Aztec C86 User Manual

Release 1.05i

June 83

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

INTRODUCTION

Welcome to the growing number of Aztec C86 users. This manual will describe the use of the various components of the Aztec C86 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 C86 can be conveniently used in conjunction with the K & R text for learning the "C" language. Aztec C86 is a complete implementation of the K & R standard "C". The K & R book is an essential part of the Aztec C86 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 C86 System

The Aztec C86 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 the MSDOS/PCDOS assembler, MASM, and Digital Research's SID86 debugging system.

1.4 Brief System Overview

The Aztec C86 compiler is a complete implementation of UNIX version 7 "C", with the exception of the bit field datatype. The compiler produces assembly language source code which can be read by the Manx AS86 assembler or the PCDOS/MSDOS assembler, MASM. Programs generated by the compiler have a physical code segment and a physical data segment, each of which can be up to 64 K bytes long.

The Manx AS86 relocating assembler accepts the same language as the PCDOS/MSDOS assembler and the Intel AS86 assembler. It has support for codemacros, but not for macros. 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 C86 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 C86 runs on any PCDOS, MSDOS, or CP/M-86 system with at least 128K of memory and two disk drives. There are no special terminal requirements for Aztec C86 other than the ability to produce upper and lower case and the special characters:

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

1.6 Cross Compilers

A UNIX/PDP11 cross compiler is available for Aztec C86. The output of the compiler, assembler, or linker can be downloaded to the target machine.

1.7 Portability

Code written for Aztec C86 can be compiled with Aztec C II, the Manx 8080 and Z80 C compiler, and with Aztec C65, the Apple DOS 3.3 "C" compiler.

SOFTWARE LICENSE

Aztec C86, Manx AS86, and Manx LN are licensed software products. Manx Software Systems reserves all distribution rights to these products. Use of these products is prohibited without a valid license agreement. The license agreement is provided with each package. Before using any of these products the license agreement must be signed and mailed to:

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

The license agreement limits use of these products to one machine and explicitly limits duplication of the products to no more than two copies whose sole purpose will be for backup. Any uses of these products that might lead to the creation of or distribution of unauthorized copies of these products will be a breach of the licensing agreement and Manx Software Systems will excercise its right to reclaim the original and any and all copies derived in whole or in part from first or later generations and to pursue any appropriate legal actions.

Software that is developed with Aztec C86, Manx AS86, or Manx LN can be run on machines that are not licensed for these products as long as no part of the Aztec C86 software, libraries, supporting files, or documentation is distributed with or required by the software. In the latter case a licensed copy of the appropriate Aztec C software is required for each machine utilizing the software. There is no licensing required for executable modules that include library routines. The only restriction is that neither the source, the libraries themselves, or the relocatable object of the library routines can be distributed.

COPYRIGHT

DISCLAIMER

Manx Software Systems makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Manx Software Systems reserves the right to revise this publication and to make changes from time to time in

the content hereof without obligation of Manx Software Systems to notify any person of such revision or changes.

TRADEMARKS

Aztec C86, Manx AS86, and Manx LN are trademarks of Manx Software Systems. CPM86 and SID86 are trademarks of Digital Research. MSDOS, MASM, and LINK are trademarks of Microsoft. UNIX is a trademark of Bell Laboratories.

CONTENTS

INSTALLATION

To begin using Aztec C86, we recommend that you first create a "working disk" by copying the files listed below from the distribution disk onto the working disk.We recommend that you then compile, assemble, link, and execute the supplied sample program, EXMPL.C. This procedure is described in chapter 2. To execute the program after it has been created, type:

EXMPL

The program will display:

enter your name

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

Your Aztec C86 system is now installed and ready to go.

A. INSTALLING AZTEC C86 FOR USE WITH MSDOS OR PCDOS

When your operating system is MSDOS or PCDOS, Aztec C86 can be used with either the Manx assembler and linker or with the MSDOS/PCDOS assembler (MASM) and linker (LINK). When the Manx assembler and linker are to be used, the following files should be copied onto your working disk:

C86.EXE
AS86.EXE
LN.EXE
LIBC.LIB
MATH.LIB
LIBC.H

When the MSDOS/PCDOS assembler and linker are to be used, the following files should be copied onto the working disk:

C86.EXE
LIBCMS.LIB
MATHMS.LIB
LIBC.H

B. INSTALLING AZTEC C86 FOR USE WITH CPM86

A CPM86 working disk is created by copying the following files from the distribution disk to the working disk:

C86.CMD
AS86.CMD
LN.CMD
LIBC.LIB

MATH.LIB
LIBC.H

OVERVIEW

This chapter describes the basic procedure for generating an executable version of a C program. Section A describes the procedure for using Aztec C86 with MSDOS/PCDOS and the Manx assembler and linker. Section B describes the procedure when using Aztec C86 with MSDOS or PCDOS and their assembler and linker. Section C describes the procedure when using CP/M-86. Section D describes the creation of object file libraries.

The use of the compiler, assembler, linker, and object file librarian are described in more detail in subsequent chapters.

A. Using Aztec C86 with MSDOS/PCDOS and the Manx assembler and linker

The following commands will produce an executable file, EXMPL.EXE, from the file EXMPL.C, which contains a C source program:

```
C86 exmpl.c
AS86 exmpl.asm
LN exmpl.o libc.lib
```

This procedure is depicted in figure 1.1

The first command activates the compiler, C86, which compiles the program in exmpl.c and writes the assembly language source to an intermediate file, exmpl.asm.

The second command activates the Manx assembler, AS86, which assembles the code in the intermediate file and writes the resulting relocatable object code to the file exmpl.o.

The third command activates the Manx linker, LN, which links exmpl.o, getting any needed modules from the object library, libc.lib, and writes the executable program to the file exmpl.exe.

If the program in exmpl.c performed floating point operations, the command to link it would be

```
LN exmpl.o math.lib libc.lib
```


Figure 1.1 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.

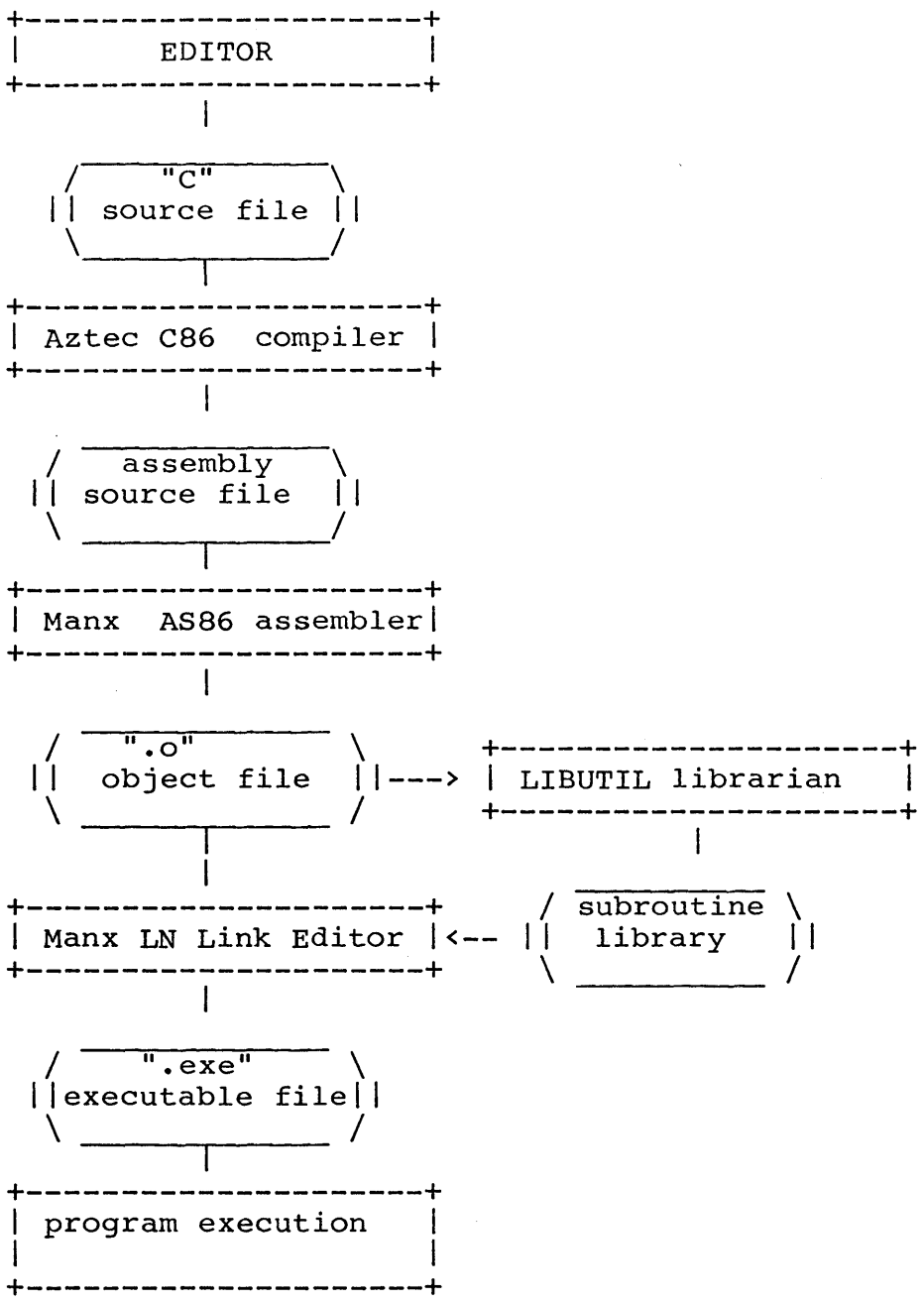

Figure 1.1: Developing "C" Programs with Aztec C86 under MSDOS or PCDOS and using the Manx assembler and linker

B. Using Aztec C86 with MSDOS/PCDOS and their assembler and linker

The following commands can be entered to produce an executable file, exmpl.exe from the C-language file exmpl.c, when using the MSDOS/PCDOS assembler and linker:

```
C86 -M exmpl.c

MASM exmpl;

LINK exmpl,,,libcms
```

This procedure is depicted in figure 2.2.

The first command activates the Aztec compiler, C86, which compiles the C program in exmpl.c and writes the resultant assembly language source to the file exmpl.asm.

The second command activates the MSDOS/PCDOS assembler MASM, which assembles the code in exmpl.asm and writes the resulting object code to the file exmpl.o.

The third command activates the MSDOS/PCDOS linker LINK, which links exmpl.o, getting any needed modules form the object library, libcms.lib, and writes the executable program to the file exmpl.exe.

If the exmpl.c program performed floating point operations, the command to link it would be:

```
LINK exmpl,,,mathms.lib+libcms.lib
```


Figure 2.2 depicts the basic steps for producing a binary image of a "C" program when using MSDOS or PCDOS and their assembler and linker. It also indicates the path for producing and using run time subroutine libraries. The process depicted is fairly basic.

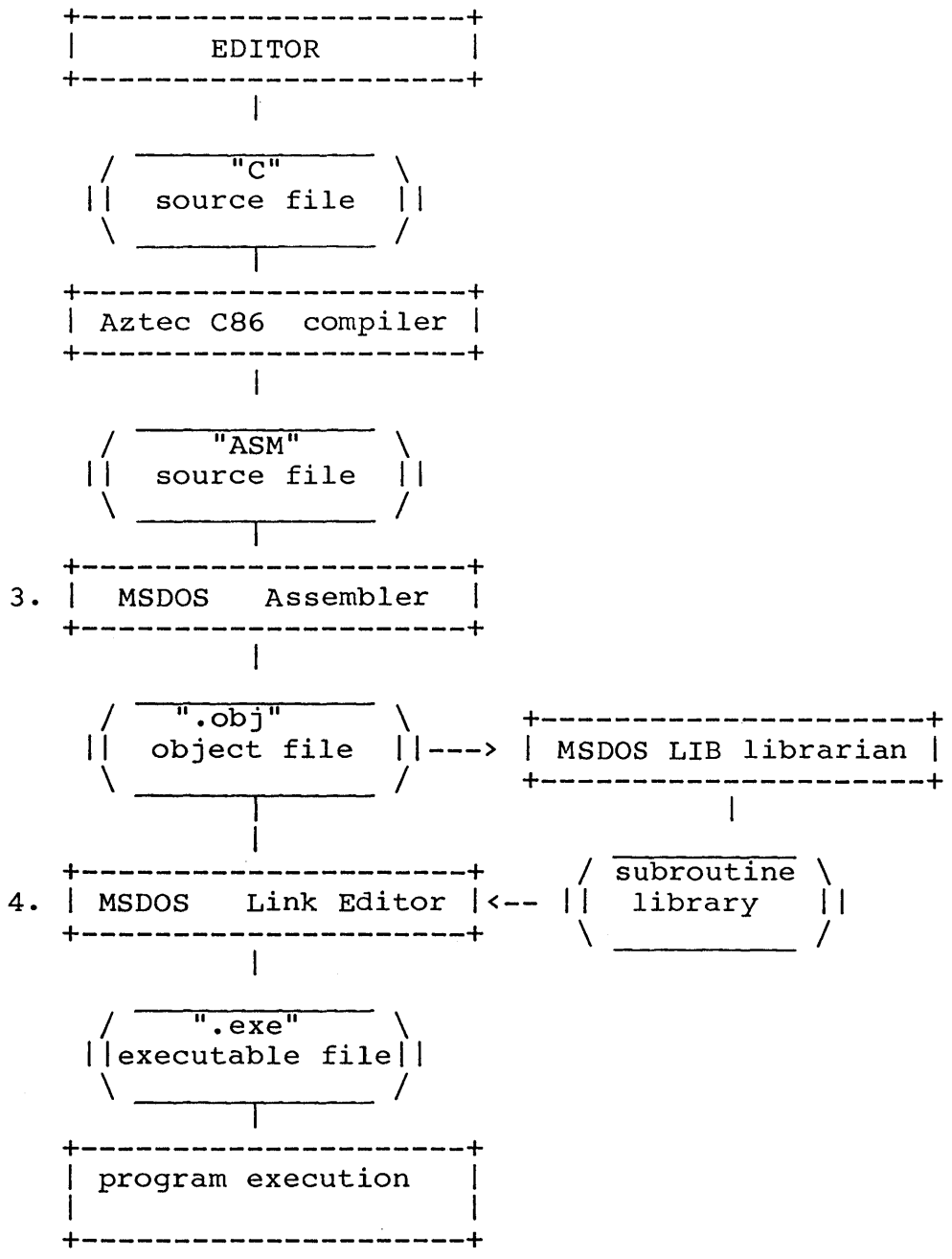

Figure 2.2: Developing "C" Programs with Aztec C86 under MSDOS or PCDOS, and using their assembler and linker

C. Using Aztec C86 with CP/M-86

The following commands can be entered to produce an executable file, exmpl.cmd, from exmpl.c:

C86 exmpl.c

LN exmpl.o libc.lib

The first command activates the compiler, C86, which compiles the C program in exmpl.c and writes the resulting assembly language source to the intermediate file $tmp.$$$. C86 then activates AS86, which assembles the code in the intermediate file and writes the resulting object code to the file exmpl.o.

The second command activates the Manx linker, LN, which links exmpl.o, getting any needed modules from the object library, libc.lib, and writes the executable program to exmpl.cmd.

If the exmpl program performed floating point operations, the link command would be:

LN exmpl.o math.lib libc.lib

D. Object libraries

The Manx linker, LN, can link any number of object files together. One way to do this is to explicitly tell LN of each object module which is to be linked together. Another way is to place commonly used modules in an object file library, using the Aztec object file librarian, LIBUTIL. When a program is linked which requires modules which are in the library, just include the object file library in the list of files passed to LN. LN will search the library, and automatically copy modules from it which are needed. Only the needed modules will be copied. Any number of object libraries can be searched by LN.

Aztec C86 for MSDOS and PCDOS comes with four object libraries:

LIBC.LIB is used when linking any program using LN.

MATH.LIB is used when linking any program using LN which performs floating point operations.

LIBCMS.LIB is used when linking any program using the MSDOS or PCDOS assembler, LINK.

MATHMS.LIB is used when linking any program using LINK which performs floating point operations.

Aztec C86 for CP/M-86 comes with two object libraries:

LIBC.LIB is used when linking any program using LN.

MATH.LIB is used when linking any program, using LN, which performs floating point operations.

C86 COMPILER

The Aztec C86 compiler uses the book The C Programming Language, by Brian W. Kernighan and Dennis M. Ritchie, for a programmer's reference manual. This book can be also be used as a tutorial, for learning C.

A. Operating Instructions

C86 is activated by entering:

C86 <command tail>

where <command tail> specifies the name of the file containing the C program, and "dash options"; that is, optional flags and parameters which override default compiler assumptions. The source file name and each of the dash options are separated by spaces.

All dash options begin with a dash (-), immediately followed by a letter which defines the option. After that, the syntax differs for the various dash options. For some, that's all there is. For others, a number follows which is the value of the option. In another case, the next field is a file name. The dash options are described below.

The compiler generates assembly language code and sends it to a disk file. There are some differences in it's operation when using MSDOS/PCDOS and when using CP/M-86. Under MSDOS and PCDOS, the assembly language code by default is sent to the file whose name is the same as that of the C language source file, with the extent changed to '.asm'. This default file name can be overriden using the '-O' option, described below. For example, to compile the program in foo.c and send the assembler source code to foo.asm when using PCDOS, enter:

C86 foo.c

Under CP/M-86, by default the assembly language source code is sent to an intermediate file, named '$tmp.$$$', and the MANX AS86 assembler is activated. The assembler generates object code, sending it to a file whose name is the same as that of the C language source file, with extent '.o'. This default name for the object file can be overriden using the '-O' option, described below. For example, to compile and assemble the program in mycprog.c and send the object code to mycprog.o, when using CP/M-86, enter:

C86 exmpl.c

Also under CP/M-86, the default chaining of the assembler to the compiler can be overriden using the '-A' option. In this case,

the compiler will stop after compilation, and won't invoke the assembler. The assembler source by default will be sent to the file whose name is derived from the C source file name by changing the extent to 'a86'. This default can be overriden using the '-O' option.

B. Compiler options

Under MSDOS or PCDOS, the dash option specifiers must be entered in upper case. With CP/M-86, they can be entered in either upper or lower case, since CP/M-86 translates the command tail to upper case.

1. '-O' option

The '-O' option allows the user to explicitly select the output file of the compiler (or the assembler, as described above). The syntax is '-O <output filename>', where there is a space between '-O' and <filename>. For example, under PCDOS, to compile exmpl.c on the default drive and send the output to exmpl.asm on the default drive, just enter:

C86 exmpl.c

Still under PCDOS, to compile exmpl.c and send the assembly language output to b:exmpl.asm, enter:

C86 exmpl.c -O b:exmpl.asm

or

C86 -O b:exmpl.asm exmpl.c

If you're running with CP/M-86, to compile and assemble exmpl.c, which is on the default drive, and send the object code to exmpl.o on the default drive, enter:

C86 exmpl.c

Still under CP/M-86, to compile and assemble exmpl.c and send the object code to b:exmpl.o, enter:

C86 exmpl.c -O b:exmpl.o

or

C86 -O b:exmpl.o86 exmpl.c

Still under CP/M-86, to compile exmpl.c, send the assembly language output to exmpl.a86, and not activate the assembler automatically, enter:

C86 exmpl.c -A

Still under CP/M-86, to compile exmpl.c, send the assembly language output to b:exmpl.a86, and not activate the assembler automatically, enter:

```
C86 exmpl.c -A -O b:exmpl.a86
```

2. '-M' option

This option causes the compiler to generate assembler source for the Microsoft MASM assembler. For example, to compile exmpl.c and then assemble the resulting assembler source file, exmpl.asm, enter:

```
C86 -M exmpl.c
MASM exmpl;
```

3. '-T' option

This option causes the compiler to copy the C source statements to the assembly language source file as comments. The C comment in the assembly language file is followed by the assembly language code generated from it. If not specified, the compiler doesn't copy C source statements to the assembly language file.

4. '-E' option

Specifies the number of entries in the expression work table. The default size is 120 entries. Each entry uses 14 bytes. If the compiler terminates with error 36, the expression work table has overflowed.

The size of the table immediately follows '-E', with no intervening spaces. For example, to compile foo.c, with an expression work table of 300 entries, enter:

```
C86 foo.c -E300
```

5. '-X' option

This option specifies the size, in bytes, of the macro table. Each '#define' in a C program creates an entry in this table. The default table size is 2000 bytes. If the compiler aborts with error 59, you need a larger macro table.

The size of the macro table immediately follows the '-X', with no intervening spaces. For example, to compile myprog.c with a macro table of 3000 bytes, enter:

```
C86 -X3000 myprog.c
```


6. '-Y' option

Specifies the maximum number of outstanding cases allowed in a switch statement. Default size: 100 outstanding cases. If the compiler terminates with error 76, you need to recompile with more outstanding cases. For example, the following code segment will use four, not five, 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 size of the case table immediately follows '-Y'. For example, to compile exmpl.c to allow 300 outstanding case statements, enter:

```
C86 exmpl.c -Y300
```

7. '-Z' option

This option specifies the size, in bytes, of the string table. The table defaults to 2000 bytes. If the compiler terminates with error 2, the program requires a larger string table.

The size of the table immediately follows the '-Z', with no intervening spaces. For example, to compile foo.c with a 3000 byte string table, enter:

C86 -Z3000 foo.c

8. '-S' option

By default, Aztec C86 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 strcture

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.

9. '-L' option

This option specifies the number of entries allowed in the local symbol table. It defaults to 40 entries. Each entry requires 26 bytes. If the compiler aborts with the message 'local symbol table full', the program being compiled requires a larger local symbol table.

The number of entries immediately follows the '-L', with no intervening spaces. For example, to compile exmpl.c with a local symbol table having 100 entries, enter:

C86 -Ll00 exmpl.c

10. '-D' option

This option allows a symbol or macro to be defined to the compiler as if it had been a #define statement in a program.

The syntax is -Dstring1[=string2], where string1 is the symbol or macro being defined and string2 is the value of the symbol. the brackets surrounding =string2 mean that =string2 is optional. If =string2 isn't entered, the value of the symbol defaults to 1. No spaces are allowed between the first character of string1 and the last character of string2.

For example, to compile exmpl.c and to define the symbol DEBUG to the program from the command line, and give it a value of 1, enter:

C86 exmpl.c -DDEBUG

To compile exmpl.c and define the macro x(y) = 10*y, enter:

C86 exmpl.c -Dx(y)=10*y

If the compiler is running under CP/M-86, all references within the program being compiled to a command line-defined symbol or macro must be in upper case, since CP/M-86 translates the command line to upper case before activating C86.

If the compiler is running under PCDOS or MSDOS, a program being compiled to a command line-defined symbol or macro must refer to it exactly as entered on the command line, since PCDOS and MSDOS don't translate the command line to upper case.

On PCDOS and MSDOS, the strings can be either upper or lower case; the program being compiled must refer to the symbol just as

it's entered.

C. Compiler 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

AS86, the MANX 8086 relocating assembler

I. OPERATING INSTRUCTIONS

AS86 is activated by entering on the command line:

AS86 <command tail>

where <command tail> specifies the source file to be assembled and the "dash options", that is, the optional flags and parameters which override default assembler assumptions.

In the command tail, the source file name and each of the dash options are separated by spaces and can be entered in any order.

All dash options begin with the dash character (-) and are immediately followed by a letter which defines the option. After that, the symtax for the various dash options can differ. In some cases, that's all there is to the option. In others, the letter defining the option is followed by a number which gives a value relating to the option. In another case, the next field following the option is a file name. The various dash options are described below.

A. Source File

The source file name can either specify the disk drive containing the file or not. If it's not specified, the assembler assumes the file is on the default drive.

B. Object File

AS86 writes the object code to a file. If the file doesn't exist, A86 will create it; if it does exist, AS86 will erase it and create a new one.

By default, the name of the object file is derived from the source file, is given extent '.o', and is placed on the same drive as the source file. For example, to assemble b:subl.asm and place the object code in b:subl.o, enter:

AS86 b:subl.asm

The default name for the object file can be overriden by specifying the '-O' dash option when the assembler is invoked. In this case, the object file name is in the field following the '-O' option. For example, to assemble b:subl.asm and place the object in a:out.o, enter:

AS86 b:subl.asm -O a:out.o

There must be spaces between '-O' and the object file name.

C. Listing File

AS86 does not currently generate a listing.

D. Dash Options

1. '-S' option: Symbol table size

By default, the symbol table can have 500 entries. To select another value, use the '-S' dash option, where the desired size immediately follows the '-S', with no intervening spaces. For example,

AS86 subl.asm -Sl000

assembles subl.asm using a symbol table having 1000 entries.

2. '-Z' option: String space size

By default, 'string space', the area where symbol names and other character strings are stored, is 200 bytes long. To select another size, use the '-Z' option, where the desired size immediately follows the '-Z', with no intervening spaces. For example,

AS86 -Z1500 subl.asm

assembles subl.asm using a 1500-byte string space.

3. '-C' option: Codemacro table size

By default, the codemacro table can have 50 entries. Use the '-C' dash option to select another value, where the desired size immediately follows '-C' with no intervening spaces. For example,

AS86 -Cl00 subl.asm

assembles subl.asm using a codemacro table having 100 entries.

4. '-V' option: Verbose option

Entering '-V' will cause the assembler to list statistics after it finishes. It will list the number of symbol table entries used, the number of bytes of string space used, and the number of entries in the codemacro table that were used.

II. Programmer Information

The Manx AS86 assembler accepts the same assembly language as does the MSDOS/PCDOS assembler, MASM, and as does the Intel ASM86 assembler, with the exceptions noted below. The Intel assembler is defined in their manual ASM86 Language Reference Manual, order

number 121703-002. The exceptions are:

1. Statement names can be up to 255 characters, and all characters are part of the name (not just the first 32 characters, as in the MSDOS/PCDOS and Intel assemblers).

Global symbols, however, while they can contain up to 255 characters within a single program, are truncated to 8 characters in the object file. Thus, if A program has an entry point named 'A_very_long_label', within the program it is refered to by its full name. Other programs in other files would refer to the entry point using its truncated name, 'A_very_l'.

2. Statement names are case sensitive; ie, the label 'A_label' is different from the label 'a_label'. All other symbols (instructions, register names, operators) aren't.

3. Only two physical segments are allowed: all logical segments (ie, those defined within a program using the 'segment' directive) with class name 'code' go in the code physical segment; all other logical segments go in the data physical segment.

4. In the MSDOS/PCDOS and Intel assemblers, code within a logical segment is contiguous in memory, even if a segment is closed, another opened and closed, and the first reopened. In AS86, the only segments for which this is true is segments whose combinability type is 'common'. Code in other segments is placed in memory in the order encountered in the program. For example, consider this program:

```
datasegl   segment
varl:      db   ?
datasegl   ends

dataseg2   segment
var2:      db   ?
dataseg2   ends

datasegl   segment
var3:      db   ?
datasegl   ends
```

With AS86, varl, var2, and var3 will be located in memory in the order in which they appear in the program; that is, varl, var2, var3. This is not true for segments having combinability type 'common'. If datasegl was of type common, the variables would be grouped in memory in this order: varl, var3, var2.

5. The ASSUME directive has no function with AS86: it's accepted, but nothing is done with it. AS86 and LN assume that the CS segment register points to the physical code segment, and DS, SS, and ES point to the physical data segment.

6. Codemacros are supported by AS86, as defined by the Intel

ASM86 Language Reference Manual (order number 121703-002), and are similar to those of the Digital Research ASM86 assembler (but the DBIT directive of the Digital Research assembler isn't supported; instead, the RECORD directive of the Intel assembler is supported).

7. Floating point instructions are not supported by AS86. You can make AS86 support them by including the codemacros for them in your programs.

8. To include a file in the assembly of a program being assembled, enter

INCLUDE <filename>

in the program being assembled, where <filename> is the name of the file to be included. Included files can be nested five deep.

9. AS86 supports conditional assembly. The statements:

```
IF <condition>
<block>
ENDIF
```

will result in the assembly language statements in <block> being assembled if <condition> has a non-zero value. If <condition> has the value zero, the <block> statements will be ignored.

The statements:

```
IF <condition>
<true block>
ELSE
<false block>
ENDIF
```

will cause <true block> to be assembled if <condition> is non-zero, and <false block>, otherwise.

The IF constructs can be nested.

LINKER

I. Operating Instructions

The Manx link editor, LN, combines object files produced by the Manx AS86 assembler, copies needed modules from object libraries which have been created using LIBUTIL, the Manx object file librarian, and produces an executable file.

To activate the linker, enter

LN <command tail>

where <command tail> is a list of file names and dash options. The command tail fields, that is, file names and dash options, are separated by spaces. A 'dash option' is used to override a default assumption by the linker. The dash options are described below.

The linker can link together any number of object files and search any number of libraries. Object files are included in the order encountered in the command tail, and libraries are searched sequentially in the order of specification.

When LN includes a module from a library which satisfies unresolved references, new unresolved references may be created, due to the external symbols referenced by the library module. The linker will not go back through libraries which have already been searched or through the part of the current library which has already been searched to try and find modules which contain symbols which satisfy the unresolved references. It will press on!

When LN is used with MSDOS or PCDOS, the resultant executable program is written to a file whose name is derived from the first object file name in the command tail by changing the extent to '.exe'. This default assumption can be overriden using the '-O' option, described below.

When LN is used with CPM86, the resultant executable program is written to the file whose name is derived from the first object file in the command tail by changing the extent to .cmd. This default assumption can be overriden using the '-O' option.

Supplied with Aztec C86 are the object libraries libc.lib and math.lib. In most cases, libc.lib must be specified when linking a program. Math.lib must be included when linking a program which performs floating point operations, and must be specified in the LN command tail before libc.lib.

To link a simple program, which doesn't perform floating point, and whose object code is in the file exmpl.o, enter

LN exmpl.o libc.lib

The linker will place the executable code in the file exmpl.exe, when running with PCDOS or MSDOS, and in exmpl.cmd, when running with CP/M-86.

If the above program performed floating point operations, it would be linked by entering:

LN exmpl.o math.lib libc.lib

If the exmpl program called functions whose object code is in sub1.o, sub2.o and sub3.o, it could be linked by entering

LN exmpl.o sub1.o sub2.o sub3.o math.lib libc.lib

If sub1, sub2, and sub3 are commonly used functions, they could be put in an object file library which we'll call sub2.lib, using the Manx program LIBUTIL; then exmpl could be linked by entering

LN exmpl.o subs.lib math.lib libc.lib

Dash Options

1. '-O' option

This option specifies the name of the file to which the executable program generated by LN is sent. If not specified, LN generates the output file name from the name of the first object module in the command tail as described above.

For example, to link exmpl.o and send the output to b:exmpl.exe, enter

LN -O b:exmpl.exe exmpl.o libc.lib

2. '-T' option

This option causes the linker to generate a symbol table which can be read by the Digital Research debugger, SID86. The name of the file to which the symbol table is sent is derived from that of the file containing the executable program by changing its extent to '.sym'.

For example, to link exmpl.o and generate a symbol table in the file exmpl.sym, enter:

LN -T exmpl.o libc.lib

To link exmpl.o, send the output to foo.exe, and send a symbol table to foo.sym, enter:

LN -T -O foo.exe exmpl.o libc.lib

3. '-C' option

This option specifies the starting address for the code portion of the output. If not specified, the starting code address is 3. In bytes 0, 1, and 2 the linker places a jump instruction to $begin, which performs system initialization functions.

The starting code address immediately follows the '-C', with no intervening spaces.

For example, to link exmpl.o, where the code starts at 0X300, enter:

LN -C300 exmpl.o libc.lib

4. '-D' option

This option specifies a starting address for the data and common segments in the data block. It defaults to 0. The address immediately follows '-D' with no intervening spaces.

5. '-F' option

This option causes LN to merge the contents of a file with the command tail. The name of the file follows the '-F', with spaces between the two.

For example, to link exmpl.o, subl.o, sub2.o, sub3.o, and edit math.lib and libc.lib, the following could be entered:

LN -F exmpl.lnk

where exmpl.lnk contains:

```
exmpl.o
subl.o
sub2.o
sub3.o
math.lib
libc.lib
```

6. '-S' option

This option is used to explicitly select the amount of extra space the linker includes in the data segment above the linked data. The size defaults to 4096 bytes.

The size, in hex, immediately follows the '-S' option, with no intervening spaces.

For example, to link exmpl.o and give it 100H extra bytes above the data, enter

LN exmpl.o -S100

II. Programmer's Information

The LN linker can create two physical segments for a program: a code segment and a data segment. Each physical segment can be up to 64 K bytes long. Logical segments from the assembler, ie, those defined by the 'segment' and 'ends' directives in an assembly language program, are grouped into physical segments as follows: logical segments whose combinability type is 'common' are included in the physical data segment. Segments whose class name is 'code' are included in the physical code segment. All other segments are included in the physical data segment.

LIBRARY MAINTENANCE

LIBUTIL

A. SUMMARY

The LIBUTIL LIBrary UTILity is used in order to:

1. create a library
2. append modules (-a)
3. produce an index list (-t)
4. extract modules (-x)
5. replace modules (-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 sub1.o and sub2.o

```
>LIBUTIL -o example.lib sub1.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.

STANDARD LIBRARY FUNCTIONS

A. SUMMARY

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

agetc	(stream)	ASCII version of getc
aputc	(c,stream)	ASCII version of putc
clrerror	(stream)	clear error on stream
fclose	(stream)	closes an I/O stream
feof	(stream)	eof on stream?
ferror	(stream)	error on 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	(stream, pos, mode)	positions stream to pos
ftell	(stream)	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 srtl to str2 at most max
strncpy	(dest, src, max)	string copy at most max characters

4. Utility Routines

alloc	(size)	allocate space
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
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
random	()	random number generator
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.

These functions provide a uniform I/O interface for all programs written in Aztec C86 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"
```

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 and returns it as a single newline character ('\n'). Also, an end of file sequence (control Z) 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).

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) before being flushed.

clrerror

```
clrerror(stream)
FILE * stream;
```

Clears all error conditions on stream.

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.

feof

```
feof(stream)
FILE * stream;
```

Returns 0 if stream is not at EOF; otherwise, it returns 1.

ferror

```
ferror (stream)
FILE * stream;
```

Returns 0 if no error has occurred on the stream; otherwise it returns 1.

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.

Under MSDOS or PCDOS, the system console has the name 'con:'. Other devices have their standard MSDOS/PCDOS names.

Under CPM86, 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.

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 The C Programming Language.

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 Oxff 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). '\n' is appended to the end of the string before sending it.

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, 0L, 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. When using PCDOS or MSDOS, the system console is named

'con:'. Other devices have their standard MSDOS/PCDOS names.

When using CPM86, 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) ();
```



```
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 for MSDOS and PCDOS

```
bdos(ah, dx, cx)
int ah, dx, cx;
```

Calls the bdos, by issuing 'int 21', with register AL set to al, register AH set to 0, DX set to dx and CX set to cx. The value returned in AL is the return value.

bdos for CPM86

```
bdos(cx, dx)
int cx,dx;
```

Calls the bdos, by issuing 'int 224', with register CX set to cx and register DX set to dx. The value returned from the bdos in register AX is returned as the function 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.

random

```
double random()
```

returns a random number in the range 0 to 1.

ERROR PROCESSING

During run time two 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. "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 C86:

```
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:
                        argument value illegal by definition
```

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 C86: 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 device. Under PCDOS and MSDOS, the console is referred to as 'con:'; other devices are called by their standard PCDOS/MSDOS names. Under CP/M-86, devices have these names:

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

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. For example, using CPM86, the following program will send a message to the printer:


```
#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.

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 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 the library functions chapter.

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.

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

The calling conventions used by the Aztec C86 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 a near 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. The segment registers, and registers BP, SI, and DI must be preserved by routines called from C. If the function returns an integer as its value it is returned in register AX; long integers are returned in the primary long register, and doubles in the primary floating point register. 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) 1983  Thomas Fenwick
Cg      group   CODESEG
CODESEG segment word public 'code'
        assume  cs:Cg
        public isupper_
isupper_ proc  near
        mov     bx,sp
        mov     al,2[bx]
        cmp     al,'A'
        jl      false
        cmp     al,'Z'
        jg      false
true:
        sub     ax,ax
        inc     ax
        ret
;
        public islower_
islower_:
        mov     bx,sp
        mov     al,2[bx]
        cmp     al,'a'
        jl      false
        cmp     al,'z'
```

```
        jle     true
false:
        sub     ax,ax
        ret
;
        ret
isupper_  endp
CODESEG ends
        end
```


Data Formats

1. character

Characters are 8 bit ASCII.

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

For computation characters are promoted to signed integers.

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. They are in standard 8087 format.

When a floating point exception occurs, in addition to returning an indicator in 'flterr', the floating point support routines will log an error message to the console. 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 the floating point support routines return to the user's program which called the support routines.

Internal representation of floating point numbers

Floats are in the standard 8087 32-bit format, and doubles are in the standard 8087 64-bit format.

The Digital Research SID86 symbolic debugger can be used with the Aztec C86 system. The -T option in the link edit step will create a symbol table.

