bt - Toggle the trace mode flag
bT - Toggle the return trace mode flag

Syntax:

bt
bT

Description:

bt and *bT* toggle the trace mode and return trace mode flags, respectively.

The state of the trace mode flag determines whether trace mode is enabled or disabled.

The state of the return trace mode flag determines whether the tracing of a function's return is enabled or disabled. If trace mode is disabled, the return trace mode flag has no effect.

2.3.2 The Clear Commands

cs - Clear symbol table

Syntax:

cs

Description:

cs removes all symbols from the debugger's memory-resident symbol table, except for symbols which the operator has entered using the *v* command.

When a program is loaded using the *lp* command, the *cs* command is automatically called.

2.3.3 The Display Commands

db - Display memory in bytes
dw - Display memory in words
d - Display memory in last format

Syntax:

db [RANGE]
dw [RANGE]
d [RANGE]

Description:

The *db* and *dw* commands display successive bytes and words of memory, respectively. *d* displays memory using the last format specified; for example, if *d* is entered, and *db* was the last 'display memory' command, then *d* will display bytes, too.

The starting address of the RANGE parameter is optional; if not specified, it defaults to the ending address of the last display's RANGE, plus one.

Each line of the display begins with the segment and address, followed by a hexadecimal display of 16 bytes or 8 words, followed by an ASCII display, by bytes, of the same data. For the ASCII display, values falling outside the range 0x20 to 0x7f are displayed as a period.

If the ending address does not fall on a multiple of 16 bytes, only the number of bytes or words specified in the last line will be displayed.

---

dc - Display all code symbols

Syntax:

dc

Description:

dc lists all the code symbols in the memory-resident symbol table and all user-defined symbols.

For each symbol, its name and address are displayed.

---

dd - Display all data symbols

Syntax:

dd

Description:

dd lists all the data symbols in the memory-resident symbol table.

For each symbol, its name and address are displayed.

---

df - Display source file lines

Syntax:

df [RANGE]

Description:

df displays lines from the source file which was specified in the last *lf* command.

The RANGE parameter specifies the numbers of the lines to be displayed.

The starting line number is optional; if not specified, the display starts with the "current" line.

The current line in a source file is set by the source file commands *lf*, *df*, and *f*, as follows:

* When the file is first loaded with the *lf* command, the first line in the file is the current line;
* When the last source file command was 'display source', *df*, the current line is the line following the last one displayed;
* When the last source file command was 'find string', *f*, the current line is the line in which the string was found.

df also sets the "F-dot" for the source file to the number of the first line displayed. The F-dot is the line referred to when the starting line number of the range in a *df* command specifies a period (.). Also, source string searches begin at the line following the F-dot line.

Each displayed line is preceded with a line number in decimal, a colon, and the line itself.

---

dg - Display global values

Syntax:

dg

Description:

For each data symbol in the debugger's symbol table, *dg* displays the contents of the 16-bit field referenced by that symbol.

---

dm - Display memory map

Syntax:

dm

Description:

dm displays a memory map of allocated memory, beginning with the debugger program itself.

An entry gives information about one allocated block of memory. It has the following form:

xxxx: owner=yyyy size=zzzzz

where *xxxx* is the paragraph number at which the block begins,

yyyy is the Program Segment Prefix of the process that owns the block, and *zzzzz* is the number of bytes in the block, in decimal.

For example:

```
10cf: owner=10d0 size=85264
25a1: owner=25aa size=112
25a9: owner=25aa size=69168
368d: owner=0000 size=38688
```

The debugger occupies the block that begins at paragraph 0x10cf; the Program Segment Prefix for *db* begins at paragraph 0x10d0; the number of bytes in *db*'s block is 85264. The environment of the program being debugged begins at paragraph 0x25a1, and the Program Segment Prefix for the program begins at paragraph 0x25aa. The program itself occupies the block beginning at paragraph 0x25a9. Memory beginning at paragraph 0x368d is unallocated.

Note that both the block containing program and the block containing its environment are owned by the PSP of the program.

---

dn - Display 8087 State

Syntax:

dn

Description:

dn displays the state of the 8087, in the following format:

```
cccc    ssss    tttt
iilo    jjjj    oolo    oohi
st(0): top stack element
st(1): next stack element
...
st(7): last stack element
```

where

* *cccc* is the control word
* *ssss* is the status word
* *tttt* is the tag word
* *iilo* is the least significant 16 bits of the instruction pointer
* *jjjj*, most significant 4 bits are the most significant 4 bits of the instruction pointer and the least significant 11 bits are the opcode
* *oolo* is the least significant 12 bits of the operand pointer
* *oohi* has as its most significant four bits the most

significant four bits of the operand pointer, and its least significant 12 bits are 0.

* *st(0)*, ... are the contents of the stack registers.

All values are in hexadecimal.

---

ds - Display Stack Backtrace

Syntax:

ds

Description:

ds displays information about the current function, the function whic called it, and so on, back to Croot, the Manx function which called the user's function *main.*

For each function, the information consists of the function's name, the parameters passed to it, and the address to which it will return.

The arguments are displayed as a series of 16-bit hex values. If an argument is actually of type long or double, it will be displayed as separate words.

ds determines the number of parameters by looking at the instructions which follow the address to which the function will return.

ds assumes that the BP register points to the C stack frame for the current function, unless the current instruction is within 6 bytes of the start of the function.

2.3.4 The 'Find Source String' Command

f - find string in source file

Syntax:

fSTRING

Description:

This command searches the current source file (that is, the one specified by the last *lf* command) for a specified string.

The search begins at the line following the current line.

If the string is found, the current line and the F-dot line of the source file is set to the line containing the string; otherwise, these values are unchanged.

STRING is the character string to be located, and consists of all characters following the *f* and preceeding the carriage return.

If the first character of the string is '^', the search will begin with the first character on a line. In this case, '^' isn't part of the search string.

The 'current line' and F-dot line for a source file are defined in the description of the *df* command.

2.3.5 The Go commands

g - Execute the program
G - Execute the program, without setting table breakpoints

Syntax:

[#]g [@ <function>] [ADDR] [;CMDLIST]
[#]G [@ <function>] [ADDR] [;CMDLIST]

Description:

The *g* commands transfer control of the processor to the user's program, at the address specified by CS:IP. The user's program then executes until it terminates, an error such as division by zero occurs, or a breakpoint is taken; control then returns to the debugger program.

The parameters to the 'g' commands allow one or two temporary breakpoints to be set in memory before the user's program is executed.

The difference between the 'g' and the 'G' command is that the 'G' command sets in memory just the breakpoints specified in the command itself, while the 'g' command also sets the breakpoints specified in the breakpoint table.

The '#' and 'ADDR' parameters define one of the temporary breakpoints that a Go command can set:

* # is the skip count for the breakpoint; it defaults to zero, meaning that the breakpoint is taken every time it's reached;
* ADDR is the address for the breakpoint;

The '@ <function>' parameter specifies that a temporary breakpoint is to be set at the return address of the specified function. If the function isn't specified, it defaults to the current function. If a function is specified, the breakpoint is set to the address to which the function will return. In this case, the breakpoint isn't set until the function is entered; thus, in programs which call the function from several different places, the breakpoint will be set at the actual address to which the function will return.

The ';CMDLIST' parameter defines a sequence of debugger commands, separated by semicolons, that the debugger is to

execute once a breakpoint which is specified in the 'go' command is taken. If this parameter isn't specified, it defaults to the command list used for the last temporary breakpoint.

If *db* is maintaining separate screens for the program and itself, it restores the program screen before transferring control to the program. For more information on this, see the discussion of separate screens in section 1 of this *db* documentation.

Before setting breakpoints and transferring control to the user's program, the debugger single-steps the user's program, (that is, causes it to execute one instruction). This allows the operator to transfer control to a location in the program at which there is a breakpoint, without immediately triggering a breakpoint and re-entry to the debugger.

2.3.6 The Input Commands

ib - Input byte from port
iw - Input word from port

Syntax:

ib PORT
iw PORT

Description:

ib and *iw* input a byte or word, respectively, from the i/o port, PORT.

2.3.7 The Load Commands

lf - Load a source file

Syntax:

lf filename

Description:

lf opens the specified source file for subsequent examination by the *df* command.

If a file has already been opened by a previous *lf*, it's closed before the new file is opened.

---

lp - Load program

Syntax:

lp
lp progfile [arg1 arg2 ...]

Description:

lp loads a program into memory. If a symbol table file can be found for the program, it will be loaded, too.

If the *lp* command is given without parameters, the last *lp* command is re-executed. The following comments describe the parameterized version of *lp*.

Loading the program

The parameter *progfile* specifies the file containing the program. The extension on the file name is optional.If not given, a search is made for the file with extension ".exe"; if this fails, another is made for the file with extension ".com". To specify a program file which doesn't have an extension, use a period after the filename.

If the program file name specifies a drive or path, the file is searched for in just that location; otherwise, it's searched for on the current directory of the default drive.

If an attempt is made to load a program before a currently loaded program has terminated, the current program will be terminated by the debugger before the new program is loaded.

Loading the symbol table

After the program is loaded, the memory-resident symbol table is cleared of all symbols except for those defined with the *v* command, and an attempt is made to locate and load the program's symbol table. The name of the file containing the symbol table is assumed to be the same as the program file name, with the extension changed to ".sym".

If the symbol table is in a file not obeying this convention, it can be explicitly loaded using the 'load symbols' command, *ls*.

And then...

After the program is loaded, its program segment prefix (PSP) is initialized with the arguments *arg1*, *arg2*, etc. These are available to the program as the *main* function's arguments *argv[1]*, *argv[2]*, and so on. *argv[0]* is set to 0, and *argc* is set to the number of "arg" parameters.

The U-dot (that is, the value of the period parameter associated with the *u* commands) is set to CS:IP. The D-dot (the value of the period parameter for the *d* and *m* commands) is set to DS:0.

Once a program exits, it must be reloaded with an *lp* command before it can begin again.

---

ls - load symbols

Syntax:

ls <filename>

Description:

ls loads symbols from the specified symbol table file into the debugger's memory-resident symbol table, after first clearing the memory-resident table of all but those symbols defined with the *v* command.

The parameter *<filename>* specifies the file to be loaded. The extension on this name is optional; if not given, it's assumed to be ".sym". To specify a file which doesn't have an extension, include a period at the end of the name.

If the file name gives a drive or path, it's searched for just on the specified area. Otherwise, it's searched for on the current directory on the default drive.

2.3.8 The Memory Modification Commands

mb - Modify bytes of memory
mw - Modify words of memory

Syntax:

mb ADDR EXPR1 [EXPR2 ...]
mw ADDR EXPR1 [EXPR2 ...]

Description:

db and *dw* modify bytes and words of memory, respectively.

The parameter ADDR specifies the address of the first byte or word to be modified.

The EXPR parameters are expressions, whose resulting values are set in memory, with EXPR1 set in the first byte or word specified, EXPR2 set in the next higher byte or word, and so on.

The EXPR parameters can be separated by spaces or commas.

---

mc - Compare memory

Syntax:

mc RANGE = ADDR

Description:

mc compares two blocks of memory and, for each comparison which fails, displays the corresponding segment, address, and

value.

RANGE specifies one of the blocks of memory. The second begins at ADDR and has the same length as the first block.

---

mf - Fill memory

Syntax:

mf RANGE = EXPR

Description:

mf sets each byte in a block of memory to a specified value.

The RANGE parameter specifies the memory block, and EXPR an expression whose resulting value is the value to be set in the range.

---

mm - Move memory

Syntax:

mm RANGE = ADDR

Description:

mm copies one block of memory to another.

The RANGE parameter specifies the source block and ADDR the starting address of the block to be modified.

---

ms - Search memory

Syntax:

ms RANGE = EXPR1 [EXPR2 ...]

Description:

ms searches a block of memory for a sequence of bytes having specified values. For each match, the corresponding address of the start of the string is displayed.

RANGE specifies the block of memory. The EXPR parameters are expressions, each of whose resulting values is one byte of the search sequence.

2.3.9 The Output Commands

ob - Output byte to i/o port
ow - Output word to i/o port

Syntax:

ob PORT=EXPR
ow PORT=EXPR

Description:

ob and *ow* output a byte or word, respectively, to an i/o port.

PORT is the address of the port. EXPR is an expression whose resulting value is the value to be output.

2.3.10 The 'Print' Command

p - formatted print

Format:

p[format] [ADDR][,COUNT]

Description:

p generates a formatted display of memory of a section of memory, by converting data items in memory to a displayable form as directed by the format conversion string *format.*

format is a list of format specifications, each of which defines the type of a data item and the conversion to be performed on it.

p works its way through the *format* string, converting and displaying data items in memory as requested by the format string items. When *p* reaches an item in the *format* string, it converts the data item at its 'current address' as directed by the format item. When it finishes processing a format string item, it increments its current address by the size of the data item that it just processed, so as to be ready to process the next data item as directed by the next format string item.

The *format* string is optional; if not specified, the *format* string used by the previous *p* command is used.

ADDR specifies the address of the first data item that *p* is to convert and display. If ADDR is not entered, the starting address is assumed to be the print command's 'current address'. Normally, this is the address of the first byte beyond the last data item converted by the last *p* command. However, there is a *format* item that causes *p* to remember the address contained in the current data item, and then make that the current address after it finishes processing the entire format string.

COUNT specifies the number of times that *p* is to work its way through the *format* string. Each time through, *p* begins at the current address that was left by the last time through. If COUNT isn't specified, it defaults to one time.

The format items have the form

[rpt][indir_flgs][size]desc_code

where

* *desc_code* is a single-letter code that defines the type of the data item and the conversion to be performed upon it. For example, the code *d* says 'take the two-byte binary value at the current address, convert it to decimal, and print it'. So if *var* is an *int*, the following command could be used to print its value in decimal:

  pd var

  The code *x* says "take the two-byte binary value at the current address, convert it to hexadecimal, and print the result". So the hexadecimal value of *var* could be printed with the command:

  px var

* *indir* is a string of zero or more * and/or # characters, which are indirection indicators specifying that the value at the current data item is a pointer to a chain of zero or more pointers, the last of which points to an object whose type and requested conversion are defined by *desc_code*.

  To find the data object corresponding to a format item that has indirection indicators, *p* begins by setting its idea of the address of the data object to the current address. It then works its way from left to right through the indirection indicators; for each indicator it replaces its current idea of the data object address with the pointer that is in the field at this address. The data object address is distinct from the current address: at the end of this process, the *p* command's current address is simply incremented past the first pointer.

  A * specifies that the pointer within the field referenced by the current data object address is two bytes long. This pointer is the offset component of the new data object address from the last segment referenced.

  A # specifies that the pointer within the field referenced by the current data object address is four bytes long. The most significant word of this pointer is

the segment component of the new data object address, and the least significant word is its offset component.

For example, if the variable *cp* is a short pointer to a character string (that is, its declaration is *char *cp*), then the string pointed at by *cp* could be printed by the command

```
p*s cp
```

Here we have made use of the *s desc_code*, which specifies that the data object is a character string, and that the string's characters are to be printed, with possible modifications as noted below, up to a terminating null character. After this command, the *p* command's current address is set to the byte immediately following *cp*.

As another example, if a module uses the 'large data' memory model (that is, its pointers to data objects are four bytes long), and if *cpp* is a pointer to an array of pointers to character strings (that is, the declaration of *cpp* is *char **cpp*), then the string pointed at by the first element of the array could be displayed with the command

```
p##s cpp
```

Following this command, the *p* command's current address is set to the byte following *cpp*.

* The *rpt* parameter of a format item defines the number of times that the item is to be processed. It allows a sequence of *rpt* identical format items to be abbreviated by just one such item with a leading *rpt* count.

  For example, if *a* is an array of *float*s, then the first five items in this array could be displayed with the command

  ```
  p5f a
  ```

  This command uses the fact that the *desc_code* to convert a four-byte floating point value at the current address to a displayable value is *f*. This command is equivalent to the command *pfffff a*. At the end of this command, the *p* command's current address is set to the address of the byte following the last displayed *float*.

* The *size* parameter of a format item defines the number of data items that are to be converted and printed. When the format item doesn't use indirection, *size* has the same effect as *rpt*; for example, in the *p5f a* command above, the 5 could be interpreted as being a

size parameter instead of a *rpt* parameter.

When the format item does use indirection, then the *size* parameter defines the number of data items to be converted and printed at the end of the indirection chain. For example, if a module using the 'small data' memory model defines *lpp* as a pointer to an array of pointers to *long*s (that is, the declaration of *lpp* is *long **ip*), then the following command would display the first four *long*s pointed at by the first element of the pointer array:

```
p**4D lpp
```

Here we have used the *D desc_code*, which specifies that a four-byte signed binary value is to converted to decimal and printed. The following command would display the first three *long*s pointed at by the first three elements of the pointer array:

```
p3*4D *lpp
```

To demonstrate further the difference between the *rpt* and *size* fields in a format item, consider the format items *4*d* and **4d*. The first causes the print command to take the item at the current address as a short pointer, increment the current address by two, convert to decimal and print the two-byte value referenced by the pointer, and then repeat the process three more times. At the end of the process, the current address has been advanced by eight.

The second item causes the print command to again take the item at the current address as a short pointer, increment the current address by two, and then convert to decimal and print the four successive two-byte values that begin at the address defined by the pointer. At the end of the process, the current address has been advanced by two.

As an example of the use of format strings containing several format items, consider the following code in a program that uses short data pointers:

```
struct {
   int *ip;
   float flt;
   char *cp;
} var = {&i, 3.14159, "ralph"};
int i=2;
```

The command

```
p*d2-xf*s2-x var
```

will print

2 xxxx 3.14159 ralph

where *xxxx* is the hexadecimal address of *i* and *yyyy* is the hexadecimal address of the string.

A complete list of the *desc_codes*

We have introduced some of the *desc_codes* above. Here is a list of the basic *desc_codes*:

b	Convert to hexadecimal and print a byte.
d	Convert to decimal and print a two-byte signed binary value.
D	Convert to decimal and print a four-byte signed binary value.
f	Convert and print a four-byte *float*.
F	Convert and print an eight-byte *double*.
.	Define the precision to be used in the display of floating point values; that is the number of digits to be displayed to the right of the decimal point. This number precedes the period. For more information, see below.
o	Convert to octal and print a two-byte field.
O	Convert to octal and print an eight-byte field.
x	Convert to hexadecimal and print a 2-byte field.
X	Convert to hexadecimal and print a 4-byte field.
u	Convert to decimal and print an unsigned, two-byte value.
U	Convert to decimal and print an unsigned, four-byte value.
p	Print a short, two-byte pointer in segment:offset form.
P	Print a long, four-byte pointer in segment:offset form.
c	Print a character.
C	Print a character.
s	Print a string up to a terminating null byte.
S	Print a string up to a terminating null byte.

Codes for printing floating point numbers

When a floating point number is displayed in response to an *f* or *F* *desc_code*, the 'precision' of the display (that is, the number of digits displayed to the right of the decimal point) is determined by the most recently-entered period *desc-code*; if a period code hasn't been entered, the precision of the display defaults to 7 digits for floats and 16 for doubles.

The period *desc_code* consists of a period preceded by the number of digits of precision. For example, the following command contains two *desc_codes*: the first sets the precision to 2 digits; the second then displays a float, listing two digits to the right of the decimal point.

p2.F

Codes for printing characters

For the C and S *desc_codes*, each character is printed "as is", with no translations.

For the c and s codes, printable ASCII characters (that is, whose hex value is between 0x20 and 0x7f) are printed "as is". A character whose hex value is less than 0x20 is printed as two characters: ^ followed by the printable character whose hex value equals the original character's value plus 0x40. A character whose hex value is 0x80 or greater is displayed as a ' character followed by the one or two characters that would be printed for the character whose hex value equals that of the original character less 0x80. For example, 0x41, 0x1, and 0x81 would be printed as A, ^A, and '^A, respectively.

Special purpose codes

The following *desc_codes* can be used to assist in the formatting of the *p* output:

character	*output*
N or n	Output a newline character
R or r	Output a blank character
T or t	Output a tab character
"string"	output "string"

These characters can be preceded by a count specifying the number of characters or strings to be output.

Codes for setting the *p* command's current address

The next group of *desc_codes* change the *p* command's notion of the current address. They don't cause any printing.

^ Back up the current address by the size of the last data item.

- or + Back up or advance, respectively, the current address by *size* bytes, where *size* is a decimal value preceding the - code. If *size* isn't specified, it defaults to one byte.

A or a Remember the long or short pointer, respectively, that is contained in the current data object; If this pointer is not null, set the *p* command's current address to this value after the entire format string has been processed.

If the pointer is null, set the *p* command's current address to the value it had before the entire format string was processed.

The A and a *desc_codes* are useful for printing the elements of a linked list. For example, consider the following code, which defines

the structure for a symbol table item, and declares *sym_head* to be a pointer to this structure. The program that uses this structure and field will chain symbol table items together, and set a pointer to the head of the chain in *sym_head.*

```
struct symbol {
   struct symbol * sym_next;
   char *sym_name;
   unsigned sym_val;
} *sym_head;
```

The following command would display the symbol table item pointed at by *sym_head* and then set the *p* command's current address to the next symbol table item, which is pointed at by the *sym_next* field in the first item:

```
pA"symbol name="#snt"value="x sym_head
```

After this command is entered, you can display successive symbol table items by simply entering

```
p
```

The *p* command's current address is correctly set to the next table item, and since a format string isn't specified, the *p* command will use the one that it last used.

You can print out multiple symbol table items by entering a single *p* command. To do this, place a comma and the maximum number of items to be printed after the command's starting address. The command will follow the chain, printing symbol table items until it either prints the specified number of items or it prints an item whose *sym_next* pointer is null. In the latter case, it will terminate and leave the *p* command's current address set to the address of the last symbol table item. For example, entering

```
pA"symbol name="#snt"value="x sym_head,100
```

will print symbol table items until it either prints 100 items or it prints an item having a null *sym_next* pointer.

2.3.11 The Quit command

q - Quit the debugger

Syntax:

q

Description:

q terminates the program being debugged, restores any modified interrupt vectors, and returns control to the operating system.

2.3.12 The Register command

r - Register display

Syntax:

r
r <reg>=EXPR

Description:

r displays and modifies the registers, including the status registers, of the program being debugged.

The parameter-less version displays the registers.

The parameterized version modifies the contents of a register, with <reg> being the name of the register to be modified, and EXPR an expression whose resulting value is to be set into the register.

2.3.13 The Single Step commands

s - Single step with display
S - Single step without display

Syntax:

[#] s [;CMDLIST]
[#] S [;CMDLIST]

Description:

These commands 'single step' the user's program; that is, execute its instructions one by one.

The optional '#' parameter specifies the number of instructions to be executed; it defaults to one instruction.

The optional CMDLIST parameter is a list of debugger commands to be executed after each single step.

The *s* and *S* commands differ in that *s* displays information after each single step, whereas *S* only displays information after the last single step.

The displayed information consists of the registers and a disassembly of the next instruction to be executed.

When single-stepping, breakpoints aren't enabled.

2.3.14 The Unassemble commands

u - Unassemble memory, with symbols
U - Unassemble memory, without symbols

Syntax:

u RANGE
U RANGE

Description:

These commands 'disassemble' a range of memory; that is, display the assembly language instructions in the range.

The *u* and *U* commands differ in that the *u* command will make use of the symbol table during disassembly and the *U* command won't. Also, the *U* command displays, for each instruction, the hex value of each byte of the instruction, whereas the *u* command won't.

With the *u* command, the disassembly of an instruction which references memory displays the location as the symbol nearest to the location plus an offset, if possible. With the *U* command, the location is displayed as a hexadecimal value.

The RANGE parameter specifies the area of memory to be disassembled. It gives the starting address, and either the number of instructions to be disassembled, or the ending address of the area.

2.3.15 The Variable commands

v - Create a new symbol
V - Modify the value of an existing symbol

Syntax:

v SYMBOL = ADDR
V SYMBOL = ADDR

Description:

The *v* and *V* commands are used to create a new symbol or modify the value for an existing symbol, respectively, in the debugger's memory resident symbol table.

SYMBOL is the name of the symbol being created or modified, and ADDR is its address.

The symbol will be classified as a code symbol unless the ADDR parameter specifies the data segment.

Symbols created using these commands remain in the memory-resident symbol table, even when the symbol table is cleared.

2.3.16 The Macro command

x - Macro command

Syntax:

xc
xc = CMDLIST
x?

Description:

The *x* command defines or executes a sequence of debugger commands, called a 'macro'. It can also list the defined macros.

A macro is associated with a letter of the alphabet, so up to 26 macros can be known to the debugger at one time. Case is not significant.

A macro is defined by typing the letter 'x', followed by the letter with which the macro is to be associated. Then follows an '=' character and the macro's list of debugger commands, with the commands separated by semicolons.

A macro is executed by typing 'x', followed by the letter with which the macro is associated, followed by a carriage return.

The macros which have been defined can be listed using the command *x?*.

2.3.17 The 'Display expression' Command

= - Display the value of an expression

Syntax:

= EXPR

Description:

This command displays the value of an expression.

The expression is displayed in several formats: hexadecimal, signed decimal, unsigned decimal, octal, binary, and ASCII. If a symbol table has been loaded, the closest symbol is displayed as well.

Some expressions involve a segment as well as an address. In this case, the segment is displayed at the beginning of the line, followed by a ':'.

2.3.18 The 'Redirect command input' Command

< - Redirect command input

Syntax:

< filename

Description:

This command causes the debugger to read and execute commands from the specified file.

When the end of the file is reached, the debugger returns to the console for commands.

This command provides a convenient means for defining macros or variables.

2.3.19 The Help command

? - list commands

Syntax:

?

Description:

This command lists the debugger commands. For groups of related commands, the listing usually lists the first letter of the commands followed by a ?. You can get a listing of all the commands in such a group by typing the the letter, the ?, and return. For example, the listing for the 'display' commands is *d?*; thus you can type *d?* followed by return to get a listing of all the 'display' commands.

3. Command Summary

breakpoint commands

bb/bw	set byte/word memory-change breakpoint
bc/bC	clear one/all breakpoints
bd	display the breakpoint table
br	reset the breakpoint counters
bs	set or modify a breakpoint
bt/bT	enable/disable trace mode

clear commands

cs	clear all symbols

display commands

db/dw/d	display memory in bytes/words/last form
dc/dd	display code/data symbols
df	display source file lines
dg	display global values
dm	display memory map
dn	display 8087 status
ds	display stack backtrace

find source string

f	find string in source file

go commands

g/G	execute user's program

port input commands

ib/iw	input byte/word from port

load commands

lf	load a source file
lp	load program
ls	load symbols

memory modification commands

mb/mw	modify bytes/words of memory
mc	compare areas of memory
mf	fill memory
mm	move memory
ms	search memory

port output commands

ob/ow	output byte/word to port

formatted print commands

p	generate formatted print

quit command

q	quit debugger

register command

r	register display

single step commands

s/S	single step with/without display

unassembly commands

u/U	unassemble memory

variable command

v/V	create/modify symbol

macro command

x	define or modify a command macro

display expression

=	display value of an expression

redirect command input

<	redirect command input

help command

?	list debugger commands

commands for screen saving and restoring

w	display saved screen (debug or program)
W	disable screen saving and restoring

commands for setting the default radix

nx	hexadecimal is default radix
no	octal is default radix
nd	decimal is default radix

OVERVIEW OF LIBRARY FUNCTIONS

Chapter Contents

Overview of Library Functions ..... libov
1. I/O Overview 4
1.1 Pre-opened devices, command line args 4
1.2 File I/O 6
1.2.1 Sequential I/O 6
1.2.2 Random I/O 6
1.2.3 Opening Files 6
1.3 Device I/O 7
1.3.1 Console I/O 7
1.3.2 I/O to Other Devices 7
1.4 Mixing unbuffered and standard I/O calls 7
2. Standard I/O Overview 9
2.1 Opening files and devices 9
2.2 Closing Streams 9
2.3 Sequential I/O 10
2.4 Random I/O 10
2.5 Buffering 10
2.6 Errors 11
2.7 The standard I/O functions 12
3. Unbuffered I/O Overview 14
3.1 File I/O 15
3.2 Device I/O 15
3.2.1 Unbuffered I/O to the Console 15
3.2.2 Unbuffered I/O to Non-Console Devices 16
4. Console I/O Overview 17
4.1 Line-oriented input 17
4.2 Character-oriented input 18
4.3 Using ioctl 19
4.4 The sgtty fields 19
4.5 Examples 20
5. Dynamic Buffer Allocation 22
6. Error Processing Overview 23

Overview of Library Functions

This chapter presents an overview of the functions that are provided with Aztec C. It's divided into the following sections:

1. *I/O:* Introduces the i/o system provided in the Aztec C package.

2. *Standard I/O:* The i/o functions can be grouped into two sets; this section describes one of them, the standard i/o functions.

3. *Unbuffered I/O:* Describes the other set of i/o functions, the unbuffered.

4. *Console I/O:* Describes special topics relating to console i/o.

5. *Dynamic Buffer Allocation:* Discusses topics related to dynamic memory allocation.

6. *Errors:* Presents an overview of error processing.

The overviews present information that is system independent. Overview information that is specific to your system is in the form of an appendix to this chapter; it accompanies the system dependent section of your manual.

1. Overview of I/O

There are two sets of functions for accessing files and devices: the unbuffered i/o functions and the standard i/o functions. These functions are identical to their UNIX equivalents, and are described in chapters 7 and 8 of *The C Programming Language.*

The unbuffered i/o functions are so called because, with few exceptions, they transfer information directly between a program and a file or device. By contrast, the standard i/o functions maintain buffers through which data must pass on its journey between a program and a disk file.

The unbuffered i/o functions are used by programs which perform their own blocking and deblocking of disk files. The standard i/o functions are used by programs which need to access files but don't want to be bothered with the details of blocking and deblocking the file records.

The unbuffered and standard i/o functions each have their own overview section (UNBUFFERED I/O and STANDARD I/O). The remainder of this section discusses features which the two sets of functions have in common.

The basic procedure for accessing files and devices is the same for both standard and unbuffered i/o: the device or file must first be "opened", that is, prepared for processing; then i/o operations occur; then the device or file is "closed".

There is a limit on the number of files and devices that can simultaneously be open; the limit on your system is defined in this chapter's system dependent appendix.

Each set of functions has its own functions for performing these operations. For example, each set has its own functions for opening a file or device. Once a file or device has been opened, it can be accessed only by functions in the same set as the function which performed the open, and must be closed by the appropriate function in the same set. There are exceptions to this non-intermingling which are described below.

There are two ways a file or device can be opened: first, the program can explicitly open it by issuing a function call. Second, it can be associated with one of the logical devices standard input, standard output, or standard error, and then opened when the program starts.

1.1 Pre-opened devices and command line arguments

There are three logical devices which are automatically opened when a program is started: standard input, standard output, and standard error. By default, these are associated with the console. The operator, as part of the command line which starts the program, can specify that these logical devices are to be "redirected" to another

device or file. Standard input is redirected by entering on the command line, after the program name, the name of the file or device, preceded by the character '<'. Standard output is redirected by entering the name of the file or device, preceded by '>'.

For example, suppose the executable program *cpy* reads standard input and writes it to standard output. Then the following command will read lines from the keyboard and write them to the display:

```
cpy
```

The following will read from the keyboard and write it to the file *testfile*:

```
cpy >testfile
```

This will copy the file *exmplfil* to the console:

```
cpy <exmplfil
```

And this will copy *exmplfil* to *testfile*:

```
cpy <exmplfil >testfile
```

Aztec C will pass command line arguments to the user's program via the user's function *main(argc, argv). argc* is an integer containing the number of arguments plus one; *argv* is a pointer to a an array of character pointers, each of which, except the first, points to a command line argument. On some systems, the first array element points to the command name; on others, it is a null pointer. Information on your system's treatment of this pointer is presented in this chapter's system dependent appendix.

For example, if the following command is entered:

```
prog arg1 arg2 arg3
```

the program *prog* will be activated and execution begins at the user's function *main*. The first parameter to main is the integer 4. The second parameter is a pointer to an array of four character pointers; on some systems the first array element will point to the string "prog" and on others it will be a null pointer. The second, third, and fourth array elements will be pointers to the strings "arg1", "arg2", and "arg3" respectively.

The command line can contain both arguments to be passed to the user's program and i/o redirection specifications. The i/o redirection strings won't be passed to the user's program, and can appear anywhere on the command line after the command name. For example, the standard output of the "prog" program can be redirected to the file outfile by any of the following commands; in each case the argc and argv parameters to the main function of 'prog' are the same as if the redirection specifier wasn't present:

```
prog arg1 arg2 arg3 >outfile
prog >outfile arg1 arg2 arg3
prog arg1 >outfile arg2 arg3
```

1.2 File I/O

A program can access files both sequentially and randomly, as discussed in the following paragraphs.

1.2.1 Sequential I/O

For sequential access, a program simply issues any of the various read or write calls. The transfer will begin at the file's "current position", and will leave the current position set to the byte following the last byte transferred. A file can be opened for read or write access; in this case, its current position is initially the first byte in the file. A file can also be opened for append access; in this case its current position is initially the end of the file.

On systems which don't keep track of the last character written to a file, it isn't always possible to correctly position a file to which data is to be appended. If this is a problem on your system, it's discussed in the system dependent appendix to this chapter, which accompanies the system dependent section of your manual.

1.2.2 Random I/O

Two functions are provided which allow a program to set the current position of an open file: *fseek*, for a file opened for standard i/o; and *lseek*, for a file opened for unbuffered i/o.

A program accesses a file randomly by first modifying the file's current position using one of the seek functions. Then the program issues any of the various read and write calls, which sequentially access the file.

A file can be positioned relative to its beginning, current position, or end. Positioning relative to the beginning and current position is always correctly done. For systems which don't keep track of the last character written to a file, positioning relative to the end of a file can't always be correctly done. For information on this, see this chapter's system dependent appendix.

1.2.3 Opening files

Opening files is somewhat system dependent: the parameters to the open functions are the same on the Aztec C packages for all systems, but some system dependencies exist, to conform with the system conventions. For example, the syntax of file names and the areas searched for files differ from system to system.

For information on the opening of files on your system, see this chapter's system dependent appendix.

1.3 Device I/O

Aztec C allows programs to access devices as well as files. Each system has its own names for devices: for the names of devices on your system, see this chapter's system dependent appendix.

1.3.1 Console I/O

Console I/O can be performed in a variety of ways. There's a default mode, and other modes can be selected by calling the function *ioctl*. We'll briefly describe console I/O in this section; for more details, see the *Console I/O* section of this chapter and the system dependent appendix to this chapter.

When the console is in default mode, console input is buffered and is read from the keyboard a line at a time. Typed characters are echoed to the screen and the operator can use the standard operating system line editing facilities. A program doesn't have to read an entire line at a time (although the system software does this when reading keyboard input into it's internal buffer), but at most one line will be returned to the program for a single read request.

The other modes of console i/o allow a program to get characters from the keyboard as they are typed, with or without their being echoed to the display; to disable normal system line editing facilities; and to terminate a read request if a key isn't depressed within a certain interval.

Output to the console is always unbuffered: characters go directly from a program to the display. The only choice concerns translation of the newline character; by default, this is translated into a carriage return, line feed sequence.

Optionally, this translation can be disabled.

1.3.2 I/O to Other Devices

On most systems, few options are available when writing to devices other than the console. For a discussion of such options, if any, that are available on your system, see this chapter's system dependent appendix.

1.4 Mixing unbuffered and standard i/o calls

As mentioned above, a program generally accesses a file or device using functions from one set of functions or the other, but not both.

However, there are functions which facilitate this dual access: if a file or device is opened for standard i/o, the function *fileno* returns a file descriptor which can be used for unbuffered access to the file or device. If a file or device is open for unbuffered i/o, the function *fdopen* will prepare it for standard i/o as well.

Care is warranted when accessing devices and files with both standard and unbuffered i/o functions.

2. Overview of Standard I/O

The standard i/o functions are used by programs to access files and devices. They are compatible with their UNIX counterparts, with few exceptions, and are also described in chapter 8 of *The C Programming Language*. The exceptions concern appending data to files and positioning files relative to their end, and are discussed below.

These functions provide programs with convenient and efficient access to files and devices. When accessing files, the functions buffer the file data; that is, handle the blocking and deblocking of file data. Thus the user's program can concentrate on its own concerns.

Buffering of data to devices when using the standard i/o functions is discussed below.

For programs which perform their own file buffering, another set of functions are provided. These are described in the section UNBUFFERED I/O.

2.1 Opening files and devices

Before a program can access a file or device, it must be "opened", and when processing on it is done it must be "closed".

An open device or file is called a "stream" and has associated with it a pointer, called a "file pointer", to a structure of type FILE. This identifies the file or device when standard i/o functions are called to access it.

There are two ways for a file or device to be opened for standard i/o: first, the program can explicitly open it, by calling one of the functions *fopen*, *freopen*, or *fdopen*. In this case, the open function returns the file pointer associated with the file or device. *fopen* just opens the file or device. *freopen* reopens an open stream to another file or device; it's mainly used to change the file or device associated with one of the logical devices standard output, standard input, or standard error. *fdopen* opens for standard i/o a file or device already opened for unbuffered i/o.

Alternatively, the file or device can be automatically opened as one of the logical devices standard input, standard output, or standard error. In this case, the file pointer is *stdin*, *stdout*, or *stderr*, respectively. These symbols are defined in the header file *stdio.h*. See the section entitled I/O for more information on logical devices.

2.2 Closing streams

A file or device opened for standard i/o can be closed in two ways: first, the program can explicitly close it by calling the function *fclose*.

Alternatively, when the program terminates, either by falling off the end of the function *main*, or by calling the function *exit*, the system will automatically close all open streams.

Letting the system automatically close open streams is error-prone: data written to files using the standard i/o functions is buffered in memory, and a buffer isn't written to the file until it's full or the file is closed. Most likely, when a program finishes writing to a file, the file's buffer will be partially full, with this information not having been written to the file. If a program calls *fclose*, this function will write the partially filled buffer to the file and return an error code if this couldn't be done. If the program lets the system automatically close the file, the program won't know if an error occurred on this last write operation.

2.3 Sequential I/O

Files can be accessed sequentially and randomly. For sequential access, simply issue repeated read or write calls; each call transfers data beginning at the "current position" of the file, and updates the current position to the byte following the last byte transferred. When a file is opened, its current position is set to zero, if opened for read or write access, and to its end if opened for append.

On systems which don't keep track of the last character written to a file, such as CP/M and Apple // DOS, not all files can be correctly positioned for appending data. See the section entitled I/O for details.

2.4 Random I/O

The function *fseek* allows a file to be accessed randomly, by changing its current position. Positioning can be relative to the beginning, current position, or end of the file.

For systems which don't keep track of the last character written to a file, such as CP/M and Apple // DOS, positioning relative to the end of a file cannot always be correctly done. See the I/O overview section for details.

2.5 Buffering

When the standard i/o functions are used to access a file, the i/o is buffered. Either a user-specified or dynamically- allocated buffer can be used.

The user's program specifies a buffer to be used for a file by calling the function *setbuf* after the file has been opened but before the first i/o request to it has been made.

If, when the first i/o request is made to a file, the user hasn't specified the buffer to be used for the file, the system will automatically allocate, by calling *malloc*, a buffer for it. When the file is closed it's buffer will be freed, by calling *free*.

Dynamically allocated buffers are obtained from the one region of memory (the heap), whether requested by the standard i/o functions or by the user's program. For more information, see the overview

section *Dynamic Buffer Allocation.*

The size of an i/o buffer differs from system to system. See this chapter's system-dependent appendix for the size of this buffer on your system.

A program which both accesses files using standard i/o functions and has overlays has to take special steps to insure that an overlay won't be loaded over a buffer dynamically allocated for file i/o. For more information, see the section on overlay support in the *Technical Information* chapter.

By default, output to the console using standard i/o functions is unbuffered; all other device i/o using the standard i/o functions is buffered. Console input buffering can be disabled using the *ioctl* function; see the overview section *Console I/O* for details.

2.6 Errors

There are three fields which may be set when an exceptional condition occurs during stream i/o. Two of the fields are unique to each stream (that is, each stream has its own pair). The other is a global integer.

One of the fields associated with a stream is set if end of file is detected on input from the stream; the other is set if an error occurs during i/o to the stream. Once set for a stream, these flags remain set until the stream is closed or the program calls the *clearerr* function for the stream. The only exception to the last statement is that when called, *fseek* will reset the end of file flag for a stream. A program can check the status of the eof and error flags for a stream by calling the functions *feof* and *ferror*, respectively.

The other field which may be set is the global integer *errno*. By convention, a system function which returns an error status as its value can also set a code in *errno* which more fully defines the error. The overview section *Errors* defines the values which may be set in *errno*.

If an error occurs when a stream is being accessed, a standard i/o function returns EOF (-1) as its value, after setting a code in *errno* and setting the stream's error flag.

If end of file is reached on an input stream, a standard i/o function returns EOF after setting the stream's eof flag.

There are two techniques a program can use for detecting errors during stream i/o. First, the program can check the result of each i/o call. Second, the program can issue i/o calls and only periodically check for errors (for example, check only after all i/o is completed).

On input, a program will generally check the result of each operation.

On output to a file, a program can use either error checking technique; however, periodic checking by calling *ferror* is more efficient. When characters are written to a file using the standard i/o functions they are placed in a buffer, which is not written to disk until it is full. If the buffer isn't full, the function will return good status. It will only return bad status if the buffer was full and an error occurred while writing it to disk. Since the buffer size is 1024 bytes, most write calls will return good status, and hence periodic checking for errors is sufficient and most efficient.

Once a file opened for standard i/o is closed, *ferror* can't be used to determine if an error has occurred while writing to it. Hence *ferror* should be called after all writing to the file is completed but before the file is closed. The file should be explicitly closed by *fclose*, and its return value checked, rather than letting the system automatically close it, to know positively whether an error has occurred while writing to the file. The reason for this is that when the writing to the file is completed, it's standard i/o buffer will probably be partly full. This buffer will be written to the file when the file is closed, and *fclose* will return an error status if this final write operation fails.

2.7 The standard i/o functions

The standard i/o functions can be grouped into two sets: those that can access only the logical devices standard input, standard output, and standard error; and all the rest.

Here are the standard i/o functions that can only access *stdin*, *stdout*, and *stderr*. These are all ASCII functions; that is, they expect to deal with text characters only.

getchar	Get an ASCII character from stdin
gets	Get a line of ASCII characters from stdin
printf	Format data and send it to stdout
puterr	Send a character to stderr
putchar	Send a character to stdout
puts	Send a character string to stdout
scanf	Get a line from stdin and convert it

Here are the rest of the standard i/o functions:

agetc	Get an ASCII character
aputc	Send an ASCII character
fopen	Open a file or device
fdopen	Open as a stream a file or device already open for unbuffered i/o
freopen	Open an open stream to another file or device
fclose	Close an open stream
feof	Check for end of file on a stream
ferror	Check for error on a stream
fileno	Get file descriptor associated with stream
fflush	Write stream's buffer
fgets	Get a line of ASCII characters
fprintf	Format data and write it to a stream
fputs	Send a string of ASCII characters to a stream
fread	Read binary data
fscanf	Get data and convert it
fseek	Set current position within a file
ftell	Get current position
fwrite	Write binary data
getc	Get a binary character
getw	Get two binary characters
putc	Send a binary character
putw	Send two binary characters
setbuf	Specify buffer for stream
ungetc	Push character back into stream

3. Overview of Unbuffered I/O

The unbuffered I/O functions are used to access files and devices. They are compatible with their UNIX counterparts and are also described in chapter 8 of *The C Programming Language.*

As their name implies, a program using these functions, with two exceptions, communicates directly with files and devices; data doesn't pass through system buffers. Some unbuffered I/O, however, is buffered: when data is transferred to or from a file in blocks smaller than a certain value, it is buffered temporarily. This value differs from system to system, but is always less than or equal to 512 bytes. Also, console input can be buffered, and is, unless specific actions are taken by the user's program.

Programs which use the unbuffered i/o functions to access files generally handle the blocking and deblocking of file data themselves. Programs requiring file access but unwilling to perform the blocking and deblocking can use the standard i/o functions; see the overview section *Standard I/O* for more information.

Here are the unbuffered i/o functions:

open	Prepares a file or device for unbuffered i/o
creat	Creates a file and opens it
close	Concludes the i/o on an open file or device
read	Read data from an open file or device
write	Write data to an open file or device
lseek	Change the current position of an open file
rename	Renames a file
unlink	Deletes a file
ioctl	Change console i/o mode
isatty	Is an open file or device the console?

Before a program can access a file or device, it must be "opened", and when processing on it is done, it must be "closed".

An open file or device has an integer known as a "file descriptor" associated with it; this identifies the file or device when it's accessed.

There are two ways for a file or device to be opened for unbuffered i/o. First, it can explicitly open it, by calling the function *open.* In this case, *open* returns the file descriptor to be used when accessing the file or device.

Alternatively, the file or device can be automatically opened as one of the logical devices standard input, standard output, or standard error. In this case, the file descriptor is the integer value 0, 1, or 2, respectively. See the section entitled I/O for more information on this.

An open file or device is closed by calling the function *close.* When a program ends, any devices or files still opened for unbuffered i/o will be closed.

If an error occurs during an unbuffered i/o operation, the function returns -1 as its value and sets a code in the global integer *errno*. For more information on error handling, see the section ERRORS.

The remainder of this section discusses unbuffered i/o to files and devices.

3.1 File I/O

Programs call the functions *read* and *write* to access a file; the transfer begins at the "current position" of the file and proceeds until the number of characters specified by the program have been transferred.

The current position of a file can be manipulated in various ways by a program, allowing both sequential and random acccess to the file. For sequential access, a program simply issues consecutive i/o requests. After each operation, the current position of the file is set to the character following the last one accessed.

The function *lseek* provides random access to a file by setting the current position to a specified character location.

lseek allows the current position of a file to be set relative to the end of a file. For systems which don't keep track of the last character written to a file, such positioning cannot always be correctly done. For more information, see the section entitled I/O.

open provides a mode, O_APPEND, which causes the file being opened to be positioned at its end. This mode is supported on UNIX Systems 3 and 5, but not UNIX version 7. As with *lseek*, the positioning may not be correct for systems which don't keep track of the last character written to a file.

3.2 Device I/O

3.2.1 Unbuffered I/O to the Console

There are several options available when accessing the console, which are discussed in detail in the Console I/O sections of this chapter and of the system-dependent appendix to this chapter. Here we just want to briefly discuss the line- or character-modes of console I/O as they relate to the unbuffered i/o functions.

Console input can be either line- or character-oriented. With line-oriented input, characters are read from the console into an internal buffer a line at a time, and returned to the program from this buffer. Line buffering of console input is available even when using the so-called "unbuffered" i/o functions.

With character-oriented input, characters are read and returned to the program when they are typed: no buffering of console input occurs.

3.2.2 Unbuffered I/O to Non-Console Devices

Unbuffered I/O to devices other than the console is truly unbuffered.

4. Overview of Console I/O

A program has control over several options relating to console i/o. The primary option allows console input to be either line- or character-oriented, as described below.

On most systems, a program can selectively enable and disable the echoing of typed characters to the screen; this is called the ECHO option. A program can also enable and disable the conversion of carriage return to newline on input and of newline to carriage return-linefeed on output; this is called the CRMOD option.

On some systems, additional options are available. If your system supports additional options, they are discussed in the system dependent appendix to this chapter.

All the console i/o options have default settings, which allow a program to easily access the console without having to set the options itself. In the default mode, console i/o is line-oriented, with ECHO and CRMOD enabled.

A program can easily change the console i/o options, by calling the function *ioctl*.

Console i/o behaves the same on all systems when the console options have their default settings. However, the behavior of console i/o differs from system to system when the options are changed from their default values. Thus, a program requiring machine independence should either use the console in its default mode or be careful how it sets the console options. In the paragraphs below, we will try to point out system dependencies.

4.1 Line-oriented input

With line-oriented input, a program issuing a read request to the console will wait until an entire line has been typed. On some systems a non-UNIX option (NODELAY) is available that will prevent this waiting. If this option is available on your system, it's discussed in the system-dependent appendix to this chapter.

The program need not read an entire line at once; the line will be internally buffered, and characters returned to the program from the buffer, as requested. When the program issues a read request to the console and the buffer is empty, the program will wait until an entire new line has been typed and stored in the internal buffer (again, on some systems programs can disable this wait by setting the non-UNIX NODELAY option).

A single unbuffered read operation can return at most one line.

On most systems , selecting line-oriented console input forces the ECHO option to be enabled. On such systems the program still has control over the CRMOD option. To find out if, on your system,

line-oriented mode always has ECHO enabled, see the system-dependent appendix to this chapter.

4.2 Character-oriented input

The basic idea of character-oriented console input is that a program can read characters from the console without having to wait for an entire line to be entered.

The behavior of character-oriented console input differs from system to system, so programs requiring both machine independence and character-oriented console input have to be careful in their use of the console. However, it is possible to write such programs, although they may not be able to take full advantage of the console i/o features available for a particular system.

There are two varieties of character-oriented console input, named CBREAK and RAW. Their primary difference is that with the console in CBREAK mode, a program still has control over the other console options, whereas with the console in RAW mode it doesn't. In RAW mode, all other console options are reset: ECHO and CRMOD are disabled.

Thus, to some extent RAW mode is simply an abbreviation for 'CBREAK on, all other options off'. However, there are some differences on some systems, as noted below and in this chapter's system-dependent appendix.

The system-dependent appendix to this chapter, which accompanies your manual, presents information about character-oriented console that is specific to your system.

4.2.1 Writing system-independent programs

To write system-independent programs that access the console in character-oriented input mode, the console should be set in RAW mode, and the program should read only a single character at a time from the console. All the non-UNIX options that are supported by some systems should be reset.

The standard i/o functions all read just one character at a time from the console, even when the calling program requests several characters. Thus, programs requiring system independence and character-oriented input can read the console using the standard i/o functions.

Some systems require a program that wants to set console option to first call *ioctl* to fetch the current console options, then modify them as desired, and finally call *ioctl* to reset the new console options. The systems that don't require this don't care if a program first fetches the console options and then modifies them. Thus, a program requiring system-independence and console i/o options other than the default should fetch the current console options before modifying them.

4.3 Using ioctl

A program selects console I/O modes using the function *ioctl.* This has the form:

```
#include <sgtty.h>

ioctl(fd, code, arg)
struct sgttyb *arg;
```

The header file *sgtty.h* defines symbolic values for the *code* parameter (which tells *ioctl* what to do) and the structure *sgttyb.*

The parameter *fd* is a file descriptor associated with the console. On UNIX, this parameter defines the file descriptor associated with the device to which the *ioctl* call applies. Here, *ioctl* always applies to the console.

The parameter *code* defines the action to be performed by *ioctl.* It can have these values:

TIOCGETP	Fetch the console parameters and store them in the structure pointed at by *arg.*
TIOCSETP	Set the console parameters according to the structure pointed at by *arg.*
TIOCSETN	Equivalent to *TIOCSETP.*

The argument *arg* points to a structure named *sgttyb* that contains the following fields:

```
int sg_flags;
char sg_erase;
char sg_kill;
```

The order of these fields is system-dependent.

The *sg_flags* field is supported by all systems, while the other fields are not supported by some systems. If these fields are supported on your system, the system-dependent appendix to this chapter that accompanies your manual says so, and describes them.

To set console options, a program should fetch the current state of the sgtty fields, using ioctl's TIOCGETP option. Then it should modify the fields to the appropriate values and call ioctl again, using ioctl's TIOCSETP option.

4.4 The *sgtty* fields

4.4.1 The sg_flags field

sg_flags contains the following UNIX-compatible flags:

RAW	Set RAW mode (turns off other options). By default, RAW is disabled.
CBREAK	Return each character as soon as typed. By default, CBREAK is disabled.

ECHO	Echo input characters to the display. By default, ECHO is enabled.
CRMOD	Map CR to LF on input; convert LF to CR-LF on output. By default, CRMOD is enabled.

On some systems, other flags are contained in *sg_flags*. If your system supports other flags, they're described in the system-dependent appendix to this chapter that accompanies your manual.

More than one flag can be specified in a single call to *ioctl*; the values are simply 'or'ed together. If the RAW option is selected, none of the other options have any effect.

When the console i/o options are set and RAW and CBREAK are reset, the console is set in line-oriented input mode.

4.5 Examples

4.5.1 Console input using default mode

The following program copies characters from stdin to stdout. The console is in default mode, and assuming these streams haven't been redirected by the operator, the program will read from the keyboard and write to the display. In this mode, the operator can use the operating system's line editing facilities, such as backspace, and characters entered on the keyboard will be echoed to the display. The characters entered won't be returned to the program until the operator depresses carriage return.

```
#include <stdio.h>

main()
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
}
```

4.5.2 Console input - RAW mode

In this example, a program opens the console for standard i/o, sets the console in RAW mode, and goes into a loop, waiting for characters to be read from the console and then processing them. The characters typed by the operator aren't displayed unless the program itself displays them. The input request won't terminate until a character is received. This example assumes that the console is named 'con:'; on systems for which this is not the case, just substitute the appropriate name.

```
#include <stdio.h>
#include <sgtty.h>
main()
{
        int c;
        FILE *fp;
        struct sgttyb stty;

        if ((fp = fopen("con:", "r") == NULL){
                printf("can't open the console\n");
                exit();
        }

        ioctl(fileno(fp),TIOCGETP, &stty);

        stty.sg_flags |= RAW;
        ioctl(fileno(fp), TIOCSETP, &stty);
        for (;;){
                c = getc(fp);
                . . .
        }
}
```

4.5.3 Console input - console in CBREAK + ECHO mode

This example modifies the previous program so that characters read from the console are automatically echoed to the display. The program accesses the console via the standard input device. It uses the function *isatty* to verify that stdin is associated with the console; if it isn't, the program reopens stdin to the console using the function *freopen.* Again, the console is assumed to be named *con:*.

```
#include <stdio.h>
#include <sgtty.h>
main()
{
        int c;
        struct sgttyb stty;

        if (!isatty(stdin))
                freopen("con:", "r", stdin);
        ioctl(0, TIOCGETP, &stty);
        stty.sg_flags |= CBREAK | ECHO;
        ioctl(0, TIOCSETP, &stty);
        for (;;){
                c = getchar();
                . . .
        }
}
```

5. Overview of Dynamic Buffer Allocation

Several functions are provided for the dynamic allocation and deallocation of buffers from a section of memory called the 'heap'. They are:

malloc	Allocates a buffer
calloc	Allocates a buffer and initializes it to zeroes
realloc	Allocates more space to a previously allocated buffer
free	Releases an allocated buffer for reuse

These standard UNIX functions are described in the System Independent Functions section of this chapter.

In addition, on some systems the UNIX-compatible functions *sbrk* and *brk* are provided that provide a more elementary means to allocate heap space. The *malloc*-type functions call *sbrk* to get heap space, which they then manage.

On some systems, non-UNIX memory allocation functions are also supported. If such functions are supported on your system, they are described in the system-dependent appendix to this chapter that accompanies your manual.

Dynamic allocation of standard i/o buffers

Buffers used for standard i/o are dynamically allocated from the heap unless specific actions are taken by the user's program. Standard i/o calls to dynamically allocate and deallocate buffers can be interspersed with those of the user's program.

Programs which perform standard i/o and which must have absolute control of the heap can explicitly define the buffers to be used by a standard i/o stream.

Where to go from here

For descriptions of the *sbrk* and *brk* functions and, when applicable, non-UNIX memory allocation functions see the System Dependent Functions chapter.

For a discussion of i/o buffer allocation, see the Standard I/O section of the Library Functions Overviews chapter.

For more information on the heap, see the Program Organization section of the Technical Information chapter.

6. Overview of Error Processing

This section discusses error processing which relates to the global integer *errno*. This variable is modified by the standard i/o, unbuffered i/o, and scientific (eg, *sin*, *sqrt*) functions as part of their error processing.

The handling of floating point exceptions (overflow, underflow, and division by zero) is discussed in the Tech Info chapter.

When a standard i/o, unbuffered i/o, or scientific function detects an error, it sets a code in *errno* which describes the error. If no error occurs, the scientific functions don't modify *errno*. If no error occurs, the i/o functions may or may not modify *errno*.

Also, when an error occurs,

* A standard i/o function returns -1 and sets an error flag for the stream on which the error occurred;
* An unbuffered i/o function returns -1;
* A scientific function returns an arbitrary value.

When performing scientific calculations, a program can check *errno* for errors as each function is called. Alternatively, since *errno* is modified only when an error occurs, *errno* can be checked only after a sequence of operations; if it's non-zero, then an error has occurred at some point in the sequence. This latter technique can only be used when no i/o operations occur during the sequence of scientific function calls.

Since *errno* may be modified by an i/o function even if an error didn't occur, a program can't perform a sequence of i/o operations and then check *errno* afterwards to detect an error. Programs performing unbuffered i/o must check the result of each i/o call for an error.

Programs performing standard i/o operations cannot, following a sequence of standard i/o calls, check *errno* to see if an error occurred. However, associated with each open stream is an error flag. This flag is set when an error occurs on the stream and remains set until the stream is closed or the flag is explicitly reset. Thus a program can perform a sequence of standard i/o operations on a stream and then check the stream's error flag. For more details, see the standard i/o overview section.

The following table lists the system-independent values which may be placed in *errno*. These symbolic values are defined in the file *errno.h*. Other, system-dependent, values may also be set in *errno* following an i/o operation; these are error codes returned by the operating system. System dependent error codes are described in the operating system manual for a particular system.

The system-independent error codes and their meanings are:

error code	*meaning*
ENOENT	File does not exist
E2BIG	Not used
EBADF	Bad file descriptor - file is not open or improper operation requested
ENOMEM	Insufficient memory for requested operation
EEXIST	File already exists on creat request
EINVAL	Invalid argument
ENFILE	Exceeded maximum number of open files
EMFILE	Exceeded maximum number of file descriptor
ENOTTY	Ioctl attempted on non-console
EACCES	Invalid access request
ERANGE	Math function value can't be computed
EDOM	Invalid argument to math function

SYSTEM-INDEPENDENT FUNCTIONS

Chapter Contents

System Independent Functions ..... lib
Index 5
The functions 8

System Independent Functions

This chapter describes in detail the functions which are UNIX-compatible and which are common to all Aztec C packages.

The chapter is divided into sections, each of which describes a group of related functions. Each section has a name, and the sections are ordered alphabetically by name. Following this introduction is a cross reference which lists each function and the name of the section in which it is described.

A section is organized into the following subsections:

TITLE

Lists the name of the section, a phrase which is intended to catagorize the functions described in the section, and one or more letters in parentheses which specify the libraries containing the section's functions.

The letters which may appear in parentheses and their corresponding libraries are:

C	c.lib
M	m.lib

On some systems, the actual library name may be a variant on the name given above. For example, on TRSDOS, the libraries are named *c/lib* and *m/lib*.

With *Apprentice C*, the functions are all in the run-time system, and not libraries.

SYNOPSIS

Indicates the types of arguments that the functions described in the section require, and the values they return. For example, the function *atof* converts character strings into double precision numbers. It is listed in the synopsis as

```
double atof(s)
char *s;
```

This means that *atof()* returns a value of type *double* and requires as an argument a pointer to a character string. Since *atof* returns a non-integer value, prior to use of the function it must be declared:

```
double atof();
```

The notation

```
#include "header.h"
```

at the beginning of a synopsis indicates that such a statement should appear at the beginning of any program calling one of the functions described in the section.

On Radio Shack systems, a header file can use either a period or a slash to separate the filename from the extent. That is, the include statement can be as listed above, or

```
#include "header/h"
```

DESCRIPTION
Describes the section's functions.

SEE ALSO
Lists relevant sections. A letter in parentheses may follow a section name. This specifies where the section is located: no letter means that the section is in the current chapter; 'O' means that it's in the Functions Overview chapter; 'S' means that it's in the System Dependent Functions chapter.

DIAGNOSTICS
Describes the error codes that the section's functions may return. The section ERRORS in the Functions Overview chapter presents an overview of error processing.

EXAMPLES
Gives examples on use of the section's functions.

Index to System Independent Functions

function	*page*	*description*
acos	SIN	compute arccosine
agetc	GETC	get ASCII char from a stream
aputc	PUTC	put ASCII char to a stream
asin	SIN	compute arcsine
atan	SIN	compute arctangent
atan2	SIN	another arctangent function
atof	ATOF	convert char string to a *double*
atoi	ATOF	convert char string to an *int*
atol	ATOF	convert char string to a *long*
calloc	MALLOC	allocate a buffer
ceil	FLOOR	get smallest integer not less than x
clearerr	FERROR	clear error flags on a stream
close	CLOSE	close of unbuffered file/device
cos	SIN	compute cosine
cosh	SINH	compute hyperbolic cosine
cotan	SIN	compute cotangent
creat	CREAT	create a file & open for unbuffered i/o
exp	EXP	compute exponential
fabs	FLOOR	compute absolute value
fclose	FCLOSE	close i/o stream
fdopen	FOPEN	open file descriptor as an i/o stream
feof	FERROR	check for eof on an i/o stream
ferror	FERROR	check for error on an i/o stream
fflush	FCLOSE	flush an i/o stream
fgets	GETS	get a line from an i/o stream
fileno	FERROR	get file descriptor for i/o stream
floor	FLOOR	get largest *int* not greater than x
fopen	FOPEN	open i/o stream
format	PRINTF	formatting utility for *printf*
fprintf	PRINTF	format string & send to i/o stream
fputs	PUTS	put char string to i/o stream
fread	FREAD	read binary data from i/o stream
free	MALLOC	release buffer
freopen	FOPEN	reopen i/o stream
frexp	FREXP	get components of a *double*
fscanf	SCANF	input string from i/o stream & convert
fseek	FSEEK	position i/o stream
ftell	FSEEK	determine position in i/o stream
ftoa	ATOF	convert float/double to char string

fwrite	FREAD	write binary data to i/o stream
getc	GETC	get binary char from i/o stream
getchar	GETC	get ASCII char from stdin
gets	GETS	get ASCII line from stdin
getw	GETW	get ASCII word from stdin
index	STRING	find char in string
ioctl	IOCTL	set mode of device
isalpha, etc.	CTYPE	char classification functions
isatty	IOCTL	is this a console?
ldexp	FREXP	build *double*
log	EXP	compute natural logarithm
log10	EXP	compute base-10 log
longjmp	SETJMP	non-local goto
lseek	LSEEK	position unbuffered i/o file
malloc	MALLOC	allocate buffer
movmem	MOVMEM	copy a block of memory
modf	FREXP	get components of *double*
open	OPEN	open file/device for unbuffered i/o
pow	EXP	compute x**y
printf	PRINTF	format data and print on stdout
putc	PUTC	put binary char to i/o stream
putchar	PUTC	put ASCII char to stdout
puterr	PUTC	put ASCII char to stderr
puts	PUTS	put ASCII string to stdout
putw	PUTC	put ASCII word to stdout
qsort	QSORT	Quick sort
ran	RAN	compute random number
read	READ	read unbuffered file/device
realloc	MALLOC	reallocate buffer
rename	RENAME	rename file
rindex	STRING	find char in string
scanf	SCANF	input string from stdin & convert
setbuf	SETBUF	set buffer for i/o stream
setjmp	SETJMP	*longjmp* partner
setmem	MOVMEM	set memory to specified byte
sin	SIN	compute sine
sinh	SINH	compute hyperbolic sine
sprintf	PRINTF	format string into buffer
sqrt	EXP	compute square root
sscanf	SCANF	convert string from buffer
strcat	STRING	concatenate two strings
strcmp	STRING	compare two strings
strcpy	STRING	copy char string
strlen	STRING	get length of char string
strncat	STRING	concatenate strings
strncmp	STRING	compare strings
strncpy	STRING	copy string
swapmem	MOVMEM	swap two blocks of memory

tan ..... SIN ..... compute tangent
tanh ..... SINH ..... compute hyperbolic tangent
tolower ..... TOUPPER ..... convert upper case char to lower
toupper ..... TOUPPER ..... convert lower case char to upper
ungetc ..... UNGETC ..... return char to i/o stream
unlink ..... UNLINK ..... delete file
write ..... WRITE ..... unbuffered write of binary data

NAME

atof, atoi, atol - convert ASCII to numbers
ftoa - convert floating point to ASCII

SYNOPSIS

double atof(cp)
char *cp;

atoi(cp)
char *cp;

long atol(cp)
char *cp;

ftoa(val, buf, precision, type)
double val;
char *buf;
int precision, type;

DESCRIPTION

atof, *atoi*, and *atol* convert a string of text characters pointed at by the argument *cp* to double, integer, and long representations, respectively.

atof recognizes a string containing leading blanks and tabs, which it skips, then an optional sign, then a string of digits optionally containing a decimal point, then an optional 'e' or 'E' followed by an optionally signed integer.

atoi and *atol* recognize a string containing leading blanks and tabs, which are ignored, then an optional sign, then a string of digits.

ftoa converts a double precision floating point number to ASCII. *val* is the number to be converted and *buf* points to the buffer where the ASCII string will be placed. *precision* specifies the number of digits to the right of the decimal point. *type* specifies the format: 0 for "E" format, 1 for "F" format, 2 for "G" format.

atof and *ftoa* are in the library *m.lib*; the other functions are in *c.lib*.

NAME

close - close a device or file

SYNOPSIS

close(fd)
int fd;

DESCRIPTION

close closes a device or disk file which is opened for unbuffered i/o.

The parameter *fd* is the file descriptor associated with the file or device. If the device or file was explicitly opened by the program by calling *open* or *creat*, *fd* is the file descriptor returned by *open* or *creat*.

close returns 0 as its value if successful.

SEE ALSO

Unbuffered I/O (O), Errors (O)

DIAGNOSTICS

If *close* fails, it returns -1 and sets an error code in the global integer *errno*.

NAME

creat - create a new file

SYNOPSIS

creat(name, pmode)
char *name;
int pmode;

DESCRIPTION

creat creates a file and opens it for unbuffered, write-only access. If the file already exists, it is truncated so that nothing is in it (this is done by erasing and then creating the file).

creat returns as its value an integer called a "file descriptor". Whenever a call is made to one of the unbuffered i/o functions to access the file, its file descriptor must be included in the function's parameters.

name is a pointer to a character string which is the name of the device or file to be opened. See the I/O overview section for details.

For most systems, *pmode* is optional: if specified, it's ignored. It should be included, however, for programs for which UNIX-compatibility is required, since the UNIX creat function requires it. In this case, *pmode* should have the octal value 0666.

For some systems, *pmode* is required and has a special meaning. If it is required for your system, the System Dependent Functions chapter will contain a description of the *creat* function, which will define the meaning.

SEE ALSO

Unbuffered I/O (O), Errors (O)

DIAGNOSTICS

If *creat* fails, it returns -1 as its value and sets a code in the global integer *errno*.

NAME

isalpha, isupper, islower, isdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii
- character classification functions

SYNOPSIS

#include "ctype.h"

isalpha(c)

. . .

DESCRIPTION

These macros classify ASCII-coded integer values by table lookup, returning nonzero if the integer is in the catagory, zero otherwise. *isascii* is defined for all integer values. The others are defined only when *isascii* is true and on the single non-ASCII value EOF (-1).

isalpha	*c* is a letter
isupper	*c* is an upper case letter
islower	*c* is a lower case letter
isdigit	*c* is a digit
isalnum	*c* is an alphanumeric character
isspace	*c* is a space, tab, carriage return, newline, or formfeed
ispunct	*c* is a punctuation character
isprint	*c* is a printing character, valued 0x20 (space) through 0x7e (tilde)
iscntrl	*c* is a delete character (0xff) or ordinary control character (value less than 0x20)
isascii	*c* is an ASCII character, code less than 0x100

NAME

exponential, logarithm, power, square root functions:
exp, log, log10, pow, sqrt

SYNOPSIS

```
#include <math.h>

double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

double pow(x, y)
double x,y;

double sqrt(x)
double x;
```

DESCRIPTION

exp returns the exponential function of x.

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

pow returns x ** y (x to the y-th power).

sqrt returns the square root of x.

SEE ALSO

Errors (O)

DIAGNOSTICS

If a function can't perform the computation, it sets an error code in the global integer *errno* and returns an arbitrary value; otherwise it returns the computed value without modifying *errno*. The symbolic values which a function can place in *errno* are EDOM, signifying that the argument was invalid, and ERANGE, meaning that the value of the function couldn't be computed. These codes are defined in the file *errno.h*.

The following table lists, for each function, the error codes that can be returned, the function value for that error, and the meaning of the error. The symbolic values are defined in the file *math.h*.

function	error	f(x)	Meaning
exp	ERANGE	HUGE	x > LOGHUGE
"	ERANGE	0.0	x < LOGTINY
log	EDOM	-HUGE	x <= 0
log10	EDOM	-HUGE	x <= 0
pow	EDOM	-HUGE	x < 0, x=y=0
"	ERANGE	HUGE	y*log(x)>LOGHUGE
"	ERANGE	0.0	y*log(x)<LOGTINY
sqrt	EDOM	0.0	x < 0.0

NAME

fclose, fflush - close or flush a stream

SYNOPSIS

#include "stdio.h"

fclose(stream)
FILE *stream;

fflush(stream)
FILE *stream;

DESCRIPTION

fclose informs the system that the user's program has completed its buffered i/o operations on a device or file which it had previously opened (by calling *fopen*). *fclose* releases the control blocks and buffers which it had allocated to the device or file. Also, when a file is being closed, *fclose* writes any internally buffered information to the file.

fclose is called automatically by *exit.*

fflush causes any buffered information for the named output stream to be written to that file. The stream remains open.

If *fclose* or *fflush* is successful, it returns 0 as its value.

SEE ALSO

Standard I/O (O)

DIAGNOSTICS

If the operation fails, -1 is returned, and an error code is set in the global integer *errno.*

NAME

feof, ferror, clearerr, fileno - stream status inquiries

SYNOPSIS

```
#include "stdio.h"

feof(stream)
FILE *stream;

ferror(stream)
FILE *stream;

clearerr(stream)
FILE *stream;

fileno(stream)
FILE *stream;
```

DESCRIPTION

feof returns non-zero when end-of-file is reached on the specified input stream, and zero otherwise.

ferror returns non-zero when an error has occurred on the specified stream, and zero otherwise. Unless cleared by *clearerr*, the error indication remains set until the stream is closed.

clearerr resets an error indication on the specified stream.

fileno returns the integer file descriptor associated with the stream.

These functions are defined as macros in the file stdio.h.

SEE ALSO

Standard I/O (O)

NAME

fabs, floor, ceil - absolute value, floor, ceiling routines

SYNOPSIS

```
#include <math.h>

double floor(x)
double x;

double ceil(x)
double x;

double fabs(x)
double x;
```

DESCRIPTION

fabs returns the absolute value of *x*.

floor returns the largest integer not greater than *x*.

ceil returns the smallest integer not less than *x*.

NAME

fopen, freopen, fdopen - open a stream

SYNOPSIS

```
#include "stdio.h"

FILE *fopen(filename, mode)
char *filename, *mode;

FILE *freopen(filename, mode, stream)
char *filename, *mode;
FILE *stream;

FILE *fdopen(fd, mode)
char *mode;
```

DESCRIPTION

These functions prepare a device or disk file for access by the standard i/o functions; this is called "opening" the device or file. A file or device which has been opened by one of these functions is called a "stream".

If the device or file is successfully opened, these functions return a pointer, called a "file pointer" to a structure of type FILE. This pointer is included in the list of parameters to buffered i/o functions, such as *getc* or *putc*, which the user's program calls to access the stream.

fopen is the most basic of these functions: it simply opens the device or file specified by the *filename* parameter for access specified by the *mode* parameter. These parameters are described below.

freopen substitutes the named device or file for the device or file which was previously associated with the specified stream. It closes the device or file which was originally associated with the stream and returns *stream* as its value. It is typically used to associate devices and files with the preopened streams *stdin*, *stdout*, and *stderr*.

fdopen opens a device or file for buffered i/o which has been previously opened by one of the unbuffered open functions *open* and *creat*. It returns as it's value a FILE pointer.

fdopen is passed the file descriptor which was returned when the device or file was opened by *open* or *creat*. It's also passed the *mode* parameter specifying the type of access desired. *mode* must agree with the mode of the open file.

The parameter *filename* is a pointer to a character string which is the name of the device or file to be opened. For details, see the I/O overview section.

mode points to a character string which specifies how the user's program intends to access the stream. The choices are as follows:

mode	*meaning*
r	Open for reading only. If a file is opened, it is positioned at the first character in it. If the file or device does not exist, NULL is returned.
w	Open for writing only. If a file is opened which already exists, it is truncated to zero length. If the file does not exist, it is created.
a	Open for appending. The calling program is granted write-only access to the stream. The current file position is the character after the last character in the file. If the file does not exist, it is created.
x	Open for writing. The file must not previously exist. This option is not supported by Unix.
r+	Open for reading and writing. Same as "r", but the stream may also be written to.
w+	Open for writing and reading. Same as "w", but the stream may also be read; different from "r+" in the creation of a new file and loss of any previous one.
a+	Open for appending and reading. Same as "a", but the stream may also be read; different from "r+" in file positioning and file creation.
x+	Open for writing and reading. Same as "x" but the file can also be read.

On systems which don't keep track of the last character in a file (for example CP/M and Apple DOS), not all files can be correctly positioned when opened in append mode. See the I/O overview section for details.

SEE ALSO

I/O (O), Standard I/O (O)

DIAGNOSTICS

If the file or device cannot be opened, NULL is returned and an error code is set in the global integer *errno*.

EXAMPLES

The following example demonstrates how *fopen* can be used in a program.

```
#include "stdio.h"

main(argc,argv)
char **argv;
{
    FILE *fopen(), *fp;

    if ((fp = fopen(argv[1], argv[2])) == NULL) {
        printf("You asked me to open %s",argv[1]);
        printf("in the %s mode", argv[2]);
        printf("but I can't!\n");
    } else
        printf("%s is open\n", argv[1]);
```

Here is a program which uses *freopen*:

```
#include "stdio.h"
main()
{
    FILE *fp;
    fp = freopen("dskfile", "w+", stdout);
    printf("This message is going to dskfile\n");
}
```

Here is a program which uses *fdopen*:

```
#include "stdio.h"

dopen_it(fd)
int fd;  /* value returned by previous call to open */
{
    FILE *fp;

    if ((fp = fdopen(fd, "r+")) == NULL)
        printf("can't open file for r+\n");
    else
        return(fp);
}
```

NAME

fread, fwrite - buffered binary input/output

SYNOPSIS

```
#include "stdio.h"

int fread(buffer, size, count, stream)
char *buffer;
int size, count;
FILE *stream;

int fwrite(buffer, size, count, stream)
char *buffer;
int size, count;
FILE *stream;
```

DESCRIPTION

fread performs a buffered input operation and *fwrite* a buffered write operation to the open stream specified by the parameter *stream*.

buffer is the address of the user's buffer which will be used for the operation.

The function reads or writes *count* items, each containing *size* bytes, from or to the stream.

fread and *fwrite* perform i/o using the functions *getc* and *putc*; thus, no translations occur on the data being transferred.

The function returns as its value the number of items actually read or written.

SEE ALSO

Standard I/O (O), Errors (O), fopen, ferror

DIAGNOSTICS

fread and *fwrite* return 0 upon end of file or error. The functions *feof* and *ferror* can be used to distinguish between the two. In case of an error, the global integer *errno* contains a code defining the error.

EXAMPLE

This is the code for reading ten integers from file 1 and writing them again to file 2. It includes a simple check that there are enough two-byte items in the first file:

```
#include "stdio.h"

main()
{
    FILE *fp1, *fp2, *fopen();
    char *buf;
    int size = 2, count = 10;

    fp1 = fopen("file1","r");
    fp2 = fopen("file2","w");
    if (fread(buf, size, count, fp1) != count)
        printf("Not enough integers in file1\n");
    fwrite(buf, size, count, fp2);
}
```

NAME

frexp, ldexp, modf - build and unbuild real numbers

SYNOPSIS

```
#include <math.h>

double frexp(value, eptr)
double value;
int *eptr;

double ldexp(value, exp)
double value;

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

DESCRIPTION

Given *value*, *frexp* computes integers x and n such that value=x*2**n. x is returned as the value of *frexp*, and n is stored in the *int* field pointed at by *eptr*.

ldexp returns the double quantity value*2**exp.

modf returns as its value the positive fractional part of value and stores the integer part in the double field pointed at by *iptr*.

NAME

fseek, ftell - reposition a stream

SYNOPSIS

```
#include "stdio.h"

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

long ftell(stream)
FILE *stream;
```

DESCRIPTION

fseek sets the "current position" of a file which has been opened for buffered i/o. The current position is the byte location at which the next input or output operation will begin.

stream is the stream identifier associated with the file, and was returned by *fopen* when the file was opened.

offset and *origin* together specify the current position: the new position is at the signed distance *offset* bytes from the beginning, current position, or end of the file, depending on whether *origin* is 0, 1, or 2, respectively.

offset can be positive or negative, to position after or before the specified origin, respectively, with the limitation that you can't seek before the beginning of the file.

For some operating systems (for example, CP/M and Apple DOS) a file may not be able to be correctly positioned relative to its end. See the overview sections I/O and STANDARD I/O for details.

If *fseek* is successful, it will return zero.

ftell returns the number of bytes from the beginning to the current position of the file associated with *stream.*

SEE ALSO

Standard I/O (O), I/O (O), lseek

DIAGNOSTICS

fseek will return -1 for improper seeks. In this case, an error code is set in the global integer *errno.*

EXAMPLE

The following routine is equivalent to opening a file in "a+" mode:

```
a_plus(filename)
char *filename;
{
    FILE *fp, *fopen();

    if ((fp = fopen(filename, r+)) == NULL)
    fp = fopen(filename, w+);
    fseek(fp, 0L, 2);   /* position 1 byte past
                           last character */
}
```

To set the current position back 5 characters before the present current position, the following call can be used:

```
fseek(fp, -5L, 1)
```

NAME

getc, agetc, getchar, getw

SYNOPSIS

```
#include "stdio.h"

int getc(stream)
FILE *stream;

int agetc(stream)       /* non-Unix function */
FILE *stream;

int getchar()

int getw(stream)
FILE *stream;
```

DESCRIPTION

getc returns the next character from the specified input stream.

agetc is used to access files of text. It generally behaves like *getc* and returns the next character from the named input stream. It differs from *getc* in the following ways:

* It translates end-of-line sequences (eg, carriage return on Apple DOS; carriage return-line feed on CP/M) to a single newline ('\\n') character.
* It translates an end-of-file sequence (eg, a null character on Apple DOS; a control-z character on CP/M) to EOF;
* It ignores null characters (' ') on all systems except Apple DOS;
* On some systems, the most significant bit of each character returned is set to zero.

agetc is not a UNIX function. It is, however, provided with all Aztec C packages, and provides a convenient, system-independent way for programs to read text.

getchar returns text characters from the standard input stream (stdin). It is implemented as the call *agetc(stdin).*

getw returns the next word from the specified input stream. It returns EOF (-1) upon end-of-file or error, but since that is a good integer value, *feof* and *ferror* should be used to check the success of *getw*. It assumes no special alignment in the file.

SEE ALSO

I/O (O), Standard I/O (O), fopen, fclose

DIAGNOSTICS

These functions return EOF (-1) at end of file or if an error occurs. The functions *feof* and *ferror* can be used to distinguish the two. In the latter case, an error code is set in the global

integer *errno*.

NAME

gets, fgets - get a string from a stream

SYNOPSIS

```
#include "stdio.h"

char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
FILE *stream;
```

DESCRIPTION

gets reads a string of characters from the standard input stream, stdin, into the buffer pointed by *s*. The input operation terminates when either a newline character (\\n) or end of file is encountered.

fgets reads characters from the specified input stream into the buffer pointer at by *s* until either (1) *n*-1 characters have been read, (2) a newline character (\\n) is read, or (3) end of file or an error is detected on the stream.

Both functions return *s*, except as noted below.

gets and *fgets* differ in their handling of the newline character: *gets* doesn't put it in the caller's buffer, while *fgets* does. This is the behavior of these functions under UNIX.

These functions get characters using *agetc*; thus they can only be used to get characters from devices and files which contain text characters.

SEE ALSO

I/O (O), Standard I/O (O), ferror

DIAGNOSTICS

gets and *fgets* return the pointer NULL (0) upon reaching end of file or detecting an error. The functions *feof* and *ferror* can be used to distinguish the two. In the latter case, an error code is placed in the global integer *errno*.

NAME

ioctl, isatty - device i/o utilities

SYNOPSIS

#include "sgtty.h"

ioctl(fd, cmd, stty)
struct sgttyb *stty;

isatty(fd)

DESCRIPTION

ioctl sets and determines the mode of the console.

For more details on *ioctl*, see the overview section on console I/O.

isatty returns non-zero if the file descriptor *fd* is associated with the console, and zero otherwise.

SEE ALSO

Console I/O (O)

NAME

lseek - change current position within file

SYNOPSIS

long int lseek(fd, offset, origin)
int fd, origin;
long offset;

DESCRIPTION

lseek sets the current position of a file which has been opened for unbuffered i/o. This position determines where the next character will be read or written.

fd is the file descriptor associated with the file.

The current position is set to the location specified by the offset and origin parameters, as follows:

* If *origin* is 0, the current position is set to *offset* bytes from the beginning of the file.
* If *origin* is 1, the current position is set to the current position plus *offset.*
* If *origin* is 2, the current position is set to the end of the file plus *offset.*

The offset can be positive or negative, to position after or before the specified origin, respectively.

Positioning of a file relative to its end (that is, calling *lseek* with *origin* set to 2) cannot always be correctly done on all systems (for example, CP/M and Apple DOS). See the section entitled I/O for details.

If *lseek* is successful, it will return the new position in the file (in bytes from the beginning of the file).

SEE ALSO

Unbuffered I/O (O), I/O (O)

DIAGNOSTICS

If *lseek* fails, it will return -1 as its value and set an error code in the global integer *errno. errno* is set to EBADF if the file descriptor is invalid. It will be set to EINVAL if the offset parameter is invalid or if the requested position is before the beginning of the file.

EXAMPLES

1. To seek to the beginning of a file:

```
lseek(fd, 0L, 0);
```

lseek will return the value zero (0) since the current position in the file is character (or byte) number zero.

2. To seek to the character following the last character in the file:

```
pos = lseek(fd, 0L, 2);
```

The variable *pos* will contain the current position of the end of file, plus one.

3. To seek backward five bytes:

```
lseek(fd, -5L, 1);
```

The third parameter, 1, sets the origin at the current position in the file. The offset is -5. The new position will be the origin plus the offset. So the effect of this call is to move backward a total of five characters.

4. To skip five characters when reading in a file:

```
read(fd, buf, count);
lseek(fd, 5L, 1);
read(fd, buf, count);
```

NAME

malloc, calloc, realloc, free - memory allocation

SYNOPSIS

```
char *malloc(size)
unsigned size;

char *calloc(nelem, elemsize)
unsigned nelem, elemsize;

char *realloc(ptr, size)
char *ptr;
unsigned size;

free(ptr)
char *ptr;
```

DESCRIPTION

These functions are used to allocate memory from the "heap", that is, the section of memory available for dynamic storage allocation.

malloc allocates a block of *size* bytes, and returns a pointer to it.

calloc allocates a single block of memory which can contain *nelem* elements, each *elemsize* bytes big, and returns a pointer to the beginning of the block. Thus, the allocated block will contain (*nelem* * *elemsize*) bytes. The block is initialized to zeroes.

realloc changes the size of the block pointed at by *ptr* to *size* bytes, returning a pointer to the block. If necessary, a new block will be allocated of the requested size, and the data from the original block moved into it. The block passed to *realloc* can have been freed, provided that no intervening calls to *calloc*, *malloc*, or *realloc* have been made.

free deallocates a block of memory which was previously allocated by *malloc*, *calloc*, or *realloc*; this space is then available for reallocation. The argument *ptr* to *free* is a pointer to the block.

malloc and *free* maintain a circular list of free blocks. When called, *malloc* searches this list beginning with the last block freed or allocated coalescing adjacent free blocks as it searches. It allocates a buffer from the first large enough free block that it encounters. If this search fails, it calls *sbrk* to get more memory for use by these functions.

SEE ALSO

Memory Usage (O), break (S)

DIAGNOSTICS

malloc, *calloc* and *realloc* return a null pointer (0) if there is no available block of memory.

free returns -1 if it's passed an invalid pointer.

NAME

movmem, setmem, swapmem

SYNOPSIS

```
movmem(src, dest, length)          /* non-Unix function */
char *src, *dest;
int length;

setmem(area,length,value)          /* non-Unix function */
char *area;

swapmem(s1, s2, len)               /* non-Unix function */
char *s1, *s2;
```

DESCRIPTION

movmem copies *length* characters from the block of memory pointed at by *src* to that pointed at by *dest.*

movmem copies in such a way that the resulting block of characters at *dest* equals the original block at *src.*

setmem sets the character *value* in each byte of the block of memory which begins at *area* and continues for *length* bytes.

swapmem swaps the blocks of memory pointed at by *s1* and *s2.* The blocks are *len* bytes long.

NAME

open

SYNOPSIS

```
#include "fcntl.h"

open(name, mode)    /* calling sequence on most systems */
char *name;

/* calling sequence on some systems (see below): */
open(name, mode, param3)
char *name;
```

DESCRIPTION

open opens a device or file for unbuffered i/o. It returns an integer value called a file descriptor which is used to identify the file or device in subsequent calls to unbuffered i/o functions.

name is a pointer to a character string which is the name of the device or file to be opened. For details, see the overview section I/O.

mode specifies how the user's program intends to access the file. The choices are as follows:

mode	*meaning*
O_RDONLY	read only
O_WRONLY	write only
O_RDWR	read and write
O_CREAT	Create file, then open it
O_TRUNC	Truncate file, then open it
O_EXCL	Cause open to fail if file already exists; used with O_CREAT
O_APPEND	Position file for appending data

These open modes are integer constants defined in the files *fcntl.h*. Although the true values of these constants can be used in a given call to open, use of the symbolic names ensures compatibility with UNIX and other systems.

The calling program must specify the type of access desired by including exactly one of O_RDONLY, O_WRONLY, and O_RDWR in the mode parameter. The three remaining values are optional. They may be included by adding them to the mode parameter, as in the examples below.

By default, the open will fail if the file to be opened does not exist. To cause the file to be created when it does not already exist, specify the O_CREAT option. If O_EXCL is given in addition to O_CREAT, the open will fail if the file already exists; otherwise, the file is created.

If the O_TRUNC option is specified, the file will be truncated so that nothing is in it. The truncation is performed by simply erasing the file, if it exists, and then creating it. So it is not an error to use this option when the file does not exist.

Note that when O_TRUNC is used, O_CREAT is not needed.

If O_APPEND is specified, the current position for the file (that is, the position at which the next data transfer will begin) is set to the end of the file. For systems which don't keep track of the last character written to a file (for example, CP/M and Apple DOS), this positioning cannot always be correctly done. See the I/O overview section for details. Also, this option is not supported by UNIX.

param3 is not needed or used on many systems. If it is needed for your system, the System Dependent Library Functions chapter will contain a description of the *open* function, which will define this parameter.

If *open* does not detect an error, it returns an integer called a "file descriptor." This value is used to identify the open file during unbuffered i/o operations. The file descriptor is very different from the file pointer which is returned by *fopen* for use with buffered i/o functions.

SEE ALSO

I/O (O), Unbuffered I/O (O), Errors (O)

DIAGNOSTICS

If open encounters an error, it returns -1 and sets the global integer *errno* to a symbolic value which identifies the error.

EXAMPLES

1. To open the file, testfile, for read-only access:

```
fd = open("testfile", O_RDONLY);
```

If testfile does not exist *open* will just return -1 and set *errno* to ENOENT.

2. To open the file, sub1, for read-write access:

```
fd = open("sub1", O_RDWR+O_CREAT);
```

If the file does not exist, it will be created and then opened.

3. The following program opens a file whose name is given on the command line. The file must not already exist.

```
main(argc, argv)
char **argv;
{
    int fd;

    fd = open(*++argv, O_WRONLY+O_CREAT+O_EXCL
    if (fd = -1)  {
    if (errno == EEXIST)
    printf("file already exists\n");
    else if (errno == ENOENT)
        printf("unable to open file\n");
    else
        printf("open error\n");
}
```

NAME

printf, fprintf, sprintf, format
- formatted output conversion functions

SYNOPSIS

```
#include "stdio.h"

printf(fmt [,arg] ...)
char *fmt;

fprintf(stream, fmt [,arg] ...)
FILE *stream;
char *fmt;

sprintf(buffer, fmt [,arg] ...)
char *buffer, *fmt;

format(func, fmt, argptr)
int (*func)();
char *fmt;
unsigned *argptr;
```

DESCRIPTION

These functions convert and format their arguments (*arg* or *argptr*) according to the format specification *fmt*. They differ in what they do with the formatted result:

printf outputs the result to the standard output stream, *stdout*;

fprintf outputs the result to the stream specified in its first argument, *stream*;

sprintf places the result in the buffer pointed at by its first argument, *buffer*, and terminates the result with the null character, ' '.

format calls the function *func* with each character of the result. In fact, *printf*, *fprintf*, and *sprintf* call *format* with each character that they generate.

These functions are in both *c.lib* and *m.lib*, the difference being that the *c.lib* versions don't support floating point conversions. Hence, if floating point conversion is required, the *m.lib* versions must be used. If floating point conversion isn't required, either version can be used. To use *m.lib*'s version, *m.lib* must be specified before *c.lib* at the time the program is linked.

The character string pointed at by the *fmt* parameter, which directs the print functions, contains two types of items: ordinary characters, which are simply output, and conversion specifications, each of which causes the conversion and output of the next successive *arg*.

A conversion specification begins with the character % and continues with:

* An optional minus sign (-) which specifies left adjustment of the converted value in the output field;
* An optional digit string specifying the 'field width' for the conversion. If the converted value has fewer characters than this, enough blank characters will be output to make the total number of characters output equals the field width. If the converted value has more characters than the field width, it will be truncated. The blanks are output before or after the value, depending on the presence or absence of the left- adjustment indicator. If the field width digits have a leading 0, 0 is used as a pad character rather than blank.
* An optional period, '.', which separates the field width from the following field;
* An optional digit string specifying a precision; for floating point conversions, this specifies the number of digits to appear after the decimal point; for character string conversions, this specifies the maximum number of characters to be printed from a string;
* Optionally, the character *l*, which specifies that a conversion which normally is performed on an *int* is to be performed on a *long*. This applies to the d, o, and x conversions.
* A character which specifies the type of conversion to be performed.

A field width or precision may be * instead of a number, specifying that the next available *arg*, which must be an *int*, supplies the field width or precision.

The conversion characters are:

d, *o*, or *x* The *int* in the corresponding *arg* is converted to decimal, octal, or hexadecimal notation, respectively, and output;

u The unsigned integer *arg* is converted to decimal notation;

c The character *arg* is output. Null characters are ignored;

s The characters in the string pointed at by *arg* are output until a null character or the number of characters indicated by the precision is reached. If the precision is zero or missing, all characters in the string, up to the terminating null, are output;

f The float or double *arg* is converted to decimal notation in the style '[-]ddd.ddd'. The number

of d's after the decimal point is equal to the precision given in the conversion specification. If the precision is missing, it defaults to six digits. If the precision is explicitly 0, the decimal point is also not printed.

e The float or double *arg* is converted to the style '[-]d.ddde[-]dd', where there is one digit before the decimal point and the number after is equal to the precision given. If the precision is missing, it defaults to six digits.

g The float or double *arg* is printed in style d, f, or e, whichever gives full precision in minimum space.

% Output a %. No argument is converted.

SEE ALSO

Standard I/O (O)

EXAMPLES

1. The following program fragment:

```
char *name; float amt;
printf("your total, %s, is $%f\n", name, amt);
```

will print a message of the form

```
your total, Alfred, is $3.120000
```

Since the precision of the %f conversion wasn't specified, it defaulted to six digits to the right of the decimal point.

2. This example modifies example 1 so that the field width for the %s conversion is three characters, and the field width and precision of the %f conversion are 10 and 2, respectively. The %f conversion will also use 0 as a pad character, rather than blank.

```
char *name; float amt;
printf("your total, %3s, is $%10.2f\n", name, amt);
```

3. This example modifies example 2 so that the field width of the %s conversion and the precision of the %f conversion are taken from the variables nw and ap:

```
char *name; float amt; int nw, ap;
printf("your total %*s,is $%10.*f\n",nw,name,ap,amt);
```

4. This example demonstrates how to use the *format* function by listing *printf*, which calls *format* with each character that it generates.

```
printf(fmt,args)
char *fmt; unsigned args;
{
    extern int putchar();
    format(putchar,fmt,&args);
}
```

NAME

putc, aputc, putchar, putw, puterr
- put character or word to a stream

SYNOPSIS

```
#include "stdio.h"

putc(c, stream)
char c;
FILE *stream;

aputc(c, stream)            /* non-Unix function */
char c;
FILE *stream;

putchar(c)
char c;

putw(w,stream)
FILE *stream;

puterr(c)                   /* non-Unix function */
char c;
```

DESCRIPTION

putc writes the character *c* to the named output stream. It returns *c* as its value.

aputc is used to write text characters to files and devices. It generally behaves like *putc*, and writes a single character to a stream. It differs from *putc* as follows:

* When a newline character is passed to *aputc*, an end- of-line sequence (eg, carriage return followed by line feed on CP/M, and carriage return only on Apple DOS) is written to the stream;
* The most significant bit of a character is set to zero before being written to the stream.
* *aputc* is not a UNIX function. It is, however, supported on all Aztec C systems, and provides a convenient, system-independent way for a program to write text.
* *putchar* writes the character *c* to the standard output stream, stdout. It's identical to *aputc(c, stdout)*.
* *putw* writes the word *w* to the specified stream. It returns *w* as its value. *putw* neither requires nor causes special alignment in the file.
* *puterr* writes the character *c* to the standard error stream, stderr. It's identical to *aputc(c, stderr)*. It is not a UNIX function.

SEE ALSO

Standard I/O

DIAGNOSTICS

These functions return EOF (-1) upon error. In this case, an error code is set in the global integer *errno*.

NAME

puts, fputs - put a character string on a stream

SYNOPSIS

```
#include "stdio.h"

puts(s)
char *s;

fputs(s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

puts writes the null-terminated string *s* to the standard output stream, stdout, and then an end-of-line sequence. It returns a non-negative value if no errors occur.

fputs copies the null-terminated string *s* to the specified output stream. It returns 0 if no errors occur.

Both functions write to the stream using *aputc*. Thus, they can only be used to write text. See the PUTC section for more details on *aputc*.

Note that *puts* and *fputs* differ in this way: On encountering a newline character, *puts* writes an end-of-line sequence and *fputs* doesn't.

SEE ALSO

Standard I/O (O), putc

DIAGNOSTICS

If an error occurs, these functions return EOF (-1) and set an error code in the global integer *errno*.

NAME

qsort - sort an array of records in memory

SYNOPSIS

```
qsort(array, number, width, func)
char *array;
unsigned number;
unsigned width;
int (*func)();
```

DESCRIPTION

qsort sorts an array of elements using Hoare's Quicksort algorithm. *array* is a pointer to the array to be sorted; *number* is the number of record to be sorted; *width* is the size in bytes of each array element; *func* is a pointer to a function which is called for a comparison of two array elements.

func is passed pointers to the two elements being compared. It must return an integer less than, equal to, or greater than zero, depending on whether the first argument is to be considered less than, equal to, or greater than the second.

EXAMPLE

The Aztec linker, LN, can generate a file of text containing a symbol table for a program. Each line of the file contains an address at which a symbol is located, followed by a space, followed by the symbol name. The following program reads such a symbol table from the standard input, sorts it by address, and writes it to standard output.

```
#include "stdio.h"
#define MAXLINES 2000
#define LINESIZE 16
char *lines[MAXLINES], *malloc();

main()
{
    int i,numlines, cmp();
    char buf[LINESIZE];

    for (numlines=0; numlines<MAXLINES; ++numlines){
        if (gets(buf) == NULL)
            break;
        lines[numlines] = malloc(LINESIZE);
        strcpy(lines[numlines], buf);
    }
    qsort(lines, numlines, 2, cmp);
    for (i = 0; i <numlines; ++i)
        printf("%s\n", lines[i]);
}

cmp(a,b)
char **a, **b;
{
    return strcmp(*a, *b);
}
```

NAME

ran - random number generator

SYNOPSIS

double ran()

DESCRIPTION

ran returns as its value a random number between 0.0 and 1.0.

NAME

read - read from device or file without buffering

SYNOPSIS

read (fd, buf,bufsize)
int fd, bufsize; char *buf;

DESCRIPTION

read reads characters from a device or disk file which has been previously opened by a call to *open* or *creat.* In most cases, the information is read directly into the caller's buffer.

fd is the file descriptor which was returned to the caller when the device or file was opened.

buf is a pointer to the buffer into which the information is to be placed.

bufsize is the number of characters to be transferred.

If *read* is successful, it returns as its value the number of characters transferred.

If the returned value is zero, then end-of-file has been reached, immediately, with no bytes read.

SEE ALSO

Unbuffered I/O (O), open, close

DIAGNOSTICS

If the operation isn't successful, *read* returns -1 and places a code in the global integer *errno.*

NAME

rename - rename a disk file

SYNOPSIS

```
rename(oldname, newname)        /* non-Unix function */
char *oldname,*newname;
```

DESCRIPTION

rename changes the name of a file.

oldname is a pointer to a character array containing the old file name, and *newname* is a pointer to a character array containing the new name of the file.

If successful, *rename* returns 0 as its value; if unsuccessful, it returns -1.

If a file with the new name already exists, *rename* sets E_EXIST in the global integer *errno* and returns -1 as its value without renaming the file.

NAME

scanf, fscanf, sscanf - formatted input conversion

SYNOPSIS

```
#include "stdio.h"

scanf(format [,pointer] ...)
char *format;

fscanf(stream, format [,pointer] ...)
FILE *stream;
char *format;

sscanf(buffer, format [,pointer] ...)
char *buffer, *format;
```

DESCRIPTION

These functions convert a string or stream of text characters, as directed by the control string pointed at by the *format* parameter, and place the results in the fields pointed at by the *pointer* parameters.

The functions get the text from different places:

scanf gets text from the standard input stream, *stdin*;

fscanf gets text from the stream specified in its first parameter, *stream*;

sscanf gets text from the buffer pointed at by its first parameter, *buffer*.

The scan functions are in both *c.lib* and *m.lib*, the difference being that the *c.lib* versions don't support floating point conversions. Hence, if floating point conversion is required, the *m.lib* versions must be used. If floating point conversions aren't required, either version can be used. To use *m.lib*'s version, *m.lib* must be specified before *c.lib* when the program is linked.

The control string pointed at by *format* contains the following 'control items':

* Conversion specifications;
* 'White space' characters (space, tab newline);
* Ordinary characters; that is, characters which aren't part of a conversion specification and which aren't white space.

A scan function works its way through a control string, trying to match each control item to a portion of the input stream or buffer. During the matching process, it fetches characters one at a time from the input. When a character is fetched which isn't appropriate for the control item being matched, the scan function pushes it back into the input stream or buffer and

finishes processing the current control item. This pushing back frequently gives unexpected results when a stream is being accessed by other i/o functions, such as *getc*, as well as the scan function. The examples below demonstrate some of the problems that can occur.

The scan function terminates when it first fails to match a control item or when the end of the input stream or buffer is reached. It returns as its value the number of matched conversion specifications, or EOF if the end of the input stream or buffer was reached.

Matching 'white space' characters

When a white space character is encountered in the control string, the scan function fetches input characters until the first non-white-space character is read. The non-white-space character is pushed back into the input and the scan function proceeds to the next item in the control string.

Matching ordinary characters

If an ordinary character is encountered in the control string, the scan function fetches the next input character. If it matches the ordinary character, the scan function simply proceeds to the next control string item. If it doesn't match, the scan function terminates.

Matching conversion specifications

When a conversion specification is encountered in the control string, the scan function first skips leading white space on the input stream or buffer. It then fetches characters from the stream or buffer until encountering one that is inappropriate for the conversion specification. This character is pushed back into the input.

If the conversion specification didn't request assignment suppression (discussed below), the character string which was read is converted to the format specified by the conversion specification, the result is placed in the location pointed at by the current pointer argument, and the next pointer argument becomes current. The scan function then proceeds to the next control string item.

If assignment suppression was requested by the conversion specification, the scan function simply ignores the fetched input characters and proceeds to the next control item.

Details of input conversion

A conversion specification consists of:

* The character '%', which tells the scan function that it

has encountered a conversion specification;
* Optionally, the assignment suppression character '*';
* Optionally, a 'field width'; that is, a number specifying the maximum number of characters to be fetched for the conversion;
* A conversion character, specifying the type of conversion to be performed.

If the assignment suppression character is present ina conversion specification, the scan function will fetch characters as if it was going to perform the conversion, ignore them, and proceed to the next control string item.

The following conversion characters are supported:

% A single '%' is expected in the input; no assignment is done.

d A decimal integer is expected; the input digit string is converted to binary and the result placed in the *int* field pointed at by the current *pointer* argument;

o An octal integer is expected; the corresponding *pointer* should point to an *int* field in which the converted result will be placed;

x A hexadecimal integer is expected; the converted value will be placed in the *int* field pointed at by the current *pointer* argument;

s A sequence of characters delimited by white space characters is expected; they, plus a terminating null character, are placed in the character array pointed at by the current *pointer* argument.

c A character is expected. It is placed in the *char* field pointed at by the current *pointer*. The normal skip over leading white space is not done; to read a single char after skipping leading white space, use '%1s'. The field width parameter is ignored, so this conversion can be used only to read a single character.

[A sequence of characters, optionally preceded by white space but not terminated by white space is expected. The input characters, plus a terminating null character, are placed in the character array pointed at by the current *pointer* argument. The left bracket is followed by:

* Optionally, a '^' or '~' character;
* A set of characters;
* A right bracket, ']'.

If the first character in the set isn't ^ or ~, the set specifies characters which are allowed; characters are fetched from the input until one is read which isn't in the set.

If the first character in the set is ^ or ~, the set specifies characters which aren't allowed; characters are fetched from the input until one is read which is in the set.

e A floating point number is expected. The input string is converted to floating point format and stored in the *float* field pointed at by the current *pointer* argument. The input format for floating point numbers consists of an optionally signed string of digits, possibly containing a decimal point, optionally followed by an exponent field consisting of an E or e followed by an optionally signed digit.

The conversion characters d, o, and x can be capitalized or preceded by *l* to indicate that the corresponding *pointer* is to a *long* rather than an *int*. Similarly, the conversion characters e and f can be capitalized or preceded by *l* to indicate that the corresponding *pointer* is to a *double* rather than a *float*.

The conversion characters o, x, and d can be optionally preceded by *h* to indicate that the corresponding *pointer* is to a *short* rather than an *int*. Since *short* and *int* fields are the same in Aztec C, this option has no effect.

SEE ALSO

Standard I/O (O)

EXAMPLES

1. In this program fragment, *scanf* is used to read values for the int x, the float y, and a character string into the char array z:

```
int x; float y; char z[50];
scanf("%d%f%s", &x, &y, z);
```

The input line

```
32 75.36e-1 rufus
```

will assign 32 to x, 7.536 to y, and "rufus " to z. *scanf* will return 3 as its value, signifying that three conversion specifications were matched.

The three input strings must be delimited by 'white space' characters; that is, by blank, tab, and newline characters. Thus, the three values could also be entered on separate

lines, with the white space character newline used to separate the values.

2. This example discusses the problems which may arise when mixing *scanf* and other input operations on the same stream.

   In the previous example, the character string entered for the third variable, z, must also be delimited by white space characters. In particular, it must be terminated by a space, tab, or newline character. The first such character read by *scanf* while getting characters for z will be 'pushed back' into the standard input stream. When another read of stdin is made later, the first character returned will be the white space character which was pushed back.

   This 'pushing back' can lead to unexpected results for programs that read stdin with functions in addition to *scanf*. Suppose that the program in the first example wants to issue a *gets* call to read a line from stdin, following the *scanf* to stdin. *scanf* will have left on the input stream the white space character which terminated the third value read by *scanf*. If this character is a newline, then *gets* will return a null string, because the first character it reads is the pushed back newline, the character which terminates *gets*. This is most likely not what the program had in mind when it called *gets*.

   It is usually unadvisable to mix *scanf* and other input operations on a single stream.

3. This example discusses the behavior of *scanf* when there are white space characters in the control string.

   The control string in the first example was "%d%f%s". It doesn't contain or need any white space, since *scanf*, when attempting to match a conversion specification, will skip leading white space. There's no harm in having white space before the %d, between the %d and %f, or between the %f and %s. However, placing a white space character after the %s can have unexpected results. In this case, *scanf* will, after having read a character string for z, keep reading characters until a non-white-space character is read. This forces the operator to enter, after the three values for x, y, and z, a non-white space character; until this is done, *scanf* will not terminate.

   The programmer might place a newline character at the end of a control string, mistakenly thinking that this will circumvent the problem discussed in example 2. One might think that *scanf* will treat the newline as it would an

ordinary character in the control string; that is, that *scanf* will search for, and remove, the terminating newline character from the input stream after it has matched the z variable. However, this is incorrect, and should be remembered as a common misinterpretation.

4. *scanf* only reads input it can match. If, for the first example, the input line had been

   32 rufus 75.36e-1

   scanf would have returned with value 1, signifying that only one conversion specification had been matched. x would have the value 32, y and z would be unchanged. All characters in the input stream following the 32 would still be in the input stream, waiting to be read.

5. One common problem in using *scanf* involves mismatching conversion specifications and their corresponding arguments. If the first example had declared y to be a double, then one of the following statements would have been required:

   ```
   scanf("%d%lf%s", &x, &y, z);
   ```

   or

   ```
   scanf("%d%F%s", &x, &y, z);
   ```

   to tell *scanf* that the floating point variable was a double rather than a float.

6. Another common problem in using *scanf* involves passing *scanf* the value of a variable rather than its address. The following call to *scanf* is incorrrect:

   ```
   int x; float y; char z[50];
   scanf("%d%f%s", x, y, z);
   ```

   scanf has been passed the value contained in x and y, and the address of z, but it requires the address of all three variables. The "address of" operator, &, is required as a prefix to x and y. Since z is an array, its address is automatically passed to *scanf*, so z doesn't need the & prefix, although it won't hurt if it is given.

7. Consider the following program fragment:

   ```
   int x; float y; char z[50];
   scanf("%2d%f%*d%[1234567890]", &x, &y, z);
   ```

   When given the following input:

   12345 678 90a65

   scanf will assign 12 to x, 345.0 to y, skip '678', and place

the string '90 ' in z. The next call to *getchar* will return 'a'.

NAME

setbuf - assign buffer to a stream

SYNOPSIS

```
#include "stdio.h"

setbuf(stream, buf)
FILE *stream;
char *buf;
```

DESCRIPTION

setbuf defines the buffer that's to be used for the i/o stream *stream*. If *buf* is not a NULL pointer, the buffer that it points at will be used for the stream instead of an automatically allocated buffer. If *buf* is a NULL pointer, the stream will be completely unbuffered.

When *buf* is not NULL, the buffer it points at must contain BUFSIZ bytes, where BUFSIZ is defined in *stdio.h*.

setbuf must be called after the stream has been opened, but before any read or write operations to it are made.

If the user's program doesn't specify the buffer to be used for a stream, the standard i/o functions will dynamically allocate a buffer for the stream, by calling the function *malloc*, when the first read or write operation is made on the stream. Then, when the stream is closed, the dynamically allocated buffer is freed by calling *free*.

SEE ALSO

Standard I/O (O), malloc

NAME

setjmp, longjmp - non-local goto

SYNOPSIS

#include "setjmp.h"

setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;

DESCRIPTION

These functions are useful for dealing with errors encountered by the low-level functions of a program.

setjmp saves its stack environment in the memory block pointed at by *env* and returns 0 as its value.

longjmp causes execution to continue as if the last call to *setjmp* was just terminating with value *val. val* cannot be zero.

The parameter *env* is a pointer to a block of memory which can be used by *setjmp* and *longjmp.* The block must be defined using the typedef *jmp_buf.*

WARNING

longjmp must not be called without *env* having been initialized by a call to *setjmp.* It also must not be called if the function that called *setjmp* has since returned.

EXAMPLE

In the following example, the function *getall* builds a record pertaining to a customer and returns the pointer to the record if no errors were encountered and 0 otherwise.

getall calls other functions which actually build the record. These functions in turn call other functions, which in turn ...

getall defines, by calling *setjmp,* a point to which these functions can branch if an unrecoverable error occurs. The low level functions abort by calling *longjmp* with a non-zero value.

If a low level function aborts, execution continues in *getall* as if its call to *setjmp* had just terminated with a non-zero value. Thus by testing the value returned by *setjmp getall* can determine whether *setjmp* is terminating because a low level function aborted.

```
#include "setjmp.h"

jmp_buf envbuf; /* environment saved here by setjmp */

getall(ptr)
char *ptr; /* ptr to record to be built */
{
    if (setjmp(envbuf))
        /* a low level function has aborted */
        return 0;
    getfield1(ptr);
    getfield2(ptr);
    getfield3(ptr);
    return ptr;
}
```

Here's one of the low level functions:

```
getsubfld21(ptr)
char *ptr;
{
    ...
    if (error)
        longjmp(envbuf, -1);
    ...
}
```

NAME

trigonometric functions:
sin, cos, tan, cotan, asin, acos, atan, atan2

SYNOPSIS

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

double tan(x)
double x;

double cotan(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

double atan2(x,y)
double x;
```

DESCRIPTION

sin, *cos*, *tan*, and *cotan* return trigonometric functions of radian arguments.

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

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

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

atan2 returns the arc tangent of x/y in the range -pi to pi.

SEE ALSO

Errors (O)

DIAGNOSTICS

If a trig function can't perform the computation, it returns an arbitrary value and sets a code in the global integer *errno*; otherwise, it returns the computed number, without modifying *errno*.

A function will return the symbolic value EDOM if the argument is invalid, and the value ERANGE if the function value can't be computed. EDOM and ERANGE are defined in the file *errno.h*.

The values returned by the trig functions when the computation can't be performed are listed below. The symbolic values are defined in *math.h.*

function	error	f(x)	meaning
sin	ERANGE	0.0	abs(x) > XMAX
cos	ERANGE	0.0	abs(x) > XMAX
tan	ERANGE	0.0	abs(x) > XMAX
cotan	ERANGE	HUGE	0<x< XMIN
cotan	ERANGE	-HUGEi	-XMIN <x <0
cotan	ERANGE	0.0	abs(x) >= XMAX
asin	EDOM	0.0	abs(x) > 1.0
acos	EDOM	0.0	abs(x) > 1.0
atan2	EDOM	0.0	x = y = 0

NAME

sinh, cosh, tanh

SYNOPSIS

```
#include <math.h>

double sinh(x)
double x;

double cosh(x)
double x;

double tanh(x)
double x;
```

DESCRIPTION

These functions compute the hyperbolic functions of their arguments.

SEE ALSO

Errors (O)

DIAGNOSTICS

If the absolute value of the argument to *sinh* or *cosh* is greater than 348.6, the function sets the symbolic value ERANGE in the global integer *errno* and returns a huge value. This code is defined in the file *errno.h*.

If no error occurs, the function returns the computed value without modifying *errno*.

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, index, rindex - string operations

SYNOPSIS

```
char *strcat(s1, s2)
char *s1, *s2;

char *strncat(s1, s2, n)
char *s1, *s2;

strcmp(s1, s2)
char *s1, *s2;

strncmp(s1, s2, n)
char *s1, s2;

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;

strlen(s)
char *s;

char *index(s, c)
char *s;

char *rindex(s, c)
char *s;
```

DESCRIPTION

These functions operate on null-terminated strings, as follows:

strcat appends a copy of string *s2* to string *s1*. *strncat* copies at most *n* characters. Both terminate the resulting string with the null character (\0) and return a pointer to the first character of the resulting string.

strcmp compares its two arguments and returns an integer greater than, equal, or less than zero, according as *s1* is lexicographically greater than, equal to, or less than *s2*. *strncmp* makes the same comparison but looks at *n* characters at most.

strcpy copies string *s2* to *s1* stopping after the null character has been moved. *strncpy* copies exactly *n* characters: if *s2* contains less than *n* characters, null characters will be appended to the resulting string until *n* characters have been moved; if *s2* contains *n* or more characters, only the first *n* will be moved, and the resulting string will not be null terminated.

strlen returns the number of characters which occur in *s* up to the first null character.

index returns a pointer to the first occurrance of the character *c* in string *s*, or zero if *c* isn't in the string.

rindex returns a pointer to the last occurrance of the character *c* in string *s*, or zero if *c* isn't in the string.

NAME

toupper, tolower

SYNOPSIS

toupper(c)

tolower(c)

#include "ctype.h"

_toupper(c)

_tolower(c)

DESCRIPTION

toupper converts a lower case character to upper case: if *c* is a lower case character, *toupper* returns its upper case equivalent as its value, otherwise *c* is returned.

tolower converts an upper case character to lowr case: if *c* is an upper case character *tolower* returns its lower case equivalent, otherwise *c* is returned.

toupper and *tolower* do not require the header file *ctype.h.*

_toupper and *_tolower* are macro versions of *toupper* and *tolower*, respectively. They are defined in *ctype.h.* The difference between the two sets of functions is that the macro versions will sometimes translate non-alphabetic characters, whereas the function versions don't.

NAME

ungetc - push a character back into input stream

SYNOPSIS

```
#include "stdio.h"

ungetc(c, stream)
FILE *stream;
```

DESCRIPTION

ungetc pushes the character *c* back on an input stream. That character will be returned by the next *getc* call on that stream. *ungetc* returns *c* as its value.

Only one character of pushback is guaranteed. EOF cannot be pushed back.

SEE ALSO

Standard I/O (O)

DIAGNOSTICS

ungetc returns EOF (-1) if the character can't be pushed back.

NAME

unlink

SYNOPSIS

unlink(name)
char *name;

DESCRIPTION

unlink erases a file.

name is a pointer to a character array containing the name of the file to be erased.

unlink returns 0 if successful.

DIAGNOSTICS

unlink returns -1 if it couldn't erase the file and places a code in the global integer *errno* describing the error.

NAME

write

SYNOPSIS

write(fd,buf,bufsize)
int fd, bufsize; char *buf;

DESCRIPTION

write writes characters to a device or disk which has been previously opened by a call to *open* or *creat.* The characters are written to the device or file directly from the caller's buffer.

fd is the file descriptor which was returned to the caller when the device or file was opened.

buf is a pointer to the buffer containing the characters to be written.

bufsize is the number of characters to be written.

If the operation is successful, *write* returns as its value the number of characters written.

SEE ALSO

Unbuffered I/O (O) , open, close, read

DIAGNOSTICS

If the operation is unsuccessful, *write* returns -1 and places a code in the global integer *errno.*

STYLE

Chapter Contents

Style ..... style
1. Introduction 3
2. Structured Programming 7
3. Top-down Programming 8
4. Defensive Programming and Debugging 10
5. Things to watch out for 15

Style

This section was written for the programmer who is new to the C language, to communicate the special character of C and the programming practices for which it is best suited. This material will ease the new user's entry into C. It gives meaning to the peculiarities of C syntax, in order to avoid the errors which will otherwise disappear only with experience.

1. Introduction

what's in it for me?

These are the benefits to be reaped by following the methods presented here:

* Reduced debugging times;
* Increased program efficiency;
* Reduced software maintenance burden.

The aim of the responsible programmer is to write straightforward code, which makes his programs more accessible to others. This section on style is meant to point out which programming habits are conducive to successful C programs and which are especially prone to cause trouble.

The many advantages of C can be abused. Since C is a terse, subtle language, it is easy to write code which is unclear. This is contrary to the "philosophy" of C and other structured programming languages, according to which the structure of a program should be clearly defined and easily recognizable.

keep it simple

There are several elements of programming style which make C easier to use. One of these is *simplicity*. Simplicity means *keep it simple*. You should be able to see exactly what your code will do, so that when it doesn't you can figure out why.

A little suspicion can also be useful. The particular "problem areas" which are discussed later in this section are points to check when code "looks right" but does not work. A small omission can cause many errors.

learn the C idioms

C becomes more valuable and more flexible with time. Obviously, elementary problems with syntax will disappear. But more importantly,

C can be described as "idiomatic." This means that certain expressions become part of a standard vocabulary used over and over.

For example,

```
while ((c = getchar()) != EOF)
```

is readily recognized and written by any C programmer. This is often used as the beginning of a loop which gets a character at a time from a source of input. Moreover, the inside set of parentheses, often omitted by a new C programmer, is rarely forgotten after this construct has been used a few times.

be flexible in using the library

The standard library contains a choice of functions for performing the same task. Certain combinations offer advantages, so that they are used routinely. For instance, the standard library contains a function, *scanf*, which can be used to input data of a given format. In this example, the function "scans" input for a floating point number:

```
scanf("%f", &flt_num);
```

There are several disadvantages to this function. An important debit is that it requires a lot of code. Also, it is not always clear how this function handles certain strings of input. Much time could be spent researching the behavior of this function. However, the equivalent to the above is done by the following:

```
flt_num = atof(gets(inp_buf));
```

This requires considerably less code, and is somewhat more straightforward. *gets* puts a line of input into the buffer, "inp_buf," and *atof* converts it to a floating point value. There is no question about what the input function is "looking for" and what it should find.

Furthermore, there is greater flexibility in the second method of getting input. For instance, if the user of the program could enter either a special command ("e" for exit) or a floating point value, the following is possible:

```
gets(inp_buf);
if (inp_buf[0] == 'e')
        exit(0);
flt_num = atof(inp_buf);
```

Here, the first character of input is checked for an "e", before the input is converted to a float.

The relative length of the library description of the *scanf* function is an indication of the problems that can arise with that and related functions.

write readable code

Readability can be greatly enhanced by adhering to what common sense dictates. For instance, most lines can easily accommodate more than one statement. Although the compiler will accept statements which are packed together indiscriminately, the logic behind the code will be lost. Therefore, it makes sense to write no more than one statement per line.

In a similar vein, it is desirable to be generous with whitespace. A blank space should separate the arithmetic and assignment operators from other symbols, such as variable names. And when parentheses are nested, dividing them with spaces is not being too prudent. For example,

```
if((fp=fopen("filename","r")==NULL))
```

is not the same as

```
if ( (fp = fopen("filename", "r")) == NULL )
```

The first line contains a misplaced parenthesis which changes the meaning of the statement entirely. (A file is opened but the file pointer will be null.) If the statement was expanded, as in the second line, the problem could be easily spotted, if not avoided altogther.

use straightforward logical expressions

Conditionals are apt to grow into long expressions. They should be kept short. Conditionals which extend into the next line should be divided so that the logic of the statement can be visualized at a glance. Another solution might be to reconsider the logic of the code itself.

learn the rules for expression evaluation

Keep in mind that the evaluation of an expression depends upon the order in which the operators are evaluated. This is determined from their relative precedence.

Item 7 in the list of "things to watch out for", below, gives an example of what may happen when the evaluation of a boolean expression stops "in the middle". The rule in C is that a boolean will be evaluated only until the value of the expression can be determined.

Item 8 gives a well known example of an "undefined" expression, one whose value is not strictly determined.

In general, if an expression depends upon the order in which it is evaluated, the results may be dubious. Though the result may be strictly defined, you must be certain you know what that definition is.

a matter of taste

There are several popular styles of indentation and placement of the braces enclosing compound statements. Whichever format you

adopt, it is important to be consistent. Indentation is the accepted way of conveying the intended nesting of program statements to other programmers. However, the compiler understands only braces. Making them as visible as possible will help in tracking down nesting errors later.

However much time is devoted to writing readible code, C is low-level enough to permit some very peculiar expressions.

```
/* It is important to insert comments on a regular basis! */
```

Comments are especially useful as brief introductions to function definitions.

In general, moderate observance of these suggestions will lessen the number of "tricks" C will play on you-- even after you have mastered its syntax.

2. Structured Programming

"Structured programming" is an attempt to encourage programming characterized by method and clarity. It stems from the theory that any programming task can be broken into simpler components. The three basic parts are statements, loops, and conditionals. In C, these parts are, respectively, anything enclosed by braces or ending with a semicolon; *for*, *while* and *do-while*; *if-else*.

modularity and block structure

Central to structured programming is the concept of modularity. In one sense, any source file compiled by itself is a module. However, the term is used here with a more specific meaning. In this context, modularity refers to the independence or isolation of one routine from another. For example, a routine such as *main()* can call a function to do a given task even though it does not know how the task is accomplished or what intermediate values are used to reach the final result.

Sections of a program set aside by braces are called "blocks". The "privacy" of C's block structure ensures that the variables of each block are not inadvertently shared by other blocks. Any left brace ({) signals the beginning of a block, such as the body of a function or a *for* loop. Since each block can have its own set of variables, a left brace marks an opportunity to declare a temporary variable.

A function in C is a special block because it is called and is passed control of execution. A function is called, executes and returns. Essentially, a C program is just such a routine, namely, *main*.

A function call represents a task to be accomplished. Program statements which might otherwise appear as several obscure lines can be set aside in a function which satisfies a desired purpose. For instance, *getchar* is used to get a single character from standard input.

When a section of code must be modified, it is simpler to replace a single modular block than it is to delete a section of an unstructured program whose boundaries may be unclear at best. In general, the more precisely a block of program is defined, the more easily it can be changed.

3. Top-down Programming

"Top-down" programming is one method that takes advantage of structured programming features like those discussed above. It is a method of designing, writing, and testing a program from the most general function (i.e., (main()) to the most specific functions (such as getchar()).

All C programs begin with a function called main(). main() can be thought of as a supervisor or manager which calls upon other functions to perform specific tasks, doing little of the work itself. If the overall goal of the program can be considered in four parts (for instance, input, processing, error checking and output), then main() should call at least four other functions.

step one

The first step in the design of a program is to identify what is to be done and how it can be accomplished in a "programmable" way. The *main* routine should be greatly simplified. It needs to call a function to perform the crucial steps in the program. For example, it may call a function, init(), which takes care of all necessary startup initializations. At this point, the programmer does not even need to be certain of all the initializations that will take place in init().

All functions consist of three parts: a parameter list, body, and return value. The design of a function must focus on each of these three elements.

During this first stage of design, each function can be considered a black box. We are concerned only with what goes in and what comes out, not with what goes on inside.

Do not allow yourself to be distracted by the details of the implementation at this point. Flowcharts, pseudocode, decision tables and the like are useful at this stage of the implementation.

A detailed list of the data which is passed back and forth between functions is important and should not be neglected. The interface between functions is crucial.

Although all functions are written with a purpose in mind, it is easy to unwittingly merge two tasks into one. Sometimes, this may be done in the interests of producing a compact and efficient program function. However, the usual result is a bulky, unmanageable function. If a function grows very large or if its logic becomes difficult to comprehend, it should be reduced by introducing additional function calls.

step two

There comes a time when a program must pass from the design stage into the coding stage. You may find the top-down approach to

coding too restrictive. According to this scheme, the smallest and most specific functions would be coded last. It is our nature to tackle the most daunting problems first, which usually means coding the low-level functions.

Whereas the top-down approach is the preferred method for designing software, the bottom-up approach is often the most practical method for writing software. Given a good design, either method of implementation should produce equally good results.

One asset of top-down writing is the ability to provide immediate tests on upper level routines. Unresolved function calls can be satisfied by "dummy" functions which return a range of test values. When new functions are added, they can operate in an environment that has already been tested.

C functions are most effective when they are as mutually independent as is possible. This independence is encouraged by the fact that there is normally only one way into and one way out of a function: by calling it with specific arguments and returning a meaningful value. Any function can be modified or replaced so long as its entry and exit points are consistent with the calling function.

4. Defensive Programming and Debugging

"Defensive programming" obeys the same edict as defensive driving: trust no one to do what you expect. There are two sides to this rule of thumb. Defend against both the possibility of bad data or misuse of the program by the user, and the possibility of bad data generated by bad code.

Pointers, for example, are a prime source of variables gone astray. Even though the "theory" of pointers may be well understood, using them in new ways (or for the first time) requires careful consideration at each step. Pointers present the fewest problems when they appear in familiar settings.

faced with the unknown

When trying something new, first write a few test programs to make sure the syntax you are using is correct. For example, consider a buffer, *str_buf*, filled with null-terminated strings. Suppose we want to print the string which begins at offset *begin* in the buffer. Is this the way to do it?

```
printf("%s", str_buf[begin]);
```

A little investigation shows that str_buf[begin] is a character, not a pointer to a string, which is what is called for. The correct statement is

```
printf("%s", str_buf + begin);
```

This kind of error may not be obvious when you first see it. There are other topics which can be troublesome at first exposure. The promotion of data types within expressions is an example. Even if you are sure how a new construct behaves, it never hurts to doublecheck with a test program.

Certain programming habits will ease the bite of syntax. Foremost among these is simplicity of style. Top-down programming is aimed at producing brief and consequently simple functions. This simplicity should not disappear when the design is coded.

Code should appear as "idiomatic" as possible. Pointers can again provide an example: it is a fact of C syntax that arrays and pointers are one and the same. That is,

```
array[offset]
```

is the same as

```
*(array + offset)
```

The only difference is that an array name is not an lvalue; it is fixed. But mixing the two ways of referencing an object can cause confusion, such as in the last example. Choosing a certain idiom, which is known to behave a certain way, can help avoid many errors in usage.

when bugs strike

The assumption must be that you will have to return to the source code to make changes, probably due to what is called a bug. Bugs are characterized by their persistence and their tendency to multiply rapidly.

Errors can occur at either compile-time or run-time. Compile-time errors are somewhat easier to resolve since they are usually errors in syntax which the compiler will point out.

from the compiler

If the compiler does pick up an error in the source code, it will send an error code to the screen and try to specify where the error occurred. There are several peculiarities about error reporting which should be brought up right away.

The most noticeable of these peculiarities is the number of spurious errors which the compiler may report. This interval of inconsistency is referred to as the compiler's recovery. The safest way to deal with an unusually long list of errors is to correct the first error and then recompile before proceeding.

The compiler will specify where it "noticed" something was wrong. This does not necessarily indicate where you must make a change in the code. The error number is a more accurate clue, since it shows what the compiler was looking for when the error occurred.

if this ever happens to you

A common example of this is error 69: "missing semicolon." This error code will be put out if the compiler is expecting a semicolon when it finds some other character. Since this error most often occurs at the end of a line, it may not be reported until the first character of the following line-- recall that whitespace, such as a newline character, is ignored.

Such an error can be especially treacherous in certain situations. For example, a missing semicolon at the end of a #*include*'d file may be reported when the compiler returns to read input in the original file.

In general, it is helpful to look at a syntax error from the compiler's point of view.

Consider this error:

```
struct structag {
        char c;
        int i;
}

int j;
```

This should generate an error 16: "data type conflict". The arrow in the error message should show that the error was detected right after the "int" in the declaration of *j*. This means that the error has to do with something before that line, since there is nothing illegal about the *int* keyword.

By inspection, we may see that the semicolon is missing from the preceding line. If this fact escapes our notice, we still know that error 16 means this: the compiler found a declaration of the form

[data type] [data type] [symbol name]

where the two data types were incompatible. So while *shortint* is a good data type, *double int* is not. A small intuitive leap leads us to assume that the compiler has read our source as a kind of "struct int" declaration; *struct* is the only keyword preceding the *int* which could have caused this error. Since the compiler is reading the two declarations as a single statement, we must be missing a delimiter.

run-time errors

It takes a bit more ingenuity to locate errors which occur at run-time. In numerical calculations, only the most anomalous results will draw attention to themselves. Other bugs will generate output which will appear to have come from an entirely different program.

A bug is most useful when it is repeatable. Bugs which show up only "sometimes" are merely vexing. They can be caused by a corrupted disk file or a bad command from the user.

When an error can be consistently produced, its source can be more easily located. The nature of an error is a good clue as to its source. Much of your time and sanity will be preserved by setting aside a few minutes to reflect upon the problem.

Which modules are involved in the computation or process? Many possibilities can be eliminated from the start, such as pieces of code which are unrelated to the error.

The first goal is to determine, from a number of possibilities, which module might be the source of the bug.

checking input data

Input to the program can be checked at a low cost. Error checking of this sort should be included on a "routine" basis. For instance, "if ((fp=fopen("file","r"))==NULL)" should be reflex when a file is

opened. Any useful error handling can follow in the body of the *if*.

It is easy to check your data when you first get your hands on it. If an error occurs after that, you have a bug in your program.

printf it

It is useful to know where the data goes awry. One brute force way of tracking down the bug is to insert *printf* statements wherever the data is referenced. When an unexpected value comes up, a single module can be chosen for further investigation.

The printf search will be most effective when done with more refinement. Choose a suspect module. There are only two keys points to check: the entry and return of the function. *printf* the data in question just as soon as the function is entered. If the values are already incorrect, then you will want to make sure the correct data was passed in the function call.

If an incorrect value is returned, then the search is confined to the guilty function. Even if the function returns a good value, you may want to make sure it is handled correctly by the calling function.

If everything seems to be working, jump to the next tricky module and perform another check. When you find a bad result, you will still have to backtrack to discover precisely where the data was spoiled.

function calls

Be aware that data can be garbled in a funtion call. Function parameters must be declared when they are not two byte integers. For instance, if a function is called:

```
fseek(fp, 0, 0);
```

in order to "seek" to the beginning of a file, but the function is defined this way:

```
fseek(fp, offset, origin)
FILE *fp;
long offset;
int origin;
```

there will be unfortunate consequences.

The second parameter is expected to be a *long* integer (four bytes), but what is being passed is a *short* integer (two bytes). In a function call, the arguments are just being pushed onto the stack; when the function is entered, they are pulled off again. In the example, two bytes are being pushed on, but four bytes (whatever four bytes are there) are being pulled off.

The solution is just to make the second parameter a long, with a suffix (0L) or by the cast operator (as in (long)i).

A similar problem occurs when a non-integer return value is not declared in the calling function. For example, if *sqrt* is being called, it must be declared as returning a *double*:

```
double sqrt();
```

This method of debugging demonstrates the usefulness of having a solid design before a function is coded. If you know what should be going into a function and what should be coming out, the process of checking that data is made much simpler.

found it

When the guilty function is isolated, the difficulty of finding the bug is proportional to the simplicity of the code. However, the search can continue in a similar way. You should have a good notion of the purpose of each block, such as a loop. By inserting a *printf* in a loop, you can observe the effect of each pass on the data.

printf's can also point out which blocks are actually being executed. "Falling through" a test, such as an *if* or a *switch*, can be a subtle source of problems. Conditionals should not leave cases untested. An *else*, or a *default* in a *switch*, can rescue the code from unexpected input.

And if you are uncertain how a piece of code will work, it is usually worthwhile to set up small test programs and observe what happens. This is instructional and may reveal a bug or two.

5. Things to Watch Out for

Some errors arise again and again. Not all of them go away with experience. The following list will give you an idea of the kinds of things that can go wrong.

* **missing semicolon or brace**

The compiler will tell you when a missing semicolon or brace has introduced bad syntax into the code. However, often such an error will affect only the logical structure of the program; the code may compile and even execute. When this error is not revealed by inspection, it is usually brought out by a test *printf* which is executed too often or not enough. See compiler error 69.

* **assignment (=) vs comparison (==)**

Since variables are assigned values more often than they are tested for equality, the former operator was given the single keystroke: =. Notice that all the comparison tests with equality are two characters: <=, >= and ==.

* **misplaced semicolon**

When typing in a program, keep in mind that all source lines do not automatically end with a semicolon. Control lines are especially susceptible to an unwanted semicolon:

```
for (i=0; i<100; i++);
        printf("%d",i);
```

This example prints the single number 100.

* **division (/) vs escape sequence (\)**

C definitely distinguishes between these characters. The division sign resides below the question mark on a standard console; the backslash is generally harder to find.

* **character constant vs character string**

Character constants are actually integers equal to the ASCII values of the respective character. A character string is a series of characters terminated by a null character (\0). The appropriate delimiter is the single quote and double quote, respectively.

* **uninitialized variable**

At some point, all variables must be given values before they are used. The compiler will set global and static variables to zero, but automatic variables are guaranteed to contain garbage every time they are created.

* **evaluation of expressions**

For most operations in C, the order of evaluation is rigidly defined; thus, many expressions can be written without lots of parentheses.

However, the order in which unparenthesized expressions are evaluated are not always what you would expect; therefore, it's usually a good idea to use parentheses liberally in expressions where there may be doubt about the order of evaluation (in your mind or in the mind of someone who may later read your program).

For example, the result of the following example is 6:

```
int a = 2, b = 3, c = 4, d;
d = a + b / a * c;
```

The above expression is equivalent to the parenthesized expression *d = a + ((b / a) * c);*. You should probably use some parentheses in this expression, to make its effect clear to yourself and to others.

Consider this example:

```
if ( (c = 0) || (c = 1) )
        printf("%d", c);
```

"1" will be printed; since the first half of the conditional evaluates to zero, the second half must be also evaluated. But in this example:

```
if ( (c = 0) && (c = 1) )
        ;
printf("%d", c);
```

a "0" is printed. Since the first half evaluates to zero, the value of the conditional must be zero, or false, and evaluation stops. This is a property of the logical operators.

* **undefined order of evaluation**

Unfortunately, not all operators were given a complete set of instructions as to how they should be evaluated. A good example is the increment (or decrement) operator. For instance, the following is undefined:

```
i = ++i + --i/++i - i++;
```

How such an expression is evaluated by a particular implementation is called a "side effect." In general, side effects are to be avoided.

* **evaluation of boolean expressions**

Ands, ors and nots invite the programmer to write long conditionals whose very purpose is lost in the code. Booleans should be brief and to the point. Also, the bitwise logical operators must be fully parenthesized. The table in sections 2.12 and 18.1 of *The C Programming Language*, by Kernighan and Ritchie, shows their precedence in relation to other operators.

Here is an extreme example of how a lengthy boolean can be reduced:

```
if ((c = getchar()) != EOF && c >= 'a' && c <= 'z' &&
(c = getchar()) >= '1' && c <= '9')
   printf("good input\n");

if ((c = getchar()) != EOF)
   if (c >= 'a' && c <= 'z')
      if ((c = getchar()) >= '0' && c <= '9')
         printf("good input\n");
```

* **badly formed comments**

The theory of comment syntax is simply that everything occurring between a left /* and a right */ is ignored by the compiler. Nonetheless, a missing */ should not be overlooked as a possible error.

Note that comments cannot be nested, that is

```
/* /*  this will cause an error  */  */
```

And this could happen to you too:

```
/* the rest of this file is ignored until another comment /*
```

* **nesting error**

Remember that nesting is determined by braces and not by indentations in the text of the source. Nested *if* statements merit particular care since they are often paired with an *else*.

* **usage of else**

Every else must pair up with an *if*. When an *else* has inexplicably remained unpaired, the cause is often related to the first error in this list.

* **falling through the cases in a switch**

To maintain the most control over the *cases* in a *switch* statement, it is advisable to end each *case* with a *break*, including the last *case* in the *switch*.

* **strange loops**

The behavior of loops can be explored by inserting *printf* statements in the body of the loop. Obviously, this will indicate if the loop has even been entered at all in course of a run. A counter will show just how many times the loop was executed; a small slip-up will cause a loop to be run through once too often or seldom. The condition for leaving the loop should be doublechecked for accuracy.

* **use of strings**

All strings must be terminated by a null character in memory. Thus, the string, "hello", will occupy a six-element array; the sixth element is ' '. This convention is essential when passing a string to a standard library function. The compiler will append the null character to string constants automatically.

* **pointer vs object of a pointer**

The greatest difficulty in using pointers is being sure of what is needed and what is being used. Functions which take a pointer argument require an address in memory. The best way to ensure that the correct value is being passed is to keep track of what is being pointed to by which pointer.

* **array subscripting**

The first element in a C array has a subscript of zero. The array name without a subscript is actually a pointer to this element. Obviously, many problems can develop from an incorrect subscript. The most damaging can be subscripting out of bounds, since this will access memory above the array and overwrite any data there. If array elements or data stored with arrays are being lost, this error is a good candidate.

* **function interface**

During the design stage, the components of a program should be associated with functions. It is important that the data which is passed among or shared by these functions be explicitly defined in the preliminary design of the program. This will greatly facilitate the coding of the program since the interface between functions must be precise in several respects.

First of all, if the parameters of a function are established, a call can be made without the reservation that it will be changed later. There is less chance that the arguments will be of the wrong type or specified in the wrong order.

A function is given only a private copy of the variables it is passed. This is a good reason to decide while designing the program how functions should access the data they require. You will be able to detail the arguments to be passed in a function call, the global data which the function will alter, the value which the function will return and what declarations will be appropriate-- all without concern for how the function will be coded.

Argument declarations should be a fairly simple matter once these things are known. Note that this declaration list must stand before the left brace of the function body.

The type of the function is the same as the type of the value it returns. Functions must be declared just like any variable. And just like variables, functions will default to type int, that is, the compiler will assume that a function returns an integer if you do not tell it otherwise with a declaration. Thus if function f calls function g which returns a variable of type double, the following declaration is needed:

```
function f()
{
        double g(), bigfloat;

        g(bigfloat);
}
double g(arg)
double arg;
{
        return(arg);
}
```

* **be sure of what a function returns**

You will probably know very well what is returned by a function you have written yourself. But care should be taken when using functions coded by someone else. This is especially true of the standard library functions. Most of the supplied library functions will return an int or a char pointer where you might expect a char. For instance, getchar() returns an int, not a char. The functions supplied by Manx adhere to the UNIX model in all but a few cases.

Of course, the above applies to a function's arguments as well.

* **shared data**

Variables that are declared globally can be accessed by all functions in the file. This is not a very safe way to pass data to functions since once a global variable is altered, there is no returning it to its former state without an elaborate method of saving data. Moreover, global data must be carefully managed; a function may process the wrong variable and consequently inhibit any other function which depends on that data.

Since C provides for and even encourages private data, this definitely should not be a common bug.

COMPILER ERROR MESSAGES

Chapter Contents

Compiler Error Codes ..... err
1. Summary 4
2. Explanations 7
3. Fatal Error Messages 35

Compiler Error Messages

This chapter discusses error messages that can be generated by the compiler. It is divided into three sections: the first summarizes the messages, the second explains them, and the third discusses fatal compiler error messsages.

1. Summary of error codes

No. Interpretation

1: bad digit in octal constant
2: string space exhausted
3: unterminated string
4: internal error
5: illegal type for function
6: inappropriate arguments
7: bad declaration syntax
8: syntax error in typecast
9: array dimension must be constant
10: array size must be positive integer
11: data type too complex
12: illegal pointer reference
13: unimplemented type
14: internal
15: internal
16: data type conflict
17: unsupported data type
18: data type conflict
19: obsolete
20: structure redeclaration
21: missing }
22: syntax error in structure declaration
23: incorrect type for library function (Apprentice C only)
obsolete (other Aztec C compilers)
24: need right parenthesis or comma in arg list
25: structure member name expected here
26: must be structure/union member
27: illegal typecast
28: incompatible structures
29: illegal use of structure
30: missing : in ? conditional expression
31: call of non-function
32: illegal pointer calculation
33: illegal type
34: undefined symbol
35: typedef not allowed here
36: no more expression space
37: invalid expression for unary operator
38: no auto. aggregate initialization allowed
39: obsolete
40: internal
41: initializer not a constant
42: too many initializers

43: initialization of undefined structure
44: obsolete
45: bad declaration syntax
46: missing closing brace
47: open failure on include file
48: illegal symbol name
49: multiply defined symbol
50: missing bracket
51: lvalue required
52: obsolete
53: multiply defined label
54: too many labels
55: missing quote
56: missing apostrophe
57: line too long
58: illegal # encountered
59: macro too long
60: obsolete
61: reference of member of undefined structure
62: function body must be compound statement
63: undefined label
64: inappropriate arguments
65: illegal argument name
66: expected comma
67: invalid else
68: syntax error
69: missing semicolon
70: goto needs a label
71: statement syntax error in do-while
72: 'for' syntax error: missing first semicolon
73: 'for' syntax error: missing second semicolon
74: case value must be an integer constant
75: missing colon on case
76: too many cases in switch
77: case outside of switch
78: missing colon on default
79: duplicate default
80: default outside of switch
81: break/continue error
82: illegal character
83: too many nested includes
84: too many array dimensions
85: not an argument
86: null dimension in array
87: invalid character constant
88: not a structure
89: invalid use of register storage class
90: symbol redeclared

91: illegal use of floating point type
92: illegal type conversion
93: illegal expression type for switch
94: invalid identifier in macro definition
95: macro needs argument list
96: missing argument to macro
97: obsolete
98: not enough arguments in macro reference
99: internal
100: internal
101: missing close parenthesis on macro reference
102: macro arguments too long
103: #else with no #if
104: #endif with no #if
105: #endasm with no #asm
106: #asm within #asm block
107: missing #endif
108: missing #endasm
109: #if value must be integer constant
110: invalid use of : operator
111: invalid use of void expression
112: invalid use function pointer
113: duplicate case in switch
114: macro redefined
115: keyword redefined
116: field width must be > 0
117: invalid 0 length field
118: field is too wide
119: field not allowed here
120: invalid type for field
121: ptr to int conversion
122: ptr & int not same size
123: function ptr & ptr not same size
124: invalid ptr/ptr assignment
125: too many subscripts or indirection on integer

Error codes between 116 and 125 will not occur on Aztec C compilers whose version number is less than 3.

Error codes greater than 200 will occur only if there's something wrong with the compiler. If you get such an error, please send us the program that generated the error.

2. Explanations

1: bad digit in octal constant

The only numerals permitted in the base 8 (octal) counting system are zero through seven. In order to distinguish between octal, hexadecimal, and decimal constants, octal constants are preceded by a zero. Any number beginning with a zero must not contain a digit greater than seven. Octal constants look like this: 01, 027, 003. Hexadecimal constants begin with 0x (e.g., 0x1, 0xAA0, 0xFFF).

2: string space exhausted

The compiler maintains an internal table of the strings appearing in the source code. Since this table has a finite size, it may overflow during compilation and cause this error code. The table default size is about one or two thousand characters depending on the operating system. The size can be changed using the compiler option -Z. Through simple guesswork, it is possible to arrive at a table size sufficient for compiling your program.

3: unterminated string

All strings must begin and end with double quotes ("). This message indicates that a double quote has remained unpaired.

4: internal error

This error message should not occur. It is a check on the internal workings of the compiler and is not known to be caused by any particular piece of code. However, if this error code appears, please bring it to the attention of MANX. It could be a bug in the compiler. The release documentation enclosed with the product contains further information.

5: illegal type for function

The type of a function refers to the type of the value which it returns. Functions return an *int* by default unless they are declared otherwise. However, functions are not allowed to return aggregates (arrays or structures). An attempt to write a function such as *struct sam func()* will generate this error code. The legal function types are *char*, *int*, *float*, *double*, *unsigned*, *long*, *void* and a pointer to any type (including structures).

6: error in argument declaration

The declaration list for the formal parameters of a function stands immediately before the left brace of the function body, as shown below. Undeclared arguments default to *int*, though it is usually better practice to declare everything. Naturally, this declaration list may be empty, whether or not the function takes any arguments at all.

No other inappropriate symbols should appear before the left (open) brace.

```
badfunction(arg1, arg2)
shrt arg 1;    /* misspelled or invalid keyword */
double arg 2;
{ /* function body */
}

goodfunction(arg1,arg2)
float arg1;
int arg2;  /* this line is not required */
{ /* function body */
}
```

7: bad declaration syntax

A common cause of this error is the absence of a semicolon at the end of a declaration. The compiler expects a semicolon to follow a variable declaration unless commas appear between variable names in multiple declarations.

```
int i, j;          /* correct */
char c d;          /* error 7 */
char *s1, *s2
float k;           /* error 7 detected here */
```

Sometimes the compiler may not detect the error until the next program line. A missing semicolon at the end of a *#include*'d file will be detected back in the file being compiled or in another *#include* file. This is a good example of why it is important to examine the context of the error rather than to rely solely on the information provided by the compiler error message(s).

8: syntax error in type cast

The syntax of the cast operator must be carefully observed. A common error is to omit a parenthesis:

```
i = 3 * (int number);          /* incorrect usage */
i = 3 * ((int)number);         /* correct usage */
```

9: array dimension must be constant

The dimension given an array must be a constant of type *char*, *int*, or *unsigned*. This value is specified in the declaration of the array. See error 10.

10: array size must be positive integer

The dimension of an array is required to be greater than zero. A dimension less than or equal to zero becomes 1 by default. As can be seen from the following example, specifying a dimension of zero is not the same as leaving the brackets empty.

```
char badarray[0];          /* meaningless */
extern char goodarray[];   /* good */
```

Empty brackets are used when declaring an array that has been defined (given a size and storage in memory) somewhere else (that is, outside the current function or file). In the above example, *goodarray* is external. Function arguments should be declared with a null dimension:

```
func(s1,s2)
char s1[], s2[];
{
    ...
}
```

11: data type too complex

This message is best explained by example:

```
char *******foo;
```

The form of this declaration implies six pointers-to-pointers. The seventh asterisk indicates a pointer to a *char*. The compiler is unable to keep track of so many "levels". Removing just one of the asterisks will cure the error; all that is being declared in any case is a single two-byte pointer. However it is to be hoped that such a construct will never be needed.

12: illegal pointer reference

The type of a pointer must be either *int* or *unsigned*. This is why you might get away with not declaring pointer arguments in functions like *fopen* which return a pointer; they default to *int*. When this error is generated, an expression used as a pointer is of an invalid type:

```
char c;
int var;                   /* any variable */
int varaddress;
varaddress = &var;         /* valid since addresses */
*(varaddress) = 'c';       /* can fit in an int */
*(expression) = 10;        /* in general, expression
                              must be an int or unsigned */
*c = 'c';                  /* error 12 */
```

13: internal [see error 4]

14: internal [see error 4]

15: storage class conflict

Only automatic variables and function parameters can be specified as *register*.

This error can be caused by declaring a *static register* variable. While structure members cannot be given a storage class at all, function

arguments can be specified only as *register.*

A *register int i* declaration is not allowed outside a function--it will generate error 89 (see below).

16: data type conflict

The basic data types are not numerous, and there are not many ways to use them in declarations. The possibilities are listed below.

This error code indicates that two incompatible data types were used in conjunction with one another. For example, while it is valid to say *long int i,* and *unsigned int j,* it is meaningless to use *double int k* or *float char c.* In this respect, the compiler checks to make sure that *int, char, float* and *double* are used correctly.

data type	*interpretation*	*size(bytes)*
char	character	1
int	integer	2
unsigned/unsigned int	unsigned integer	2
short	integer	2
long/long integer	long integer	4
float	floating point number	4
long float/double	double precision float	8

17: Unsupported data type

This message occurs only when data types are used which are supported by the extended C language, such as the *enum* data type.

18: data type conflict

This message indicates an error in the use of the *long* or *unsigned* data type. *long* can be applied as a qualifier to *int* and *float. unsigned* can be used with *char, int* and *long.*

```
long i;                    /* a long int */
long float d;              /* a double */
unsigned u;                /* an unsigned int */
unsigned char c;
unsigned long l;
unsigned float f;          /* error 18 */
```

19: obsolete

Error codes interpreted as obsolete do not occur in the current version of the compiler. Some simply no longer apply due to the increased adaptability of the compiler. Other error codes have been translated into full messages sent directly to the screen. If you are using an older version of the product and have need of these codes, please contact Manx for information.

20: structure redeclaration

The compiler is able to tell you if a *struct*ure has already been defined. This message informs you that you have tried to redefine a *struct*ure.

21: missing }

The compiler expects to find a comma after each member in the list of fields for a *struct*ure initialization. After the last field, it expects a right (close) brace.

For example, the following program fragment will generate error 21, since the initialization of the structure named 'harry' doesn't have a closing brace:

```
struct sam {
    int bone;
    char license[10];
} harry = {
    1,
    "23-4-1984";
```

22: syntax error in structure declaration

The compiler was unable to find the left (open) brace which follows the tag in a *struct*ure declaration. In the example for error 21, "sam" is the structure tag. A left brace must follow the keyword *struct* if no structure tag is specified.

23: incorrect type for library function (Apprentice C only)

For Apprentice C, this error means that your program has either explicitly or implicitly incorrectly declared the type of a function that's in the run-time system. For example, you will get this error if you call the run-time system function *sqrt* without declaring that it returns a *double*.

23: obsolete (Other Aztec C Compilers)

For Compilers other than Apprentice C, this error should not occur.

24: need right parenthesis or comma

The right parenthesis is missing from a function call. Every function call must have an argument list enclosed by parentheses even if the list is empty. A right parenthesis is required to terminate the argument list.

In the following example, the parentheses indicate that *getchar* is a function rather than a variable.

```
getchar();
```

This is the equivalent of

```
CALL getchar
```

which might be found in a more explicit programming language. In general, a function is recognized as a name followed by a left parenthesis.

With the exception of reserved words, any name can be made a function by the addition of parentheses. However, if a previously defined variable is used as a function name, a compilation error will result.

Moreover, a comma must separate each argument in the list. For example, error 24 will also result from this statement:

```
funccall(arg1, arg2 arg3);
```

25: structure member name expected here

The symbol name following the dot operator or the arrow must be valid. A valid name is a string of alphanumerics and underscores. It must begin with an alphabetic (a letter of the alphabet or an underscore). In the last line of the following example, "(salary)" is not valid because '(' is not an alphanumeric.

```
empptr = &anderson;
empptr->salary = 12000;          /* these three lines */
(*empptr).salary = 12000;        /* are */
anderson.salary = 12000;         /* equivalent */
empptr = &anderson.;             /* error 25 */
empptr-> = 12000;                /* error 25 */
anderson.(salary) = 12000;       /* error 25 */
```

26: must be structure/union member

The defined structure or union has no member with the name specified. If the -S option was specified, no previously defined structure or union has such a member either.

Structure members cannot be created at will during a program. Like other variables, they must be fully defined in the appropriate declaration list. Unions provide for variably typed fields, but the full range of desired types must be anticipated in the union declaration.

27: illegal type cast

It is not possible to cast an expression to a function, a structure, or an array. This message may also appear if a syntax error occurs in the expression to be cast.

```
structure sam { ... } thom;
thom = (struct sam)(expression);          /* error 27 */
```

28: incompatible structures

C permits the assignment of one structure to another. The compiler will ensure that the two structures are identical. Both structures must have the same structure tag. For example:

```
struct sam harry;
struct sam thom;
    ...
harry = thom;
```

29: illegal use of structure

Not all operators can accept a structure as an operand. Also, structures cannot be passed as arguments. However, it is possible to take the address of a structure using the ampersand (&), to assign structures, and to reference a member of a structure using the dot operator.

30: missing : in ? conditional expression

The standard syntax for this operator is:

```
expression ? statement1 : statement2
```

It is not desirable to use ?: for extremely complicated expressions; its purpose lies in brevity and clarity.

31: call of non-function

The following represents a function call:

```
symbol(arg1, arg2, ..., argn);
```

where "symbol" is not a reserved word and the expression stands in the body of a function. Error 31, in reference to the expression above, indicates that "symbol" has been previously declared as something other than a function.

A missing operator may also cause this error:

```
a(b + c);                    /* error 31 */
a * (b + c);                 /* intended */
```

The missing '*' makes the compiler view "a()" as a function call.

32: illegal pointer calculation

Pointers may be involved in three calculations. An integral value can be added to or subtracted from a pointer. Pointers to objects of the same type can be subtracted from one another and compared to one another. (For a formal definition, see Kernighan and Ritchie pp. 188-189.) Since the comparison and subtraction of two pointers is dependent upon pointer size, both operands must be the same size.

33: illegal type

The unary minus (-) and bit complement (~) operators cannot be applied to structures, pointers, arrays and functions. There is no reasonable interpretation for the following:

```
int function();
char array[12];
struct sam { ... } harry;
a = -array;              /* ? */
b = -harry;
c = ~function & WRONG;
```

34: undefined symbol

The compiler will recognize only reserved words and names which have been previously defined. This error is often the result of a typographical error or due to an omitted declaration.

35: typedef not allowed here

Symbols which have been defined as types are not allowed within expressions. The exception to this rule is the use of *sizeof(expression)* and the cast operator. Compare the accompanying examples:

```
struct sam {
    int i;
} harry;
typedef double bigfloat;
typedef struct sam foo;

j = 4 * bigfloat f;      /* error 35 */
k = &foo;                /* error 35 */
x = sizeof(bigfloat);
y = sizeof(foo);         /* good */
```

The compiler will detect two errors in this code. In the first assignment, a typecast was probably intended; compare error 8. The second assignment makes reference to the address of a structure type. However, the structure type is just a template for instances of the structure (such as "harry"). It is no more meaningful to take the address of a structure type than any other data type, as in *&int.*

36: no more expression space

This message indicates that the expression table is not large enough for the compiler to process the source code. It is necessary to recompile the file using the -E option to increase the number of available entries in the expression table. See the description of the compiler in the manual.

37: invalid expression

This error occurs in the evaluation of an expression containing a unary operator. The operand either is not given or is itself an invalid expression.

Unary operators take just one operand; they work on just one variable or expression. If the operand is not simply missing, as in the example below, it fails to evaluate to anything its operator can accept. The unary operators are logical not (!), bit complement (~), increment (++), decrement (--), unary minus (-), typecast, pointer-to (*), address-of (&), and sizeof.

```
if (!) ;
```

38: no auto. aggregate initialization

It is not permitted to initialize automatic arrays and structures. Static and external aggregates may be initialized, but by default their members are set to zero.

```
char array[5] = { 'a', 'b', 'c', 'd' };
function()
{
    static struct sam {
        int bone;
        char license[10];
    } harry = {
        1,
        "123-4-1984"
    };
    char autoarray[2] = { 'f', 'g' };    /* no good */
    extern char array[];
}
```

There are three variables in the above example, only two of which are correctly initialized. The variable "array" may be initialized because it is external. Its first four members will be given the characters as shown. The fifth member will be set to zero.

The structure "harry" is static and may be initialized. Notice that "license" cannot be initialized without first giving a value to "bone". There are no provisions in C for setting a value in the middle of an aggregate.

The variable "autoarray" is an automatic array. That is, it is local to a function and it is not declared to be static. Automatic variables reappear automatically every time a function is called, and they are guaranteed to contain garbage. Automatic aggregates cannot be initialized.

39: obsolete [see error 19]

40: internal [see error 4]

41: initializer not a constant

In certain initializations, the expression to the right of the equals sign (=) must be a constant. Indeed, only automatic and register variables may be initialized to an expression. Such initializations are meant as a convenient shorthand to eliminate assignment statements. The initialization of statics and globals actually occurs at link-time, and not at run-time.

```
{
    int i = 3;
    static int j = (2 + i);        /* illegal */
}
```

42: too many initializers

There were more values found in an initialization than array or structure members exist to hold them. Either too many values were specified or there should have been more members declared in the aggregate definition.

In the initialization of a complex data structure, it is possible to enclose the initializer in a single set of braces and simply list the members, separated by commas. If more than one set of braces is used, as in the case of a structure within a structure, the initializer must be entirely braced.

```
struct {
    struct {
        char array[];
    } substruct;
} superstruct =
```

version 1:

```
{
    "abcdefghij"
};
```

version 2:

```
{
    {
        { 'a','b','c',...,'i','j'}
    }
};
```

In version 1, the initializers are copied byte-for-byte onto the structure, *superstruct.*

Another likely source of this error is in the initialization of arrays with strings, as in:

```
char array[10] = "abcdefghij";
```

This will generate error 42 because the string constant on the right is null-terminated. The null terminator (' ' or 0x00) brings the size of the initializer to 11 bytes, which overflows the ten-byte array.

43: undefined structure initialization

An attempt has been made to assign values to a structure which has not yet been defined.

```
struct sam {...};
struct dog sam = { 1, 2, 3};  /* error 43 */
```

44: obsolete [see error 19]

45: bad declaration syntax

This error code is an all purpose means for catching errors in declaration statements. It indicates that the compiler is unable to interpret a word in an external declaration list.

46: missing closing brace

All the braces did not pair up at the end of compilation. If all the preceding code is correct, this message indicates that the final closing brace to a function is missing. However, it can also result from a brace missing from an inner block.

Keep in mind that the compiler accepts or rejects code on the basis of syntax, so that an error is detected only when the rules of grammar are violated. This can be misleading. For example, the program below will generate error 46 at the end even though the human error probably occurred in the *while* loop several lines earlier.

As the code appears here, every statement after the left brace in line 6 belongs to the body of the *while* loop. The compilation error vanishes when a right brace is appended to the end of the program, but the results during run time will be indecipherable because the brace should be placed at the end of the loop.

It is usually best to match braces visually before running the compiler. A C-oriented text editor makes this task easier.

```
main()
{
    int i, j;
    char array[80];

    gets(array);
    i = 0;
    while (array[i])  {
        putchar(array[i]);
        i++;
    for ( i=0; array[i];i++) {
        for (j=i + 1; array[j]; j++)  {
            printf("elements %d and %d are ", i, j);
            if (array[i] == array[j])
                printf("the same\n");
            else
                printf("different\n");
        }
        putchar('\n');
    }
}
```

47: open failure on include file

When a file is #*included*, the compiler will look for it in a default area (see the manual description of the compiler). This message will be generated if the file could not be opened. An open failure usually occurs when the included file does not exist where the compiler is searching for it. Note that a drive specification is allowed in an include statement, but this diminishes flexibility somewhat.

48: illegal symbol name

This message is produced by the preprocessor, which is that part of the compiler which handles lines which begin with a pound sign (#). The source for the error is on such a line. A legal name is a string whose first character is an alphabetic (a letter of the alphabet or an underscore). The succeeding characters may be any combination of alphanumerics (alphabetics and numerals). The following symbols will produce this error code:

```
2nd_time,
dont_do_this!
```

49: multiply defined symbol

This message warns that a symbol has already been declared and that it is illegal to redeclare it. The following is a representative example:

```
int i, j, k, i;          /* illegal */
```

50: missing bracket

This error code is used to indicate the need for a parenthesis, bracket or brace in a variety of circumstances.

51: lvalue required

Only *lvalues* are are allowed to stand on the left-hand side of an assignment. For example:

```
int num;
num = 7;
```

They are distinguished from *rvalues*, which can never stand on the left of an assignment, by the fact that they refer to a unique location in memory where a value can be stored. An *lvalue* may be thought of as a bucket into which an *rvalue* can be dropped. Just as the contents of one bucket can be passed to another, so can an lvalue *y* be assigned to another lvalue, *x*:

```
#define NUMBER  512
x = y;
1024 = z;              /* wrong; l/rvalues are reversed */
NUMBER = x;            /* wrong; NUMBER is still an rvalue */
```

Some operators which require *lvalues* as operands are increment (++), decrement (--), and address-of (&). It is not possible to take the address of a register variable as was attempted in the following example:

```
register int i, j;
i = 3;
j = &i;
```

52: obsolete [see error 19]

53: multiply defined label

On occasions when the goto statement is used, it is important that the specified label be unique. There is no criterion by which the computer can choose between identical labels. If you have trouble finding the duplicate label, use your text editor to search for all occurrences of the string.

54: too many labels

The compiler maintains an internal table of labels which will support up to several dozen labels. Although this table is fixed in size, it should satisfy the requirements of any reasonable C program. C was structured to discourage extravagance in the use of goto's. Strictly speaking, goto statements are not required by any procedure in C; they are primarily recommended as a quick and simple means of exiting from a nested structure.

This error indicates that you should significantly reduce the number of goto's in your program.

55: missing quote

The compiler found a mismatched double quote (") in a *#define* preprocessor command. Unlike brackets, quotes are not paired innermost to outermost, but sequentially. So the first quote is associated with the second, the third with the fourth, and so on. Single quotes (') and double quotes (") are entirely different characters and should not be confused. The latter are used to delimit string constants. A double quote can be included in a string by use of a backslash, as in this example:

```
"this is a string"
"this is a string with an embedded quote: \". "
```

56: missing apostrophe

The compiler found a mismatched single quote or apostrophe (') in a *#define* preprocessor command. Single quotes are paired sequentially (see error 55). Although quotes can not be nested, a quote can be represented in a character constant with a backslash:

```
char c = '\'';          /* c is initialized to
                           single quote  */
```

57: line too long

Lines are restricted in length by the size of the buffer used to hold them. This restriction varies from system to system. However, logical lines can be infinitely long by continuing a line with a backslash-newline sequence. These characters will be ignored.

58: illegal # encountered

The pound sign (#) begins each command for the preprocessor: *#include, #define, #if, #ifdef, #ifndef, #else, #endif, #asm, #endasm, #line and #undef.* These symbols are strictly defined. The pound sign (#) must be in column one and lower case letters are required.

59: macro too long

Macros can be defined with a preprocessor command of the following form:

```
#define [identifier] [substitution text]
```

The compiler then proceeds to replace all instances of "identifier" with the substitution text that was specified by the *#define.*

This error code refers to the substitution text of a macro. Whereas ideally a macro definition may be extended for an arbitrary number of lines by ending each line with a backslash (), for practical purposes the size of a macro has been limited to 255 characters.

60: obsolete [see error 19]

61: reference of member of undefined structure

Occurs only under compilation without the -S option. Consider the following example:

```
int bone;
struct cat {
   int toy;
} manx;
struct dog *samptr;
manx.toy = 1;
bone = samptr->toy;      /* error 61 */
```

This error code appears most often in conjunction with this kind of mistake. It is possible to define a pointer to a structure without having already defined the structure itself. In the example, *samptr* is a structure pointer, but what form that structure ("dog") may take is still unknown. So when reference is made to a member of the structure to which *samptr* points, the compiler replies that it does not even known what the structure looks like.

The *-S* compiler option is provided to duplicate the manner in which earlier versions of UNIX treated structures. Given the example above, it would make the compiler search all previously defined structures for the member in question. In particular, the value of the member "toy" found in the structure "manx" would be assigned to the variable "bone". The -S option is not recommended as a short cut for defining structures.

62: function body must be compound statement

The body of a function must be enclosed by braces, even though it may consist of only one statement:

```
function()
{
   return 1;
}
```

This error can also be caused by an error inside a function declaration list, as in:

```
func(a, b)
int a; chr b;
{
   ...
```

63: undefined label

A *goto* statement is meaningless if the corresponding label does not appear somewhere in the code. The compiler disallows this since it must be able to specify a destination to the computer.

It is not possible to goto a label outside the present function (labels are local to the function in which they appear). Thus, if a label does not exist in the same procedure as its corresponding goto, this message will be generated.

64: inappropriate arguments

When a function is declared (as opposed to defined), it is poor syntax to specify an argument list:

```
function(string)
char *string;
{
   char *func1();              /* correct */
   double func2(x,y);          /* wrong */
   ...
}
```

In this example, function() is being defined, but func1() and func2() are being declared.

65: illegal or missing argument name

The compiler has found an illegal name in a function argument list. An argument name must conform to the same rules as variable names, beginning with an alphabetic (letter or underscore) and continuing with any sequence of alphanumerics and underscores. Names must not coincide with reserved words.

66: expected comma

In an argument list, arguments must be separated by commas.

67: invalid else

An *else* was found which is not associated with an *if* statement. *else* is bound to the nearest *if* at its own level of nesting. So if-else pairings are determined by their relative placement in the code and their grouping by braces.

```
if(...) {
   ...
   if (...) {
      ...
   } else if (...)
      ...
   } else {
      ...
   }
```

The indentation of the source text should indicate the intended structure of the code. Note that the indentation of the if and else-if means only that the programmer wanted both conditionals to be nested at the same level, in particular one step down from the presiding if

statement. But it is the placement of braces that determines this for the compiler. The example above is correct, but probably does not conform to the expectations revealed by the indentation of the else statement. As shown here, the else is paired with the first if, not the second.

68: syntax error

The keywords used in declaring a variable, which specify storage class and data type, must not appear in an executable statement. In particular, all local declarations must appear at the beginning of a block, that is, directly following the left brace which delimits the body of a loop, conditional or function. Once the compiler has reached a non-declaration, a keyword such as *char* or *int* must not lead a statement; compare the use of the casting operator:

```
func()
{
    int i;
    char array[12];
    float k = 2.03;

    i = 0;
    int m;                          /* error 68 */
    j = i + 5;
    i = (int) k;                    /* correct */
    if (i) {
        int i = 3;
        j = i;
        printf("%d",i);
    }
    printf("%d%d\n",i,j);
}
```

This trivial function prints the values 3, 2 and 3. The variable i which is declared in the body of the conditional (if) lives only until the next right brace; then it dies, and the original i regains its identity.

69: missing semicolon

A semicolon is missing from the end of an executable statement. This error code is subject to the same vagaries as its cousin, error 7. It will remain undetected until the following line and is often spuriously caused by a previous error.

70: bad goto syntax

Compare your use of goto with an example. This message says that you did not specify where you wanted to goto with a label:

```
        goto label;
        ...
    label:
        ...
```

It is not possible to goto just any identifier in the source code; labels are special because they are followed by a colon.

71: statement syntax error in do-while

The body of a *do-while* may consist of one statement or several statements enclosed in braces. A *while* conditional is required after the body of the loop. This is true even if the loop is infinite, as it is required by the rules of syntax. After typing in a long body, don't forget the *while* conditional.

72: 'for' syntax error: missing first semicolon

This error focuses on another control flow statement, the *for*. The keyword, *for*, must be followed by parentheses. In the parentheses belong three expressions, any or all of which may be null. For the sake of clarity, C requires that the two semicolons which separate the expressions be retained, even if all three expressions are empty.

```
for (;                  /* an infinite loop which does */
    ;                   /* absolutely nothing */
```

Error 72 signifies that the compiler didn't find the first semicolon within the parentheses.

73: 'for' syntax error: missing second semicolon

This error is similar to error 72; it means that the compiler didn't find the second semicolon within the parenthesized expression following the 'for'.

74: case value must be integer constant

Strictly speaking, each value in a *case* statement must be a constant of one of three types: *char*, *int* or *unsigned*. This is similar to the rule for a *switched* variable. In the following example, a float must be cast to an int in order to be switched; however, notice that the programmer did not check his case statements. The second case value is invalid, and the code will not compile.

```
float k = 5.0;
switch((int)k) {
case 4:
    printf("good case value\n");
    break;
case 5.0:
    printf("bad case value\n");
    break;
}
```

The programmer must replace "case 5.0:" with "case 5".

75: missing colon on case

This should be straightforward. If the compiler accepts a case value, a colon should follow it. A semi-colon must not be accidently entered in its place.

76: too many cases in switch

The compiler reserves a limited number of spaces in an internal table for *case* statements. If a program requires more cases than the table initially allows, it becomes necessary to tell the compiler what the table value should be changed to. It is not necessary to know exactly how many are needed; an approximation is sufficient, depending on the requirements of the situation.

77: case outside of switch

The keyword, *case*, belongs to just one syntactic structure, the *switch*. If "case" appears outside the braces which contain a switch statement, this error is generated. Remember that all keywords are reserved, so that they cannot be used as variable names.

78: missing colon

This message indicates that a colon is missing after the keyword, *default*. Compare error 75.

79: duplicate default

The compiler has found more than one *default* in a *switch*. Switch will compare a variable to a given list of values. But it is not always possible to anticipate the full range of values which the variable may take. Nor is it feasible to specify a large number of cases in which the program is not particularly interested.

So C provides for a default case. The default will handle all those values not specified by a case statement. It is analogous to the else companion to the conditional, if. Just as there is one else for every if, only one default case is allowed in a switch statement. However, unlike the else statement, the position of a default is not crucial; a default can appear anywhere in a list of cases.

80: default outside of switch

The keyword, *default*, is used just like *case*. It must appear within the brackets which delimit the switch statement.

81: break/continue error

Break and continue are used to skip the remainder of a loop in order to exit or repeat the loop. Break will also end a switch statement. But when the keywords, *break* or *continue*, are used outside of these contexts, this message results.

82: illegal character

Some characters simply do not make sense in a C program, such as '$' and '@'. Others, for instance the pound sign (#), may be valid only in particular contexts.

83: too many nested includes

#includes can be nested, but this capacity is limited. The compiler will balk if required to descend more than three levels into a nest. In the example given, file D is not allowed to have a #include in the compilation of file A.

```
file A          file B          file C          file D
#include "B"   #include "C"   #include "D"
```

84: too many array dimensions

An array is declared with too many dimensions. This error should appear in conjunction with error 11.

85: not an argument

The compiler has found a name in the declaration list that was not in the argument list. Only the converse case is valid, i.e., an argument can be passed and not subsequently declared.

86: null dimension in array

In certain cases, the compiler knows how to treat multidimensional arrays whose left-most dimensions are not given in its declaration. Specifically, this is true for an extern declaration and an array initialization. The value of any dimension which is not the left-most must be given.

```
extern char array[][12];        /* correct */
extern char badarray[5][];      /* wrong */
```

87: invalid character constant

Character constants may consist of one or two characters enclosed in single quotes, as 'a' or 'ab'. There is no analog to a null string, so '' (two single quotes with no intervening white space) is not allowed. Recall that the special backslash characters (\b, \n, \t etc.) are singular,

so that the following are valid: '\n', '\na', 'a\n'; 'aaa' is invalid.

88: not a structure

Occurs only under compilation without the -S option. A name used as a structure does not refer to a structure, but to some other data type.

```
int i;
i.member = 3;          /* error 88 */
```

89: invalid storage class

A globally defined variable cannot be specified as register. Register variables are required to be local.

90: symbol redeclared

A function argument has been declared more than once.

91: illegal use of floating point type

Floating point numbers can be negated (unary minus), added, subtracted, multiplied, divided and compared; any other operator will produce this error message.

92: illegal type conversion

This error code indicates that a data type conversion, implicit in the code, is not allowed, as in the following piece of code:

```
int i;
float j;
char *ptr;
...
i = j + ptr;
```

The diagram shows how variables are converted to different types in the evaluation of expressions. Initially, variables of type *char* and *short* become *int*, and *float* becomes *double*. Then all variables are promoted to the highest type present in the expression. The result of the expression will have this type also. Thus, an expression containing a *float* will evaluate to a *double*.

hierarchy of types:

```
double <-- float
long
unsigned
int <-- short, char
```

This error can also be caused by an attempt to return a structure, since the structure is being cast to the type of the function, as in:

```
int func()
{
   struct tag sam;
   return sam;
}
```

93: illegal expression type for switch

Only a *char*, *int* or *unsigned* variable can be switched. See the example for error 74.

94: bad argument to define

An illegal name was used for an argument in the definition of a macro. For a description of legal names, see error 65.

95: no argument list

When a macro is defined with arguments, any invocation of that macro is expected to have arguments of corresponding form. This error code is generated when no parenthesized argument list was found in a macro reference.

```
#define  getchar()  getc(stdin)
...
c = getchar;                    /* error 95 */
```

96: missing argument to macro

Not enough arguments were found in an invocation of a macro. Specifically, a "double comma" will produce this error:

```
#define reverse(x,y,z)  (z,y,x)
func(reverse(i,,k));
```

97: obsolete [see error 19]

98: not enough args in macro reference

The incorrect number of arguments was found in an invocation of a previously defined macro. As the examples show, this error is not identical to error 96.

```
#define  exchange(x,y)  (y,x)
func(exchange(i));              /* error 98 */
```

99: internal [see error 4]

100: internal [see error 4]

101: missing close parenthesis on macro reference

A right (closing) parenthesis is expected in a macro reference with arguments. In a sense, this is the complement of error 95; a macro argument list is checked for both a beginning and an ending.

102: macro arguments too long

The combined length of a macro's arguments is limited. This error can be resolved by simply shortening the arguments with which the macro is invoked.

103: #else with no #if

Correspondence between #if and #else is analogous to that which exists between the control flow statements, if and else. Obviously, much depends upon the relative placement of the statements in the code. However, #if blocks must always be terminated by #endif, and the #else statement must be included in the block of the #if with which it is associated. For example:

```
#if ERROR > 0
   printf("there was an error\n");
#else
   printf("no error this time\n");
#endif
```

#if statements can be nested, as below. The range of each #if is determined by a #endif. This also excludes #else from #if blocks to which it does not belong:

```
#ifdef   JAN1
   printf("happy new year!\n");
#if      sick
   printf("i think i'll go home now\n");
#else
   printf("i think i'll have another\n");
#endif
#else
   printf("i wonder what day it is\n");
#endif
```

If the first #endif was missing, error 103 would result. And without the second #endif, the compiler would generate error 107.

104: #endif with no #if

#endif is paired with the nearest #if, #ifdef or #ifndef which precedes it. (See error 103.)

105: #endasm with no #asm

#endasm must appear after an associated #asm. These compiler-control lines are used to begin and end embedded assembly code. This error code indicates that the compiler has reached a #endasm without having found a previous #asm. If the #asm was simply missing, the error list should begin with the assembly code (which are undefined symbols to the compiler).

106: #asm within #asm block

There is no meaningful sense in which in-line assembly code can be nested, so the #asm keyword must not appear between a paired #asm/#endasm. When a piece of in-line assembly is augmented for temporary purposes, the old #asm and #endasm can be enclosed in comments as place-holders.

```
#asm
   /* temporary asm code */
/*  #asm          old beginning  */
   /* more asm code */
#endasm
```

107: missing #endif

A #endif is required for every #if, #ifdef and #ifndef, even if the entire source file is subject to a single conditional compilation. Try to assign pairs beginning with the first #endif. Backtrack to the previous #if and form the pair. Assign the next #endif with the nearest unpaired #if. When this process becomes greatly complicated, you might consider rethinking the logic of your program.

108: missing #endasm

In-line assembly code must be terminated by a #endasm in all cases. #asm must always be paired with a #endasm.

109: #if value must be integer constant

#if requires an integral constant expression. This allows both integer and character constants, the arithmetic operators, bitwise operators, the unary minus (-) and bit complement, and comparison tests.

Assuming all the macro constants (in capitals) are integers,

```
#if DIFF >= 'A'-'a'
#if (WORD &= ~MASK) >> 8
#if MAR | APR | MAY
```

are all legal expressions for use with #if.

110: invalid use of colon operator

The colon operator occurs in two places: 1. following a question mark as part of a conditional, as in (flag ? 1 : 0); 2. following a label inserted by the programmer or following one of the reserved labels, *case* and *default.*

111: illegal use of a void expression

This error can be caused by assigning a *void* expression to a variable, as in this example:

```
void func();
int h;

h = func(arg);
```

112: illegal use of function pointer

For example,

```
int (*funcptr) ();
   ...
funcptr++;
```

funcptr is a pointer to a function which returns an integer. Although it is like other pointers in that it contains the address of its object, it is not suject to the rules of pointer arithmetic. Otherwise, the offending statement in the example would be interpreted as adding to the pointer the size of the function, which is not a defined value.

113: duplicate case in switch

This simply means that, in a *switch* statement, there are two *case* values which are the same. Either the two *case*s must be combined into one, or one of them must be discarded. For instance:

```
switch (c) {
case NOOP:
   return (0);
case MULT:
   return (x * y);
case DIV:
   return (x / y);
case ADD:
   return (x + y);
case NOOP:
default:
   return;
}
```

The case of NOOP is duplicated, and will generate an error.

114: macro redefined

For example,

```
#define   islow(n)  (n>=0&&n<5)
   ...
#define   islow(n)  (n>=0&&n<=5)
```

The macro, *islow*, is being used to classify a numerical value. When a second definition of it is found, the compiler will compare the new substitution string with the previous one. If they are found to be different, the second definition will become current, and this error code will be produced.

In the example, the second definition differs from the first in a single character, '='. The second definition is also different from this one:

```
#define  islow(n)  n>=0&&n<=5
```

since the parentheses are missing.

The following lines will not generate this error:

```
#define  NULL 0
  ...
#define  NULL 0
```

But these are different from:

```
#define  NULL ' '
```

In practice, this error message does not affect the compilation of the source code. The most recent "revision" of the substitution string is used for the macro. But relying upon this fact may not be a wise habit.

115: keyword redefined

Keywords cannot be defined as macros, as in:

```
#define  int  foo
```

If you have a variable which may be either, for instance, a short or a long integer, there are alternative methods for switching between the two. If you want to compile the variable as either type of integer, consider the following:

```
#ifdef   LONGINT
   long i;
#else
   short i;
#endif
```

Another possibility is through a *typedef*:

```
#ifdef   LONGINT
   typedef  long    VARTYPE;
#else
   typedef  short   VARTYPE;
#endif

VARTYPE i;
```

116: field width must be > 0

A field in a bit field structure can't have a negative number of bits.

117: invalid 0 length field

A field in a bit field structure can't have zero bits.

118: field is too wide

A field in a bit field structure can't have more than 16 bits.

119: field not allowed here

A bit field definition can only be contained in a structure.

120: invalid type for field

The type of a bit field can only be of type *int* of *unsigned int.*

121: ptr/int conversion

The compiler issues this warning message if it must implicitly convert the type of an expression from pointer to *int* or *long*, or vice versa.

If the program explicitly casts a pointer to an *int* this message won't be issued. However, in this case, error 122 may occur.

For example, the following will generate warning 121:

```
char *cp;
int i;
...
i = cp; /* implicit conversion of char * to int */
```

When the compiler issues warning 121, it will generate correct code if the sizes of the two items are the same.

122: ptr & int not same size

If a program explicitly casts a pointer to an *int*, and the sizes of the two items differ, the compiler will issue this warning message. The code that's generated when the converted pointer is used in an expression will use only as much of the least significant part of the pointer as will fit in an *int.*

123: function ptr & ptr not same size

If a program explicitly casts a pointer to a data item to be a pointer to a function, or vice versa, and the sizes of the two pointers differ, the compiler issues this warning message.

If the program doesn't explicitly request the conversion, warning 124 will be issued instead of warning 123.

124: invalid ptr/ptr assignment

If a program attempts to assign one pointer to another without explicitly casting the two pointers to be of the same type, and the types of the two pointers are in fact different, the compiler will issue this warning message.

The compiler will generate code for the assignment, and if the sizes of the two pointers are the same, the code will be correct. But if the

sizes differ, the code may not be correct.

125: too many subscripts or indirection on integer

This warning message is issued if a program attempts to use an integer as a pointer; that is, as the operand of a star operator.

If the sizes of a pointer and an *int* are the same, the generated code will access the correct memory location, but if they don't, it won't.

For example,

```
char c;
long g;
*0x5c=0; /* warning 125, because 0x5c is an int */
c[i]=0;  /* warning 125, because c+i is an int */
g[i]=0;  /* error 12, because g+i is a long */
```

3. Fatal Compiler Error Messages

If the compiler encouters a "fatal" error, one which makes further operation impossible, it will send a message to the screen and end the compilation immediately.

Out of disk space!

There is no room on the disk for the output file of the compiler. Previous disk files will not be overwritten by the compiler's assembly language output. To make room on the disk, it is usually sufficient to remove unneeded files from the disk.

unknown option:

The compiler has been invoked with an option letter which it does not recognize. The manual explicitly states which options the compiler will accept. The compiler will specify the invalid option letter.

duplicate output file

If an output file name has been specified with the -o option and that file already exists on the disk, the compiler will not overwrite it. -O must specify a new file.

too few arguments for -o option

The compiler expected to find the output filename following the "-o", but didn't find it. The output file name must follow the option letter and the name of the file to be compiled must occur last in the command line.

Open failure on input

The input file specified in the command line does not exist on the disk or cannot be opened. A path or drive specification can be included with a filename according to the operating system in use.

No input!

While the compiler was able to open the input file given in the command line, that file was found to be empty.

Open failure on output

The compiler was unable to create an output file. On some systems, this error could occur if a disk's directory is full.

Local table full! (use -L)

The compiler maintains an internal table of the local variables in the source code. If the number of local symbols in use exceeds the available entries in the table at any time during compilation, the compiler will print this message and quit. The default size of the local symbol table (40 entries) can be changed with the -L option for the

compiler. Local variables are those defined within braces, i.e., in a function body or in a compound statement. The scope of a local variable is the body in which it is defined, that is, it is defined until the next right brace at its own nesting level.

Out of memory!

Since the compiler must maintain various tables in memory as well as manipulate source code, it may run out of memory during operation. The more immediate solution is to vary the sizes of the internal tables using the appropriate compiler options. Often, a compilation will require fewer than the default number of entries in a particular table. By reducing the size of that table, memory space is freed up during compile time. The amount of memory used while compiling does not affect the size or content of the assembly or object file output. If this stategy fails to squeeze the compilation into the available memory, the only solution is to divide the source file into modules which can be compiled separately. These modules can then be linked together to produce a single executable file.

INDEX

Order of chapters in manual

System Dependent Chapters

title	*code*
Overview	ov
Tutorial Introduction	tut
The Compiler	cc
The Assembler	as
The Linker	ln
Utility Programs	util
Library Functions Overview: 8086 Information	libov86
8086 Functions	lib86
Technical Information	tech
Unitools	unitools
Source Level Debugger	sdb
Assembly Language Debugger	db

System Independent Chapters

Overview of Library Functions	libov
System-Independent Functions	lib
Style	style
Compiler Error Messages	err

Index

Index	index

__ptradd cc.48
__ptrdiff cc.48

A
absolute value lib.16
abstoptr cc.48; lib86.33
access lib86.6-7
accessing data in memory
asm.20
accessing devices libov.8
accessing files
unitools.73-78
acos lib.59-60
adding modules after existing
modules in a library util.16
adding modules at beginning
or end of a library util.16-17
adding modules to a library
util.15
addr db.13-15; sdb.12
addresses in ex commands
unitools.67-68
adjusting the screen
unitools.55
agetc lib.25-26
aputc lib.41-42
arcv util.4
arithmetic operators asm.23
array subscripting style.18
asctime lib86.55-56
asin lib.59-60
assembler operating
instructions asm.5
assembler options asm.5,9
-186 asm.9
-c asm.9
-ca asm.9
-cs asm.9
-dsym[=const] asm.9
-i asm.6-9
-l asm.6,9
-la asm.9
-ls asm.9
-o asm.6,9
-sn asm.9
-x asm.9
-zap asm.2,5,6,9
assembly-language functions
tech.30-40
assembly-language macros
tech.33-38
entrdef tech.36
finish tech.38
internal tech.36
intrdef tech.36
ldptr tech.36
pend tech.35
pret tech.36
procdef tech.34-36
retnull tech.38
retptrm tech.37
retptrr tech.37
assert lib86.8
assign buffer to a stream
lib.56
atan lib.59-60
atan2 lib.59-60
atof lib.8
atoi lib.8
atol lib.8
autoindent unitools.61
aztec to microsoft format
util.26

B
backtracing db.9; sdb.9
bdos lib86.9,19
bdosx lib86.10
boolean expressions
style.16-17
breakpoints
db.7-8,16-18,40
sdb.7-8,13-15,38
brk lib86.11-12
buffer size libov86.6
buffered binary input
lib.20-21
buffered output lib.20-21
buffering libov.10-11
build and unbuild real
numbers lib.22

C

c idioms style.3
c source file cc.6
c86 libraries tech.25
calloc lib.31-32
case table cc.24
cbreak libov.21
ceil lib.16
ceiling lib.16
change current position within a file lib.29-30
char cc.41
character classification funtions lib.11
character-oriented input libov.18; libov86.6-7
chdir lib86.18
chmod lib86.13
circle lib86.14
clear db.19
clearerr lib.15
clock lib86.15
close lib.9,14
close a device or a file lib.9
close a stream lib.14
closing streams libov.9
cmdlist db.16; sdb.13
cnm util.5-8
code area tech.5
codemacros asm.30,60-71
colon commands unitools.89
color lib86.16
command line arguments libov.4-6
command line arguments libov86.3
command summary unitools.85-89
comments style.17
common problems style.15-19
compatablity of Aztec products cc.42
compiler error checking cc.50
compiler operating instructions cc.5
compiler options cc.7-10,19,22-23,50
-a cc.8-9,19
-d cc.19,22
-i cc.19,22
-o cc.7,19
-s cc.19,23
-t cc.9,19
-b cc.19,50
-table manipulation options cc.19,23-25
-e cc.19,24
-l cc.19,23-24
-y cc.19,24-25
-z cc.20,25
-options for the optimizing compilers cc.20,26-29
+f cc.20,26
+c cc.20,26
+n cc.20,26
+d cc.20,26
+df cc.20,26
+0 cc.20,27
+1 cc.20,27
+2 cc.20,27
+r cc.20,27
+u cc.20,27-28
+a cc.20,28-29
+m cc.20,29
-options for the non-optimizing compilers cc.21,29-30
+f cc.21,29
+u cc.21,30
+j cc.21,30
concatenating parameters to parameters asm.48
conditional compilation statements cc.34-37
console i/o libov.17-21; libov86.6-7
convert ascii to numbers lib.8
convert floating point to ascii lib.8
cos lib.59-60
cosh lib.61
cotan lib.59-60
crc util.9
crclist util.9

creat lib.10
create a new file lib.10
creating a library util.13
creating a root and overlays tech.20
creating an assembly language file cc.8
creating an object code file cc.7
cross development tech.26
csread lib86.17
ctags utility unitools.77-78
ctime lib86.55-56

D
data formats cc.41-43
default mode libov.7,17,20
default segment attribute-overriding operators asm.25
default segment asm.15
defensive programming style.10
deleting line unitools.57
deleting modules form a library util.18
deleting text unitools.42-43,56-57
desc_codes db.33-35; sdb.27-30
determine accessiblity of a file lib86.6-7
device i/o libov.7 libov86.6
device i/o utilities lib.28
diff unitools.6-9
directives asm.15-19,31-60
assume asm.15-16,31
bss asm.31-32
codemacro asm.64
else asm.60
end asm.34-35
endif asm.60
endm asm.56
equ asm.35-36
equal sign asm.36
exitm asm.56
extrn asm.17-18,37
global asm.17,18,37
group asm.37
if asm.58
if1 asm.59
if2 asm.59
ifb asm.59
ifdef asm.59
ifdif asm.60
ife asm.59
ifidn asm.60
ifnb asm.59
ifndef asm.59
include asm.37-38
irp asm.54-55,56
irpc asm.55,56
label asm.38
largecode asm.38-39
local asm.56
macro asm.44-46,56-57
modrm asm.65
name asm.39
nosegfix asm.65
org asm.39
public asm.16-17,18-19,42
purge asm.57
record asm.42-44,67
relb asm.66
relw asm.66
rept asm.53-54,57
segment and ends asm.13-14,44
user-defined record asm.67
disabling options unitools.80
display commands db.19-23,40; sdb.16-20,38
display object file info util.5
displaying source files db.9,40; sdb.9,17,38
displaying unprintable characters unitools.46
dos lib86.19
dostime lib86.55-56
dosx lib86.10
dot operator to shift

parameters asm.67
double cc.42
dup lib86.20
duplicating blocks of text
unitools.58-59
dynamic buffer allocation
libov.11,22

E
echo mode libov.21; libov86.6
editing an existing file
unitools.40-45
editing another file
unitools.74-76
embedded assembler source
tech.39
enabling options
unitools.80
end of a file libov86.4
entrdef macro tech.36
errno lib86.44
error messages form linker
ln.17-22
error messages from ovloader
tech.23
error processing
libov.23-24
evaluation of expressions
style.16
ex-like commands
unitools.67-69
examine memory lib86.43
execl lib86.21-23
execlp lib86.21-23
executable program tech.4
executing system commands
unitools.79
execv lib86.21-23
execvp lib86.21-23
exit lib86.24
exiting z unitools.39
exp lib.12-13
exponetial functions
lib.12-13
expr db.11-12; sdb.11-12
expression evaluation
style.5
expression table cc.24
extended pattern matching
unitools.50-51,80
extracting modules from a
library util.19-20

F
fabs lib.16
far call asm.13
farcall lib86.25
fcbint lib86.26
fclose lib.14
fdopen lib.17-19
fdup lib86.20
feof lib.15
ferror lib.15
fexecl lib86.27-28
fexecv lib86.27-28
fflush lib.14
fgets lib.27
file comparison utility
unitools.6-9
file i/o libov.6,9-13,15;
libov86.4-6
file lists unitools.70,76
filenames unitools.73
fileno lib.15
find source string
db.23,40; sdb.20,38
finish macro tech.38
float cc.42
floating point exceptions
cc.42
floor lib.16
flterr cc.42
flush a stream lib.15
fopen lib.17-19
format lib.37-40
formatted input conversion
lib.49-55
formatted output conversion
functions lib.37-40
fprintf lib.37-40
fputs lib.43
fread lib.20-21
free lib.31-32,56
freopen lib.17-19

frexp lib.22
fscanf lib.49-55
fseek lib.23-24
ftell lib.23-24
ftime lib86.29-30
ftoa lib.8
function calls and returns tech.32
function key macros unitools.83-84
functions calls style.13-14
fwrite lib.20-21

G
generating romable code tech.41-46
get a string from a stream lib.27
get time lib86.15
getc lib.25-26
getchar lib.25-26
getenv lib86.31
gets lib.27
getusr lib86.59
getw lib.25-26
getwd lib86.18
global variables cc.39-41; tech.28-29,30-31
globally-accessible symbols asm.16
gmtime lib86.55-56
go db.24-25,40; sdb.23-24,40; unitools.41,83
grep unitools.10-15
grep options unitools.10
matching character strings unitools.12
matching repeating characters unitools.12
matching single characters unitools.11
pattern matching program unitools.10-15
patterns unitools.11
ground lib86.16

H
help in lb util.21
hex86 tech.41-46
-j tech.45
-z tech.45
-e tech.45
-o tech.45
-s tech.45
-p tech.46
-b tech.46
hex86 util.10
high operand asm.23
huge arrays 48-49
hyperbolic functions lib.61

I
immediate macro definition unitools.63-64
immediate operands asm.19
include environment variable cc.10
index lib.62-63
indirect macro definition unitools.64-65
inportb lib86.46
inportw lib86.46
insert commands unitools.38,43-44,61
insert mode unitools.38,61
inserting text unitools.44,61
int,short cc.41
int_sp lib86.39-41
intel hex generator util.10
internal macro tech.36
intrdef macro tech.36
ioctl lib.28; libov.19
isalnum lib.11
isalpha lib.11
isascii lib.11
isatty lib.28
iscntrl lib.11
isdigit lib.11
islower lib.11
isprint lib.11
ispunct lib.11
isspace lib.11

isupper lib.11

L
labels asm.12-13
large code cc.16
large data cc.17;
tech.6-7,8
lb util.11-21
lb arguments util.14
lb options util.11
ldexp lib.22
ldptr macro tech.36
learning c idioms style.3
libraries cc.14-15
library module names
util.13
library order util.14
library table of contents
util.13
line lib86.32
line-oriented input
libov.17-18
lines longer than screen size
unitools.46
lineto lib86.32
linker error messages
ln.17-22
linker options ln.9-10
-o <file> ln.9,11
-l <name> ln.9,11
-f <file> ln.9,12,19
-t ln.9,12
-m ln.9,12-13
-n ln.9,13
-s <size> ln.9,13
-x <size> ln.9,13-14
-v ln.9,14
-options for segment address
specification ln.9,14-16
-b <address> ln.9,14-16
-c <address> ln.9,14-16
-d <address> ln.9,14-16
-u <address> ln.9,14-16
-options for overlay usage
ln.10,16
-r ln.10,16
+c <size> ln.10,16
+d <address> ln.10,16
linking process ln.4
list directory util.22-24
list object code util.25
loading programs
db.6-7,25-27,40
sdb.6-7,23,38
local moves unitools.52-55
local symbol table cc.23
local symbols asm.46-47
localtime lib86.55-56
log lib.12-13
logarithm lib.12-13
logical operators asm.24
long cc.41-42
long pointer cc.44-48
long pointer conversion
functions lib86.33
longjmp lib.57-58
low operand asm.23
ls util.22-24
lseek lib.29-30

M
macros cc.31-37
unitools.63-66,88
make unitools.16-33
aborting unitools.26
built-in rules unitools.25
logging commands
unitools.26
macro capability
unitools.22-24
makefile unitools.17-19
standard output unitools.29
starting make unitools.28
malloc lib.31-32,56
marking unitools.54,86
memccpy lib86.34-35
memchr lib86.34-35
memcmp lib86.34-35
memory allocation lib.31-32
memory models cc.11-14
memory modification commands
db.27-28,40; sdb.24-25,38
memory operands asm.20
memory operations lib86.34-35

memory-change breakpoints
db.8; sdb.8,38
memset lib86.34-35
missing semicolon style.15
mkdir lib86.18
mktemp lib86.36-37
mode lib86.38
modes of z unitools.37-38
modf lib.22
modifiers asm.63,69-70
modify memory lib86.43
modularity style.7
monitor lib86.39-41
movblock lib86.42
moves within c programs
unitools.53
moving around on the screen
unitools.52
moving blocks unitools.57
moving modules to the
beginning or end
of a library util.18
moving modules within a
library util.17
moving text between files
unitools.60
moving within a line
unitools.52-53
movmem lib.33
ms-dos linker tech.27-29
msdos source files cc.6
multi-module programs
cc.13-14

N
named buffers unitools.59
names db.5-6; sdb.8
near call asm.12
nested segments asm.14
nesting errors style.17
nodelay libov.17
non-local gotto lib.57-58

O
obd util.25
obj tech.27-29
obj util.26
object file librarian
util.11-21
offset attribute asm.12
offset operator asm.28
open lib.34-36
open a stream lib.17-19
opening files libov86.5
opening files and devices
libov.6,9
operand expressions asm.23
operands and expressions
asm.19
operands to jump and call
asm.21
operator precedence asm.30
option codes unitools.80
ord util.27
order of evaluation
style.16
order of library modules
ln.5-6
outport lib86.46
outportb lib86.46
overlay code area tech.5
overlay data area tech.5
overlay usage options
ln.10,16
overlays tech.4-5,11-24
ovloader tech.15-16,19-23
+c and +d options tech.19

P
paging unitools.48
palette lib86.16
passing comma-containing
arguments to macros asm.50
passing data to functions
style.18
passing pointers between
functions style.18
pcz unitools.82
peekb lib86.43
peekw lib86.43
pend macro tech.35
perform bdos call with a far
pointer lib86.10

perror lib86.44
point lib86.45
pointer cc.41
pointers cc.41,44-48
pokeb lib86.43
pokew lib86.43
port lib86.46
pow lib.12-13
power lib.12-13
pre-opened devices
libov.4; libov86.3
preprocessor statements
cc.31-37
pret macro tech.36
printf lib.37-40
procdef macro tech.34-36
proclen symbol asm.68
prof util.28
profiler report program
util.28
profiling functions
lib86.39-41
program areas tech.5
program maintenance utility
unitools.16-33
program organization
tech.4-14
ptr operator asm.25-26
ptrtoabs cc.48
ptrtoabs lib86.33
push a character back into
input stream lib.65
put a character string to a
stream lib.43
putc lib.41-42
putchar lib.41-42
puterr lib.41-42
puts lib.43
putw lib.41-42

Q
qsort lib.44-45
quit db.35,41; sdb.32,39

R
ran lib.46
random i/o
libov.6,10; libov86.4
random number generator
lib.46
range db.15-16; sdb.12
range specifiers asm.63,70
raw mode libov.20-21
re-executing macros
'unitools.65
read lib.47
readable code style.5
reading files unitools.74
realloc lib.31-32
rebuilding a library
util.20
register commands
db.36; sdb.33,39
register usage tech. 27, 33
registers asm.19
relational operators asm.24
relocatable object files
ln.3
rename a disk file lib.48
repalcing library modules
util.19
repeat last substitution(&)
unitools.69
reposition a stream
lib.23-24
retnull macro tech.38
retptrm macro tech.37
rindex lib.62-63
rmdir lib86.18
romable code tech.41-46
root tech.20
rstusr lib86.59
rsvstk lib86.11-12
run-time errors style.12

S
sbrk lib86.11-12
scanf lib.49-55
screen functions
lib86.47-50
scrolling
unitools.40-41,44,48
search order of #include

files cc.10
seg operator asm.28
segment address specification options ln.9,14-16
segment override operator asm.25
segment register asm.22
segmentation asm.13
segread lib86.51
sequential i/o libov.6,10; libov86.4
set_asp lib86.14
setbuf lib.56
setjmp lib.57-58
setmem lib.33
setting options for a file unitools.71
setusr lib86.59
sgtty fields libov.19; libov86.7
shared data style.19
shift operators asm.24
shifting text unitools.60
short operator asm.27
signal lib86.52-53
silence library option util.20
sin lib.59-60
single step db.36,41; sdb.33,39
sinh lib.61
size operator asm.29
small code cc.16
small data cc.17; tech. 6-7,9-13
sort an array lib.44-45
sort object module list util.27
source dearchiver util.4
special keys unitools.47
specifiers asm.62-62,69
sprintf lib.37-40
sqrt lib.12-13
square root lib.12-13
squeeze an object library util.29
sqz util.29
sscanf lib.49-55
stack above heap tech.6,10,11
stack and heap areas tech.6
stack below heap tech.6,9,10
standard i/o libov.9-13
standard i/o functions libov.12-13
starting and stopping z unitools.37,70-72
starting db db.11
starting sdb sdb.11
startup routine termination codes tech.14
stat lib86.13
strcat lib.62-63
strcmp lib.62-63
strcpy lib.62-63
stream status inquiries lib.15
string merging cc.39
string operations lib.62-63
string searching unitools.41-42,49
string table cc.25
strlen lib.62-63
strncat lib.62-63
strncmp lib.62-63
strncpy lib.62-63
structure assign cc.37
structured programming style.7
substitute command unitools.68-69
supported language features cc.31
swapmem lib.33
symbol names asm.11
symbols asm.11-12
symbols related to program organization tech.13-14
syntax asm.10-11
sys_errlist lib86.44
sys_nerr lib86.44
sysint lib86.25
system lib86.54
system error messages lib86.44

system-dependent features
unitools.82
system-independent programs
libov.18

T
tags unitools.76-77
tan lib.59-60
tanh lib.61
term db.12-13;
util.29
terminal emulation util.29
termination codes tech.14
text editor unitools.34-89
this operator asm.27
time lib86.55-56
tmpfile lib86.57
tmpnam lib86.58
tolower lib.64
top-down programming
style.8-9
toupper lib.64
trace mode db.9,19,40
trace mode sdb.9,15,38
trigonometric functions:
lib.59-60
type operator asm.29

U
unassemble db.37,41;
sdb.34,39
unbuffered and standard i/o
calls libov.7
unbuffered i/o libov.14-16
undoing changes unitools.60
ungetc lib.65
uninitialized data area
tech.5
uninitialized variables
style.15
unlink lib.66
using the linker ln.7
utime lib86.29-30

V
verbose library option
util.20
verify program assertion
lib86.8
void data type cc.38

W
word movements unitools.53
write lib.67
writing files
unitools.73-74
writing machine independent
code cc.42

Y
yank unitools.58-59,88

Z
z unitools.34-89
accessing files
unitools.73-78
adjusting the screen
unitools.55
autoindent unitools.61
colon commands
unitools.89
command summary
unitools.85-89
ctags utility
unitools.77-78
deleting line
unitools.57
deleting text
unitools.42-43,56-57
disabling options
unitools.80
displaying unprintable
characters unitools.46
duplicating blocks of text
unitools.58-59
editing an existing file
unitools.40-45
editing another file
unitools.74-76
enabling options

unitools.80
ex-like commands
unitools.67-69
addresses in ex commands
unitools.67-68
substitute command
unitools.68-69
repeat last substitution(&) unitools.69
executing system commands
unitools.79
exiting z unitools.39
extended pattern matching
unitools.50-51,80
file lists
unitools.70,76
filenames unitools.73
function key macros
unitools.83-84
go unitools.41,83
insert commands
unitools.38,43-44,61
insert mode
unitools.38,61
inserting text
unitools.44,61
lines longer than screen size unitools.46
local moves
unitools.52-55
macros unitools.63-66,88
immediate macro definition
unitools.63-64
indirect macro definition
unitools.64-65
re-executing macros
unitools.65
marking unitools.54,86
modes of z
unitools.37-38
moves within c programs
unitools.53
moving around on the screen unitools.52
moving blocks
unitools.57
moving text between files
unitools.60
moving within a line
unitools.52-53
named buffers
unitools.59
option codes unitools.80
paging unitools.48
pcz unitools.82
reading files
unitools.74
scrolling
unitools.40-41,44,48
setting options for a file
unitools.71
shifting text
unitools.60
special keys unitools.47
starting and stopping z
unitools.37,70-72
string searching
unitools.41-42,49
system-dependent features
unitools.82
tags unitools.76-77
undoing changes
unitools.60
word movements
unitools.53
writing files
unitools.73-74
yank unitools.58-59,88
z vs vi unitools.81