AZTEC C OWNER'S MANUAL

MANX SOFTWARE SYSTEMS

C

C C

C Co

C Com

C Comp

C Compi

C Compil

C Compile

C Compiler

Aztec C II User Manual

Release 1.05

9/9/83

Distributed by:
Manx Software Systems
P. O. Box 55
Shrewsbury, N. J. 07701
201-780-4004

CONTENTS

SECTION

Installation I

Overview II

Aztec C II Compiler III

MANX AS Relocating Assembler IV

MANX LN Linker V

Library Functions VI

Microsoft Compatibility VII

Error Codes and Error Processing VIII

LIBUTIL Library Utility IX

I/O Redirection and Buffered I/O X

MANX Overlay Linker XI

Data Formats XII

Assembly Language Support XIII

Producing ROMable Code XIV

Floating Point Support XV

SID/ZSID Support XVI

Unbuffered I/O Support XVII

User Submitted Software XVIII

Sample Program .... Appendix A

Programming for Efficiency .... Appendix B

INTRODUCTION

Welcome to the growing number of Aztec C II users. This manual will describe the use of the various components of the Aztec C II system.

1.1 Origin of "C"

Dennis Ritchie originally designed "C" for the UNIX project at Bell Telephone Laboratories. All of the UNIX operating system, its utilities, and application programs are written in "C".

1.2 Standard Reference Manual for "C"

The standard reference for the "C" language is:

Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language. Prentice-Hall Inc., 1978, (Englewood Cliffs, N. J.)

The above text besides providing the standard definition and reference for the "C" language is an excellent tutorial. Aztec C II can be conveniently used in conjunction with the K & R text for learning the "C" language. Aztec C II is a complete implementation of the K & R standard "C". The K & R book is an essential part of the Aztec C II documentation. Most questions regarding the "C" language and many questions on the run time library package will only be answered in the K & R text.

1.3 Basic Components of the Aztec C II System

The Aztec C II system consists of a comprehensive set of tools for producing software using the "C" programming language. The system includes a full feature "C" compiler, a relocating assembler, a linkage editor, an object library maintenance utility, plus an extensive set of run time library routines. Also included are interfaces to MICROSOFT's MACRO-80 assembler (M80) and Digital Research's SID/ZSID debugging system.

1.4 Brief System Overview

The Aztec C II compiler is a complete implementation of UNIX version 7 "C", with the exception of the bit field datatype. The compiler produces relocatable 8080 source code. The compiler can optionally produce Z80 instructions for some optimization on Z80 systems. It does not, however, generate Z80 mnemonics. The source output of the Aztec C II compiler can be assembled by the MICROSOFT MACRO-80 (M80) assembler.

The MANX AS relocating assembler is an 8080 mnemonic assembler that accepts a subset of the MICROSOFT MACRO-80 assembler syntax. The assembler is used to assemble the output of the compiler and for writing assembly language subroutines to be combined with "C" routines.

The relocatable object files produced by the assembler are combined with other relocatable files and library routines by the MANX LN linkage editor. The linkage editor will scan through one or more run time libraries and incorporate any routines that are referenced by the linked modules.

The Aztec C II system also includes LIBUTIL, an object library utility. LIBUTIL allows a user to change the contents of the standard MANX supplied run time library or to create private run time library.

The run time library is included in the standard package in source form, in MANX library format, and in MICROSOFT library format.

1.5 System Requirements

Aztec C II runs on any CP/M or HEATH HDOS system with at least 56K of memory and one disk drive. There are no special terminal requirements for Aztec C II other than the ability to produce upper and lower case and the special characters:

( ) [ ] < > - + = ~ ! ? \ / ^ % * & : ; | " '

1.6 Cross Compilers

A UNIX cross compiler is available for Aztec C II. The cross compiler produces 8080 or Z80 code that can be downloaded to the target machine. Cross compilers are also available from PC-DOS, MS-DOS, and CP/M-86. Cross compilers are in development, for VAX VMS, RSX and RT11 on the PDP 11, and XENIX.

01.7 Portability

Code written for Aztec C II can be compiled with Aztec C][, the Apple DOS 3.3 "C" compiler, Computer Innovations C86 compiler for the IBM PC, CP/M-86, and MSDOS, and UNIX v7 "C". MANX "C" compilers for 8088/8086 and 68000 systems will be available in early 1983.

SOFTWARE LICENSE

COPYRIGHT

INSTALLATION

Aztec C II is distributed on one or more diskettes. Two compilers and two libraries are supplied. The CII compiler and LIBC.LIB library will run on 8080 systems and produce 8080 code. The CZII and LIBCZ80.LIB library require a Z80 processor for compilation and execution. The Z80 version is more efficient than the 8080 version in the use of memory and CPU resources. The Z80 version does not, however, generate Z80 mnemonics, nor is it fully optimized for the Z80. Any other libraries supplied on the distribution disk are 8080. To produce code that will run on both 8080 and Z80 systems the 8080 compiler and libraries must be used.

Instructions for generating a "working disk" from the distribution disk(s) will be found in the release document supplied with the disks. Because some releases have significant changes in the form and contents of the distribution disk, it is very important to read the release document thoroughly to insure proper creation of a working disk.

After creating a working disk, it is advisable to compile, assemble, link edit, and execute the sample program, EXMPL.C. To produce an executable absolute file, follow the procedure described in the OVERVIEW section of this manual. To execute the program once created, type in EXMPL. The program will display the following:

```
                    enter your name
```

When you enter your name followed by a carriage return, the program will display a simple greeting.

Your Aztec CII compiler, assembler, and link editor are now installed and ready to go.

OVERVIEW

Figure 1 depicts the basic steps for producing a binary image of a "C" program. It also indicates the path for producing and using run time subroutine libraries. The process depicted is fairly basic.

```
          +---------------------+
       1. |       EDITOR        |
          +---------------------+
                    |
            / ------------- \
           |       "C"       |
           |   source file   |
            \ ------------- /
                    |
          +---------------------+
       2. | Aztec C II compiler |
          +---------------------+
                    |
            / ------------- \
           |      "ASM"      |
           |   source file   |
            \ ------------- /
                    |
          +---------------------+
       3. |  MANX AS Assembler  |
          +---------------------+
                    |
            / ------------- \        +---------------------+
           |       "O"       |       |                     |
           |   object file   |---> | LIBUTIL librarian   |
            \ ------------- /        +---------------------+
                    |                           |
                    |                  / ----------- \
          +---------------------+     |  subroutine   |
       4. | MANX LN Link Editor |<-- |    library    |
          +---------------------+      \ ----------- /
                    |
            / ------------- \
           |      "COM"      |
           | executable file |
            \ ------------- /
                    |
          +---------------------+      +------------------+
       5. | program execution   |<---->| Digital Research |
          |                     |      |   SID debugger   |
          +---------------------+      +------------------+
```

Figure 1. Developing "C" Programs with Aztec C II

In developing a large system, many "C" programs would be compiled and assembled into object files. A private library might be built to contain frequently used subroutines. Object modules would be combined with library routines by the linkage editor to produce an executable binary image.

Figure 2 depicts the basic steps for producing a binary image of a "C" program using Aztec C II and the MICROSOFT MACRO-80 (M80) assembler, LINK-80 linker (L80), and LIB-80 (LIB) librarian. The basic procedure is the same.

```
     +----------------------+
  1. |        EDITOR        |
     +----------------------+
                |
        /  ____________  \
       |       "C"        |
       |   source file    |
        \  ____________  /
                |
     +----------------------+
  2. | Aztec C II compiler  |
     +----------------------+
                |
        /  ____________  \
       |      "ASM"       |
       |   source file    |      at this step ".ASM" must be
        \  ____________  /       renamed ".MAC"
                |
     +----------------------+
  3. |  MACRO-80 assembler  |     the .8080 option must be used
     +----------------------+
                |
        /  ____________  \         +----------------------+
       |      "REL"       |        |                      |
       |   object file    |---->   | LIB-80  librarian    |
        \  ____________  /         +----------------------+
                |                             |
                |                     /  __________  \
     +----------------------+        |  subroutine    |
  4. | LINK-80 Link Editor  |<--     |   library      |
     +----------------------+         \  __________  /
                |
        /  ____________  \
       |      "COM"       |
       | executable file  |
        \  ____________  /
                |
     +----------------------+      +-------------------+
  5. | program execution    |<---->| Digital Research  |
     |                      |      |   SID debugger    |
     +----------------------+      +-------------------+
```

Figure 2.
Developing "C" Programs with Aztec C II and the MICROSOFT System

In the following text several references are made to "COM" files. CP/M absolute modules usually have ".COM" sufffixed to the filename and so "COM" refers to an executable module. For HEATH HDOS the corresponding type of file has ".ABS" appended to the file name.

Source programs for Aztec C II are created with a text editor. A text editor is not supplied with the Aztec CII, but there are numerous excellent editors available for CP/M and HDOS.

There are three steps to follow to create a "COM" (CP/M) or "ABS" (HDOS) file from a "C" source file. The first step is the compile step that translates "C" source into assembler mnemonics. The second step is an assembly of the assembler source file generated by the "C" compiler. Either the MANX AS assembler or the MICROSOFT MACRO-80 assembler may be used. The third step is the link edit step. If the MANX AS assembler was used in the first two steps, then the MANX LN linker must be used to combine the object files and library object routines to produce an executable file. If the MICROSOFT assembler was used then the MICROSOFT linker must be used. The Manx and MICROSOFT object files are not compatible.

Assume that a "C" source program, EXMPL.C, exists. Then the following procedure would produce a EXMPL.COM (CP/M) or EXMPL.ABS (HDOS) file.

step 1:

CII exmpl.c <u>compile</u>

step 2:

AS exmpl.asm <u>assemble</u>

step 3:

LN exmpl.o libc.lib <u>link</u>

In the above example, the output file from the compile step, "exmpl.asm", is specified as the input to the assembly step. The output file from the assembler, "exmpl.o", is specified as the input file to the linkage editor. The output of the linkage editor is named "exmpl.com" (CP/M) or "exmpl.abs" (HDOS).

Any number of object files can be linked together. Common subroutines can be automatically included through a library search. The "-L" option specifies the library name. "libc.lib" is the name of a run time library supplied with the compiler package. It must be included in every link. Additional libraries can be supplied with additional "-L" specifications. The linker can tell the difference between a library and a simple object file allowing the "-L" to be omitted. A library is created by the

LIBUTIL program.

In order to use Aztec C II with the Microsoft assembler and linker, the "-M" option must be specified on the compile step. To create a library for use with the Microsoft link editor, the "C" library source supplied with this package must be compiled with the "-M" option, assembled with the Microsoft assembler, and placed in a Microsoft library. Library assembler source must be assembled and placed in the same library (see section VI, Library Functions).

The CII command and "libc.lib" library are 8080 compatible. The CZII command and "libcz80.lib" library are Z80 equivalents and can be used on Z80 systems in place of CII and "libc.lib" for some improvement in memory utilization and execution speed.

COMPILER

The Aztec C II compiler is implemented according to the language description supplied by Brian W. Kernighan and Dennis M. Ritchie, in The C Programming Language. The user should refer to that document for a description and definition of the "C" language. This document will detail areas where the Aztec C II compiler differs from the description in that book.

The reader who is not familiar with "C" and does not have a copy of the Kernighan and Ritchie book is strongly advised to acquire one. The book provides an excellant tutorial for learning and using C. The program examples given in the book, can be entered, compiled with Aztec C II and executed to reinforce the instruction given in the text.

The library routines defined in standard C that are supported by Aztec "C" are identical in syntax to the standard. The library routines that are supported are defined in the library section of this manual. In order to allow access to native operating system functions, Aztec CII includes some extended library routines that do not exist in the standard C. These also are described in the library section. The system dependent functions should be avoided in favor of the standard functions in order to reduce future conversion problems.

Aztec C II requires the following statement:

```
#include "libc.h"
```

If none of the special open options are used (see Library Function section), then the following can be used instead of libc.h:

```
#include "stdio.h"
```

Aztec C II is invoked by the command:

```
CII name.c
```

It is recommended that the filename end in ".c", but it is not necessary. "C" source statements found in the "name.c" file are translated to assembler source statements and written to a file named "name.asm". If some other name is wanted then the "-O" option is used (O is a letter). For example

```
CII -O temp.asm exmpl.c
```

will process the "C" statements in exmpl.c and write the translated assembler source to temp.asm.

If the Microsoft assembler is to be used, the "-M" option is required.

By default Aztec C II expects that pointer references to members within a structure are limited to the structure associated with the pointer. To support programs written for other compilers where this is not the case, the "-S" option is provided. If "-S" is specified as a compile time option and a pointer reference is to a structure member name that is not defined in the structure associated with the pointer, then all previously defined structures will be searched until the specified member is found. The search will begin with the structure most recently defined and search backwards from there.

Compiler Options

"-T" option	will copy the "C" source statements as comments in the assembly language output file. Each "C" statement is followed by the assembly language code generates from the statement.

There are four options for changing default internal table sizes:

-E option	specifies the size of the expression work table.
-X option	specifies the size of the macro (#define) work table. The -Y option specifies the maximum number of outstanding cases allowed in a switch statement.
-Y option	specifies the maximum number of outstanding cases allowed in a switch statement.
-Z option	specifies the size of the string literal table.

The default value for -E is 120 entries. Each entry uses 14 bytes. Each operand and operator in an expression requires one entry in the expression table. Each function and each comma within an argument list is an operator. There are some other rules for determining the number of entries that an expression will require. Since they are not straightforward and are subject to change, they will not be defined here. The best advice is that if a compile terminates because of an expression table overflow (error 36), recompile with a larger value for -E.

The following expression uses 15 entries in the expression table:

```
a = b + function( a + 7, b, d) * x
```

The following will reserve space for 300 entries in the expression table:

```
cii -E300 prog.c
```

There must be no space between the -E and the entry size.

The macro table size defaults to 2000 bytes. Each "#define" uses four bytes plus the total number of bytes in the two strings. The following macro uses 9 bytes of table space:

```
#define v 0X1F
```

The following will reserve 4000 bytes for the macro table:

```
cii -X4000 prog.c
```

The macro table needs to be expanded if an error 59 (macro table exhausted) is encountered.

The default size for the case table is 200 entries, with each entry using 4 bytes.

The following will use 4 (not 5) entries in the case table:

```
switch (a) {
case 0:
    a+=1;
    break;
case 1:
    switch (x) {
    case 'a':
        funct1(a);
        break;
    case 'b':
        funct2(b);
        break;
    }
    a = 5;
case 3:
    funct2(a);
    break;
}
```

The following allows for 300 outstanding case statements:

```
cii -Y300 prog.c
```

The size of the case table needs to be increased if an error 76 (case table exhausted) is encountered.

The size of the string table defaults to 2000. Each string literal occupies a number of bytes equal to the size of the string. The size of a string is equal to the number of characters in the string plus one (for the null terminator).

The following will reserve 3000 bytes for the string table:

```
cii -Z3000 prog.c
```

The size of the string table needs to be increased if an error 2 (string space exhausted) is encountered.

The name of the "C" source file must always be the last argument in the command line.

ASSEMBLER

The MANX AS assembler accepts a subset of the Microsoft MACRO-80 assembler language. The Manx AS assembler does not support macroes or Z80 mnemonics.

The MANX AS assembler is a relocating assembler and is invoked by the command line:

```
AS name.asm
```

The relocatable object file produced by the assembly will be named name.o where name is the same name as the name on the .asm file. An alternate object filename can be supplied by specifiying -O filename (O is a letter). The object file will be written to the filename following "-O". The filename does not have to end with ".o", it is, however, the recommended format. The file "name.asm" is the assembly language source file. The filename does not have to end in ".asm".

To produce an assembly listing, specify "-L". The assembler is a one pass assembler so forward address references will not appear on the listing.

The following defines the syntax for the AS assembler:

STATEMENTS

Source files for the MANX AS assembler consist of statements of the form:

```
[label[:]] [opcode] [argument] [;comment]
```

The brackets "[...]" indicate an optional element.

LABELS

A label consists of 1 to 8 alphanumerics followed by an optional colon. A label must start in column one. If a statement is not labeled then column one must be left blank. A label must start with an alphabetic. An alphabetic is defined to be any letter or one of the special characters:

@ $ _ .

An alphanumeric is an alphabetic, or a digit from 0 to 9.

A label followed by "##" is declared external.

EXPRESSIONS

Expressions are evaluated from left to right with no precedence as to operator or parentheses. Operators are:

+ - * / AND OR XOR NOT SHL SHR MOD

CONSTANTS

The default base for numeric constants is decimal. A number suffixed by a "B" is binary, ie. 10010110B. A number suffixed by a "D" is decimal, ie. 765D. A number suffixed by an O or Q is octal, eg. 126O or 126Q. A number or alphabetic A-F suffixed by an "H" is hexadecimal, ie. 0FEEH.

A character constant is of the form 'character': 'A'.

ASSEMBLER DIRECTIVES

The MANX AS assembler supports the following pseudo operations:

COMMON /<block name>/	sets the location counter to the selected common block.
CSEG	select code segment
DB <exp>	define byte constant
DSEG	select data segment
DW	define word constant (2 bytes)
END	end of assembler source statements
FUNC label	if label is not defined then it is declared external
NLIST	turn off listing
LIST	turn on listing
MACLIB/XTEXT filename	include statements from another file
PUBLIC/EXT/EXTRN label	declares label to be external or entry

LINKER

Overview

A. SUMMARY

The aztec link editor will:

- a. combine object files produced by the Aztec II assembler
- b. select routines from object libraries
- c. produce an executable .COM (CP/M) or ABS (HDOS) file

The following are the options available with this linker:

1.	-l	specifies an input library of subroutines
2.	-o	specifies the output file
3.	-r	generates a symbol table for overlays
4.	-t	creates a symbol table file
5.	-b	sets the base address
6.	-c	sets the base address for the code portion of the output
7.	-d	sets the base address for the data area
8.	-f	allows command arguments to be taken from the file

The MANX LN link editor will combine object files produced by the MANX AS assembler, select routines from object libraries, and produce an executable "COM" (CP/M) or "ABS" (HDOS) file.

Supplied with Aztec C II is the libc.lib object library. In most cases this library must be specified. To link a simple single module routine, the following command will suffice:

```
LN  name.o  libc.lib
```

The operand "name.o" is the name of the object file. The executable file created by LN will be named name.COM (CP/M) or name.ABS (HDOS). The -O option followed by a filename can be used to create an alternative name for the LN output file.

Several modules can be linked together as in the following example:

```
LN -O name.com mod1.o mod2.o mod3.o  libc.lib
```

Also several libraries can be searched as in the following:

```
LN -O name.com mod1.o mod2.o  mylib.lib  lib.lib  libc.lib
```

Libraries are searched sequentially in order of specification. It is expected that all external references will be forward. One way to deal with the problem of routines that make external reference to a routine already passed by the librarian is the following:

```
 LN -O name.com mod1.o mod2.o  mylib.1  mylib.1  libc.lib
```

The link editor will read the "mylib.l" library twice. The second time through it will resolve backward references encountered on the first pass.

B. DETAILED LISTING OF LINKER OPTIONS

-T

to create a symbol table file for the ZSID debugging aid. The symbol table file will have the same prefix name as the ".COM" or ".ABS" file with a suffix of ".SYM".

-B address

to specify a base address other than hex 100. The "base address" is assumed to be in hex. A space is required between the B and the hex address.

-C address

to specify a starting address for code portion of the output. The default is the base address + 3. The first three bytes are usually occupied by a jump instruction to system initialization code. It is assumed that the code starting address is specified as a hex number.

-D address

to specify a data address. Data is usually placed behind the end of the code segment.

-F filename

to merge contents of "filename" with command line arguments. More than one specification of -F can be supplied. There are several advantageous uses for this command. The most obvious is to supply the names of modules that are commonly linked together. All records in the file are read. There is no need to squeeze everything into one record.

STANDARD LIBRARY FUNCTIONS

A. SUMMARY

1. Buffered File I/O (K & R chapter 7)

agetc	(stream)	ASCII version of getc
aputc	(cstream)	ASCII version of putc
fclose	(stream)	closes an I/O stream
fgets	(buffer, max, stream)	reads text from stream to buffer
fopen	(name, how)	opens file name according to how
fprintf	(strm, format, arg1...)	writes formatted print to stream
fputs	(cp, stream)	writes string cp to stream
fread	(buf, sz, cnt, strm)	reads cnt items from strm to buffer
fscanf	(fp, control, p1, p2, ...)	converts input string
fseek	(strm, pos, mode)	positions stream to pos
ftell	(strm)	returns current file position
fwrite	(buf, sz, cnt, strm)	writes count items from buf to strm
getc	(stream)	gets a character from file stream
getchar	()	read from standard input
gets	(buffer)	reads a line from the console
getw	(stream)	returns a word from stream
printf	(format, arg1, arg2...)	writes formatted data on console
putc	(c, stream)	writes character c into stream
putchar	(c)	writes to standard output
puts	(cp)	writes string cp onto console
putw	(c, stream)	writes a word c to stream
scanf	(control, p1, p2, ...)	formats input from standard in
sscanf	(str, control, p1, p2, ...)	reverse of sprintf
ungetc	(c, stream)	pushes c back into stream

2. Unbuffered I/O (K & R chapter 8)

close	(fd)	closes file fd
creat	(name, mode)	creates a file
lseek	(fd, pos, mode)	positions file desc according to mode
open	(name, rwmode)	opens file according to read/write mode
posit	(fd, num)	positions file fd to number record
read	(fd, buf, BUFSIZE)	reads from fd to buf BUFSIZE bytes
rename	(oldname, newname)	renames a disk file
unlink	(filename)	erases a disk file
write	(fd,buf, BUFSIZE)	writes from buffer to fd BUFSIZE bytes

3. String Manipulation

atof	(cp)	converts ASCII to floating
atoi	(cp)	converts ASCII to integer
atol	(cp)	converts ASCII to long
ftoa	(m, cp, prec, type)	converts floating point to ASCII
index	(cp, c)	returns cp from beginning of string
rindex	(cp, c)	returns cp from end of string
strcmp	(str1, str2)	compares str1 with str2
strcpy	(dest, src)	string copy routine
strlen	(cp)	returns length of string
strncmp	(str1, str2, max)	compares $\overset{\text{str}}{\cancel{\text{str}}}$1 to str2 at most max
strncpy	(dest, src, max)	string copy at most max characters

4. Utility Routines

alloc	(size)	allocates memory
blockmv	(dest, src, length)	moves length bytes from src to dest
clear	(area, length, value)	initializes area to value
exit	(n)	stop program
format	(func, format, argptr)	formats data using routine function
isdigit	(c)	checks for digits 0...9
islower	(c)	checks for lower case
isupper	(c)	checks for upper case
sprintf	(buff, form, arg1,arg2)	places string format data in buffer
tolower	(c)	converts to lower case
toupper	(c)	converts to upper case

5. Operating System Interface

bdos	(bc, de)	calls bdos
bios	(n, bc, de)	calls the n'th entry into BIOS
bioshl	(n, bc, de)	calls the n'th entry into BIOS
CPM	(bc, de)	calls bdos
exit	(n)	returns to the operating system
fcbinit	(name, fcbptr)	initializes file control blocks
settop	(size)	bumps top of program memory

6. Math and Scientific Routines

acos	(x)	inverse cosine of x (arcos x)
asin	(x)	inverse sine of x (arcsin x)
atan	(x)	inverse tangent of (arctan x)
atan2	(x,y)	arctangent of x divided by y
cos	(x)	cosine of x
cosh	(x)	hyperbolic cosine
cotan	(x)	cotangent of x
exp	(x)	exponential function of x
log	(x)	natural log of x
log10	(x)	logarithm basi of x
pow	(x, y)	raise x to the y-th power
sin	(x)	sine of x
sinh	(x)	hyperbolic sine function
sqrt	(x)	returns the square root of x

tan	(x)	tangent of x
tanh	(x)	hyperbolic tangent function

B. DETAILED LISTING OF LIBRARY FUNCTIONS

Explanation of Format of Library Descriptions

The following is a sample library function description. Each of its parts is numbered and explained in the paragraphs below. All the library functions found in this section of the manual follow this format:

```
     1.fseek
               2.int 3.fseek 4.(stream, pos, mode)
               5.FILE *stream;
               6.int pos, mode
```

1.fseek
The word located in the left margin is the name of the function to be described. The functions are listed in alphabetical order according to category.

2.int
This is a definition of the type of value returned. Here, it is an integer. (Other types could be longs, characters, doubles, pointers, etc).

3.fseek
This again is the name of the function.

4.(stream, pos, mode)
This is a prototype of the parameter list. In this example, "stream" is a pointer (*) to a structure of type "FILE". The parameters of "pos" and "mode" are integers.

5.FILE *stream This defines the "stream" parameter as type FILE. All parameters must be defined as they are in the function definition.

6.int pos,mode
This defines defines pos and mode as integers.

NOTES:

1. FILE is defined in file libc.h or stdio.h.
2. When calling ANY library function or using MACRO-80, libc.h **MUST BE INCLUDED.**

Standard I/O functions

These functions provide a uniform I/O interface for all programs written in Aztec C II regardless of the operating system being used. They also provide a byte stream orientated view of a file even under systems which do not support byte I/O. To use the standard I/O package you should insert the statement:

```
#include "libc.h"

or

#include "stdio.h"
```

into your programs to define the FILE data type and miscellaneous other things needed to use the functions.

1. Buffered File I/O (K & R chapter 7)

agetc

```
int agetc(stream)
FILE *stream;
```

This is an ASCII version of getc which recognizes an end of line sequence (CR LF on CPM) and returns it as a single newline character ('\n'). Also, an end of file sequence (control Z on CPM) is recognized and returned as EOF. This routine provides a uniform way of reading ASCII data across several different systems.

aputc

```
int aputc(c, stream)
int c; FILE * stream;
```

ASCII version of putc which operates in the same manner as putc. However, when a newline ('\n') is put into the file, an end of line sequence is written to the file (CR LF on CPM).

Note: If a partial data block is written as the last block in a file, it is padded with an end of file sequence (control Z on CP/M) before being flushed.

fclose

```
int fclose(stream)
FILE *stream;
```

The function "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 the function "fopen"). fclose releases the control blocks and buffers which it had allocated to the device or file, thus allowing them to be used when other devices or files are opened for buffered i/o. Also, when a disk file is being closed, fclose writes the internally buffered information, if any, to the file.

If the close operation is successful, fclose returns a non-negative integer as its value. If it isn't successful, "fclose" returns -1 as its value, and sets an error code in the global integer errno. If the close was successful, errno is not modified.

fgets

```
char *fgets (buffer, max, stream)
char *buffer;  int max;
FILE *stream
```

The function "fgets" reads characters from a device or file which has been previously opened for buffered i/o (by a call to "fopen") into the caller's buffer. The operation continues until either (1) a newline character ('\n') is read, or (2) the maximum number of characters specified by the caller have been transferred. If the newline character is read, it will appear in the caller's buffer.

If the read operation is successful, "fgets" returns as its value a pointer to the start of the caller's buffer. Otherwise, it returns the pointer NULL and sets a code in the global integer errno. If it is successful, errno is not modified.

The parameter "stream" identifies the device or file; it contains the pointer which was returned by the function "fopen" when the device or file was opened for buffered i/o.

The parameter "buffer" is a pointer to a character array into which "fgets" can put characters.

The parameter "max" is an integer specifying the maximum number of characters to be transferred.

fopen

```
FILE *fopen(name,how)
char *name; char *how;
```

The function "fopen" prepares a device or disk file for subsequent buffered i/o operations; this is called "opening" the device or file.

If the device or file is successfully opened, fopen returns as its value a pointer to a control block of type FILE. When the user's program issues subsequent buffered i/o calls to this device, the pointer to its control block must be included in the list of parameters. In the descriptions of the other buffered i/o functions which require this pointer, the FILE pointer is called "stream".

If fopen can't open the device or file, it returns the pointer NULL and sets an error code in the global integer "errno". If the open was successful, errno isn't modified.

The parameter "name" is a pointer to a character array which contains the name of the device or file to be opened. The devices which can be opened have the following names:

device name	device
con:	system console
lst: or prn:	line printer
pun:	punch device
rdr:	reader device

The device name can be in upper or lower case.

When a disk file is to be opened, the drive identifier in the name parameter is optional. If its included, the file is assumed to be on the specified drive; otherwise, its assume to be on the default drive.

The "how" parameter specifies how the user's program intends to access the device or file. The allowed values and their meanings are:

"how" value	meaning
"r"	Open for reading. The device or file is opened. If a file is opened, its current position is set to the first character in the file. If the device or file doesn't exist, NULL is returned.
"w"	Open for writing. If a file is being opened, and if it already exists, it is truncated to zero length. If it's a file and the file doesn't exist, it is created.
"a"	Open for append. The calling program is granted write-only access to the device or file. For disk files, if the file exists, the its current position is set to the character which follows the last character in the file. Also, for disk files, if the file doesn't exist, it is created and its current position is set to the start of the file.

"r+"	Open for reading and writing. Same as "r" but the device or file may also be written to.
"w+"	Open for reading and writing. Same as "w" but the device or file may also be read.
"a+"	Open for append and read. Same as "a" but the device or file may also be read.

fprintf

```
fprintf(stream,format,arg1,arg2,...)
FILE *stream;
char *format;...
```

The function "fprintf" formats the caller's parameters as specified by the caller and writes the result to a device or disk file. Formatting is done as described in chapter 7, entitled "Input and Output", of The C Programming Language. Note: Long and floating point conversions are supported by Aztec CII, but not by Aztec C.

The parameter "stream" identifies the device or file. It contains the pointer which "fopen" returned to the caller when the device or file was opened for buffered i/o.

The parameter "format" specifies how the formating is to be done.

The parameters "arg1", etc, are the parameters which are to be formatted.

fputs

```
int fputs(cp,stream)
char *cp; FILE *stream;
```

The function "fputs" writes a character string to a device or disk file. "fputs" uses the function "aputc" to write the string, so newline translation may occur.

If the operation is successful, "fputs" returns zero as its value. Otherwise, it returns EOF.

The parameter "stream" identifies the device or file. It contains the pointer which was returned by "fopen" to the caller when the device or file was opened for buffered i/o.

The parameter "cp" is a pointer to a character array containing the string to be written.

fread

```
int fread(buffer,size,count,stream)
char *buffer;
int size,count;
FILE *stream;
```

Reads count items of size bytes into buffer from stream. Returns the number of items actually read.

fscanf

```
int fscanf(stream,control, arg1, arg2, ...)
FILE *stream;
char *control;
```

Formats data according to control. Data is read from stream file. Formating is done as described in chapter 7, Input and Output, of <u>The C Programming Language</u>.

fseek

```
int fseek (fp,pos,mode)
FILE *fp;
long pos;
int mode;
```

Positions the stream according to pos and mode. Mode is interpreted as follows:

0- seek from 0. Pos is treated as an unsigned number and fp is positioned pos bytes from the beginning of the file.

1- seek relative from the current position.

2- seek relative from the end of the file.

ftell

```
long ftell(stream)
FILE stream;
```

Returns the current byte position of stream from the beginning of the file.

fwrite

```
int fwrite(buffer,size,count,stream)
char *buffer;
```

```
int size,count;
FILE *stream;
```

Writes count items of size bytes from buffer into stream. Returns the number of items actually written.

getc

```
int getc(stream)
FILE *stream;
```

Returns the next character from stream. The unique value EOF is returned if an error is encountered or when reaching end of file. The character is not sign extended so that the unique value EOF (-1) is distinguishable from an 0xff byte in the file.

getchar C MACRO

```
int getchar()
```

Returns the next character from standard input (stdin).

gets

```
char *gets(buffer)
char *buffer;
```

Reads a line from the standard input. The returned value is buffer. All of the usual line editing facilities are available if input is from the console. This is not the case with getchar. Note: the end of line sequence is not left in the buffer. This is different from fgets for compatibility reasons.

getw

```
int getw(stream)
FILE *stream;
```

Returns a word from stream. The least significant byte is read first, followed by the most significant byte. Returns EOF if errors or end of file occur. However, since EOF is a good integer value, errno should be checked to determine if an error has occurred.

printf

```
printf(format,arg1,arg2,...)
char *format; ...
```

Formats data according to format and writes the result to

the console. Formating is done as described in chapter 7, Input and Output, of The C Programming Language.

putc

```
int putc(c,stream)
int c; FILE *stream;
```

Writes character c into stream at the current position. Returns c if all is okay and returns EOF if an error occurs.

putchar

```
int putchar(c)
int c;
```

Writes c to the standard output (stdout)

puts

```
int puts(cp)
char *cp;
```

Writes string cp to the standard output (stdout).

putw

```
int putw(c, stream)
int c; FILE *stream;
```

Writes a word, c, to stream. The least significant byte is written first, followed by the most significant byte. Returns c if all is okay and EOF if error occurs. However, since EOF is a good integer value, errno should be checked to determine if an error has occurred.

scanf

```
int scanf(control, arg1, arg2, ...)
char *control;
```

Formats data according to control. Data is read from standard in. Formating is done as described in chapter 7, Input and Output, of The C Programming Language.

ungetc

```
int ungetc(c stream)
int c; FILE *stream;
```

Pushes c back onto stream so that the next call to getc will return c. Normally returns c, and returns EOF if c cannot be pushed back. Only one character of push back is guaranteed and EOF cannot be pushed back.

2. Unbuffered I/O

Unbuffered I/O is described in chapter 8 of The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie. the chapter is captioned "The UNIX System Interface".

close

```
close(fd)
int fd;
```

An open device or disk file is closed.

The parameter "fd" specifies the device or file to be closed. It is the file descriptor which was returned to the caller by the open function when the device or file was opened.

If the close operation is successful, close returns as its value the value of the fd parameter.

If the close operation fails, close returns -1 and sets a code in the global integer errno. If the close was successful, errno is not modified. The only symbolic value which close may set in errno is EBADF, meaning that the file descriptor parameter was invalid.

creat

```
creat(name, pmode)
char *name;
int pmode;
```

The function "creat" creates a file and opens it for 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).

If "creat" is successful, it returns as its value a "file descriptor", that is, a positive integer which is an index into a table of device and file control blocks. 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.

If "creat" fails, it returns -1 and sets a code in the global integer "errno". If it succeeds, errno is not modified.

The parameter "name" is a pointer to a character array containing the name of the file. The drive identifier in the name is optional. If its included, the file will be created on the specified drive; otherwise, it will be created on the default drive.

The parameter "pmode" is optional; if specified, it is ignored. The pmode parameter 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 an octal value of 0666.

lseek

```
long int lseek(fd, offset, origin)
int fd, origin;
long offset;
```

lseek sets the current position in the file specified by the fd parameter to the position specified by the offset and origin parameters.

The current position is set to the location specified by the origin parameter plus the offset specified by the offset parameter,where the offset is a number of characters.

The value of the parameter "origin" determines the basis for the offset as follows:

0	offset is from beginning of file
1	offset from the current position
2	offset is from the end of file

If lseek is successful, it returns as its value the new current position for the file; otherwise, it returns -1. In the latter case, the global integer errno is set to a symbolic value which defines the error. The symbolic values which lseek may set in errno are: EBADF, if the fd parameter is invalid; EINVAL, if the offset parameter is invalid or if the requested current position is less than zero. If lseek is successful, errno is not modified.

Examples:

1. To set the current position to the beginning of the file:

   ```
   lseek(fd, OL, 0)
   ```

   lseek returns as its value 0, meaning that the current position for the file is character 0.

2. To set the current position to the character following the last character in the file:

```
lseek(fd, 0L, 2)
```

lseek returns as its value the current position of the end of the file, plus 1.

3. To set the current position 5 characters before the present current position:

```
lseek(fd,-5L,1)
```

4. To set the current position 5 characters after the present current position:

```
lseek(fd,5L,1)
```

open

```
open(name,rwmode)
char *name;
```

The function "open" prepares a device or file for unbuffered i/o and returns as its value an integer which must be included in the list of parameters for the i/o function calls which refer to this device or file.

The name parameter is a pointer to a character string which is the name of the device or file which is to be opened. The names of the devices which can be opened are :

device name	device
con:	system console
lst: or prn:	line printer
pun:	punch device
rdr:	reader device

The names can be either upper or lower case.

When a disk file is to be opened, the name string can be a complete name; for example, "b:sample.ext". The drive identifier and the colon character can be omitted; in this case the file is assumed to be on the default drive. The extent and preceeding period can also be omitted, if the file doesn't have an extent field.

The "mode" parameter specifies the type of access to the device or file which is desired, and optionally, for a disk file, specifies other functions which open should perform. The mode values are:

mode value	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	if O_EXCL and O_CREAT are both set, open will fail if the file exists

The integer values associated with the symbolic values for mode are defined in the file "fcntl.h", which can be included in a user's program. To guarantee UNIX compatibility, a program should set the "mode" parameter using these symbolic names.

The calling program must specify the type of access desired by including exactly one of O_RDONLY, O_WRONLY, or O_RDWR in the mode parameter. The other values for mode are optional, and if specified,are "or-ed" into one of the type-of-access values.

If only the O_CREAT option is specified, the file will be created, if it doesn't exist, and then opened. If the file does exist it is simply opened.

If the O_CREAT and O_EXCL options are both specified, and if it didn't previously exist, it will be created and then opened. If it did previously exist, the open will fail.

If the O_TRUNC option is specified, the file will be truncated so that nothing is in it, and then will be opened. The truncation is performed by erasing the file, if it exists, then creating it. It's not an error to truncate a file which doesn't previously exist.

If both O_CREAT and O_TRUNC are specified, open proceeds as if only O_TRUNC was specified.

If open doesn't detect an error, it returns as its value an integer, called a "file descriptor", which must be included in the list of parameters which are passed to the other unbuffered i/o functions when performing i/o operations on the file. The file descriptor is different from the file pointer which is used for buffered i/o.

If open does detect an error, it returns as its value -1, and sets a code in the global integer errno which defines the error. The symbolic values which open may set in errno and their meanings are:

errno value	meaning
EMFILE	maximum number of open devices and files exceeded (11's the limit)
EACCES	invalid access requested
ENFILE	maximum number of open files exceeded
EEXIST	file already exists (when O_CREAT and O_EXCL are both specified)

ENOENT	unable to open file

The file errno.h defines the integer values of the symbolic values. If open doesn't detect an error, errno isn't modified.

Examples:

1. To open the system console for read access:

```
fd = open("con:",O_RDONLY)
```

2. To open the line printer for write access:

```
fd = open("lst",O_WRONLY)
```

3. To open the file "b:sample.ext" for read-only access (the file must already exist):

```
fd = open("b:sample.ext",O_RDONLY)
```

4. To open the file subl.c on the default drive, for read-write access (if the file doesn't exist, it will be created first):

```
fd = open("subl.c",O_RDWR+O_CREAT)
```

5. To create the file "main.txt", if it doesn't exist, or to truncate it to zero length, if it already exists, and then to open it for write-only access:

```
fd = open("main.txt",O_WRONLY+O_TRUNC)
```

posit

```
posit(fd,num)
int fd,num;
```

posit will set the current position for a disk file to a specified 128-byte record.

This function should not be used when UNIX compatibility is required, because it isn't supported by UNIX.

The parameter "fd" identifies the file; fd is the file descriptor which was returned to the caller by open when the file was opened.

The parameter "num" is the number of the specified record, where the number of the first record in the is zero.

If posit is successful, it returns 0 as its value.

If no error occurs, posit returns -1, and sets an error code in the global integer errno. The only symbolic value which may be set in errno is EBADF, in response to a bad file descriptor. If no error occurs, errno isn't modified.

Examples:

1. to set the current position to the first byte in the first record:

```
posit(fd,0)
```

2. To set the current position to the first byte of the fourth record:

```
posit(fd,3)
```

read

```
read (fd, buf,bufsize)
int fd, bufsize; char buf;
```

The read function reads characters from a device or disk file into the caller's buffer. In most cases, the characters are read directly into the caller's buffer.

The fd parameter specifies the file; it contains the file descriptor which was returned to the caller when the file was opened.

The parameter buf is a pointer to the buffer into which the characters from the deive or file are to be placed.

The parameter bufsize specifies the number of characters to be transfered.

If the read operation is successful, it returns as its value the number of characters transfered.

If the operation isn't successful, read returns -1 and places a code in the global integer errno.

For more information, see the description on the unbuffered read operation for the various devices and for disk files in the chapter on unbuffered i/o.

rename

```
rename(oldname, newname)
char oldname[],newname[];
```

The function "rename" changes the name of a file.

The parameter "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 a file with the new name already exists, it is erased before the rename occurs.

The value returned by rename is undefined. Unlike many other i/o functions, rename never modifies the global integer errno.

unlink

```
unlink(name)
char name[];
```

The function "unlink" erases a file.

The parameter "name" is a pointer to a character array containing the name of the file to be erased.

unlink returns 255 as its value if the operation wasn't successful; otherwise it returns a value in the range 0 to 3. Unlike many other i/o functions, unlink never modifies the global integer errno.

write

```
write(fd,buf,bufsize)
int fd, bufsize; char buf;
```

The write function writes characters to a device or disk file from the caller's buffer. The characters are written to the device or file directly from the caller's buffer.

The parameter "fd" specifies the device or file. It contains the file descriptor which was returned by the open function to the caller when the device or file was opened.

The parameter "buf" is a pointer to the buffer containing the characters to be written.

The parameter "bufsize" specifies the number of characters to be written.

If the operation is successful, write returns as its value the number of characters written.

If the operation is unsuccessful, write returns -1 and places a code in the global integer errno. If the operation is successful, errno is not modified.

For more information on the detailed operation of the write function when writing to the different devices and to disk files, see the chapter on unbuffered i/o.

3. String Manipulation

These functions allow manipulation of "C" style strings as described in The C Programming Language by Kernighan and Ritchie.

atof

```
double atof(cp)
char *cp;
```

ASCII to float conversion routine.

atoi

```
int atoi(cp)
char *cp;
```

Converts ASCII string of decimal digits into an integer. Atoi will stop as soon as it encounters a non-digit in the string.

atol

```
long atol(cp)
char *cp;
```

ASCII to long conversion routine.

ftoa

```
int ftoa (m,cp,precision,type)
double m;
char *cp;
int precision;
int type;
```

convert from float/double format to character format. The value of m is converted to ans ASCII string and assigned to *c. The precision operand specifies the number of digits to the right of the decimal point. Type can be

0 for E format

1 for F format.

index

```
char *index(cp,c)
char *cp, c;
```

Searches string cp for the letter specified by parameter "c". If the letter is found then the function returns a pointer to its position. Othersise a 0 is returned.

rindex

```
char *rindex(cp,c)
```

Functions the same as index, but the scan begins from the end of the string and moves towards the beginning.

sscanf

```
int sscanf(string,control, arg1, arg2, ...)
char *string
char *control;
```

Formats string according to control. Formating is done as described in chapter 7, Input and Output, of The C Programming Language.

strcmp

```
strcmp(str1,str2)
char *str1, *str2;
```

Compares str1 to str2 and returns: 0 (zero) if strings are equal, -1 (negative one) if str1 is less than str2, and 1 (one) if str1 is greater than str2.

strcpy

```
strcpy(dest,src)
char *dest, *src;
int max;
```

Copies the string pointed to by src into destination.

strlen

```
strlen(str)
char *str;
```

Returns the length of str. The length does not include the null at the end of the string.

strncmp

```
strncmp(str1,str2,max)
char *str1, *str2;
int max;
```

Compares str1 to str2 the same as strcmp, but compares at most max characters.

strncpy

```
strncpy(dest,src,max)
char *dest, *src;
int max;
```

copies the string pointed to by src into dest, but copies at most max characters. The destination may not be null terminated when copy is done.

4.Utility Routines

alloc

```
char *alloc(size)
int size;
```

Allocates memory with size numer of bytes and returns pointer to beginning.

blockmv

```
blockmv(dest, src, length)
char *dest, *src;
int dest;
```

Moves data from src to dest. The number of bytes is specified by parameter length. N o checking for overlap is performed.

clear

```
clear(area,length,value)
char *area; int length, value;
```

Initializes length bytes starting at area with value.

exit

```
exit(n)
int n;
```

Returns to the operating system. Any streams which have been opened with fopen but not closed with fclose will be closed at this time. If N is non zero then any submit that was in progress will abort.

format

```
format(function,format,argptr)
int (*function) ();
char *format; unsigned *argptr;
```

Formats data according to the string format and calls the given function with each character of the result. Formatting is done as described in chapter 7, Input and Output, of The C Programming Language. Note: the long and floating point conversions are not yet supported.

e.g. The printf routine looks like this:

```
printf(fmt,args)
char *fmt; unsigned args;

      int putchar();
      format(putchar,fmt,&args);
```

isdigit

```
isdigit(c)
int c;
```

Returns one if c is a digit, zero otherwise.

islower

```
islower(c)
int c;
```

Returns one if c is a lower case alphabetic, zero otherwise.

isupper

```
issupper(c)
char c;
```

Tests whether argument is an uppper case letter and returns non zero if it is and zero if not.

sprintf

```
sprintf(buffer,format,arg1,arg2,...)
char *buffer, format;...
```

Formats data according to the string format and leaves the result in buffer. Formatting is done as described in chapter 7, " Input and Output", of The C Programming Language

tolower

```
tolower(c)
int c;
```

If c is upper case,c is mapped to lower case and the new value returned; otherwise c is returned.

toupper

```
toupper(c)
int c;
```

If c is lower case, it is mapped to upper case and the new value returned; otherwise c is returned.

5. Operating System Interface

bdos

```
bdos(bc,de)
int bc,de;
```

Calls the bdos with register pair BC set to bc and DE set to de. The value returned in register A is the return value.

bdos

```
bdoshl(bc,de)
int bc,de;
```

Calls the bdos with register pair BC set to bc and DE set to de. The value returned in register pair HL is the return value.

bios

```
bios(n,bc,de)
int n,bc,de;
```

Calls the n'th entry into the bios with BC set to bc and DE set to de. The returned value is the accumulator contents on return from the CP/M BIOS. N equal to zero is a cold boot.

bioshl

```
bioshl(n,bc,de)
int n,bc,de;
```

Calls the n'th entry into the bios with BC set to bc and DE set to de. The returned value is the HL register contents on return from the CP/M BIOS. N equal to zero is a cold boot.

CPM

```
CPM(bc,de)
int bc,de;
```

Calls the bdos with register pair BC set to bc and DE set to de. The value returned in A is the return value.

exit

```
exit(n)
int n;
```

Returns to the operating system. Any streams which have been opened with fopen but not closed with fclose will be closed at this time. N is the return code, which is ignored in this release but may be used by future versions.

fcbinit

```
fcbinit(name,fcbptr)
char *name; struct _fcb *fcbptr;
```

The _fcb structure is initialized to zeros and name is unpacked into the proper places. The _fcb structure is defined in "io.c". The structure need not be used; however, fcbptr must point to an area at least 36 bytes long.

settop

```
char *settop(size)
unsigned size;
```

The current top of available memory is moved up by size bytes and the old value of the top is returned. If the new top is within 512 bytes of the stack pointer, NULL will be returned.

6. Math and Scientific Routines

sqrt

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

sqrt is a function of one argument which returns as its value the square root of the argument. The type of the returned value is double.

The argument which is passed to sqrt must be of type double and must be greater than or equal to zero.

If sqrt detects an error, it sets a code in the global integer variable ERRNO and returns an arbitrary value to the caller. If sqrt doesn't detect an error, it returns to the caller without modifying ERRNO. Table 2.1.1 lists the symbolic values which sqrt may set in ERRNO and their meanings. The file MATH.H, which can be included in a user's module, declares ERRNO to be a global integer and defines the numeric value associated with each symbolic value.

EXAMPLE

In the following program sqrt computes the square root of 2. If the computation returns a non-zero value in ERRNO, the program prints an error message.

```
#include "libc.h"
#include "errno.h"
main() {
     double sqrt(),a;

     errno = 0;
     a = sqrt((double) 2);
     if (errno != 0) {
          if (errno == EDOM)
              printf("errno set to EDOM by sqrt\n");
          else
               printf("invalid errno=%d returned by sqrt\n");
     }
}
```

Table 2.1.1 Error codes returned in ERRNO by sqrt

Code	sqrt(x)	Meaning
EDOM	0.0	x < 0.0

log

```
double log(x);
double x;
```

log is a function of one argument which returns the natural logarithm of the argument as its value, as a double precision floating point number.

The argument which is passed to log must be a double precision floating point number and must be greater than zero.

If log detects an error, it sets a code in the global variable ERRNO and returns an arbitrary value;otherwise, it returns to the caller without modifying ERRNO. Table 2.2.1 lists the symbolic values which log may set in ERRNO, the associated values returned by log, and the meaning.

Table 2.2.1 Error codes returned in ERRNO by log

Code	log(x)	Meaning
EDOM	-HUGE	x <= 0.0

log10

```
double log10(x);
double x;
```

log10 is a function of one argument which returns as its value the base-10 logarithm of the argument. The type of the returned value is double.

The argument must be greater than zero, and must be of type double.

If log10 detects an error, it sets a code in the global integer ERRNO and returns an arbitrary value to the caller; otherwise, it returns to the caller without modifying ERRNO. Table 2.3.1 lists the symbolic values which log10 may set in

ERRNO, the associated value returned by log10, and the meaning.

Table 2.1.1 Error codes returned in ERRNO by log10

Code	log10(x)	Meaning
EDOM	-5.2e151	x <= 0.0

exp

```
double exp(x);
double x;
```

exp is a function of one argument which returns as its value e**(argument). The type of the returned value is double.

The argument must be greater than -354.8 and less than 349.3; it must be of type double.

If exp is unable to perform the computation, it sets a code in the global integer ERRNO and returns an arbitrary value; otherwise, it returns the computed value without modifying ERRNO. Table 2.4.1 lists the symbolic values that exp may set in ERRNO, the associated value of exp, and the meaning.

Table 2.1.1 Error codes returned in ERRNO by exp

Code	exp(x)	Meaning
ERANGE	5.2e151	x > 349.3
ERANGE	0.0	x < -354.8

pow

```
double pow(x,y);
double x,y;
```

pow is a function of two arguments, for example, x and y, which, when called, returns as its value x to the y-th power (x**y, in FORTRAN notation). x is the first argument to pow, and y the second. The value returned is of type double.

The arguments must meet the following requirements:
x cannot be less than zero;
if x equals zero, y must be greater than zero;

```
if x is greater than zero, then
     -354.8 < y*log(x) < 349.3
```

If pow is unable to perform the calculation, it sets a code in the global integer ERRNO and returns an arbitrary value; otherwise it returns the computed number as its value without modifying ERRNO. Table 2.6.1 lists the symbolic codes which pow may set in ERRNO, the associated value returned by pow, and the meaning.

Table 2.1.1 Error codes returned in ERRNO by pow

Code	pow(x,y)	Meaning
EDOM	-5.2e151	x<0 or x=y=0
ERANGE	5.2e151	y*log(x) > 349.3
ERANGE	0.0	y*log(x) < -354.8

sin

```
double sin(x);
double x;
```

sin is a function of one argument which, when called, returns as its value the sine of the argument. The value returned is of type double.

The argument is in radians, and its absolute value must be less than 6.7465e9. The type of the argument is double.

If sin 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. Table 2.7.1 lists the symbolic codes which sin may set in ERRNO, the associated values returned by sin, and the meaning.

Table 2.1.1 Error codes returned in ERRNO by sin

Code	sin(x)	Meaning
ERANGE	0.0	abs(x) >= 6.7465e9

cos

```
double cos(x);
double x;
```

cos is a function of one argument which, when called, returns as its value the cosine of the argument. The returned value is of type double.

The argument is in radians, and its absolute value must be less than 6.7465e9. The type of the argument is double.

If cos can't perform the computation, it returns an arbitrary value and sets a code in the global integer ERRNO; otherwise, it returns the computed value, without modifying the associated value returned by cos, and the meaning.

Table 2.1.1 Error codes returned in ERRNO by cos

Code	cos(x)	Meaning
ERANGE	0.0	abs(x) >= 6.7465e9

tan

```
double tan(x);
double x;
```

tan is a function of one argument which, when called, returns as its value the tangent of the argument. The type of the value returned is double.

The argument is in radians, and its absolute value must be less than 6.7465e9. The type of the argument is double.

If tan can't perform the computation, it returns an arbitrary value and sets a code in the global integer ERRNO; otherwise, it returns the computed value without modifying ERRNO. Table 2.8.1 lists the codes which tan may set in ERRNO, the associated value returned by tan, and the meaning.

Table 2.1.1 Error codes returned in ERRNO by tan

Code	tan(x)	Meaning
ERANGE	0.0	abs(x) >= 6.7465e9

cotan

```
double cotan(x);
double x;
```

cotan is a function of one argument which, when called, returns as its value the cotangent of the argument. The returned value is of type double.

The argument is in radians, and its absolute value must be greater than 1.91e-152 and less than 6.7465e9. The type of the argument is double.

If cotan can't perform the computation, it returns an arbitrary value and sets a code in the global integer ERRNO; otherwise, it returns the computed value without modifying ERRNO. Table 2.9.1 lists the symbolic codes which cotan may set in ERRNO, the associated value returned by cotan, and the meaning.

Table 2.1.1 Error codes returned in ERRNO by cotan

Code	cotan(x)	Meaning
ERANGE	5.2e151	0<x<1.91e-152
ERANGE	-5.2e151	-1.91e-152 <x<0
ERANGE	0.0	abs(x) >= 6.7465e9

asin

```
double asin(x);
double x;
```

asin is a function of one argument which, when called, returns as its value the arcsine of the argument. The returned value is of type double.

The absolute value of the argument must be less than or equal to 1.0. Its type is double.

If asin can't perform the computation, it returns an arbitrary value and sets a code in the global integer ERRNO; otherwise, it returns the computed value without modifying ERRNO. Table 2.10.1 lists the symbolic codes which asin may set in ERRNO, the associated values returned by asin, and the meaning.

Table 2.1.1 Error codes returned in ERRNO by asin

Code	asin(x)	Meaning
EDOM	0.0	abs(x) > 1.0

acos

```
double acos(x);
double x;
```

acos is a function of one argument which, when called, returns as its value the arcosine of the argument. The returned value is of type double.

The absolute value of the argument must be less than or equal to 1.0. It must be of type double.

If acos can'g perform the computation, it returns an arbitrary value and sets a code in the global integer ERRNO; otherwise, it returns the computed value without modifying ERRNO. Table 2.11.1 lists the symbolic codes which acos may set in ERRNO, the associated value returned by acos, and the meaning.

Table 2.1.1 Error codes returned in ERRNO by acos

Code	acos (x)	Meaning
EDOM	0.0	abs(x) > 1.0

atan

```
double atan(x);
double x;
```

atan is a function of one argument which, when called, returns as its value the arctangent of the argument. The returned value is of type double.

The argument can be any real value, and must be of type double.

Unlike many of the other math functions, atan never returns code in ERRNO.

atan2

```
double atan2(y,x);
double y,x;
```

atan2 is a function of two arguments, say x and y, which, when called, returns as its value the arctangent of y/x, in radians. y is the first argument, and x is the second. The returned value is of type double.

The arguments can assume any real values, except that x and y cannot both be zero. If x equals zero, the value returned is also zero.

If atan2 can't perform the computation, it returns an arbitrary value and sets a code in the global integer ERRNO; otherwise, it returns the computed value without modifying ERRNO. Table 2.12.1 lists the symbolic codes which atan2 may set in ERRNO, the associated values returned by atan2, and the meaning.

Table 2.1.1 Error codes returned in ERRNO by atan2

Code	atan2(x)	Meaning
EDOM	0.0	x = y = 0

sinh

```
double sinh(x);
double x;
```

sinh is a function of one argument which returns as its value the hyperbolic sine of the argument. The returned value is of type double.

The absolute value of the argument must be less than 348.606839, and is of type double.

If sinh can't perform the computation, it sets a code in the global integer ERRNO and returns an arbitrary value; otherwise, it returns the computed value without modifying ERRNO. Table 2.13.1 lists the symbolic codes which sinh may set in ERRNO, the value returned by sinh, and the meaning.

Table 2.1.1 Error codes returned in ERRNO by sinh

Code	sinh(x)	Meaning
ERANGE	5.2e151	abs(x) > 348.606839

cosh

```
double cosh(x);
double x;
```

cosh is a function of one argument which returns as its value the hyperbolic cosine of the argument. The value returned is of type double.

The absolute value of the argument must be less than 348.606839, and it must be of type double.

If cosh can't perform the computation, it returns an arbitrary value and sets a code in the global integer ERRNO; otherwise, it returns the computed value without modifying ERRNO. Table 2.14.1 lists the symbolic codes which cosh may set in ERRNO, the associated values returned by cosh, and the meaning.

Table 2.1.1 Error codes returned in ERRNO by cosh

Code	cosh(x)	Meaning
ERANGE	5.2e151	abs(x) > 348.606839

tanh

```
double tanh(x);
double x;
```

tanh is a function of one argument which returns as its value the hyperbolic tangent of its argument. The value returned is of type double.

The argument can be any real number whatsoever. It must, however, be of type double.

Unlike some of the other math functions, tanh never modifies ERRNO, and always returns the computed value.

MICROSOFT COMPATIBILITY

The Microsoft assembler (M80) and linker (L80) can be used with the Aztec C II compiler. The "-M" option must be specified on all compilations targeted for the M80 assembler as in the following:

```
CII -M crtdrivr.c
```

Some older versions of L80 will not work with Aztec C II.

You must specify:

```
#include "libc.h"
```

in every module that will be used with the MICROSOFT system.

An 8080 library for use with the MICROSOFT L80 linker is supplied on the distribution disk. Read the release document for more details. Generally a Z80 Microsoft library will not be included on the distribution disk and must be created. To create a library to use with the L80 linker, all of the ".C" programs supplied on the distribution disk must be compiled using the "-M" option, assembled with the M80 assembler, and placed in a Microsoft library. The ".ASM" files supplied on the distribution disk must also be assembled with M80 and placed in the library. Some of the supplied assembler source has "8080" or "Z80" as part of the filename. Only the 8080 versions should be assembled for an 8080 system, and only the Z80 versions should be assembled for a Z80 system. Code intended for assembly with the MICROSOFT MACRO-80 assembler should not include labels with leading "_".

You must specify .8080 to the MICROSOFT M80 assembler to assemble source files created by Aztec C II. Z80 assembler subroutines using Z80 mnemonics can be combined with the Aztec C II modules by specifying .Z80 to the MICROSOFT assembler for the Z80 source modules. The "rel" file outputs for .8080 and .Z80 are compatible.

ERROR MESSAGES

ERROR NUMBER	EXPLANATION
1	bad digit in octal constant
2	string space exausted (see COMPILER -Z option)
3	unterminated string
4	compiler error in effaddr
5	illegal type for function
6	inappropriate arguments
7	bad declaration syntax
8	name not allowed here
9	must be constant
10	size must be positive integer
11	data type too complex
12	illegal pointer reference
13	unimplemented type
14	unimplemented type
15	storage class conflict
16	data type conflict
17	unsupported data type
18	data type conflict
19	too many structures
20	structure redeclaration
21	missing)'s
22	struct decl syntax
23	undefined struct name
24	need right parenthesis
25	expected symbol here
26	must be structure/union member
27	illegal type CAST
28	incompatable structures
29	structure not allowed here
30	missing : on ? expr
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 (see COMPILER -E option)
37	invalid expression
38	no auto. aggregate initialization
39	no strings in automatic
40	this shouldn't happen
41	invalid initializer
42	too many initializers
43	undefined structure initialization
44	too many structure initializers
45	bad declaration syntax
46	missing closing bracket
47	open failure on include file
48	illegal symbol name
49	already defined
50	missing bracket

51	must be lvalue
52	symbol table overflow
53	multiply defined label
54	too many labels
55	missing quote
56	missing apostrophe
57	line too long
58	illegal # encountered
59	macro table full (see COMPILER -X option)
60	output file error
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	bad goto syntax
71	statement syntax
72	statement syntax
73	statement syntax
74	case value must be integer constant
75	missing colon on case
76	too many cases in switch (see COMPILER -Y OPTION)
77	case outside of switch
78	missing colon
79	duplicate default
80	default outside of switch
81	break/continue error
82	illegal character
83	too many nested includes
84	illegal character
85	not an argument
86	null dimension
87	invalid character constant
88	not a structure
89	invalid storage class
90	symbol redeclared
91	illegal use of floating point type
92	illegal type conversion
93	illegal expression type for switch
94	bad argument to define
95	no argument list
96	missing arg
97	bad arg
98	not enough args
99	conversion not found in code table

ERROR PROCESSING

During run time three variables are used to enhance error handling. An external variable "errno" is an integer that is set to an error code by the I/O and scientific math routines. "Sysvec" is an array used to control error processing for floating point numbers. "flterr" is set to indicate floating point arithmetic errors. "flterr" set to 0 indicates a good result, a non-zero value indicates a bad result. See the section on floating point support for more details.

"errno" is set to 0 at the beginning of each I/O request and is set to a non-zero value if an error occurred.

"errno" is set to a non-zero value if an error occurred in processing a scientific math function see section VI, Library Functions for more information.

The definition for the various settings for errno is in file errno.h. The following is the contents of errno.h for v1.05 of Aztec C II:

```
int errno;
#define ENOENT  -1      file does not exist
#define E2BIG   -2      not used
#define EBADF   -3      bad file descriptor - file is not open or
                        improper operation
#define ENOMEM  -4      insufficient memory for requested
                        operation
#define EEXIST  -5      file already exists on create request
#define EINVAL  -6      invalid argument
#define ENFILE  -7      exceeded maximum number of disk files
#define EMFILE  -8      exceeded maximum number of file
                        descriptors
#define ENOTTY  -9      not used
#define EACCES  -10     invalid access request

#define ERANGE  -20     invalid argument to math function:
                        function value can't be computed
#define EDOM    -21     invalid argument to math function:
```

LIBRARY MAINTENANCE

LIBUTIL

A. SUMMARY

The LIBUTIL LIBrary UTILity is used in order to:

```
1. create a library
2. append a library        (-a)
3. produce an index list   (-t)
4. extract members         (-x)
5. replace a library       (-r)
6. create a library using an
   extended command line   (.)
```

1. LIBUTIL -o example.lib x.o x.o

USE -	to create a library
FUNCTION	the following creates a private library, example.lib, containing modules subl.o and sub2.o

```
>LIBUTIL -o example.lib subl.o sub2.o
```

2. LIBUTIL option -a

USE -	to append to a library
FUNCTION-	the following appends exmpl.o to the example.lib

```
>LIBUTIL  -o example.lib -a exmpl.o
```

this function can be used to append any number of .o files to the library. For example, the following appends exmpl.o and smpl.o to the example.lib

```
>LIBUTIL  -o example.lib -a exmpl.o smpl.o
```

NB If a large number of files need to be appended to a library, it is advantageous to use the SUBMIT option (see item 7)

3. LIBUTIL option -t

USE -	to produce an index listing of modules in a given library
FUNCTION-	the following displays a listing of all

modules in a particular library, example.lib:

```
>LIBUTIL  -o example.lib -t
```

NB this function will allow only one library to be listed at a time

4. LIBUTIL option -x

USE - a. copies a particular library module into a relocatable object file
b. copies a complete library into relocatable object files

FUNCTION- a. the following copies library module, exmpl into a relocatable object file:

```
>LIBUTIL  -o example.lib -x exmpl
```

b. the following copies a complete library, example.lib, (including all modules contained within it) into relocatable object files:

```
>LIBUTIL  -o example.lib -x
```

NB. It should be noted that when copying a single module the LIBUTIL executes the command and returns. When copying a complete library, the LIBUTIL lists the modules being copied.

5. LIBUTIL option -r

USE - to replace a library module with the contents of a relocatable object file

FUNCTION- the following replaces the library module subl with the relocatable object file subl.o

```
>LIBUTIL  -o example.lib -r  subl.o
```

6. LIBUTIL -o library name .

USE to create a library using an extended command line

FUNCTION the following creates a library, charles.lib and appends to it subl.o, sub2.o, sub3.o, sub4.o, etc.

```
>xsub
```

```
                LIBUTIL -o charles lib .
                sub1.o sub2.o sub3.o sub4.o
                .
```

B. DETAILED EXPLANATION

Creating a Library

The command for creating a library has the following two formats:

format 1:

```
LIBUTIL [-o <output library name>] <input file list>
```

format 2:

```
LIBUTIL [-o <output library name>] <input file list>
one or more lists, each an <input file list>
```

If the optional parameter [-o <output library name>] is specified, the name of the file containing the library to be created is <output library name>; if this parameter is not specified, the name of the file containing the library to be created is "libc.lib". In either case, LIBUTIL proceeds by first creating the library in a new file having a temporary name; if the creation is successful, LIBUTIL then erases the file named <output library file>, if it exists, and renames the file containing the newly created library to <output library file>.

<input file list> defines the files containing the modules which are to be placed in the library. An input file can be either (1) a file created by the Manx assembler, AS, in which case it contains a single relocatable object module, or it can be (2) another library which was created by LIBUTIL. In either case, the input files are not modified by LIBUTIL; LIBUTIL just copies the modules in the input files to the output library.

An <input file list> is one or more names, separated by spaces. A name can be one of the following: (1) a complete CP/M file name; eg, b:sub1.o; (2) a CP/M file name which doesn't specify the disk drive on which the file resides; eg, sub1.o; in this case, LIBUTIL assumes the file is on the default disk drive; (3) a name which doesn't specify an extension; in this case, LIBUTIL assumes the file name is <name>.o. For example, if the name is sub1, LIBUTIL assumes the file name is sub1.o and is on the default disk drive. If the name is b:sub1, LIBUTIL assumes the file name is b:sub1.o.

When an input file contains a single relocatable object module, the name by which the module is known in the library is the filename, less the disk drive identifier and the extension. For example, if the input file is b:sub1.o, then the module name within the created library is sub1.

When an input file is itself a library, the member names in the created library are the same as the member names in the input library. For example, if an input file is a library containing modules sub1, sub2, and sub3, then the name of these modules in the created library are also sub1, sub2, and sub3.

To specify that there are additional lines of <input file lists>, a period surrounded by at least one space on either side must appear in the <input file list> on the first line of the command. Of course, LIBUTIL doesn't assume that such a period is a name; it just acts as a flag to LIBUTIL, specifying that there are additional lines of <input file list>s. Also, names can both preceed and follow the period flag.

The order in which modules are placed in the created library is specified by the order of the names in the input file lists. If there is only one input file list, for example:

```
          sub1.o sub2.o sub3.o ,
```

where the input files each contain a single relocatable object module, then the order of the modules in the library would be: sub1, sub2, sub3.

If an input module is itself a library, then its modules are copied to the created library in the same order. If there is only one input file list, for example

```
          sub1.o lib1.lib sub2.o
```

where sub1.o and sub2.o each contain a single relocatable object module and lib1.lib is a library containing modules sub3, sub4, and sub5, in that order, then the created library would contain modules in the following order:

```
          sub1, sub3, sub4, sub5, sub2.
```

If there are additional lines of input file lists, then modules are placed in the created library in the following order: first, the modules in the files preceeding the period flag are placed in the created library, as defined above; second, the modules in the additional input file lists are placed in the created library, third, the modules in the files succeeding the period flag are placed in the created library. For example, suppose LIBUTIL is invoked with the following sequence:

```
LIBUTIL -o newlib.lib sub1.o . sub2.o
sub3.o sub4.o
sub5.o sub6.o
   .
```

If each of the input files contains a single relocatable object module, then the created library would contain the following modules in the specified order: sub1, sub3, sub4, sub5, sub6, sub2.

Listing the modules in a library

To have LIBUTIL produce a listing of the modules in a library, LIBUTIL must be invoked with a "dash parameter" which contains the character 't'. A dash parameter is simply a parameter which has a dash (-) as its first character. LIBUTIL lists only the modules in the library, not the functions.

The user can explicitly tell LIBUTIL the name of the library file to be listed by including the character 'o' in a dash parameter; in this case, LIBUTIL assumes that the following parameter is the name of the library file.

The user can implicitly tell LIBUTIL which library file is to be listed by not including the character 'o' in a dash parameter; in this case, LIBUTIL assumes that the file libc.lib is to be listed.

LIBUTIL will not perform multiple functions during a single invocation. For example, you can't make it create a library and then list the contents with only a single activation of LIBUTIL; you would have to activate it to create the library, then activate it again to list the contents.

The parameter list to LIBUTIL, when it is to perform a listing, can include either one or two dash parameters. If one is used, then both the 't' character and the 'o' character (if specified) are in it; in this case, they can appear in any order. If two dash parameters are used, then one contains the single character 't' and the other the single character 'o'. The only restriction in this case is that the name of the library file must be the parameter string immediately following the dash parameter which has the 'o'.

EXAMPLES:

```
LIBUTIL -t
          lists the modules in the library file libc.lib
LIBUTIL -ot example.lib
LIBUTIL -t -o example.lib
LIBUTIL -o example.lib -t
          each of these three lines causes LIBUTIL to list the
          modules in the library example.lib
```

Adding modules to a library and replacing modules in a library

LIBUTIL can be told to add modules to a library or replace modules in a library by including one of the characters 'a' or 'r' in a dash parameter. There is only one function, which performs both an 'add' operation and a 'replace' operation. Either character, 'a' or 'r' causes LIBUTIL to perform the function. The user also tells LIBUTIL, either explicitly or implicitly, the name of the library file on which the operation is to occur and gives LIBUTIL a list of files whose modules are to be added to or replaced in the library. Each of these files can contain either a single relocatable object module or can be itself a library. In the following paragraphs, the library file on which the operation is to occur is called the 'subject library file' and each file which is to be added or replaced is called an 'input file'.

LIBUTIL proceeds as follows: it creates a library file with a temporary name. Then it copies modules one at a time from the subject library to the new library; before copying each module, it checks whether there is a file in the input file list whose name, less drive specification and extent, is the same as that of the module; if not, the module is copied. If they do match, LIBUTIL copies the contents of the matching file to the new library, and the module from the subject library is not copied. If, after LIBUTIL has processed all modules in the subject library in this manner, any files in the input file list remain which haven't been copied to the new library, LIBUTIL then copies the contents of these files to the new library. Finally, LIBUTIL erases the original subject library and renames the new library, giving it the name of the subject library file.

The user can give LIBUTIL the name of the subject library either explicitly or implicitly. To explicitly define it, the user includes the character 'o' in a dash parameter; the parameter immediately following this dash parameter must then be the name of the subject library file. To implicitly define it, the user simply doesn't include the 'o' character in adash parameter; LIBUTIL then assumes that the name of the subject library file is 'libc.lib'.

All parameters which follow the dash parameters and the subject file name are names of input files. The drive identifier and/or the extent of these names can be optionally ommitted. If the drive identifier is omitted, LIBUTIL assumes the file is on the default drive. If the extent is omitted, LIBUTIL assumes the extent is 'ext'.

LIBUTIL can be told to read additional input file names from one or more lines on the console device by including the character '.' in place of one of the input file names on the LIBUTIL command line. In this case, LIBUTIL will read input file names from the console until another '.' is read where a file name was expected. LIBUTIL then continues reading input file names from the original command line.

Once LIBUTIL has finished its copy-with-replace function from the subject library to the new library, it will append the input files which haven't been copied to the the new library in the same order in which it read their names from the command lines.

EXAMPLES

1. Let example.lib be a library file on the default disk drive which contains the modules sub1, sub2, and sub3. To append the module in the file newsub.o, which is also on the default drive, to example.lib any ofthe following commands could be issued:

```
LIBUTIL -oa example.lib newsub
LIBUTIL -oa example.lib newsub.o
LIBUTIL -ao example.lib newsub
LIBUTIL -a -o example.lib newsub.o
LIBUTIL -o example.lib -a newsub
```

After LIBUTIL is done, there will be a new library file named example.lib, and it will contain the following modules, in the order specified: sub1, sub2, sub3, and newsub. The module in the file newsub.o doesn't have a name; it only gets one when a copy of it is placed in a library. The name of the module is derived from the name of the file in which it was originally contained by stripping that file name of the disk drive prefix and extent suffix. In this example, the name of the module which is appended to example.lib is thus 'newsub'. Just to beat this example to death, suppose that we are back at the point at which we have the original example.lib, containing modules sub1, sub2, and sub3, and that we have the file newsub.o. After entering the following commands:

```
rename sub4.o=newsub.o
LIBUTIL -oa example.lib sub4
```

example.lib will contain modules named sub1, sub2, sub3, sub4.

2. Let example.lib contain the modules sub1, sub2, and sub3; and let newlib.lib contain the modules newsub1, newsub2, and newsub3. We can tell LIBUTIL to append the modules in newlib.lib to example.lib by entering any of the following lines:

```
LIBUTIL -oa example.lib newlib.lib
LIBUTIL -a -o example.lib newlib.lib
LIBUTIL -o example.lib -a newlib.lib
```

After LIBUTIL is done, there will be a new example.lib, and it will contain the following modules, in the specified order: sub1, sub2, sub3, newsub1, newsub2, newsub3.

To illustrate another point, let's rerun LIBUTIL again with the comand specified above, starting with the original example.lib, containing sub1, sub2, and sub3, and with the library file newlib.lib containing the modules sub3, newsub1, sub1, and newsub2. After LIBUTIL completes, there will be a new example.lib, and it will contain the following modules, in the specified order: sub1, sub2, sub3, sub3, newsub1, sub1, newsub2. The first sub1 module in the new example.lib will be that from the original example.lib, and the second will be from newlib.lib. The first sub3 module in the new example.lib will be from the original example.lib, and the second will be from newlib.lib. The point being exemplified is that LIBUTIL will not replace modules in the original library with modules from an input library; it will only append modules in the input library to the subject library.

3. Let example.lib be a library containing the modules sub1, sub2, and sub3. To replace module sub2 with the contents of the file named sub2.o and to append the modules in the library file newlib.lib (which are mod1, mod2, and mod3) and the module in the file
newsub1.o to example.lib any of the following commands could be entered:

```
LIBUTIL -oa example.lib sub2 newlib.lib newsub1
LIBUTIL -a -o example.lib sub2.o newlib.lib newsub1.o
```

After LIBUTIL is done, there will be a new example.lib file, and it will contain the following modules, in the order specified: sub1, sub2, sub3, mod1, mod2, mod3, and newsub1. The sub2 module in the new example.lib is the same as that in sub2.o.

4. Let example.lib be a library containing the modules sub1, sub2, and sub3. The following submit file, when
activated, will cause LIBUTIL to replace module sub2 with the module in file sub2.o, and append the modules in the library newlib.lib (which are mod1, mod2, and mod3), and the modules in the files newsub1.o, newsub2.o, newsub3.o, newsub4.o, newsub5.o, newsub6.o, and newsub7.o:

```
xsub
LIBUTIL -oa example.lib newsub1.o . newsub7 sub2
newsub2 newsub3 newsub4
newlib.lib newsub5
newsub6
    .
```

After LIBUTIL is done, there will be a new example.lib, containing the following modules, in the specified order: sub1, sub2, sub3, newsub1, newsub2, newsub3, newsub4, mod1, mod2, mod3, newsub5, newsub6, newsub7. The module sub2 will be a copy of that in the file sub2.o.

I/O Redirection and Buffered I/O

"C" has two basic types of I/O, namely buffered, sometimes called stream I/O, and unbuffered. Unbuffered I/O is discussed in another section. Buffered I/O tends to be less efficient than unbuffered I/O, but is easier to use.

There are three standard files in Aztec C II: stdin, stdout, and stderr. When a program is started these three files are opened automaticaly and file pointers are provided for them. The getchar and scanf functions read from the stdin file. The putchar and printf functions output to the stdout file. Run time error messages are directed to stderr.

The default device for stdin, stdout, and stderr is "CON:". The destination for stdin and stdout can be "redirected" to a disk file or another device. To redirect stdin, specify on the command line a "<" followed by the file name or device, for example:

```
myprog parm1 parm2 < input.fil
```

When "myprog" executes, all getchar requests and scanf requests will read from file input.fil.

To redirect stdout, specify on the command line a ">" followed by the file name or device, for example:

```
myprog parm1 parm2 > prn:
```

When "myprog" executes, all output requests to putchar and printf will be directed to the printer device PRN:.

"stdin" and "stdout" can be used just as any other file pointer. Any I/O performed with these file pointers will be redirected if redirection was requested.

I/O can be redirected to any file or the devices:

```
LST:
PRN:
PUN:
RDR:
```

The above devices can be specified as the "file name" to fopen and open. Any I/O to the returned file pointer (fp) or file descriptor (fd) will be directed to the specified device. "CON:" can also be specified as a device to fopen and open. For example:

```
#include "libc.h"
main()
{
char c;
FILE *fl;
```

```
            fl=fopen("lst:","w");
            fputs("this is going to the list device LST:\n",fl);
        }
```

There are a number of library routines for buffered I/O. The reader is directed to the LIBRARY section of this manual and chapter 7 of The C Programming Language for more information.

MANX 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 is only one area of memory into which the overlays are loaded.

If an overlay is in the overlay area of memory when the root requests that another be loaded, the newly specified overlay segment overlays the first.

MANX allows overlays to be "nested"; that is, an overlay at one level can call an overlay at another level. However, an overlay cannot call another overlay which is at the same level.

How to Make an Overlay File

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...)
```

The ovloader function's first parameter must be the name of the overlay file. The parameters p1, p2, 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 name of the function must be changed to 'ovmain'. The overlay doesn't have to know that it's an overlay. An overlay is activated in the normal way, and performs a normal return.

What Files are Created on the Disc?

.com	The file which contains the root has the extent of .com

.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 any overlay that invokes another overlay.

Sample Run:

```
1)    ln -r myroot.o ovloader.o libc.lib math.lib

2)    ln mysub1.o myroot.rsm ovbgn.o libc.lib math.lib

3)    ln mysub2.o myroot.rsm ovbgn.o libc.lib math.lib
```

In the first step, the executable file myroot.com is created. The file myroot.rsm is also created, for use in steps 2 and 3.

In step 2, an overlay will be created and placed in the file mysub1.ovr. The file myroot.rsm, which was created during step 1, tells the linker that an overlay is to be created. The -r option is not specified on the second link edit since mysub1 does not invoke another overlay.

In step 3, another overlay is created and placed in the file mysub2.ovr.

The overlays in mysub1.ovr and mysub2.ovr run in the same memory space.

```
     address

      x'100'    +------------------------------+
                |         base module          |
      x'9F0'    +------------------------------+
                |           module 1           |
      x'1C20'   +------------------------------+
                |           module 2           |
                +------------------------------+
```

Figure 1

A single binary image with 3 segments

```
                 +------------------------------+    x'100'
                 |   base "root" module         |
                 +------------------------------+
  x'9F0'              |                   |                x'9F0'
+------------------------------+  +------------------------------+
|           module 1           |  |           module 2           |
+------------------------------+  +------------------------------+
```

Figure 2

Layout of the Program in Figure 1 as an Overlay

Figure 1 shows a program that can be logicaly divided into three segments as it would look if run as a single module. 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.

Programmer Information

The root causes an overlay to be loaded into memory and control to be passed to it by calling the MANX-supplied function "ovloader", which must reside in the root segment. The parameters to ovloader are a character string, giving the name of the overlay to be loaded, followed by the parameters which are to be passed to the overlay. "ovloader" is the same type as the overlays.

When the overlay is loaded, control passes to the MANX-supplied function "ovbgn", which must be the first function of the overlay. In turn, ovbgn transfers control to the function "ovmain" in the overlay. ovmain is passed the parameters which were passed to ovloader.

When ovmain completes its processing, it simply returns. Control then passes back to the root segment, at the instruction in the user's program following the one that called ovloader. The value returned by ovloader is that returned by ovmain.

Overlays can be nested; that is an overlay can call another overlay, as long as they aren't both on the same 'level'. Also, an overlay can access any global functions and variables which are defined within itself or within the calling segment.

There are three caveats for overlay usage. First, the root segment must have a call to the function 'settop' before any files are opened or any overlays are called. Settop must be passed a value greater than or equal to the size of the largest overlay, or the longest 'thread' of overlays, if overlays are nested. The size of an overlay is displayed on the console, in hex, when it is linked. For example, if your program uses 3 overlays, and the linker says their sizes are 125ah, 236h, and 837h, then the program should make the following call: settop(0x125a). The parameter to settop in this case could be larger, if desired.

Second, if buffered I/O is used in an overlay, then at least one buffered I/O call must exist in the root. This can be accomplished, for example, by calling the 'printf' function in the root. It isn't necessary that printf actually be called, only that it be present.

Thirdr, if an overlay performs floating point or long integer operations, the root segment must also. It isn't necessary that these operations in the root be executed, only that they exist in the root.

Two examples follow. The first demonstrates overlay usage when overlays are not 'nested'. The second demonstrates nested overlays.

Example

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 as parameter 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:

```
     #define MAXOVLY 0x1000
     main() {
          int a;

          settop(MAXOVLY);
          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 root.o ovloader.o libc.lib
     ln ovly1.o ovbgn.o root.rsm libc.lib
     ln ovly2.o ovbgn.o root.rsm libc.lib
```

When the segments are generated and the com file 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. ovly1 calls ovly2. ovly2 just returns.

Here is the root:

```
     #define MAXOVLY 0x1000
     main()
     {
          settop(MAXOVLY);
          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 libc.lib
     ln -r ovly1.o ovbgn.o ovloader.o root.rsm libc.lib
     ln ovly2.o ovbgn.o ovly1.rsm libc.lib
```

When executed, the following messages appear on the console:

```
     in ovly1
     in ovly2
```

Creating a root segment and overlay segments

To create a root segment and one or more overlay segments, the MANX linker "ln" must be run several times. Each execution creates one segment and places it in a separate disk file. The first execution must create the root segment. This execution also creates a file containing a symbol table, which must be specified during the subsequent executions of ln which create the overlay segments.

When creating a segment which calls an overlay, the option '-r' must be specified; this causes ln to generate a symbol table for use in linking the called overlays. 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 a segment which is called by another segment, 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 a segment is both called by another segment and calls another segment, the command line to the linker must specify both the '.rsm' file of the calling segment and the '-r' option.

Data Formats

1. character

Characters are 8 bit ASCII.

Strings are terminated by a NULL (X'00').

For computation characters are promoted to integers with a value range from 0 to 255.

2. pointer

Pointers are two bytes (16 bits) long. The internal representation of the address F0AB stored in location 100 would be:

location	contents in hex format
100	AB
101	F0

3. int, short

Integers are two bytes long. A negative value is stored in two's compliment format. A -2 stored at location 100 would look like:

location	contents in hex format
100	FE
101	FF

4. long

Long integers occupy four bytes. Negative values are stored in two's complement representation. Longs are stored sequentially with the least significant byte stored at the lowest memory addres and the most significant byte at the highest memory address.

5. float and double

Floating point numbers are stored as 32 bits, doubles are stored as 64 bits. The first bit is a sign bit. The next 7 bits are the exponent in excess 64 notation. The base for the exponent is 256. The remaining bytes are the fractional data stored in byte normalized format. A zero is a special case and is all 0 bits. The hexadecimal representation for a float with a value of 1.0 is:

```
                    41 01 00 00
```

A 255.0 would be:

```
                    41 FF 00 00
```

A 256.0 would be:

```
                    42 01 00 00
```

A. Imbedded Assembler Source

Assembly language statements can be imbedded in a "C" program between an "#ASM" and "#ENDASM" statement. Both statements must begin in column one. No assumptions should be made concerning the contents of registers. The environment should be preserved and restored. Caution should be used in writing code that depends on the current code generating techniques of the compiler. There is no guarantee that future releases will generate the same or similar patterns.

B. Assembler Subroutines

INTERFACING ASSEMBLY LANGUAGE ROUTINES WITH "C" PROGRAMS

The calling conventions used by the Aztec C II compiler are very simple. The arguments to a function are pushed onto the stack in reverse order, i.e. the first argument is pushed last and the last argument is pushed first. The function is then called using the 8080 CALL instruction. When the function returns, the arguments are removed from the stack. A function is required to return with the arguments still on the stack unless something is pushed back in place of them. Registers BC, IX, and IY must be preserved by routines called from C. The function's return value should be in HL and the Z flag set according to the value in HL. For examples of assembly code called by "C" programs refer to the string.asm and toupper.asm files supplied with the package.

Example:

```
; Copyright (C) 1981  Thomas Fenwick
        public isupper_
isupper_:
        lxi h,2     ; hl := stack pointer + 2 (arguement address)
        dad sp
        mov a,m     ; load argument into accumulator via hl
        cpi 'A'
        jc  false
        cpi 'Z'+1
        jnc false
true:
        lxi h,1
        mov a,l
        ora a
        ret
;
        public islower_
islower_:
        lxi h,2
        dad sp
        mov a,m
        cpi 'a'
        jc  false
        cpi 'z'+1
        jc true
false:
        lxi h,0
        mov a,l
        ora a
        ret
```

Aztec C II produces reentrant code that is ROMable. The basic tools for producing ROMable code are provided by the Manx LN linkage editor. With the -B option of the linkage editor, the ROM address for the code can be set. With the -D option of the linkage editor, the RAM address of the variable data can be set. The code and data are written to a single file. The code precedes the data. The linkage editor produces a message showing the code size and the data size. Separating the code from the data is the responsibility of the user.

The user must also rewrite CALLCPM.ASM. This routine sets up the stack and calls croot.c to handle the command line arguments and the settup for I/O redirection. In most cases all that needs to be done is to preserve the environment, set up the stack and call the "C" routine. Exit returns to this routine. In most cases exit processing will involve restoring the entry environment and perhaps passing back data.

If space is tight, the standard library functions can be eliminated. Most library routines can be easily eliminated. The support routines cannot. The compiler generates calls to these routines for standard processing. A basic library can be created by breaking the support modules, like supp8080.asm, into their basic components. If float and long support are not needed the supp8080.asm routines should be sufficient. If this module is not broken up its full size is less than .5K with most of the space taken up by int multiply and divide routines. If these routines are not needed then they can be eliminated. I/O in a "strippped" system would be performed by bios or bdos calls.

Aztec C II Floating Point Support

Aztec C II supports floating point numbers of type float and double. All arithmetic operations (add, subtract, multiply, and divide) can be performed on floating point numbers, and conversions can be made from floating point representation to other other representations and vice versa.

The common conversions are performed automatically, as specified in the K & R text. For example, automatic conversion occurs when a variable of type 'float' is assigned to a variable of type 'int', or when a variable of type 'int' is assigned to a variable of type 'float', or when a 'float' variable is added to an 'int' variable.

Other conversions can be expicitly requested, either by using a 'cast' operator or by calling a function to perform the conversion. For example, if a function expects to be passed a value of type 'int', the (int) cast operator can be used to convert a variable of type 'float' to a value of type 'int', which is then passed to the function. For another example, the function 'atof' can be called to convert a character string to a value of type 'double'.

The following sections provide more detailed information of the floating point system. One section describes the internal representation of floating point numbers and another describes the handling of exceptional conditions by the floating point system.

Floating point exceptions

When a c program requests that a floating point arithmetic operation be performed, a call will be made to functions in the floating point support software. While performing the operation, these functions check for the occurence of the floating point exception conditions; namely, overflow, underflow, and division by zero. On return to the caller, the global integer 'flterr' indicates whether an exception has occurred. If the value of this integer is zero, no error occurred, and the value returned is the computed value of the operation. Otherwise, an error has occurred, and the value returned is arbitrary. Table A lists the possible settings of flterr, and for each setting, the associated value returned and the meaning.

flterr	value returned	meaning
0	computed value	no error has occurred
1	+/- 2.9e-157	underflow
2	+/- 5.2e151	overflow
3	+/- 5.2e151	division by zero

When a floating point exception occurs, in addition to returning an indicator in 'flterr', the floating point support routines will either log an error message to the console or call a user-specified function. The error message logged by the support routines define the type of error that has occurred (overflow, underflow, or division by zero) and the address, in hex, of the instruction in the user's program which follows the call to the support routines.

Following the error-message-logging or user-function-calling, the floating point support routines return to the user's program which called the support routines.

To determine whether to log an error message itself or to call a user's function, the support routines check the first pointer in Sysvec, the global array of function pointers. If it contains zero (which it will, unless the user's program explicitly sets it), the support routines log a message; otherwise, the support routines call the function pointed at by this field.

A user's function for handling floating point exceptions can be written in C. The function can be of any type, since the support routines don't use the value returned by the user's function. The function has two parameters: the first, which is of type 'int', is a code identifying the type of exception which has occurred. 1 indicates underflow, 2 overflow, and 3 division by zero.

The second parameter passed to the user's exception-handling routine is a pointer to the instruction in the user's program which follows the call instruction to the floating point support routines. One way to use this parameter would be to declare it to be of type 'int'. The user's routine could then convert it to a character string for printing in an error message.

Two programs follow. One is a sample routine for handling floating point , followed by exceptions. The routine displays an error message, based on the type of error that has occurred, and returns to the floating point support routines. The other is main(), which sets a pointer to the error-handling routine in the sysvec array.

```
#include "libc.lib"

main() {
     Sysvec[FLT_FAULT] = usertrap;
}

usertrap(errcode,addr)
int errcode,addr;
{
     char buff[4];

     convert(addr,buff); /* convert addr to hex char string in buff */
     switch (errcode) {
          case '1':
               printf("floating point underflow at %s\n",buff);
               break;
          case '2':
               printf("floating point overflow at %s\n",buff);
               break;
          case '3':
               printf("floating point division by zero at %s\n",buff);
               break;
          default:
               printf("invalid code %d passed to usertrap\n",errcode);
               break;
     }
```

Internal representation of floating point numbers

Floats

A variable of type 'float' is repesented internally by a sign flag, a base-256 exponent in excess-64 notation, and a three-character, base-256 fraction. All variables are normalized.

The variable is stored in a sequence of four bytes. The most significant bit of byte 0 contains the sign flag; 0 means it's positive, 1 negative.

The remaining seven bits of byte 0 contain the excess-64 exponent.

Bytes 1,2, and 3 contain the three-character mantissa, with the most significant character in byte 1 and the least in byte 3. The 'decimal point' is to the left of the most significant byte.

As an example, the internal representation of decimal 1.0 is 41 01 00 00.

Doubles

A floating point number of type 'double' is represented internally by a sign flag, a base-256 exponent in excess-64 notation, and a seven-character, base-256 fraction.

The variable is stored in a sequence of eight bytes. The most significant bit of byte 0 contains the sign flag; 0 means positive, 1 negative.

The excess-64 exponent is stored in the remaining seven bits of byte 0.

The seven-character, base-256 mantissa is stored in bytes 1 through 7, with the most significant character in byte 1, and the least in byte 7. The 'decimal point' is to the left of the most significant character.

As an example, (256**3)*(1/256 + 2/256**2) is represented by the following bytes: 43 01 02 00 00 00 00 00.

Floating Point Operations

For accuracy, floating point operations are performed using mantissas which are 16 characters long. Before the value is returned to the user, it is rounded.

Digital Researches SID and ZSID symbolic debuggers can be used with the Aztec C II system. The -T option in the link edit step wil create a symbol table. PIP or some other utility can be used to upper case the symbols for SID if necessary.

Unbuffered I/O

This section describes how a program accesses devices and files using the functions defined in chapter 8 of the K&R text. A program which acccesses devices and files using these functions will also be able to run on a UNIX system.

The basic input/output support functions allow a program to access the console, printer, reader, punch, and the files on any disk. The support functions are:

creat	creates a disk file
unlink	deletes a disk file
rename	renames a disk file
open	prepares a device or file for I/O
close	concludes the I/O operations on a device or file
read	reads data from a device or file
write	writes data to a device or file
posit	positions a disk file to a specific record
lseek	positions a disk file to a specific character

Generally, to access a device or file, a program first must call the "open" function, passing it the name of the device or file and a code indicating the type of operations the program intends to perform. Open returns a "file descriptor" which the program must include in the parameters which are passed to other functions when accessing the device or file. This file descriptor is an integer which is an index into a table, called the "channel table". Each entry in this table is a control block describing a device or file on which the program is performing I/O operations. For more details on the "open" function, see the chapter on the unbuffered i/o functions.

The only exception to the rule requiring the opening of devices and files prior to the issuance of program i/o with them regards the logical devices stdin, stdout, and stderr. When the program first gets control, these logical devices have already been opened by the system; hence, the program can issue i/o calls to them without opening them itself.

Generally,after a program has completed its i/o to a device or file, it must call the "close" function to allow the system to release the control blocks which it has allocated to the device or file. The only exception to this rule is that the logical devices stdin, stdout, and stderr never need be closed.

In the remainder of this section, the details of program i/o to the various devices and disk files are presented.

Console I/O

There are two ways for a program to access the system console using UNIX-compatible i/o functions. One is to issue read and write calls to the "logical devices" stdin, stdout, and/or stderr. These three devices are opened by the Aztec system before a user's program gains control. Thus the user's program can access these devices without performing an initial "open" function on them, and without performing a "close" function on them before terminating. The default condition is for these "logical devices" to all be the system console. However, the operator, when activating the user's program, can specify that the stdin or stdout logical device be associated with another device or a disk file; that is, that the stdin and stdout i/o be "redirected". Thus, if the user's program must communicate with the operator, and can't be assured that the stdin and/or stdout i/o has been redirected, then the program must use the other method of communicating with the console, which is described in this section. For more information on using the UNIX-compatible i/o functions to communicate with the stdin, stdout, and stderr devices, see the appropriate section which follows.

The other method for a program to access the system console is to explicitly open the console, issue read and write function calls to it, and then close it. The open and close calls were described above, so the rest of this section just covers the details of reading and writing to the console.

Console input

To read characters from the system console, a program issues read function calls, passing as parameters the file descriptor which was returned to the program when it opened the console, the address of a character buffer into which characters from the console are to be placed, and a number which specifies the maximum number of characters to be returned to the program. The read function will place characters in the buffer, as described below, and return as its value an integer specifying the number of characters placed in the buffer.

The system maintains an internal 256-character buffer into which it reads console keyboard input. The read function returns characters to the calling program from this buffer. If the internal buffer is empty when a program requests console input, the read function will perform its own read operation to the console, putting the characters obtained in its internal buffer. While the read function's read operation is in progress, the console operator can use the normal CP/M editing characters, such as rub out, backspace, etc. These editing characters do not appear in the internal read buffer. The read function's read operation terminates when the operator depresses the carriage return key, the line feed key, or ctl-z, or when there are 256 characters in the internal buffer. Following the characters in the internal buffer which were input by the user, the read function places a carriage return, line feed sequence.

The read function returns characters to the calling program from the internal buffer. If there are characters in the buffer which haven't yet been passed to the caller, the read function transfers some to the caller's buffer, with the number transfered being either the number requested by the caller, or the number remaining in the internal buffer from the last actual console read operation which haven't been passed to the caller. If the internal buffer is empty when the caller makes a request of the read function, the read function performs an actual console read operation to refill the internal buffer, as described above, and then transfers characters from it to the caller's buffer.

The read function returns to the caller as its value the number of characters placed in the caller's buffer, or zero, if the operator typed ctl-z in response to a console read operation by the read function, or -1 if an error occurred. If an error occurred, the read function also places a code in the global integer errno which defines the error. If no error occurred, read returns without modifying errno. The only symbolic value which read may place in errno is EBADF, in response to an invalid file descriptor from the caller. The integer value of EBADF is defined in the file errno.h, which may be included in the user's program.

Writing to the system console, the line printer, or the punch

To send characters to the system console, the line printer, or the punch device, a program calls the function "write", passing it as parameters the file descriptor which was passed to it by the function "open" when it opened the device, the address of a buffer containing characters to be sent, and an integer specifying the number of characters to be sent. The write function sends the characters directly to the device and returns as its value the number of characters sent. If the write function encounters a carriage return character in the caller's buffer, it sends it to the device, then sends a line feed character, then continues with the next character in the caller's buffer.

If the write function detects an error, it returns -1 as its value and places an error code in the global integer errno. If an error was not detected, errno is not modified. The only symbolic value which write may place in errno is EBADF, signifying that an invalid file descriptor was passed to write. The file errno.h defines the integer value of EBADF.

Reading from the "reader" device

A program gets characters from the "reader" device by calling the "read" function, passing it as parameters the file descriptor which was passed to it by open when it opened the reader device, the address of a buffer into which characters from the device are to be placed, and an integer specifying the number of character to be read.

The read function reads characters directly into the caller's buffer. The operation continues until "read" reads the number of characters specified by the caller. It then returns as its value the number of characters read.

If read detects an error, it returns as its value -1, and sets a code in the global integer errno.h. If no error was detected, errno.h is not modified. The only symbolic value which read may set in errno is EBADF; this means that an invalid file descriptor was passed to read. The file errno.h, which can be included in the user's program, defines the integer value of EBADF.

UNIX-compatible I/O to the stdin, stdout, and stderr devices

As was mentioned in the section on console i/o, when a user's program is activated, three "logical devices" are always open; these are called "stdin", "stdout", and "stderr". By default, these are associated with the system console; however, the operator can specify, when activating the program, that read operations directed to stdin and write operations directed to stdout be redirected to an operator-specified device or disk file. The user's program needn't be aware of the actual device associated with stdin, stdout, or stderr; it simply issues read and write function calls as it would to the system console.

If the user's program is to communicate with stdin and stdout where the possibility exists that either or both of them are a device, such as the console, then the user's program should restrict itself to just issuing read and write function calls to these logical devices. However, if the operator always redirects the stdin or stdout i/o to a disk file, then the program can access the redirected device as it would a normal disk file. That is, it can reposition the "current position" of the logical device using the "posit" and/or "lseek" function calls. These calls are described below, in the section on file i/o.

When accessing any device or file, including stdin, stdout, or stderr, the user's program must include a "file descriptor" with the function call parameters which identifies the device with which the user's program wants to communicate. In the case of devices and files other than stdin, stdout, and stderr, the file descriptor is that which the open function returned to the user's program when it opened the device or file. Since the user's program doesn't itself open the stdin, stdout, and stderr logical devices, there has to be another way for it to determine the file descriptors to use when commincating with these devices. The way is this: to communicate with stdin, use a file descriptor having value 0; for stdout, use 1; and for stderr, use 2.

File I/O

When communicating with disk files, in addition to the open and close function calls, which were described above, and the read, write, posit, and lseek function calls, which are described below, there are three other function calls which can be made: creat, to create a non-existant file, or to truncate an existing file so that it doesn't contain anything; unlink, to erase a disk file; and rename, to rename a disk file. These function are described in chapter VI.

Programs call the functions read and write to transmit characters between the program and a disk file. The transfer begins at the "current position" of the file and proceeds until the number of characters specified by the calling program have been transfered.

The current position of a file can be manipulated in various ways by a program, allowing the program to access the file both sequentially and randomly. To read a file sequentially from the beginning of the file, the program simply issues repeated read requests. After each read operation, the current position of the file is set to the character following the last one returned to the calling program. Similarly, to write a file sequentially from the beginning of the file, the program issues repeated write requests. After each write operation, the current position of the file is set to the character following the last one written.

Two additional functions, "lseek" and "posit", are provided to allow programs to access files randomly. lseek sets the current position of a file to a specified character location. posit sets the current position to a specified record. The program can then issue read and/or write requests to transfer data beginning at the new current position. If UNIX compatibility is a requirement, don't use the function "posit" - it's not supported by UNIX.

To perform a sequential update of a file, a program would repeatedly perform the following sequence: read in a buffer's worth of data; update the buffer; reset the current position in the file to the location before the read operation; and finally, write the buffer back to the file. The sequence for updating a file randomly would be the same, except that the program would explicitly set the current position of the file before each read operation.

User Submitted Software

Some user submitted software is distributed with this system. The convention for naming the file extension for user submitted software is as follows:

.ual	- assembler source for a library routine
.ucl	- "C" source for a library routine
.uau	- assembler source for a utility routine
.ucu	- "C" source for a utility routine

Included in the user routines are an alloc and free function plus an in port and out port function.

Anyone wishing to share software is welcome to do so. Each routine should have:

- a copyright notice
- a statement granting the free use of the software
- documentation describing the function and use of the software