AZTEC C OWNER'S MANUAL

MANX SOFTWARE SYSTEMS

C

C C

C Co

C Com

C Comp

C Compi

C Compil

C Compile

C Compiler

AZTEC C II User Manual

Release 1.06

March 1984

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

Contents

Overview

This manual was written to provide the shortest path between any two points. The table of contents provides a straightforward outline of the text. This section explains in more detail what is discussed in each section of the manual.

If you are new to compilers and the C language, you will want to read section **II**, since it gives a more detailed introduction to the package as a whole.

Each of the subsequent sections explores a new topic. Sections **III - VI** describe the principal programs available with Aztec C and give complete, specific information on their use.

Section **VII** shows which functions are available in the standard libraries, and how they can be called from a C program. The section has a short preface which explains how to use the summaries effectively. For all users, the library section is a handy reference and source of examples.

Miscellaneous topics are treated in section **VIII**. The summary of data formats explains how C data types are handled by the compiler. It is worth reading this section even if you are just learning C, since it offers insight into the differences between the data types available to you. The guide to the assembly language interface demonstrates how you can use your own assembly code with the Aztec compiler. However, it also provides a closer look at function calls in C, at the assembly language level.

Section **IX** describes the extensions which are available for the Aztec C development system. The introduction specifies which programs and features are included in the standard package, and which can be purchased separately.

Section **X** provides a closer look at the C language and this package. It will be most useful if you are just learning the language, as it clarifies the problems that are frequently encountered by beginners-- and non-beginners as well.

The final section lists and explains the error messages which are generated by the Aztec software. Although there is always a new way to produce a given message, this section should start you looking in the right direction for a cause. It is conveniently located in the rear of the manual for easy access.

SECTION II

A Tutorial Introduction

Introduction

The software provided by Manx is comprised of four indispensible tools. They are called the compiler, assembler, linker and librarian. These are generic names. The Manx compiler is known as **Aztec C II** (see two). The assembler, designed with CII in mind, is called **AS**. The Aztec linker is called simply **LN**. Another word for "linker" is "link editor". As its name might imply, LN is what ties together the process of developing a program. The development of larger applications is made easier with the help of a librarian such as LIBUTIL. It will help to manage your files when a program grows very large.

Just what these programs are and how they are used is the subject of the next several sections. However, before you move on to the complete descriptions, you may want to read the more general section which follows. Through it, you will become acquainted with the structure of the package and how to use the tools it makes available.

Getting Started

Manx has sent you one or more diskettes, or floppy disks. Each diskette is labeled with the name of the product, the version number and a fraction indicating which member of the set it is. If the diskette is reversible (a flippy), be sure to note that both sides may be used, in which case each side will have a different label.

The diskettes in your package are not bootable. In order to use them, you will first have to boot CP/M.

Checking the Files

You should take a directory listing of your disks and check the results against the list given in the release document enclosed with your package. If at any time you believe that a Manx file is corrupted or bad, you can test it using the program called CRC.COM. For a given file or for an entire disk, the CRC (cyclic redundancy check) program will compute a unique hexadecimal number. When a file is altered in any way, its CRC number must change also. In order to test the entire disk, simply run the program by typing in "crc" to the CP/M prompt. The general form of the command is:

```
crc filename
```

This will run the CRC check on the given file. A drive can also be specified:

```
crc a:exmpl.c
```

The CRC numbers of the files supplied with the package are listed in the release document. If a different value is generated by the CRC program for a file, the file is "bad". A CRC number remains the same only until the file is changed.

Back up the Disks!

Before going too far with your disks, it is important that you back them up. Backup disks are nothing more than copies of the original disks which insure against accidental erasure of valuable files. The CP/M copy utility is generally adequate for backing up your distribution disks.

Any further considerations are explained in the release document accompanying this manual. The release document should be read over at this point. It is the real introduction to your version of the package. It explains the changes that have been made since the last release as well as any problems reported by our users.

When you have made your backup disks, store the originals away in a safe place-- but not somewhere so safe that you can't find them later on. You should also copy the version number of the package from the label of your distribution disks. This version number may differ from that which appears when a particular program is run.

The Working Disk

It is a good idea to create a working disk, that is, a disk which contains all the files you will need right away. A working disk will eliminate all the extraneous files which appear on the originals. For now, you will need the following files: cc.com, as.com, ln.com, c.lib and exmpl.c. These files can be transferred to a formatted CP/M disk with the utility, PIP.

If these files do not fit onto a single disk, you may want to put ln.com and c.lib on a second disk.

That little extra trouble may prove to be worthwhile in the future, if only for your own peace of mind. Now we can leave the disk swapping behind and see how this package is put together.

The Z80 Compiler

The Aztec C II development system contains two compilers, **cc** and **cz**. While either compiler will run on the Z80 microprocessor, **cz** makes special use of the Z80. On the 8080 chip, only **cc** can be used.

For the purposes of this introduction, either compiler will work just as well on the Z80. The differences between them are described in section III.

If you wish to use the **cz** compiler in the following example, simply copy the file, cz.com, onto your working disk instead of cc.com. Anywhere reference is made to **cc**, read **cz**.

An Example

As depicted in the diagram, there are five steps to developing an executable program. Most of your work is done in steps one and five. The intervening steps are accomplished with the aid of the software in this package.

For now, we will assume you are interested in seeing the example program, exmpl.c, run on your computer. Although the program itself is not very engaging, we hope that compiling it will familiarize you with the Aztec development system.

Compiling the Example

"exmpl.c" is an ordinary text file. Normally, you will have created it with a text editor. It is special only in that its contents are a C language program.

Aztec C II translates this program into an assembly language program. This translation process is called compilation. It is begun by running the compiler with this command:

```
cc exmpl.c
```

If exmpl.c is not on the disk in the default disk drive, you can specify a drive as follows:

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

The same convention is used for running any CP/M program.

CP/M should go to the appropriate disk drive to find the file, CC.COM. That is the compiler. Then the compiler will go searching for the file, exmpl.c. When the compiler is finished, it will leave behind a new file on the disk, EXAMPL.ASM. This too is just a text file, which can be printed out with the CP/M command, **type**, and edited with a text editor. Assembly language programmers can edit this file to "hand optimize" certain parts of the code to make it run faster. Otherwise, though, you should not have to concern yourself with this file directly.

```
           +---------------------+
        1. |       EDITOR        |
           +---------------------+
                     |
              _____________
            /      "C"      \
           |   source file   |
            \ ______________ /
                     |
           +---------------------+
        2. | Aztec C II compiler |
           +---------------------+
                     |
              _____________
            /     "ASM"     \
           |   source file   |
            \ ______________ /
                     |
           +---------------------+
        3. |  MANX AS Assembler  |
           +---------------------+
                     |
              _____________
            /      "O"      \        +---------------------+
           |   object file   |---> | LIBUTIL librarian   |
            \ ______________ /       +---------------------+
                     |                          |
                     |                   ____________
           +---------------------+      / subroutine \
        4. | MANX LN Link Editor |<--  |   library    |
           +---------------------+      \ __________ /
                     |
              _____________
            /     "COM"     \
           | executable file |
            \ ______________ /
                     |
           +---------------------+     +------------------+
        5. | program execution   |<---->|    debugging     |
           |                     |     |   the program    |
           +---------------------+     +------------------+
```

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

The Assembly Step

The next step is to convert this assembly code into what is called object code. This is the job of the assembler. AS was specifically designed to handle the output of the Aztec compiler, so this step should run very smoothly unless: 1. there is not enough space on the disk, or 2. you are including assembly code you have written yourself. If neither of these things are true, everything will run very smoothly.

The assembly is begun with the following command:

```
as exmpl.asm
```

As always, a drive identifier can be given to specify on which disk either of the files are located.

The assembler will leave its output in a file called exmpl.o. The assembler output is known as relocatable object code because it can be relocated anywhere in memory; this is done by the linker. The linker will convert this code from relocatable object format to absolute data, that is, a program which will be loaded and run at a specific address in memory.

Linking the Example

When the assembler is finished, we are left with exmpl.o. The command to the linker is:

```
ln exmpl.o c.lib
```

The file, c.lib, is a library full of object modules similar to exmpl.o. They are relocatable object code produced by compiling and assembling the routines which are available in the library. In the example program, **printf** is a function which comes from this library. During the link step, the linker will search the library for this function and "pull in" the object module in which it is defined. More about libraries later.

Most programs will need to be linked with c.lib, since there are many functions in it which remain invisible to you. Try the following link command:

```
ln exmpl.o
```

The linker will report that several names were undefined. These are needed support functions which your program called without knowing it.

The output of the linker is an executable program file. Here, "exmpl.com" was produced. This program can be run by entering the name of the file minus the extent:

```
exmpl
```

This is the way any CP/M program is run.

So the process of going from source code to an executable program consists of these three steps:

cc exmpl.c	**compile**
as exmpl.asm	**assemble**
ln exmpl.o c.lib	**link**

If your system allows it, you can use a submit file to perform any or all of these steps.

If You Run Out of Disk Space

The only difficulty you may have is running out of disk space. The danger of this occurring varies from system to system. If you are having trouble in this respect, here are a few suggestions:

1. Use two disks. The first disk might contain the compiler and assembler, while the second disk contains the linker and library.

2. If the three programs from Manx and the library all fit on a single disk, then leave all of your software on a second disk, perhaps in drive B. Make B the default drive by entering "B:". Then, for example, your link command would look like this:

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

3. The assembler has an option called -ZAP which will cause it to delete its input file. This keeps assembly files, which are not always useful to you, from cluttering up the disk. So the following command:

```
as -ZAP exmpl.asm
```

will leave "exmpl.o" on the disk but delete "exmpl.asm".

The Compiler

The Compiler

The Aztec C II compiler is implemented according to the language description supplied by Brian W. Kernighan and Dennis M. Ritchie, in The C Programming Language. Where discrepancy or ambiguity is found in that text, reference is made to the implementation of the language under UNIX version 7. The Aztec C manual should bring to light any areas where there may be confusion as regards Aztec C.

Since this manual is not intended as a complete guide to the C language, you may need another text handy to answer questions about proper syntax and usage. Several strong tutorials are available; some are suggested in the appendix. The Kernighan and Ritchie book is generally considered the place to turn to for the final word. You may want to have a copy, whatever other books you might own.

This section will explain how to use the compiler. There are a variety of options which can be specified at compile time. These enhance the flexibility of the system, so that you will eventually want to become familiar with what is available. If you are just trying out some small programs similar to the example program of the last section, not all of the material in this section will be of immediate importance to you.

Running the Compiler

The compiler is invoked by a command of the format:

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

If the filename does not have an extension, the compiler will assume it ends in ".c". It is recommended that C source files have this extension although the compiler will allow a different one, as in "filename.src".

The C source statements found in the given file are translated into assembly language and written to a file. This output file is named "filename.asm" by default. (The assembler will expect the ".asm" extent the same way that the compiler expected the ".c".) An alternate output file can be specified with the "-O" option. For example,

```
cc -o file.a80 file
```

will compile the program in "file.c" and write the assembly language equivalent to "file.a80".

The compiler will append the ".c" only if it doesn't see a period (.) in the filename. So that if you want to name a source file without any extension at all, as in "srcfil", you will have to compile it in this way:

```
          cc srcfil.
```

The period at the end of the filename stops the compiler from tacking on the ".c".

The remaining compiler options have more specialized uses. They are described below.

The 8080 and Z80 Compilers

The Aztec C II development system includes two compilers, **cc** and **cz.** Both support the full C language, as explained above. Both compilers generate 8080 assembler mnemonics.

cc can be used in conjunction with either the 8080 or Z80. In either situation, one register is available for use by a variable through the C language storage class, **register.**

cz can be run on only the Z80, and produces code intended for the Z80. It uses the Z80 index registers, IX and IY, to hold additional register variables, for a total of three. This results in a higher throughput.

To simplify the descriptions in this manual, reference is made only to **cc** throughout. On Z80 systems, the two compilers can be used interchangeably. The assembly output of either compiler can be assembled by the **Manx AS** assembler and linked with the standard libraries, **c.lib** and **m.lib.** A call to either **csave** or **zsave** is made at the entry point of each C function. Both functions are pulled into every linked program.

Compiler Options

Utility Options

-D	Defines a symbol for the preprocessor.
-F	Forces frame allocation to take place in-line rather than through a call to a library function.
-I	Causes search for included files in specified areas.
-M	This option causes the compiler to produce code for the Microsoft assembler (see section XI for details).
-O	Used to specify an alternate output file.
-P	Sends error messages to the printer.
-Q	Converts default automatic variables to statics for efficiency.
-R	This option is a special extension to the compiler which causes it to produce code for the Digital Research assembler (see section XI for details).
-S	Causes search for undefined structure members as described below.
-T	This option will insert the C source statements as comments in the assembly code output. Each source statement appears before the assembly code it generates.
-U	Converts default global variables into externs (except initialized data).

Table Manipulation

-E	Specifies the size of the expression table.
-L	Specifies the size of the local symbol table.
-Y	Specifies the maximum number of outstanding cases allowed in a switch.
-Z	Specifies the size of the table for literal strings.

Utility Options

-D Option

The **-D** option defines a symbol in the same way as the preprocessor directive, **#define**. Its usage is as follows:

```
cc -Dmacro[=text] prog.c
```

For example,

```
cc -DMAXLEN=1000 prog.c
```

is equivalent to inserting the following line at the beginning of the program:

```
#define    MAXLEN    1000
```

Since the **-D** option causes a symbol to be defined for the preprocessor, this can be used in conjunction with the preprocessor directive, **#ifdef**, to selectively include code in a compilation. A common example is code such as the following:

```
#ifdef    DEBUG
          printf("value: %d\n", i);
#endif
```

This debugging code would be included in the compiled source by the following command:

```
cc -dDEBUG program.c
```

When no substitution text is specified, the symbol is defined as the numerical value, one.

This capability is useful when small pieces of code must be altered for different operating environments. Rather than maintaining two copies of such a program, this compile time switch can be used to generate the code needed for a specific environment. For example,

```
#ifdef    APPLE
          appleinit();
#else
          ibminit();
#endif
```

-F Option

The **-F** option causes function entry code to be generated in-line. Normally, every compiled C function begins with a call to a routine in **c.lib**. This option replaces this call with the equivalent code.

This results in a small savings in execution speed every time the compiled function is called. If the function is called repeatedly, the savings can add up to a large difference in the execution time of the program. As a side effect, this option will slightly increase the size of the compiled code.

-I Option

The **-I** option causes the compiler to search in a specified area for files included in the source code.

By default, the compiler will search for included files in the current user and user 0 on the default drive and in user 0 on drive A, if that drive has been logged in (i.e., if drive A has already been accessed).

The **-I** option is used to specify a more extended search. For example, **#include**'d header files might be kept in particular user, such as user 5 on drive A. Then a compile command might be this:

```
cc -i 5/a: program.c
```

The parameter for the option has the form:

[user number]/[drive identifier]

Each user area to be searched requires its own option letter. The -I can be specified up to eight times in a single command.

-P Option

The **-P** option redirects the screen output of the compiler to the printer. This produces a hard copy of the error messages generated during compilation.

-Q Option

-Q is an option which causes the compiler to treat automatic variables as statics. This will essentially convert an automatic "int i" within a function to a "static int i".

This can cause a significant increase in execution speed, since it is much less expensive to address statics than variables on the stack.

As an empirical example, a version of the infamous Eratosthenes' sieve program which ran in 33 seconds was reduced to a run of only 24 seconds, just by specifying the -Q option.

A declaration using the "auto" keyword, such as "auto int i", will not be affected by the -Q option. "auto" forces the variable to remain an automatic.

If a variable is declared as a "register int i", but there are no available registers, the variable defaults to the automatic storage class. So a "register int i" will become a static under the -Q option, if there is not a free register.

Like any other static data, an auto-turned-static is initialized to zero before the program begins.

Note that calling a function recursively may cause problems when the -Q option is used. Consider a function with a "static int i" which increments i and then calls itself:

```
qtest()
{
     static int i;

     if (++i < 100)
          qtest(i);
     return (i);
}
```

The following program will print out "100":

```
main()
{
     printf("%d", qtest());
}
```

If the integer variable in qtest() was not static, the recursive call would have to be "i = qtest(i)". Although qtest() does not seem to save the value returned by the call to itself, the static variable retains its value throughout the nesting.

-U Option

The **-U** option performs a different storage class conversion. It converts **global** variables into **externs.** That is, under -U, "int i" outside any function becomes "extern int i". This is useful in that it allows all global variables to be defined in a single file without having to specify an "extern" with each other declaration.

The universal way of defining a global integer, i, is to have the statement, "int i", in one file and the statement, "extern int i" in all other program files in which the variable is used. The "int i" is a "definition" of the variable since it causes space to be reserved in memory for the variable. The "extern" causes no memory to be reserved; it says, "This variable is defined somewhere else but it is going to be used in this file of the program."

When using the Aztec assembler and linker, the only requirement is that a global variable must be defined at least once. So in this example, it is also possible to have "int i" in every file; the "extern" keyword is not extremely significant in this case. Although there may turn out to be more than one global "int i" in the program, memory will be allocated for just one. This is also the behavior under UNIX.

The situation is slightly different when employing the assembler and linker provided by Microsoft or Digital Research. (These programs can be used only with a compiler which has the extended options, -M and -R.)

When using the Microsoft or DRI assembler and linker, a global variable must be defined exactly once. That is, "extern int i" must appear in every declaration except one, which must be an "int i". This is where the -U option is useful. By specifying it for all but a single source file, you will not have to worry about having too many or not enough externs; the "externs" can be left off entirely since they will be tacked on under the -U option.

A global initialization is immune to the -U option. Hence, "int i = 3;" is unchanged by it. Initializing a global variable to zero will cause it to be ignored by -U. This is one means for forcing a data definition when using this option.

-S Option

The **-S** option is best illustrated by an example:

```
struct atype {
        char a1, a2;
} a;

struct btype {
        char b1, b2;
} b;

a.b1 = 4;
b.c2 = 6;
```

Normally, both of the assignments will cause a compiler error, since "b1" is not a member of "a", and "c2" is not a member of "a". However, under the -S option, the first assignment will be legal and the second will be illegal.

Under -S, the compiler will not generate an error when it notices that "b1" is not a member of "a". Instead, it will proceed to search through all the previously defined structures until it finds the member "b1". The member of structure "b", namely "bl", is taken to be referenced by "a.bl".

The second assignment will generate an error with or without the -S option, since "c2" is not a member of a previously defined structure.

The -S option refers only to previously defined structures.

Table Manipulation

An explanation of the remaining options requires a little background. As the compiler is compiling a C program, it has to keep track of all the symbols in the source code-- mainly variable and function names. It has to remember some information about each variable, such as its data type. All this is stored in symbol tables in memory.

The symbol tables start out with a certain default size. This size is usually sufficient for compiling a moderately sized source file. However, depending on the complexity of the program, the compiler might use up all the entries in a table during compilation. In this case, the compiler will terminate with an appropriate error message.

If this happens, you will need to adjust the sizes of the tables. Usually, this will call for just increasing the size of a single table. The default sizes are given below, along with examples for each option.

The amount of memory available to the compiler is obviously limited. It will read into memory only as much of the source file as it needs in order to generate output. Aside from the work space needed for this task, it maintains the following tables: expression work table, macro/global work table, case table, string table, label table, and local symbol table.

The **label table** holds information on all the labels in the program (a label is the destination of a **goto**). It is fixed at a generous size. If it overflows, the compiler will generate error code 54, and you will have to decrease the number of labels in your program.

The **macro table** is where macros defined with "#define" are remembered. It also contains information about all global symbols. Unlike the other symbol tables, it is self-adjusting, and is never larger than it needs to be. Note that this is different from versions previous to 1.06 of the compiler. This change has made the "-X" option obsolete.

If the Compiler Runs Out of Memory

If the compiler aborts with a message indicating that it ran out of memory, you will have to decrease the size of one or more of the tables. There is no harm in doing so, since the compiler will always complain when a table overflows. Decreasing the size of a table will free up space in memory for the compiler.

As indicated in section XI, the compiler may run out of memory without overflowing any particular table. In this case, a generic "out of memory" message will appear. If the module you are compiling is extremely large, the simplest solution may be to break it into two or more separate modules. However, if the module is of reasonable size, it is possible to decrease the size of a table which is not fully used, thereby freeing up memory for the compilation.

Choosing which table to decrease and by how much is a matter of estimation.

If a particular table overflows, the appropriate error will be generated. It is then necessary to increase the size of that table. Table sizes specified by option letters are used for only a single compilation.

The Expression Table:

This is the area where the "current" expression is handled. It is the compiler's work space as it interprets a line of C code. The various parts of the line are stored here while the statement is being compiled. When the compiler moves on to the next expression, this space is again freed for use.

The default value for -E is 60 entries. Each "entry" in the table consumes 14 bytes in memory. So the expression table starts at 840 bytes. Each operand and operator in an expression is one entry in the symbol table-- another fourteen bytes. The term, "operator", includes each function and each comma in an argument list, as well as the symbols you would normally expect (+, &, ~, etc.). There are some other rules for determining the number of entries an expression will require. Since they are not straightforward and are subject to change, they will not be discussed here.

The following expression uses 15 entries in the table:

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

Everything is an entry except for the ")", including the commas which separate the function arguments.

If the expression table overflows, the compiler will generate error number 36, "no more expression space."

This command will reserve space for 100 entries (1800 bytes) in the expression table:

```
cc -E100 filename
```

The option must be given before the filename. There can be no space between the option letter and the value.

The Local Symbol Table:

New symbols can be declared after any open brace. Most commonly, a declaration list appears at the beginning of a function body. The symbols declared here are added to the local symbol table. If a variable is declared in the body of, say, a **for** loop, it is added to the table. When the compiler has finished compiling the loop, that entry in the table is freed up. And when it has finished the function, the table will be empty.

The default size of the table is 30 entries. Since each entry consumes 26 bytes, the table begins at 520 bytes. If the table overflows, the compiler will send a message to the screen and stop.

The number of entries in the table can be adjusted with the -L option. The following compilation will use a table of 75 entries, or almost 2000 bytes:

```
cc -L75 program.c
```

The Case Table:

When the compiler looks at a switch statement, it builds a table of the cases in it. When it "leaves" the switch statement, it frees up the entries for that switch. For example, the following will use a maximum of four entries in the case table:

```
switch (a)  {
case 0:                              /* one */
     a += 1;
     break;
case 1:                              /* two */
     switch (x)  {
     case 'a':                       /* three */
          func1 (a);
          break;
     case 'b':                       /* four */
          func2 (b);
          break;
     }                         /* release the last two */
     a = 5;
case 3:                         /* total ends at three */
     func2 (a);
     break;
}
```

The table defaults to 40 entries, each using up four bytes. If the compiler returns with an error 76 ("case table exhausted"), you will have to recompile with a new size, as in:

```
cc -Y100 file
```

The String Table:

This is where the compiler saves "literals", or strings. The size of this area defaults to 1000 bytes. Each string occupies a number of bytes equal to the size of the string. The size of a string is just the number of characters in it plus one (for the null terminator).

If the string table overflows, the compiler will generate error 2, "string space exhausted".

The following command will reserve 2000 bytes for the string table:

```
cc -Z2000 file
```

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

Error checking

Compiler errors come in two varieties-- fatal and not fatal. Fatal errors cause the compiler to make a final statement and stop. Running out of memory and finding no input are examples of fatal errors. Both kinds of errors are described in section XII. The non-fatal sort are introduced below.

The compiler will report any errors it finds in the source file. It will first print out a line of code. The up-arrow (carot) in this line indicates how far the compiler went before it was able to detect the error. The name of the source file will appear, followed by a line number, an error number and the symbol which may have caused the error.

The compiler is not always able to give a precise description of an error. Usually, it must proceed to the next item in the file to ascertain that an error was encountered. Once an error is found, it is not obvious how to interpret the subsequent code, since the compiler cannot second-guess the programmer's intentions. This may cause it to flag perfectly good syntax as an error.

If errors arise at compile time, it is a general rule of thumb that the very first error should be corrected first. This may clear up some of the errors which follow.

The best way to attack an error is first to look up the meaning of the error code in the back of this manual. Some hints are given there as to what the problem might be. And you will find it easier to understand the error and the message if you know why the compiler produced that particular code. The error codes indicate what the compiler was doing when the error was found.

The Assembler

The Assembler

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

The Manx AS assembler is a relocating assembler. It is invoked by the command line:

```
AS filename.asm
```

The relocatable object file produced by the assembly will be named "filename.o". An alternate object filename can be supplied by specifiying -O filename (O is a letter). The object file will be written to the filename following "-O", as in the following example:

```
as -o newfil.obj filename.asm
```

The output filename does not have to end with ".o", but that is the recommended format. The assembly language source file can also have any extension. If none is given, the extension defaults to ".asm".

When assembling compiler output, a useful option is **-ZAP.** After creating the object code output file, the assembler will delete the (intermediate) assembly language file. This conserves disk space and is especially useful when compiling a large number of C source files.

It is common practice to create a C language source file ending in ".c", such "prog.c", and leave off the extension entirely when compiling and assembling:

```
cc prog
as -ZAP prog
```

To produce an assembly listing, specify the **-L** option, as in the following example. The assembler is a one pass assembler so forward address references will not appear on the listing.

```
as -l prog
```

The following summaries define the syntax for the AS assembler:

STATEMENTS

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

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

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

LABELS

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

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

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

EXPRESSIONS

Expressions are evaluated from left to right according to parenthesization, with precedence given unary operators. Operators are:

+ - * / AND OR XOR NOT SHL SHR MOD

CONSTANTS

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

A character constant is of the form: 'A', that is, a character enclosed by single quotes.

ASSEMBLER DIRECTIVES

The Manx AS assembler supports the following pseudo operations:

BSS sym_name, size	creates symbol (sym_name) of 'size' bytes in the BSS segment (which contains **static** data from C source code).
COMMON /<block name>/	sets the location counter to the selected common block.
CSEG	select code segment.
DB <exp>	define byte constant.
DSEG	select data segment.
DW	define word constant (2 bytes).
END	end of assembler source statements.
GLOBAL sym_name, size	creates an external symbol with size, 'size' bytes. An uninitialized global **char** in C compiles to a **global** of one byte. If a symbol is defined with **global** more than once, storage is allocated by the linker for the largest size given.
NLIST	turn off listing
LIST	turn on listing
MACLIB/XTEXT filename	include statements from another file
PUBLIC/EXT/EXTRN label	declares label to be external or entry

SECTION V

The Linker

The Linker

The Aztec linker is the software which ties together the pieces of a program which were compiled and assembled separately. The assembler produces a file containing what is called relocatable object code. The Manx AS assembler generates object files with a specific object file format. The Aztec linker expects that the files it links will be in this format. That is why AS and LN must be used together.

The following pages are a brief introduction to linking and what the linker does. If you have had previous experience with linkage editors, you may wish to continue reading with the paragraph heading, "Using the Linker." There you will find a concise description of the command format for the linker.

Relocatable Object Files

The object code produced by the assembler is "relocatable" because it can be loaded anywhere in memory. One task of the linker is to assign specific addresses to the parts of the program. This tells the operating system where to load the program when it is run.

Linking hello.o

It is very unusual for a C program to consist of a single, self-contained module. Let's consider a simple program which prints "hello, world" using the function, **printf**. The terminology here is precise; **printf** is a function and not an intrinsic feature of the language. It is a function which you might have written, but it already happens to be provided in the file, "c.lib". This file is a library of all the standard i/o functions. It also contains many support routines which are called in the code generated by the compiler. These routines aid in integer arithmetic, operating system support, etc.

When the linker sees that a call to **printf** was made, it pulls the function from the library and combines it with the "hello, world" program. The link command would look like this:

```
ln hello.o c.lib
```

When "hello.c" was compiled, calls were made to some invisible support functions in the library. So linking without the standard library will cause some unfamiliar symbols to be undefined. All programs will need to be linked with "c.lib".

The Linking Process

Since the standard library contains only a limited number of general purpose functions, all but the most trivial programs are certain to call user-defined functions. It is up to the linker to connect a function call with the definition of the function somewhere in the code.

In the example given below, the linker will find two function calls in file 1. The reference to func1 is "resolved" when the definition of func1 is found in the same file. The following command

```
ln file1.o c.lib
```

will cause an error indicating that "func2" is an undefined symbol. The reason is that the definition of func2 is in another file, namely file2.o. The linkage has to include this file in order to be successful:

```
ln file1.o file2.o c.lib
```

```
          file 1                       file 2

     main()                       func2()
     {                            {
          func1();                     return;
          func2();                }
     }

     func1()
     {
          return;
     }
```

Libraries

A library is a collection of object files put together by a librarian. Libraries intended for use with LN must be built with the Aztec librarian, LIBUTIL. This utility is described in the next section.

All the object files specified to the linker will be "pulled into" the linkage; they are automatically included in the final executable file. However, when a library is encountered, it is searched. Only those modules in the library which satisfy a previous function call are pulled in.

For Example

Consider the "hello, world" example. Having looked at the module, "hello.o", the linker has built a list of undefined symbols. This list includes all the global symbols that have been referenced but not defined. Global variables and all function names are considered to be global symbols.

The list of undefined's for "hello.o" includes the symbol, "printf". When the linker reaches the standard library, this is one of the symbols it will be looking for. It will discover that "printf" is defined in a library module whose name also happens to be "printf". (There is not any necessary relation between the name of a library module and the functions defined within it.)

The linker pulls in the "printf" module in order to resolve the reference to the "printf" function.

Files are examined in the order in which they are specified on the command line. So the following linkages are equivalent:

```
ln hello.o

ln c.lib hello.o
```

Since no symbols are undefined when the linker searches c.lib in the second line, no modules are pulled in. It is good practice to leave all libraries at the end of the command line, with the standard library last of all.

The Order of Library Modules

For the same reason, the order of the modules within a library is significant. The linker searches a library once, from beginning to end. If a module is pulled in at any point, and that module introduces a new undefined symbol, then that symbol is added to the running list of undefined's. The linker will not search the library twice to resolve any references which remain unresolved. A common error lies in the following situation:

module of program	references (function calls)
main.o	**getinput, do_calc**
input.o	**gets**
calc.o	**put_value**
output.o	**printf**

Suppose we build a library to hold the last three modules of this program. Then our link step will look like this:

```
ln main.o proglib.lib c.lib
```

But it is important that "proglib.lib" is built in the right order. Let's assume that main() calls two functions, getinput() and do_calc(). getinput() is defined in the module, input.o. It in turn calls the standard library function, gets(). do_calc() is in calc.o and calls put_value(). put_value() is in output.o and calls printf().

What happens at link time if proglib.lib is built as follows?

```
proglib.lib:        input.o
                    output.o
                    calc.o
```

After main.o, the linker has "getinput" and "do_calc" undefined (as well as some other obscure functions in c.lib). Then it begins the search of proglib.lib. It looks at the library module, "input", first. Since that module defines "getinput", that symbol is taken off the list of undefined's. But "gets" is added to it.

The symbols "do_calc" and "gets" are undefined when the linker examines the module, "output". Since neither of these symbols are defined there, that module is ignored. In the next module, "calc", the reference to "do_calc" is resolved but "put_value" is a new undefined symbol.

The linker still has "gets" and "put_value" undefined. It then moves on to c.lib, where "gets" is resolved. But the call to "put_value" is never satisfied. The error from the linker will look like this:

```
Undefined symbol: put_value_
```

This means that the module defining "put_value" was not pulled into the linkage. The reason, as we saw, was that "put_value" was not an undefined symbol when the "output" module was passed over. This problem would not occur with the library built this way:

```
proglib.lib:        input.o
                    calc.o
                    output.o
```

The standard libraries were put together with much care so that this kind of problem would not arise.

Occasionally it becomes difficult or impossible to build a library so that all references are resolved. In the example, the problem could be solved with the following command:

```
ln main.o proglib.lib proglib.lib c.lib
```

The second time through proglib.lib, the linker will pull in the module "output". The reason this is not the most satisfactory solution is that the linker has to search the library twice; this will lengthen the time needed to link.

Using the Linker

The general form of a linkage is as follows:

```
ln [-options] file1.o [file2.o etc] [lib1.lib etc]
```

The linker will essentially combine any number of object files produced by the Aztec assembler into an executable program. It will also search a library of object modules for functions needed to complete the linkage. Only those modules needed will be pulled out of the library. The linker makes just a single pass through a library, so that only forward references within a library will be resolved.

By default, the executable output file will be named after the first object file given on the command line. It will have the extension ".com". The following linkage:

```
ln prog.o c.lib
```

will produce the disk file, prog.com, which can be run under CP/M. The standard library, c.lib, will have to be included in most linkages.

A different output file can be specified with the -O option, as in the following command:

```
ln -o program.com mod1.o mod2.o c.lib
```

The name given with -O must have the extension, ".com".

This command also shows how several individual modules can be linked together. A "module", in this sense, is a section of a program containing a limited number of functions, usually related. These modules are compiled and assembled separately and linked together to produce a ".com" file. Modules are useful because a change can be made to a single module without having to recompile the source for the entire program.

More About Libraries

When certain modules are used over and over by different programs, it is often expedient to build a library containing these commonly used modules. This library can then be included in the linkage of any of these programs. Any number of libraries can be included in a given linkage:

```
ln -o program.o mylib.lib new.lib m.lib c.lib
```

Each of the libraries will be searched once in the order in which they appear on the command line. In this example, the m.lib library is the math library provided by Manx.

Libraries can be named more conveniently with the **-L** option to the linker. The previous linkage is identical to the following:

```
ln -o program.o -lmylib -lnew -lm -lc
```

The -L option will take the string following it and append a ".lib". The resulting filename will be treated as the name of a library.

The options recognized by the linker are summarized in the table below. Further explanations follow.

Linker Options

General Purpose Options

-F	This option allows command arguments to be taken from the file specified.
-L	Specifies a library of routines.
-O	Specifies alternate output filename.
-R	Generates a symbol table for overlays.
-T	Creates a symbol table file.
-V	Verbose mode.

Segment Address Specification

-B	Sets the base address of the program.
-C	Sets the beginning address for the code portion of the program.
-D	Sets the beginning address for the data area.
-U	Sets the beginning address for the uninitialized data area.

Memory Usage

+C	Reserves specified number of bytes at end of code segment of linked program (for use with overlays).
+D	Reserves specified number of bytes at end of data segment of linked program (for use with overlays).

General Purpose Options:

-F causes the linker to merge the contents of the given file with the command line arguments. For example,

```
ln myprog.o -f argfil -lc
```

where the file, argfil, contains the following:

```
mod1.o mod2.o
mylib.lib
```

All records (that is, all lines) of the file are read. There is no need to squeeze everything into one record.

There are several advantageous uses for this command. The most obvious is to supply the names of modules that are commonly linked together. Since all the modules named are automatically pulled into the linkage, the linker does not spend any time in searching, as with a library. Furthermore, any linker option except -F can be given in a -F file. -F can appear on the command more than once, and in any order. The arguments are processed in the order in which they are read, as always.

The **-R** option is used only when portions of a program are being linked as overlays. This option is fully described in section IX.

The **-T** option creates a disk file which contains a symbol table for the linkage. This file is just a text file which lists each symbol with a hexadecimal address. This address is either the entry point for a function or the location in memory of a data item. A perusal of this file will indicate which functions were actually pulled into the linkage.

The symbol table file will have the same name as the ".com" file, except that its extension will be ".sym". This ".sym" file can be used in conjunction with the SID or ZSID debugging aid available from Digital Research.

There are six special symbols which will appear in the table. They are as follows:

Corg	origin of code area (cf. -C option)
Cend	end of code area
Dorg	origin of data area (-D)
Dend	end of data area
Uorg	origin of uninitialized data (-U)
Uend	end of uninitialized data area

The **-V** option causes the linker to send a progress report of the linkage to the screen as each input file is processed. This is useful in tracking down undefined symbols and other errors which may error while linking.

segment address specification:

There are four crucial addresses which may be specified at link time. They are the base address and the starting addresses of the three major parts of the program. A linked program normally looks like this:

```
 ________________
|                |        Uend
|                |
|________________|        Uorg
|                |        Dend
|                |
|________________|        Dorg
|                |        Cend
|                |
|                |
|                |
|________________|        Corg      (hex address 103)
|________________|        base      (hex address 100)
```

The symbols depicted in the figure can be seen in the symbol table produced with the **-T** linker option.

At the base address is an instruction which initiates execution; it is a jump to the beginning of the program. By default, this instruction is located at hex address 100.

Again by default, the code portion of the program starts right after this instruction-- three bytes higher at hex address 103.

The area containing initialized data is placed directly above the code. At the top comes the uninitialized data area.

This order can be rearranged. The only restriction is that the base address must be lowest and the code and data regions must not overlap. If either condition is not satisfied, the linker will print an error message and abort.

For example, the base address of a program is set by the following command:

```
ln -b 500 prog.o -lc
```

The base address for prog.com will be hex address 500. The address specified for any of these options is assumed to be hexadecimal.

When just the base address is fixed, as in the example, the code and data remain as a continguous span of memory above the base address. However, the remaining three addresses can also be specified:

```
ln -b 200 -c 500 -d 1000 -u 3000 prog.o -lc
```

This capability is needed for ROM based applications and for situations requiring that certain areas of memory not be overwritten by the program.

Memory Usage:

The **+C** and **+D** options effectively increase the size of the code and data segments of the linked program. For example,

```
ln +d 1000 prog.o -lc
```

will increase the data area of the linked program by hex 1000 bytes. These options are specifically provided for the use of overlays. See the section describing software extensions for more details.

SECTION VI

The Librarian

Library Maintenance

LIBUTIL

Summmary

The LIBUTIL LIBrary UTILity is used in order to:

1. create a library	
2. append a library	(-a)
3. produce an index list	(-t)
4. extract members	(-x)
5. replace a module	(-r)
6. create a library using an extended command line	(.)
7. report progress of library build	(-v)

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

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

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

2. LIBUTIL option -a

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

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

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

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

NB If a large number of files need to be appended to a library, it is advantageous to use the dot option (see item 6).

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; also, this lists only module names, and not the functions which each module may contain.

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 sub1.o, sub2.o, sub3.o, sub4.o, etc.

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

7. LIBUTIL option -v

USE	to report the current status of a library during an operation by LIBUTIL.

In More Detail...

Creating a Library

The command for creating a new library has this format:

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

The -O option specifies the name of the library being created. If the option is not given, then the library name is assumed to be "libc.lib". It is not recommended that LIBUTIL be used without naming a library with this option.

How it Works

First, LIBUTIL creates the library in a new file with a temporary name. If this file was successfully written, LIBUTIL erases the file with the same name as the library, if one exists. In effect, it makes sure that the new library can be created before destroying the old. Then the temporary file is renamed to the library name.

Note that there must be room on the disk for both the old library and the new.

The <input file list> is a list of the object files which are to be included in the library. These are usually files generated by the Manx assembler.

Naming Conventions

An input filename can include a drive specification, as in the name, b:module1.o. Otherwise, the file is assumed to be on the default drive. Also, the ".o" ending can be left off altogether. When an input filename lacks an extent, ".o" is added on.

When an input file contains a single relocatable object module, the name of the module in the library will be the filename, less the drive specification and the extension. For example, if the input file is b:subl.o, then the module name inside the new library will be subl.

An input file can be a library itself. In this case, the module names in the new library are the same as those in the input library. For example, if the input file is a library containing modules subl, sub2 and sub3, then the names of these modules in the created library will also be subl, sub2 and sub3.

Since the list of input files for a library often will not fit on a single line, there is a convenient way to extend the command line. A period on the command line directs the linker to start reading filenames from standard input. When another period is read, the linker returns to the command line to read in the remaining filenames.

Order in a Library

The order in which a library is built is often crucial for easy linking. Modules go into a new library in the order in which they are read by LIBUTIL. Consider the following example:

Let's assume there is currently a library, oldlib.lib, which contains three modules:

subl sub2 sub3

The following command might be given:

```
LIBUTIL -o newlib.lib oldlib.lib sub4 . sub5
sub6.o sub7.o
sub8
.
```

This will create a library called newlib.lib. The first three modules copied into it come from oldlib.lib. Then the contents of sub4.o become the module, sub4, in the library.

When LIBUTIL finds a period, it continues reading the filenames from standard input. So the next three files copied into newlib.lib are sub6.o, sub7.o and sub8.o. Notice that ".o" after a filename in the command is assumed.

The last module read in the example is in sub5.o. So the final makeup of newlib.lib is:

subl sub2 sub3 sub4 sub6 sub7
sub8 sub5

Listing the Modules in a Library

A listing such as this can be obtained with the **-T** option. This option simply produces a listing of the modules in the order in which they appear in a library. The -O option is used in this case to specify which library is to be listed. For example, the listing above would be produced by entering:

```
libutil -o newlib.lib -t
```

If the -O option is missing, the library, libc.lib, is assumed.

LIBUTIL will not perform multiple functions during a single invocation. For example, you cannot make it create a library and then list its contents with a single command; you would need to run LIBUTIL for each task.

There are just a few ways to use the -T option, such as:

```
libutil -t
libutil -ot example.lib
libutil -t -o example.lib
```

Note that the listing the modules of a library does not give a true representation of what functions are defined within the library. For instance, a module named "prog_inp" might contain the functions, "get_record", "get_name" and "get_num".

Adding and Replacing Modules

The **-A** and **-R** options are used to add or replace modules in a library. These options actually refer to the same process. The method used by LIBUTIL is fairly simple.

The -O option is used to specify the library that going to be modified; as always, this defaults to c.lib.

LIBUTIL creates a temporary file, just as it did when making a new library. Each module of the old library is then copied, in order, to the new file. Whenever a module name matches a name given on the command line, the old library module is ignored, and the contents of the file given in the command are copied to that module in the new file.

When the last module in the old library has either been copied or skipped over, LIBUTIL returns to the command line. The files which have already been copied to the new library are checked off. LIBUTIL then copies to the new library all the remaining files on the command line, which have not been copied to the new library.

For example, given an obsolete library, obslib.lib:

mod1 mod2 mod3

and the following command:

```
libutil -oa obslib.lib mod2 . sub2
sub1
```

LIBUTIL first copies mod1 from obslib to the temporary file. Since mod2 is specified on the command line, it copies the contents of mod2.o to the temporary file and ignores the mod2 in obslib. It continues to copy mod3 from obslib, sub1 and sub2 to the temporary file, in that order. Then the temporary file is renamed to obslib.lib and the old library is erased.

Just as in library creation, the old and the new libraries exist on disk at the same time, before the old is erased. There has to be enough room for both.

Consider the following command:

```
libutil -oa obslib.lib obslib.lib
```

LIBUTIL will copy obslib to the temporary file, since none of the module names appear on the command line. Then the remaining files from the input list are copied to the temporary file. So that a listing of the resulting obslib.lib would be:

mod1 mod2 mod3 mod1 mod2 mod3

This curious naming of modules does not affect the way their contents are treated by the linker. For example, the first mod1 might contain a single function, "get_value", while the second contains a function, "get_num". If "get_value" is an undefined symbol when the linker searches the library, just the first mod1 will be pulled into the link, and similarly the second will be pulled in for an undefined "get_num".

Library Functions

Library Functions

This chapter describes the functions which come with the Aztec C package. It's divided into four subchapters: introduction, overview, functions, and system dependent functions.

The 'overview' subchapter presents an overview of several topics, including i/o processing, memory usage, and error handling.

The 'functions' subchapter describes in detail the functions in the Aztec C package which are common to all systems supported by Aztec C. Most of these functions are also supported by Unix; those which aren't are clearly identified.

The 'system dependent functions' subchapter describes functions which are unique to a system.

A table of contents is provided at the beginning of the two functions subchapters.

Library Overview

Overview of Library Functions

This subchapter contains several sections, each of which presents an overview of a different topic. The following sections are provided:

I/O
: Introduces the i/o system provided in the Aztec C package.

STANDARD I/O
: The i/o functions can be grouped into two sets; this section describes one of them, the standard i/o functions.

UNBUFFERED I/O
: This section describes the other set of i/o functions, the unbuffered.

CONSOLE I/O
: Describes special topics relating to console i/o.

DYNAMIC BUFFER ALLOCATION
: Discusses related to dynamic memory allocation.

ERRORS
: Presents an overview of error processing.

Overview of I/O

There are two sets of functions for accessing files and devices: the unbuffered i/o functions and the standard i/o functions. These functions are identical to their UNIX equivalents, and are described in chapters 7 and 8 of _The C Programming Language_.

The unbuffered i/o functions are so called because, with few exceptions, they transfer information directly between a program and a file or device. By contrast, the standard i/o functions maintain buffers through which data must pass on its journey between a program and a disk file.

The unbuffered i/o functions are used by programs which perform their own blocking and deblocking of disk files. The standard i/o functions are used by programs which need to access files but don't want to be bothered with the details of blocking and deblocking the file records.

The unbuffered and standard i/o functions each have their own overview section (UNBUFFERED I/O and STANDARD I/O). The remainder of this section discusses features which the two sets of functions have in common.

The basic procedure for accessing files and devices is the same for both standard and unbuffered i/o: the device or file must first be "opened", that is, prepared for processing; Then i/o operations occur; then the device or file is "closed".

A maximum of eleven files and devices can be open at once for both standard and unbuffered i/o. When this limit is reached, an open file or device must be closed before another can be opened.

Each set of functions has its own functions for performing these operations. For example, each set has its own functions for opening a file or device. Once a file or device has been opened, it can be accessed only by functions in the same set as the function which performed the open, and must be closed by the appropriate function in the same set. There are exceptions to this non-intermingling which are described below.

There are two ways a file or device can be opened: first, the program can explicitly open it by issuing a function call. Second, it can be associated with one of the logical devices standard input, standard output, or standard error, and then opened when the program starts.

Standard input, standard output, and standard error devices

There are three logical devices which are automatically opened when a program is started: standard input, standard output, and standard error. By default, these are associated with the console. The operator, as part of the command line which starts the program, can specify that these logical devices are to be "redirected" to another device or file. Standard input is redirected by entering on the command line, after the program name, the name of the file or device, preceeded by the character '<'. Standard output is redirected by entering the name of the file or device, preceeded by '>'. For example, suppose the executable program **copy** reads standard input and writes it to standard output. Then the following command will read lines from the keyboard and write them to the display:

```
copy
```

The following will read from the keyboard and write it to the file testfile:

```
copy >testfile
```

This will copy the file exmplfil to the console:

```
copy <exmplfil
```

And this will copy exmplfil to testfile:

```
copy <exmplfil >testfile
```

Aztec C will pass command line arguments to the user's program via the user's function **main(argc, argv)**. **argc** is an integer containing the number of arguments plus one; **argv** is a pointer to a an array of character pointers, each of which, except the first, points to a command line argument. The first array element on some systems points to the command; on other systems, for example, CP/M and CP/M-86, the first pointer is null.

For example, if the following command is entered:

```
cat arg1 arg2 arg3
```

the program **cat** will be activated and execution begins at the user's function **main**. The first parameter to main is the integer 4. The second parameter is a pointer to an array of four character pointers; on some systems the first array element will point to the string "cat" and on others it will be a null pointer. The second, third, and fourth array elements will be pointers to the strings "arg1", "arg2", and "arg3" respectively.

The command line can contain both arguments to be passed to the user's program and i/o redirection specifications. The i/o redirection strings won't be passed to the user's program, and can appear anywhere on the command line after the command name. For example, the standard output of the cat program can be redirected to the file outfile by any of the following commands; in each case the argc and argv parameters to the main function of cat are the same as if the redirection specifier wasn't present:

```
cat arg1 arg2 arg3 >outfile
cat >outfile arg1 arg2 arg3
cat arg1 >outfile arg2 arg3
```

Sequential I/O

A program can access files both sequentially and randomly. For sequential access, a program simply issues any of the various read or write calls. The transfer will begin at the file's "current position", and will leave the current position set to the byte following the last byte transferred. A file can be opened for read or write access; in this case, its current position is initially the first byte in the file. A file can also be opened for append access; in this case its current position is initially the end of the file.

On systems which don't keep track of the last character written to a file, such as CP/M and Apple DOS, it isn't always possible to correctly position a file to which data is to be appended. See below for details.

Random I/O

Two functions are provided which allow a program to set the current position of an open file: **fseek**, for a file opened for standard i/o; and **lseek**, for a file opened for unbuffered i/o.

A program accesses a file randomly by first modifying the file's current position using one of the seek functions. Then the program issues any of the various read and write calls, which sequentially access the file.

A file can be positioned relative to its beginning, current position, or end. Positioning relative to the beginning and current position is always correctly done. For systems which don't keep track of the last character written to a file, such as CP/M and Apple DOS, positioning relative to the end of a file can't always be correctly done. See below for details.

Finding the end of a file

UNIX keeps track of the last character written to a file. Since the Aztec I/O functions attempt to make a file look like a UNIX file to a program, when a program requests that a file be positioned relative to its end (that is, relative to the last character which was written to it), the Aztec C routines must try to locate the last character which was written to it. This can always be done if the operating system on which Aztec C is running also keeps track of the last character written to a file.

However, CP/M, CP/M-86, and Apple DOS only keep track of the last record written to a file. For these systems, it is not always possible for the Aztec C i/o functions to determine the last character written to the file, and hence for these systems it is not always possible to position a file relative to its end.

When a program running on one of the systems mentioned in the last paragraph requests positioning of a file relative to its end, the Aztec i/o functions try to find the last character written to the file. They always succeed if the file contains only text; for files containing arbitrary data, they may not succeed.

To locate the last valid character in a file on one of these systems, the Aztec routines use the following fact: when a file is created on these systems using Aztec C, the last record in the file is padded at the end with the special character which denotes the end of a text file. For CP/M and CP/M-86, the special character is control-z; for Apple DOS, it's a null character. If the program exactly filled the last record, it won't have any padding.

When a program requests that a file be positioned relative to its end, the Aztec C i/o routines search the file's last record; end of file is declared to be located at the position following the last non-end-of-file character.

For files of text, this algorithm always correctly determines the last character in the file, so appending to text files is always correctly done.

For other files, this algorithm will still correctly determine the last valid character in the file...most of the time. However, if the last valid characters in the file are end-of-file characters, the file will be incorrectly positioned.

Opening files

Opening files is somewhat system dependent: the parameters to the open functions are the same on the Aztec C packages for all systems, but some system dependencies exist, to conform with the system conventions. For example, the syntax of file names and and the areas searched for files differ from system to system.

The following paragraphs describe, for the systems supported by Aztec C, system dependent information related to the opening of files.

Opening files on CP/M, CP/M-86, and related systems

The character string which specifies the file to be opened has the following fields, which must be in the order listed: (1) a user number followed by a forward slash, (2) a drive identifier followed by a colon, (3) the filename, (4) a period followed by an extension. Only the third field is mandatory. If a user number isn't specified, the file is assumed to be on the current user. If the drive isn't specified, the file is assumed to be on the default drive.

For example, the following are valid file names:

file.ext	file.ext is on default drive, current user
b:file.ext	file.ext is on b: drive, current user
15/file.ext	file.ext is on default drive, user 15
12/c:file.ext	file.ext is on c: drive, user 12

A program can have files located in several different user areas open at once.

There are several functions which may be useful to programs which need to access files in various user areas: **getusr**, which returns the current user number; **setusr**, which sets the current user number; and **rstusr**, which resets the current user number. See the section USER in the system dependent subchapter for more details.

Opening files on TRSDOS and related systems

When opening a file on TRSDOS or related systems, the filename has the standard TRSDOS format; that is, (1) filename, (2) followed by a slash and an extention, (3) followed by a period and a password, (4) followed by a colon and a drive number. Only the first field is mandatory.

If a drive specifier is given, the file will be searched for, and created if necessary, on that drive. Otherwise, following the TRSDOS convention, a search for the file will be made on all drives, beginning with drive :0. If the file is found, and must be recreated, it will be recreated on the same drive.

Accessing Devices

Aztec C allows programs to access devices as well as files. Each system has its own names for devices, so the following table lists the devices and, for each system, its name. In this table, "CPM" refers to CP/M, CP/M-86, and related systems; PCDOS also includes MSDOS; TRSDOS includes LDOS and DOSPLUS.

device	CPM	PCDOS	Apple DOS	TRSDOS
keyboard	con:	con:	ki:	*ki
display	con:	con:	do:	*do
printer	prn:	prn	pr:	*pr
"	lst:	-	-	-
RS232 in	rdr:	-	-	*ri
RS232 out	pun:	-	-	*ro

On model 4 TRSDOS, dynamically created devices can also be accessed.

Mixing unbuffered and standard i/o calls

As mentioned above, a program generally accesses a file or device using functions from one set of functions or the other, but not both.

However, there are functions which facilitate this dual access: if a file or device is opened for standard i/o, the function **fileno** returns a file descriptor which can be used for unbuffered access to the file or device. If a file or device os open for unbuffered i/o, the function **fdopen** will prepare it for standard i/o as well.

Care is warranted when accessing devices and files with both standard and unbuffered i/o functions.

Overview: Standard I/O

The standard i/o functions are used by programs to access files and devices. They are compatible with their UNIX counterparts, with few exceptions, and are also described in chapter 8 of The C Programming Language. The exceptions concern appending data to files and positioning files relative to their end, and are discussed below.

These functions provide programs with convenient and efficient access to files and devices. When accessing files, the functions buffer the file data; that is, handle the blocking and deblocking of file data. Thus the user's program can concentrate on its own concerns.

Buffering of data to devices when using the standard i/o functions is discussed below.

For programs which perform their own file buffering, another set of functions are provided. These are described in the section UNBUFFERED I/O.

Opening files and devices

Before a program can access a file or device, it must be "opened", and when processing on it is done it must be "closed".

An open device or file is called a "stream" and has associated with it a pointer, called a "file pointer", to a structure of type FILE. This identifies the file or device when standard i/o functions are called to access it.

There are two ways for a file or device to be opened for standard i/o: first, the program can explicitly open it, by calling one of the functions **fopen**, **freopen**, or **fdopen.** In this case, the open function returns the file pointer associated with the file or device. **fopen** just opens the file or device. **freopen** reopens an open stream to another file or device; it's mainly used to change the file or device associated with one of the logical devices standard output, standard input, or standard error. **fdopen** opens for standard i/o a file or device already opened for unbuffered i/o.

Alternatively, the file or device can be automatically opened as one of the logical devices standard input, standard output, or standard error. In this case, the file pointer is **stdin**, **stdout**, or **stderr**, respectively. These symbols are defined in the header file **stdio.h.** See the section entitled I/O for more information on logical devices.

Closing streams

A file or device opened for standard i/o can be closed in two ways: first, the program can explicitly close it by calling the function **fclose**.

Alternatively, when the program terminates, either by falling off the end of the function **main**, or by calling the function **exit**, the system will automatically close all open streams.

Letting the system automatically close open streams is error-prone: data written to files using the standard i/o functions is buffered in memory, and a buffer isn't written to the file until it's full or the file is closed. Most likely, when a program finishes writing to a file, the file's buffer will be partially full, with this information not having been written to the file. If a program calls **fclose**, this function will write the partially filled buffer to the file and return an error code if this couldn't be done. If the program lets the system automatically close the file, the program won't know if an error occurred on this last write operation.

Sequential I/O

Files can be accessed sequentially and randomly. For sequential access, simply issue repeated read or write calls; each call transfers data beginning at the "current position" of the file, and updates the current position to the byte following the last byte transferred. When a file is opened, its current position is set to zero, if opened for read or write access, and to its end if opened for append.

On systems which don't keep track of the last character written to a file, such as CP/M and Apple DOS, not all files can be correctly positioned for appending data. See the section entitled I/O for details.

Random I/O

The function **fseek** allows a file to be accessed randomly, by changing its current position. Positioning can be relative to the beginning, current position, or end of the file.

For systems which don't keep track of the last character written to a file, such as CP/M and Apple DOS, positioning relative to the end of a file cannot always be correctly done. See the I/O overview section for details.

Buffering

When the standard i/o functions are used to access a file, the i/o is buffered. Either a user-specified or dynamically-allocated buffer can be used.

The user's program specifies a 1024-byte buffer to be used for a file by calling the function **setbuf** after the file has been opened but before the first i/o request to it has been made.

If, when the first i/o request is made to a file, the user hasn't specified the buffer to be used for the file, the system will automatically allocate, by calling **malloc**, a 1024-byte buffer for it. When the file is closed it's buffer will be freed, by calling **free**.

Dynamically allocated buffers are obtained from the one region of memory (the heap), whether requested by the standard i/o functions or by the user's program. For more information, see the overview section DYNAMIC BUFFER ALLOCATION.

A program which both accesses files using standard i/o functions and has overlays has to take special steps to insure that an overlay won't be loaded over a buffer dynamically allocated for file i/o. For more information, see section IX on overlay support.

By default, output to the console using standard i/o functions is unbuffered; all other device i/o using the standard i/o functions is buffered. Console input buffering can be disabled using the **ioctl** function; see the CONSOLE I/O overview for details. Buffering of standard i/o to other devices can be disabled using the **setbuf** function. See the description of **setbuf** for details.

Errors

There are three fields which may be set when an exceptional condition occurs during stream i/o. Two of the fields are unique to each stream (that is, each stream has its own pair). The other is a global integer.

One of the fields associated with a stream is set if end of file is detected on input from the stream; the other is set if an error occurs during i/o to the stream. Once set for a stream, these flags remain set until the stream is closed or the program calls the **clearerr** function for the stream. The only exception to the last statement is that when called, **fseek** will reset the end of file flag for a stream. A program can check the status of the eof and error flags for a stream by calling the functions **feof** and **ferror**, respectively.

The other field which may be set is the global integer **errno**. By convention, a system function which returns an error status as its value can also set a code in **errno** which more fully defines the error. The section ERRORS defines the values which may be set in **errno**.

If an error occurs when a stream is being accessed, a standard i/o function returns EOF (-1) as its value, after setting a code in **errno** and setting the stream's error flag.

If end of file is reached on an input stream, a standard i/o function returns EOF after setting the stream's eof flag.

There are two techniques a program can use for detecting errors during stream i/o. First, the program can check the result of each i/o call. Second, the program can issue i/o calls and only periodically check for errors (for example, check only after all i/o is completed).

On input, a program will generally check the result of each operation.

On output to a file, a program can use either error checking technique; however, periodic checking by calling **ferror** is more efficient. When characters are written to a file using the standard i/o functions they are placed in a buffer, which is not written to disk until it is full. If the buffer isn't full, the function will return good status. It will only return bad status if the buffer was full and an error occurred while writing it to disk. Since the buffer size is 1024 bytes, most write calls will return good status, and hence periodic checking for errors is sufficient and most efficient.

Once a file opened for standard i/o is closed, **ferror** can't be used to determine if an error has occurred while writing to it. Hence **ferror** should be called after all writing to the file is completed but before the file is closed. The file should be explicitly closed by **fclose**, and its return value checked, rather than letting the system automatically close it, to know positively whether an error has occurred while writing to the file. The reason for this is that when the writing to the file is completed, it's standard i/o buffer will probably be partly full. This buffer will be written to the file when the file is closed, and **fclose** will return an error status if this final write operation fails.

The standard i/o functions

The standard i/o functions can be grouped into two sets: those that can access only the logical devices standard input, standard output, and standard error; and all the rest.

Here are the standard i/o functions that can only access **stdin, stdout,** and **stderr.** These are all ASCII functions; that is, they expect to deal with text characters only.

getchar	-	get an ASCII character from stdin
gets	-	get a line of ASCII characters from stdin
printf	-	format data and send it to stdout
puterr	-	send a character to stderr
putchar	-	send a character to stdout
puts	-	send a character string to stdout
scanf	-	get a line from stdin and convert it

Here are the rest of the standard i/o functions:

agetc	-	get an ASCII character
aputc	-	send an ASCII character
fopen	-	open a file or device
fdopen	-	open as a stream a file or device already open for unbuffered i/o
freopen	-	open an open stream to another file or device
fclose	-	close an open stream
feof	-	check for end of file on a stream
ferror	-	check for error on a stream
fileno	-	get file descriptor associated with stream
fflush	-	write stream's buffer
fgets	-	get a line of ASCII characters
fprintf	-	format data and write it to a stream
fputs	-	send a string of ASCII characters to a stream
fread	-	read binary data
fscanf	-	get data and convert it
fseek	-	set current position within a file
ftell	-	get current position
fwrite	-	write binary data
getc	-	get a binary character
getw	-	get two binary characters
putc	-	send a binary character
putw	-	send two binary characters
setbuf	-	specify buffer for stream
ungetc	-	push character back into stream

Overview of Unbuffered I/O

The unbuffered I/O functions are used to access files and devices. They are compatible with their UNIX counterparts and are also described in chapter 8 of _The C Programming Language_.

As their name implies, a program using these functions, with two exceptions, communicates directly with files and devices; data doesn't pass through system buffers. Some unbuffered I/O, however, is buffered: when data is transferred to or from a file in blocks smaller than a certain value, it is buffered temporarily. This value differs from system to system, but is always less than or equal to 512 bytes. Also, console input can be buffered, and is, unless specific actions are taken by the user's program.

Programs which use the unbuffered i/o functions to access files generally handle the blocking and deblocking of file data themselves. Programs requiring file access but unwilling to perform the blocking and deblocking can use the standard i/o functions; see the section STANDARD I/O for more information.

Here are the unbuffered i/o functions:

open	- prepares a file or device for unbuffered i/o
creat	- creates a file and opens it
close	- concludes the i/o on an open file or device
read	- read data from an open file or device
write	- write data to an open file or device
lseek	- change the current position of an open file
rename	- renames a file
unlink	- deletes a file
ioctl	- change console i/o mode
isatty	- is an open file or device the console?

Before a program can access a file or device, it must be "opened", and when processing on it is done, it must be "closed".

An open file or device has an integer known as a "file descriptor" associated with it; this identifies the file or device when it's accessed.

There are two ways for a file or device to be opened for unbuffered i/o. First, it can explicitly open it, by calling the function **open**. In this case, **open** returns the file descriptor to be used when accessing the file or device.

Alternatively, the file or device can be automatically opened as one of the logical devices standard input, standard output, or standard error. In this case, the file descriptor is the integer value 0, 1, or 2, respectively. See the section entitled I/O for more information on this.

An open file or device is closed by calling the function **close**. When a program ends, any devices or files still opened for unbuffered i/o will be closed.

If an error occurs during an unbuffered i/o operation, the function returns -1 as its value and sets a code in the global integer **errno**. For more information on error handling, see the section ERRORS.

The remainder of this section discusses unbuffered i/o to the various devices and to files.

Console I/O

Console I/O can be performed in a variety of ways. There's a default mode, and other modes can be selected by calling the function **ioctl**.

When the console is in default mode, console input is buffered and is read from the keyboard a line at a time. Typed characters are echoed to the screen and the operator can use the standard operating system line editing facilities. A program doesn't have to read an entire line at a time (although the system software does this when reading keyboard input into it's internal buffer), but at most one line will be returned to the program for a single read request.

The other modes of console i/o allow a program to get characters from the keyboard as they are typed, with or without their being echoed to the display; to disable normal system line editing facilities; and to terminate a read request if a key isn't depressed within a certain interval.

Output to the console is always unbuffered: characters go directly from a program to the display. The only choice concerns translation of the newline character; by default, this is translated into a carriage return, line feed sequence. Optionally, this translation can be disabled.

For more information see the section CONSOLE I/O.

I/O to Non-console Devices

I/O to devices other than the system console is always unbuffered, with no translations.

File I/O

Programs call the functions **read** and **write** to access a file; the transfer begins at the "current position" of the file and proceeds until the number of characters specified by the program have been transferred.

The current position of a file can be manipulated in various ways by a program, allowing both sequential and random acccess to the file. For sequential access, a program simply issues consecutive i/o requests. After each operation, the current position of the file is set to the character following the last one accessed.

The function **lseek** provides random access to a file by setting the current position to a specified character location.

lseek allows the current position of a file to be set relative to the end of a file. For systems which don't keep track of the last character written to a file, such positioning cannot always be correctly done. For more information, see the section entitled I/O.

open provides a mode, O_APPEND, which causes the file being opened to be positioned at its end. This mode is supported on UNIX Systems 3 and 5, but not UNIX version 7. As with **lseek**, the positioning may not be correct for systems which don't keep track of the last character written to a file.

Overview of Console I/O

Console I/O can be performed in a variety of ways:

- o console input can be buffered;
- o console input can be unbuffered, with the program receiving characters as they're typed;
- o echoing of typed characters to the display can be enabled or disabled;
- o an input operation can be automatically terminated if a character isn't received in a certain interval;
- o mapping of CR to LF on input and LF to CR-LF on output can be enabled or disabled.

There is a default mode for console i/o, which is in effect unless changed by a call to the function **ioctl**. Thus, programs can access the console in the default mode without doing anything special. Or, console i/o can easily be customized to behave as desired by the programmer.

In the default mode for console input, the system maintains an internal buffer of characters which it has read from the keyboard. Characters are returned to the program from this buffer. When the buffer is empty, the system reads a line of characters from the keyboard into the buffer; the program is suspended while this occurs. The operator can use the line editing facilities provided by the operating system, and typed characters are echoed to the display. Finally, carriage return characters are converted to newline characters.

In the default mode, console output is unbuffered (as it is for all other modes). Newline characters are converted to carriage return - line feed sequence.

A program selects console I/O modes using the function **ioctl**. This has the form:

```
#include "sgtty.h"

ioctl(fd, code, arg)
struct sgttyb *arg;
```

The header file **sgtty.h** defines symbolic values for the **code** parameter (which tells **ioctl** what to do) and the structure **sgttyb**.

The parameter **fd** is a file descriptor associated with the console. On UNIX, this parameter defines the file descriptor associated with the device to which the **ioctl** call applies. Here, **ioctl** always applies to the console.

The parameter **code** defines the action to be performed by **ioctl**. It can have these values:

TIOCGETP	- fetch the console parameters and store them in the structure pointed at by **arg**
TIOCSETP and **TIOCSETN**	- set the console parameters according to the structure pointed at by **arg**

The argument **arg** points to a structure having the following format:

```
struct sgttyb {
     char sg_erase;
     char sg_kill;
     int  sg_flags;
}
```

Only **sg_flags** is used by Aztec C; the rest are provided for UNIX compatibility.

sg_flags determines the console I/O mode. These are the symbolic values it can assume:

RAW	- set RAW mode (turns off CBREAK, ECHO, & CRMOD)
CBREAK	- return each character as soon as typed
ECHO	- echo input characters to the display
CRMOD	- map CR to LF on input; convert LF to CR-LF on output

More than one of these codes can be specified in a single call to **ioctl**; the values are simply 'or'ed together. If the RAW option is selected, none of the other options have any effect.

When the console is in RAW mode, console input has the following features:

- o no character translations are performed
- o the operator can't use the operating system's line editing facilities
- o typed characters aren't echoed to the screen
- o the system will attempt to read the number of characters requested by the read request; however, after one character has been received, the operation will be terminated if a character isn't received within a certain period of time. This period is hard coded into **ioctl.c**.

If CBREAK is selected, and RAW is not, console input has the following features:

- o the system will attempt to read the number of characters requested. if ECHO isn't specified, the operation will time out after one character has been received if a character isn't received within a certain period of time. If ECHO is specified, the operation won't terminate until all characters requested have been received.
- o If ECHO is selected, input characters are echoed to the display; otherwise, they're not.
- o If CRMOD is selected carriage returns are translated into line feeds.

If neither RAW nor CBREAK is selected, a mode is selected which is either similar or identical to the default mode described above. If CRMOD is set, the mode is identical. Otherwise, the only difference is that the translation of carriage return into newline on input and the translation of newline to carriage return - linefeed sequence on output doesn't occur.

As mentioned above, when the console is in RAW mode or CBREAK without ECHO mode, a read request to the console will always return at least one character, but will terminate without having read the specified number of characters if a character isn't received within a certain interval. Actually, only the unbuffered read request may time out; the standard i/o functions are implemented in such a way that they will return the requested number of characters.

For example, when the console is in RAW or CBREAK without ECHO mode,

read(fd, buf, 80)

will always return at least one character, but may not return return all 80 if the operator delays too long between subsequent key strokes.

With the console still in a time-out-able mode

getc(fp)

directed to the console will always return one character, and

gets(buf, 80, fp)

will always return an entire line of characters.

EXAMPLES

1. Console input using default mode

The following program copies characters from stdin to stdout. The console is in default mode, and assuming these streams haven't been redirected by the operator, the program will read from the keyboard and write to the display. In this mode, the operator can use the operating system's line editing facilities, such as backspace, and characters entered on the keyboard will be echoed to the display. The characters entered won't be returned to the program until the operator depresses carriage return.

```
#include "stdio.h"
#include "sgtty.h"
main()
{
     int c;
     while ((c = getchar()) != EOF)
          putchar(c);
}
```

2. Console input - RAW mode

In this example, a program opens the console for standard i/o, sets the console in RAW mode, and goes into a loop, waiting for characters to be read from the console and then processing them. The characters typed by the operator aren't displayed unless the program itself displays them. The input request won't terminate until a character is received. This example assumes that the console is named 'con:'; on systems for which this is not the case, just substitute the appropriate name.

```
#include "stdio.h"
#include "sgtty.h"
main()
{
     int c;
     FILE *fp;
     struct sgttyb stty;

     if ((fp = fopen("con:", "r") == NULL){
          printf("can't open the console\n");
          exit();
     }
     stty.sg_flags = RAW;
     ioctl(fileno(fp), TIOCSETP, &stty);
     for (;;){
          c = getc(fp);
          . . .
     }
}
```

3. Console input - console in CBREAK + ECHO mode

This example modifies the previous program so that characters read from the console are automatically echoed to the display. The program accesses the console via the standard input device. It uses the function **isatty** to verify that stdin is associated with the console; if it isn't, the program reopens stdin to the console using the function **freopen**. Again, the console is assumed to be named 'con:'.

```
#include "stdio.h"
#include "sgtty.h"
main()
{
     int c;
     struct sgttyb stty;

     if (!isatty(stdin))
          freopen("con:", "r", stdin);
     stty.sg_flags = CBREAK | ECHO;
     ioctl(0, TIOCSETP, &stty);
     for (;;){
          c = getchar();
          . . .
     }
}
```

Overview of Dynamic Buffer Allocation

Several functions are provided for the dynamic allocation and deallocation of buffers from a section of memory called the 'heap'. They are:

malloc -	allocates a buffer
calloc -	allocates a buffer and initializes it to zeroes
realloc -	allocates more space to a previously allocated buffer
free -	releases an allocated buffer for reuse

In addition, the function **sbrk** allows users to implement their own dynamic buffer allocation scheme: when passed an integer, **sbrk** increments an internal pointer by that amount and returns the original value of the pointer. The pointer initially points to the base of the heap.

Dynamic allocation of standard i/o buffers

Buffers used for standard i/o are dynamically allocated from the heap unless specific actions are taken by the user's program. Standard i/o calls to dynamically allocate and deallocate buffers can be interspersed with those of the user's program.

Programs which perform standard i/o and which must have absolute control of the heap can explicitly define the buffers to be used by a standard i/o stream. See the standard i/o overview for details.

Heap - stack positioning

The starting address of a program's heap is assigned when the program is linked. The linker chapter describes how this assignment is made.

When a program is activated, a 2048-byte block of memory is reserved for the stack. The location of this block is system dependent, but it's always above the beginning of the heap. The heap's ending address is then set to the beginning of the stack segment.

On CP/M, the top of the stack is the base of the CP/M bdos; on TRSDOS, it's $HIGH; on Apple DOS, it's the base of the DOS file buffers.

Buffers can't be allocated above the heap-stack boundary, but nothing prevents the stack from growing below the boundary. This, of course, presents the possibility of the stack overwriting a dynamically allocated buffer. The function **rsvstk** can be used to change the heap-stack boundary.

Overview of Error Processing

This section discusses error processing which relates to the global integer **errno.** This variable is modified by the standard i/o, unbuffered i/o, and scientific (eg, **sin, sqrt**) functions as part of their error processing.

The handling of floating point exceptions (overflow, underflow, and division by zero) is discussed in chapter VIII.

When a standard i/o, unbuffered i/o, or scientific function detects an error, it sets a code in **errno** which describes the error. If no error occurs, the scientific functions don't modify **errno**. If not error occurs, the i/o functions may or may not modify **errno.**

Also, when an error occurs,

- o a standard i/o function returns -1 and sets an error flag for the stream on which the error occurred;
- o an unbuffered i/o function returns -1;
- o a scientific function returns an arbitrary value.

When performing scientific calculations, a program can check **errno** for errors as each function is called. Alternatively, since **errno** is modified only when an error occurs, **errno** can be checked only after a sequence of operations; if it's non-zero, then an error has occurred at some point in the sequence. This latter technique can only be used when no i/o operations occur during the sequence of scientific function calls.

Since **errno** may be modified by an i/o function even if an error didn't occur, a program can't perform a sequence of i/o operations and then check **errno** afterwards to detect an error. Programs performing unbuffered i/o must check the result of each i/o call for an error.

Programs performing standard i/o operations cannot, following a sequence of standard i/o calls, check **errno** to see if an error occurred. However, associated with each open stream is an error flag. This flag is set when an error occurs on the stream and remains set until the stream is closed or the flag is explicitly reset. Thus a program can perform a sequence of standard i/o operations on a stream and then check the stream's error flag. For more details, see the standard i/o overview section.

The following table lists the values which may be placed in **errno**. In addition, positive values may be set in **errno** following an i/o operation; these are error codes returned by the CP/M bdos.

The symbolic values listed below are defined in the header file **errno.h**.

#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 requested
#define ENOMEM -4	insufficient memory for requested operation
#define EEXIST -5	file already exists on creat request
#define EINVAL -6	invalid argument
#define ENFILE -7	exceeded maximum number of open files
#define EMFILE -8	exceeded maximum number of file descriptors
#define ENOTTY -9	ioctl attempted on non-console
#define EACCES -10	invalid access request
#define ERANGE -20	math function value can't be computed
#define EDOM -21	invalid argument to math function

System Independent Functions

This subchapter describes in detail the functions which are common to all Aztec C packages.

The chapter is divided into sections, each of which describes a group of related functions. Each section has a name, and the sections are ordered alphabetically by name. Following this introduction is a cross reference which lists each function and the name of the section in which it is described.

A section is organized into the following subsections:

TITLE

Lists the name of the section, a phrase which is intended to catagorize the functions described in the section, and one or more letters in parentheses which specify the libraries containing the section's functions.

The letters which may appear in parentheses and their corresponding libraries are:

C	**c.lib**
M	**m.lib**

On some systems, the actual library name may be a variant on the name given above. For example, on TRSDOS, the libraries are named c/lib and m/lib.

SYNOPSIS

Indicates the types of arguments that the functions described in the section require, and the values they return. For example, the function **atof** converts character strings into double precision numbers. It is listed in the synopsis as

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

This means that **atof()** returns a value of type **double** and requires as an argument a pointer to a character string. Since **atof** returns a non-integer value, prior to use of the function it must be declared:

```
double atof();
```

The notation

#include "header.h"

at the beginning of a synopsis indicates that such a statement should appear at the beginning of any program calling one of the functions described in the section.

On Radio Shack systems, a header file can use either a period or a slash to separate the filename from the extent.

That is, the include statement can be as listed above, or

```
#include "header/h"
```

DESCRIPTION

Describes the section's functions.

SEE ALSO

Lists relevant sections. A letter in parentheses may follow a section name. This specifies where the section is located: no letter means that the section is in the current subchapter; 'O' means that it's in the overview subchapter; 'S' means that it's in the system dependent subchapter.

DIAGNOSTICS

Describes the error codes that the section's functions may return. The section ERRORS presents an overview of error processing.

EXAMPLES

Gives examples on use of the section's functions.

Index to System Independent Functions

function	page
acos	SIN
agetc	GETC
aputc	PUTC
asin	SIN
atan	SIN
atan2	SIN
atof	ATOF
atoi	ATOF
atol	ATOF
calloc	MALLOC
ceil	FLOOR
clearerr	FERROR
close	CLOSE
cos	SIN
cosh	SINH
cotan	SIN
creat	CREAT
exp	EXP
fclose	FCLOSE
fdopen	FOPEN
feof	FERROR
ferror	FERROR
fflush	FCLOSE
fgets	GETS
fileno	FERROR
floor	FLOOR
fopen	FOPEN
format	PRINTF
fprintf	PRINTF
fputs	PUTS
fread	FREAD
free	MALLOC
freopen	FOPEN
frexp	FREXP
fscanf	SCANF
fseek	FSEEK
ftell	FSEEK
ftoa	ATOF
fwrite	FREAD
getc	GETC
getchar	GETC
gets	GETS
getw	GETW
index	STRING
ioctl	IOCTL
isalpha	CTYPE
isatty	IOCTL
ldexp	FREXP
log	EXP

log10	EXP
longjmp	SETJMP
lseek	LSEEK
malloc	MALLOC
movmem	MOVMEM
modf	FREXP
open	OPEN
pow	EXP
printf	PRINTF
putc	PUTC
putchar	PUTC
puterr	PUTC
puts	PUTS
putw	PUTC
ran	RAN
read	READ
rename	RENAME
rindex	STRING
scanf	SCANF
setbuf	SETBUF
setjmp	SETJMP
setmem	MOVMEM
sin	SIN
sinh	SINH
sprintf	PRINTF
sqrt	EXP
sscanf	SCANF
strcat	STRING
strcmp	STRING
strcpy	STRING
strlen	STRING
strncat	STRING
strncmp	STRING
strncpy	STRING
swapmem	MOVMEM
tan	SIN
tanh	SINH
tolower	TOUPPER
toupper	TOUPPER
ungetc	UNGETC
unlink	UNLINK
write	WRITE

Function Definitions

NAME

atof, atoi, atol - convert ASCII to numbers
ftoa - convert floating point to ASCII

SYNOPSIS

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

atoi(cp)
char *cp;

long atol(cp)
char *cp;

ftoa(val, buf, precision, type)
double val;
char *buf;
int precision, type;
```

DESCRIPTION

atof, **atoi**, and **atol** convert a string of text characters pointed at by the argument **cp** to double, integer, and long representations, respectively.

atof recognizes a string containing leading blanks and tabs, which it skips, then an optional sign, then a string of digits optionally containing a decimal point, then an optional 'e' or 'E' followed by an optionally signed integer.

atoi and **atol** recognize a string containing leading blanks and tabs, which are ignored, then an optional sign, then a string of digits.

ftoa converts a double precision floating point number to ASCII. **val** is the number to be converted and **buf** points to the buffer where the ASCII string will be placed. **precision** specifies the number of digits to the right of the decimal point. **type** specifies the format: 0 for "E" format, 1 for "F" format, 2 for "G" format.

atof and **ftoa** are in the library m.lib; the other functions are in c.lib.

NAME

close - close a device or file

SYNOPSIS

close(fd)
int fd;

DESCRIPTION

close closes a device or disk file which is opened for unbuffered i/o.

The parameter **fd** is the file descriptor associated with the file or device. If the device or file was explicitly opened by the program by calling **open** or **creat**, **fd** is the file descriptor returned by **open** or **creat**.

close returns 0 as its value if successful.

SEE ALSO

UNBUFFERED I/O (O), ERRORS (O)

DIAGNOSTICS

If **close** fails, it returns -1 and sets an error code in the global integer **errno**.

NAME

creat - create a new file

SYNOPSIS

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

DESCRIPTION

creat creates a file and opens it for unbuffered, write-only access. If the file already exists, it is truncated so that nothing is in it (this is done by erasing and then creating the file).

creat returns as its value an integer called a "file descriptor". Whenever a call is made to one of the unbuffered i/o functions to access the file, its file descriptor must be included in the function's parameters.

The parameter **name** is a pointer to a character string which is the name of the device or file to be opened. See the I/O overview section for details.

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.

SEE ALSO

UNBUFFERED I/O (O), ERRORS (O)

DIAGNOSTICS

If **creat** fails, it returns -1 as its value and sets a code in the global integer **errno**.

NAME

isalpha, isupper, islower, isdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii
- character clasification functions

SYNOPSIS

#include "ctype.h"

isalpha(c)

. . .

DESCRIPTION

These macros classify ASCII-coded integer values by table lookup, returning nonzero if the integer is in the catagory, zero otherwise. **isascii** is defined for all integer values. The others are defined only when **isascii** is true and on the single non-ASCII value EOF (-1).

isalpha	**c** is a letter
isupper	**c** is an upper case letter
islower	**c** is a lower case letter
isdigit	**c** is a digit
isalnum	**c** is an alphanumeric character
isspace	**c** is a space, tab, carriage return, newline, or formfeed
ispunct	**c** is a punctuation character
isprint	**c** is a printing character, valued 0x20 (space) through 0x7e (tilde)
iscntrl	**c** is a delete character (0xff) or ordinary control character (value less than 0x20)
isascii	**c** is an ASCII character, code less than 0x100

NAME

exponential, logarithm, power, square root functions:

exp, log, log10, pow, sqrt

SYNOPSIS

```
double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

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

double sqrt(x)
double x;
```

DESCRIPTION

exp returns the exponential function of x.

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

pow returns x ** y (x to the y-th power).

sqrt returns the square root of x.

SEE ALSO

ERRORS

DIAGNOSTICS

If a function can't perform the computation, it sets an error code in the global integer **errno** and returns an arbitrary value; otherwise it returns the computed value without modifying **errno**. The symbolic values which a function can place in **errno** are EDOM, signifying that the argument was invalid, and ERANGE, meaning that the value of the function couldn't be computed. These codes are defined in the file **errno.h**.

The following table lists, for each function, the error codes that can be returned, the function value for that error, and the meaning of the error. The symbolic values are defined in the file math.h.

function	error	f(x)	Meaning
exp	ERANGE	HUGE	x > LOGHUGE
"	ERANGE	0.0	x < LOGTINY
log	EDOM	-HUGE	x <= 0
log10	EDOM	-HUGE	x <= 0
pow	EDOM	-HUGE	x < 0, x=y=0
"	ERANGE	HUGE	y*log(x)>LOGHUGE
"	ERANGE	0.0	y*log(x)<LOGTINY
sqrt	EDOM	0.0	x < 0.0

NAME

fclose, fflush - close or flush a stream

SYNOPSIS

```
#include "stdio.h"

fclose(stream)
FILE *stream;

fflush(stream)
FILE *stream;
```

DESCRIPTION

fclose informs the system that the user's program has completed its buffered i/o operations on a device or file which it had previously opened (by calling **fopen**). **fclose** releases the control blocks and buffers which it had allocated to the device or file. Also, when a file is being closed, **fclose** writes any internally buffered information to the file.

fclose is called automatically by **exit**.

fflush causes any buffered information for the named output stream to be written to that file. The stream remains open.

If **fclose** or **fflush** is successful, it returns 0 as its value.

SEE ALSO

STANDARD I/O (O)

DIAGNOSTICS

If the operation fails, -1 is returned, and an error code is set in the global integer **errno**.

NAME

feof, ferror, clearerr, fileno - stream status inquiries

SYNOPSIS

```
#include "stdio.h"

feof(stream)
FILE *stream;

ferror(stream)
FILE *stream;

clearerr(stream)
FILE *stream;

fileno(stream)
FILE *stream;
```

DESCRIPTION

feof returns non-zero when end-of-file is reached on the specified input stream, and zero otherwise.

ferror returns non-zero when an error has occurred on the specified stream, and zero otherwise. Unless cleared by **clearerr**, the error indication remains set until the stream is closed.

clearerr resets an error indication on the specified stream.

fileno returns the integer file descriptor associated with the stream.

These functions are defined as macros in the file stdio.h.

SEE ALSO

STANDARD I/O (O)

NAME

fabs, floor, ceil - absolute value, floor, ceiling routines

SYNOPSIS

```
double floor(x)
double x;

double ceil(x)
double x;

double fabs(x)
double x;
```

DESCRIPTION

fabs returns the absolute value of **x**.

floor returns the largest integer not greater than **x**.

ceil returns the smallest integer not less than **x**.

NAME

fopen, freopen, fdopen - open a stream

SYNOPSIS

```
#include "stdio.h"

FILE *fopen(filename, mode)
char *filename, *mode;

FILE *freopen(filename, mode, stream)
char *filename, *mode;
FILE *stream;

FILE *fdopen(fd, mode)
char *mode;
```

DESCRIPTION

These functions prepare a device or disk file for access by the standard i/o functions; this is called "opening" the device or file. A file or device which has been opened by one of these functions is called a "stream".

If the device or file is successfully opened, these functions return a pointer, called a "file pointer" to a structure of type FILE. This pointer is included in the list of parameters to buffered i/o functions, such as **getc** or **putc**, which the user's program calls to access the stream.

fopen is the most basic of these functions: it simply opens the device or file specified by the **filename** parameter for access specified by the **mode** parameter. These parameters are described below.

freopen substitutes the named device or file for the device or file which was previously associated with the specified stream. It closes the device or file which was originally associated with the stream and returns **stream** as its value. It is typically used to associate devices and files with the preopened streams **stdin, stdout,** and **stderr.**

fdopen opens a device or file for buffered i/o which has been previously opened by one of the unbuffered open functions **open** and **creat.** It returns as it's value a FILE pointer.

fdopen is passed the file descriptor which was returned when the device or file was opened by **open** or **creat.** It's also passed the **mode** parameter specifying the type of access desired. **mode** must agree with the mode of the open file.

The parameter **filename** is a pointer to a character string which is the name of the device or file to be opened. For details, see the I/O overview section.

mode points to a character string which specifies how the user's program intends to access the stream. The choices are as follows:

mode	meaning
"r"	Open for reading only. If a file is opened, it is positioned at the first character in it. If the file or device does not exist, NULL is returned.
"w"	Open for writing only. If a file is opened which already exists, it is truncated to zero length. If the file does not exist, it is created.
"a"	Open for appending. The calling program is granted write-only access to the stream. The current file position is the character after the last character in the file. If the file does not exist, it is created.
"x"	Open for writing. The file must not previously exist. This option is not supported by Unix.
"r+"	Open for reading and writing. Same as "r", but the stream may also be written to.
"w+"	Open for writing and reading. Same as "w", but the stream may also be read; different from "r+" in the creation of a new file and loss of any previous one.
"a+"	Open for appending and reading. Same as "a", but the stream may also be read; different from "r+" in file positioning and file creation.
"x+"	Open for writing and reading. Same as "x" but the file can also be read.

On systems which don't keep track of the last character in a file (for example CPM and Apple DOS), not all files can be correctly positioned when opened in append mode. See the I/O overview section for details.

SEE ALSO

I/O (O), STANDARD I/O (O)

DIAGNOSTICS

If the file or device cannot be opened, NULL is returned and an error code is set in the global integer **errno**.

EXAMPLES

The following example demonstrates how **fopen** can be used in a program.

```
#include "stdio.h"

main(argc,argv)
char **argv;
{
     FILE *fopen(), *fp;

     if ((fp = fopen(*argv[1], *argv[2])) == NULL)  {
          printf("You asked me to open %s",*argv[1]);
          printf("in the %s mode", *argv[2]);
          printf("but I can't!\n");
     }  else
          printf("%s is open\n", *argv[1]);
}
```

Here is a program which uses **freopen**:

```
#include "stdio.h"
main()
{
     FILE *fp;
     fp = freopen("dskfile", "w+", stdout);
     printf("This message is going to dskfile\n");
}
```

Here is a program which uses **fdopen**:

```
#include "stdio.h"

dopen_it(fd)
int fd;   /* value returned by previous call to open */
{
     FILE *fp;

     if ((fp = fdopen(fd, "r+")) == NULL)
          printf("can't open file for r+\n");
     else
          return(fp);
}
```

NAME

fread, fwrite - buffered binary input/output

SYNOPSIS

```
#include "stdio.h"

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

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

DESCRIPTION

fread performs a buffered input operation and **fwrite** a buffered write operation to the open stream specified by the parameter **stream.**

buffer is the address of the user's buffer which will be used for the operation.

The function reads or writes **count** items, each containing **size** bytes, from or to the stream.

fread and **fwrite** perform i/o using the functions **getc** and **putc**; thus, no translations occur on the data being transferred.

The function returns as its value the number of items actually read or written.

SEE ALSO

STANDARD I/O (O), FOPEN, ERRORS (O), FERROR

DIAGNOSTICS

fread and **fwrite** return 0 upon end of file or error. The functions **feof** and **ferror** can be used to distinguish between the two. In case of an error, the global integer **errno** contains a code defining the error.

EXAMPLE

This is the code for reading ten integers from file 1 and writing them again to file 2. It includes a simple check that there are enough two-byte items in the first file:

```
#include "stdio.h"

main()
{
     FILE *fp1, *fp2, *fopen();
     char *buf;
     int size = 2, count = 10;

     fp1 = fopen("file1","r");
     fp2 = fopen("file2","w");
     if (fread(buf, size, count, fp1) !=  count)
         printf("Not enough integers in file1\n");
     fwrite(buf, size, count, fp2);
}
```

NAME

frexp, ldexp, modf - build and unbuild real numbers

SYNOPSIS

```
double frexp(value, eptr)
double value;
int *eptr;

double ldexp(value, exp)
double value;

double modf(value, iptr)
double value, *iptr;
```

DESCRIPTION

frexp computes for its argument, **arg**, a double quantity x and an integer quantity n such that value = x*2**n. x is returned as the value of **frexp**, and n is returned in the integer field pointed at by eptr.

ldexp returns the double quantity value*2**exp.

modf returns as its value the positive fractional part of value and stores the integer part in the double field pointed at by **iptr**.

NAME

fseek, ftell - reposition a stream

SYNOPSIS

```
#include "stdio.h"

int fseek(stream, offset, origin)
FILE *stream;
long offset;
int origin;

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

DESCRIPTION

fseek sets the "current position" of a file which has been opened for buffered i/o. The current position is the byte location at which the next input or output operation will begin.

stream is the stream identifier associated with the file, and was returned by **fopen** when the file was opened.

offset and **origin** together specify the current position: the new position is at the signed distance **offset** bytes from the beginning, current position, or end of the file, depending on whether **origin** is 0, 1, or 2, respectively.

offset can be positive or negative, to position after or before the specified origin, respectively, with the limitation that you can't seek before the beginning of the file.

For some operating systems (for example, CPM and Apple DOS) a file may not be able to be correctly positioned relative to its end. See the overview sections I/O and STANDARD I/O for details.

If fseek is successful, it will return zero.

ftell returns the number of bytes from the beginning to the current position of the file associated with **stream.**

SEE ALSO

STANDARD I/O (O), I/O (O), LSEEK

DIAGNOSTICS

fseek will return -1 for improper seeks. In this case, an error code is set in the global integer **errno.**

EXAMPLE

The following routine is equivalent to opening a file in "a+" mode:

```
a_plus(filename)
char *filename;
{
	FILE *fp, *fopen();

	if ((fp = fopen(filename, r+)) == NULL)
		fp = fopen(filename, w+);
	fseek(fp, 0L, 2);    /* position 1 byte past
	                          last character */
}
```

To set the current position back 5 characters before the present current position, the following call can be used:

```
		fseek(fp, -5L, 1)
```

NAME

getc, agetc, getchar, getw

SYNOPSIS

```
#include "stdio.h"

int getc(stream)
FILE *stream;

int agetc(stream)           /* non-Unix function */
FILE *stream;

int getchar()

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

DESCRIPTION

getc returns the next character from the specified input stream.

agetc is used to access files of text. It generally behaves like **getc** and returns the next character from the named input stream. It differs from **getc** in the following ways:

- o it translates end-of-line sequences (eg, carriage return on Apple DOS; carriage return-line feed on CPM) to a single newline ('\n') character.
- o it translates an end-of-file sequence (eg, a null character on Apple DOS; a control-z character on CPM) to EOF;
- o it ignores null characters ('\0') on all systems except Apple DOS;
- o the most significant bit of each character returned is set to zero.

agetc is not a Unix function. It is, however, provided with all Aztec C packages, and provides a convenient, system-independent way for programs to read text.

getchar returns text characters from the standard input stream (stdin). It is implemented as the call **agetc(stdin).**

getw returns the next word from the specified input stream. It returns EOF (-1) upon end-of-file or error, but since that is a good integer value, **feof** and **ferror** should be used to check the success of **getw.** It assumes no special alignment in the file.

SEE ALSO

I/O (O), STANDARD I/O (O), FOPEN, FCLOSE

DIAGNOSTICS

These functions return EOF (-1) at end of file or if an error occurs. The functions **feof** and **ferror** can be used to distinguish the two. In the latter case, an error code is set in the global integer **errno.**

NAME

gets, fgets - get a string from a stream

SYNOPSIS

```
#include "stdio.h"

char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
FILE *stream;
```

DESCRIPTION

gets reads a string of characters from the standard input stream, stdin, into the buffer pointed by **s.** The input operation terminates when either a newline character (\n) or end of file is encountered.

fgets reads characters from the specified input stream into the buffer pointer at by **s** until either (1) **n**-1 characters have been read, (2) a newline character (\n) is read, or (3) end of file or an error is detected on the stream.

Both functions return **s**, except as noted below.

gets and **fgets** differ in their handling of the newline character: **gets** doesn't put it in the caller's buffer, while **fgets** does. This is the behavior of these functions under UNIX.

These functions get characters using **agetc**; thus they can only be used to get characters from devices and files which contain text characters.

SEE ALSO

I/O (O), STANDARD I/O (O), FERROR

DIAGNOSTICS

gets and **fgets** return the pointer NULL (0) upon reaching end of file or detecting an error. The functions **feof** and **ferror** can be used to distinguish the two. In the latter case, an error code is placed in the global integer **errno**.

NAME

ioctl, isatty - device i/o utilities

SYNOPSIS

```
#include "sgtty.h"

ioctl(fd, cmd, stty)
struct sgttyb *stty;

isatty(fd)
```

DESCRIPTION

ioctl sets and determines the mode of the console.

For more details on **ioctl**, see the overview section CONSOLE I/O.

isatty returns non-zero if the file descriptor **fd** is associated with the console, and zero otherwise.

SEE ALSO

CONSOLE I/O (O)

NAME

lseek - change current position within file

SYNOPSIS

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

DESCRIPTION

lseek sets the current position of a file which has been opened for unbuffered i/o. This position determines where the next character will be read or written.

fd is the file descriptor associated with the file.

The current position is set to the location specified by the offset and origin parameters, as follows:

> If **origin** is 0, the current position is set to **offset** bytes from the beginning of the file.
>
> If **origin** is 1, the current position is set to the current position plus **offset**.
>
> If **origin** is 2, the current position is set to the end of the file plus **offset**.

The offset can be positive or negative, to position after or before the specified origin, respectively.

Positioning of a file relative to its end (that is, calling **lseek** with **origin** set to 2) cannot always be correctly done on all systems (for example, CPM and Apple DOS). See the section entitled I/O for details.

If **lseek** is successful, it will return the new position in the file (in bytes from the beginning of the file).

SEE ALSO

UNBUFFERED I/O (O), I/O (O)

DIAGNOSTICS

If **lseek** fails, it will return -1 as its value and set an error code in the global integer **errno**. **errno** is set to EBADF if the file descriptor is invalid. It will be set to EINVAL if the offset parameter is invalid or if the requested position is before the beginning of the file.

EXAMPLES

1. To seek to the beginning of a file:

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

lseek will return the value zero (0) since the current position in the file is character (or byte) number zero.

2. To seek to the character following the last character in the file:

```
pos = lseek(fd, 0L, 2);
```

   The variable, pos, will contain the current position of the end of file, plus one.

3. To seek backward five bytes:

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

   The parameter, 1, sets the origin at the current position in the file. The offset is -5. The new position will be the origin plus the offset. So the effect of this call is to move backward a total of five characters.

4. To skip five characters when reading in a file:

```
read(fd, buf, count);
lseek(fd, 5L, 1);
read(fd, buf, count);
```

NAME

malloc, free, calloc, - memory allocation

SYNOPSIS

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

char *calloc(nelem, elemsize)
unsigned nelem, elemsize;

char *realloc(ptr, size)
char *ptr;
unsigned size;

free(ptr)
char *ptr;
```

DESCRIPTION

These functions are used to allocate memory from the "heap", that is, the section of memory available for dynamic storage allocation.

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

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

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

free deallocates a block of memory which was previously allocated by **malloc**, **calloc**, or **realloc**; this space is then available for reallocation. The argument **ptr** to **free** is a pointer to the block.

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

SEE ALSO

MEMORY USAGE (O), BREAK (S)

DIAGNOSTICS

malloc, calloc and **realloc** return a null pointer (0) if there is no available block of memory.

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

NAME

movmem, setmem, swapmem

SYNOPSIS

```
movmem(src, dest, length)          /* non-Unix function */
char *src, *dest;
int length;

setmem(area, length, value)        /* non-Unix function */
char *area;

swapmem(s1, s2, len)               /* non-Unix function */
char *s1, *s2;
```

DESCRIPTION

movmem copies **length** characters from the block of memory pointed at by **src** to that pointed at by **dest.**

movmem copies in such a way that the resulting block of characters at **dest** equals the original block at **src.**

setmem sets the character **value** in each byte of the block of memory which begins at **area** and continues for length bytes.

swapmem swaps the blocks of memory pointed at by **s1** and **s2.** The blocks are **len** bytes long.

NAME

open

SYNOPSIS

```
#include "fcntl.h"

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

DESCRIPTION

This function will open a device or file for unbuffered i/o. It returns an integer value called a file descriptor which is used to identify the file or device in subsequent calls to unbuffered i/o functions.

The parameter **name** is a pointer to a character string which is the name of the device or file to be opened. For details, see the overview section I/O.

The parameter **mode** specifies how the user's program intends to access the file. The choices are as follows:

mode	meaning
O_RDONLY	read only
O_WRONLY	write only
O_RDWR	read and write
O_CREAT	Create file, then open it
O_TRUNC	Truncate file, then open it
O_EXCL	Cause open to fail if file already exists; used with O_CREAT
O_APPEND	Position file for appending data

These open modes are integer constants defined in the files fcntl.h. Although the true values of these constants can be used in a given call to open, use of the symbolic names ensures compatibility with UNIX and other systems.

The calling program must specify the type of access desired by including exactly one of O_RDONLY, O_WRONLY, and O_RDWR in the mode parameter. The three remaining values are optional. They may be included by adding them to the mode parameter, as in the examples below.

By default, the open will fail if the file to be opened does not exist. To cause the file to be created when it does not already exist, specify the O_CREAT option. If O_EXCL is given in addition to O_CREAT, the open will fail if the file already exists; otherwise, the file is created.

If the O_TRUNC option is specified, the file will be truncated so that nothing is in it. The truncation is performed by simply erasing the file, if it exists, and then creating it. So it is not an error to use this option when the file does not exist.

Note that when O_TRUNC is used, O_CREAT is not needed.

If O_APPEND is specified, the current position for the file (that is, the position at which the next data transfer will begin) is set to the end of the file. For systems which don't keep track of the last character written to a file (for example, CPM and Apple DOS), this positioning cannot always be correctly done. See the I/O overview section for details. Also, this option is not supported by UNIX.

If open does not detect an error, it returns an integer called a "file descriptor." This value is used to identify the open file during unbuffered i/o operations. The file descriptor is very different from the file pointer which is returned by fopen for use with buffered i/o functions.

SEE ALSO

I/O (O), UNBUFFERED I/O (O), ERRORS (O)

DIAGNOSTICS

If open encounters an error, it returns -1 and sets the global integer **errno** to a symbolic value which identifies the error.

EXAMPLES

1. To open the file, testfile, for read-only access:

```
fd = open("testfile", O_RDONLY);
```

   If testfile does not exist **open** will just return -1 and set **errno** to ENOENT.

2. To open the file, subl, for read-write access:

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

   If the file does not exist, it will be created and then opened.

3. The following program opens a file whose name is given on the command line. The file must not already exist.

```
main(argc, argv)
char **argv;
{
     int fd;

     fd = open(*++argv, O_WRONLY+O_CREAT+O_EXCL);
     if (fd = -1)  {
          if (errno == EEXIST)
               printf("file already exists\n");
          else if (errno == ENOENT)
               printf("unable to open file\n");
          else
               printf("open error\n");
}
```

NAME

printf, fprintf, sprintf
- formatted output conversion functions

SYNOPSIS

```
#include "stdio.h"

printf(fmt [,arg] ...)
char *fmt;

fprintf(stream, fmt [,arg] ...)
FILE *stream;
char *fmt;

sprintf(buffer, fmt [,arg] ...)
char *buffer, *fmt;

format(func, fmt, argptr)
int (*func)();
char *fmt;
unsigned *argptr;
```

DESCRIPTION

These functions convert and format their arguments (**arg** or **argptr**) according to the format specification **fmt**. They differ in what they do with the formatted result:

> **printf** outputs the result to the standard output stream, **stdout**;
>
> **fprintf** outputs the result to the stream specified in its first argument, **stream**;
>
> **sprintf** places the result in the buffer pointed at by its first argument, **buffer**, and terminates the result with the null character, '\0'.
>
> **format** calls the function **func** with each character of the result.

These functions are in both **c.lib** and **m.lib**, the difference being that the c.lib versions don't support floating point conversions. Hence, if floating point conversion is required, the m.lib versions must be used. If floating point conversion isn't required, either version can be used. To use m.lib's version, m.lib must be specified before c.lib at the time the program is linked.

The character string pointed at by the **fmt** parameter, which directs the print functions, contains two types of items: ordinary characters, which are simply output, and conversion specifications, each of which causes the conversion and output of the next successive **arg**.

A conversion specification begins with the character % and continues with:

- o an optional minus sign (-) which specifies left adjustment of the converted value in the output field;
- o an optional digit string specifying the 'field width' for the conversion. If the converted value has fewer characters than this, enough blank characters will be output to make the total number of characters output equals the field width. If the converted value has more characters than the field width, it will be truncated. The blanks are output before or after the value, depending on the presence or absence of the left-adjustment indicator. If the field width digits have a leading 0, 0 is used as a pad character rather than blank.
- o an optional period, '.', which separates the field width from the following field;
- o an optional digit string specifying a precision; for floating point conversions, this specifies the number of digits to appear after the decimal point; for character string conversions, this specifies the maximum number of characters to be printed from a string;
- o optionally, the character l, which specifies that a conversion which normally is performed on an **int** is to be performed on a **long**. This applies to the d, o, and x conversions.
- o a character which specifies the type of conversion to be performed.

A field width or precision may be * instead of a number, specifying that the next available **arg**, which must be an **int**, supplies the field width or precision.

The conversion characters are:

d, o, or x
The **int** in the corresponding **arg** is converted to decimal, octal, or hexadecimal notation, respectively, and output;

u The unsigned integer **arg** is converted to decimal notation;

c The character **arg** is output. Null characters are ignored;

s The characters in the string pointed at by **arg** are output until a null character or the number of characters indicated by the precision is reached. If the precision is zero or missing, all characters in the string, up the terminating null, are output;

f The float or double **arg** is converted to decimal notation in the style '[-]ddd.ddd'. The number of d's after the decimal point is equal to the precision given in the conversion specification. If the precision is missing, it defaults to six digits. If the precision is explicitly 0, the decimal point is also not printed.

e The float or double **arg** is converted to the style '[-]d.ddde[-]dd', where there is one digit before the decimal point and the number after is equal to the precision given. If the precision is missing, it defaults to six digits.

g The float or double **arg** is printed in style d, f, or e, whichever gives full precision in minimum space.

% output a **%**. No argument is converted.

SEE ALSO

STANDARD I/O (O)

EXAMPLES

1. The following program fragment:

```
char *name; float amt;
printf("your total, %s, is $%f\n", name, amt);
```

will print a message of the form

```
your total, Alfred, is $3.120000
```

Since the precision of the %f conversion wasn't specified, it defaulted to six digits to the right of the decimal point.

2. This example modifies example 1 so that the field width for the %s conversion is three characters, and the field width and precision of the %f conversion are 10 and 2, respectively. The %f conversion will also use 0 as a pad character, rather than blank.

```
char *name; float amt;
printf("your total, %3s, is $%10.2f\n", name, amt);
```

3. This example modifies example 2 so that the field width of the %s conversion and the precision of the %f conversion are taken from the variables nw and ap:

```
char *name; float amt; int nw, ap;
printf("your total %*s,is $%10.*f\n", nw,name,ap,amt);
```

NAME

putc, aputc, putchar, putw, puterr
- put character or word to a stream

SYNOPSIS

```
#include "stdio.h"

putc(c, stream)
char c;
FILE *stream;

aputc(c, stream)                    /* non-Unix function */
char c;
FILE *stream;

putchar(c)
char c;

putw(w,stream)
FILE *stream;

puterr(c)                           /* non-Unix function */
char c;
```

DESCRIPTION

putc writes the character **c** to the named output stream. It returns **c** as its value.

aputc is used to write text characters to files and devices. It generally behaves like **putc**, and writes a single character to a stream. It differs from **putc** as follows:

- o when a newline character is passed to **aputc**, an end-of-line sequence (eg, carriage return followed by line feed on CPM, and carriage return only on Apple DOS) is written to the stream;
- o the most significant bit of a character is set to zero before being written to the stream.

aputc is not a Unix function. It is, however, supported on all Aztec C systems, and provides a convenient, system-independent way for a program to write text.

putchar writes the character **c** to the standard output stream, stdout. It's identical to **aputc(c, stdout)**.

putw writes the word **w** to the specified stream. It returns **w** as its value. **putw** neither requires nor causes special alignment in the file.

puterr writes the character **c** to the standard error stream, stderr. It's identical to **aputc(c, stderr)**. It is not a Unix function.

SEE ALSO

STANDARD I/O

DIAGNOSTICS

These functions return EOF (-1) upon error. In this case, an error code is set in the global integer **errno.**

NAME

puts, fputs - put a character string on a stream

SYNOPSIS

```
#include "stdio.h"

puts(s)
char *s;

fputs(s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

puts writes the null-terminated string **s** to the standard output stream, stdout, and then an end-of-line sequence. It returns a non-negative value if no errors occur.

fputs copies the null-terminated string **s** to the specified output stream. It returns 0 if no errors occur.

Both functions write to the stream using **aputc**. Thus, they can only be used to write text. See the PUTC section for more details on **aputc**.

Note that **puts** and **fputs** differ in this way: On encountering a newline character, **puts** writes an end-of-line sequence and **fputs** doesn't.

SEE ALSO

STANDARD I/O (O), PUTC

DIAGNOSTICS

If an error occurs, these functions return EOF (-1) and set an error code in the global integer **errno**.

NAME

qsort - sort an array of records in memory

SYNOPSIS

```
qsort(array, number, width, func)
char *array;
unsigned number;
unsigned width;
int (*func)();
```

DESCRIPTION

qsort sorts an array of elements using Hoare's Quicksort algorithm. **array** is a pointer to the array to be sorted; **number** is the number of record to be sorted; **width** is the size in bytes of each array element; **func** is a pointer to a function which is called for a comparison of two array elements.

func is passed pointers to the two elements being compared. It must return an integer less than, equal to, or greater than zero, depending on whether the first argument is to be considered less than, equal to, or greater than the second.

EXAMPLE

The Aztec linker, LN, can generate a file of text containing a symbol table for a program. Each line of the file contains an address at which a symbol is located, followed by a space, followed by the symbol name. The following program reads such a symbol table from the standard input, sorts it by address, and writes it to standard output.

```
#include "stdio.h"
#define MAXLINES 2000
#define LINESIZE 16
char *lines[MAXLINES];

main()
{
     int i,numlines, cmp();
     char buf[LINESIZE];

     for (numlines=0; numlines<MAXLINES; ++numlines){
          if (gets(buf) == NULL)
               break;
          lines[numlines] = alloc(LINESIZE);
          strcpy(lines[numlines], buf);
     }
     qsort(lines, numlines, 2, cmp);
     for (i = 0; i <numlines; ++i)
          printf("%s\n", lines[i]);
}

cmp(a,b)
char **a, **b;
{
     return strcmp(*a, *b);
}
```

NAME

ran - random number generator

SYNOPSIS

double ran()

DESCRIPTION

ran returns as its value a random number between 0.0 and 1.0.

NAME

read - read from device or file without buffering

SYNOPSIS

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

DESCRIPTION

read reads characters from a device or disk file which has been previously opened by a call to **open** or **creat.** In most cases, the information is read directly into the caller's buffer.

fd is the file descriptor which was returned to the caller when the device or file was opened.

buf is a pointer to the buffer into which the information is to be placed.

bufsize is the number of characters to be transfered.

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

If the returned value is zero, then end-of-file has been reached, immediately, with no bytes read.

SEE ALSO

UNBUFFERED I/O (O), OPEN, CLOSE

DIAGNOSTICS

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

NAME

rename - rename a disk file

SYNOPSIS

```
rename(oldname, newname)          /* non-Unix function */
char *oldname,*newname;
```

DESCRIPTION

rename changes the name of a file.

oldname is a pointer to a character array containing the old file name, and **newname** is a pointer to a character array containing the new name of the file.

If successful, **rename** returns 0 as its value; if unsuccessful, it returns -1.

If a file with the new name already exists, **rename** sets E_EXIST in the global integer **errno** and returns -1 as its value without renaming the file.

NAME

scanf, fscanf, sscanf - formatted input conversion

SYNOPSIS

```
#include "stdio.h"

scanf(format [,pointer] ...)
char *format;

fscanf(stream, format [,pointer] ...)
FILE *stream;
char *format;

sscanf(buffer, format [,pointer] ...)
char *buffer, *format;
```

DESCRIPTION

These functions convert a string or stream of text characters, as directed by the control string pointed at by the **format** parameter, and place the results in the fields pointed at by the **pointer** parameters.

The functions get the text from different places:

scanf gets text from the standard input stream, **stdin**;

fscanf gets text from the stream specified in its first parameter, **stream**;

sscanf gets text from the buffer pointed at by its first parameter, **buffer**.

The scan functions are in both c.lib and m.lib, the difference being that the c.lib versions don't support floating point conversions. Hence, if floating point conversion is required, the m.lib versions must be used. If floating point conversions aren't required, eithe version can be used. To use m.lib's version, m.lib must be specified before c.lib when the program is linked.

The control string pointed at by **format** contains the following 'control items':

o conversion specifications;

o 'white space' characters (space, tab newline);

o ordinary characters; that is, characters which aren't part of a conversion specification and which aren't white space.

A scan function works its way through a control string, trying to match each control item to a portion of the input stream or buffer. During the matching process, it fetches characters one at a time from the input. When a character is fetched which isn't appropriate for the control item being matched, the scan function pushes it back into the input stream or buffer and finishes processing the current control item. This pushing back frequently gives unexpected results when a stream is being accessed by other i/o functions, such as **getc**, as well as the scan function. The examples below demonstrate some of the problems that can occur.

The scan function terminates when it first fails to match a control item or when the end of the input stream or buffer is reached. It returns as its value the number of matched conversion specifications, or EOF if the end of the input stream or buffer was reached.

Matching 'white space' characters

When a white space character is encountered in the control string, the scan function fetches input characters until the first non-white-space character is read. The non-white-space character is pushed back into the input and the scan function proceeds to the next item in the control string.

Matching ordinary characters

If an ordinary character is encountered in the control string, the scan function fetches the next input character. input. If it matches the ordinary character, the scan function simply proceeds to the next control string item. If it doesn't match, the scan function terminates.

Matching conversion specifications

When a conversion specification is encountered in the control string, the scan function first skips leading white space on the input stream or buffer. It then fetches characters from the stream or buffer until encountering one that is inappropriate for the conversion specification. This character is pushed back into the input.

If the conversion specification didn't request assignment suppression (discussed below), the character string which was read is converted to the format specified by the conversion specification, the result is placed in the location pointed at by the current pointer argument, and the next pointer argument becomes current. The scan function then proceeds to the next control string item.

If assignment suppression was requested by the conversion specification, the scan function simply ignores the fetched input characters and proceeds to the next control item.

Details of input conversion

A conversion specification consists of:

- o the character '%', which tells the scan function that it has encountered a conversion specification;
- o optionally, the assignment suppression character '*';
- o optionally, a 'field width'; that is, a number specifying the maximum number of characters to be fetched for the conversion;
- o a conversion character, specifying the type of conversion to be performed.

If the assignment suppression character is present ina conversion specification, the scan function will fetch characters as if it was going to perform the conversion, ignore them, and proceed to the next control string item.

The following conversion characters are supported:

% a single '%' is expected in the input; no assignment is done.

d a decimal integer is expected; the input digit string is converted to binary and the result placed in the **int** field pointed at by the current **pointer** argument;

o an octal integer is expected; the corresponding **pointer** should point to an **int** field in which the converted result will be placed;

x a hexadecimal integer is expected; the converted value will be placed in the **int** field pointed at by the current **pointer** argument;

s a sequence of characters delimited by white space characters is expected; they, plus a terminating null character, are placed in the character array pointed at by the current **pointer** argument.

c a character is expected. It is placed in the **char** field pointed at by the current **pointer**. The normal skip over leading white space is not done; to read a single char after skipping leading white space, use '%1s'. The field width parameter is ignored, so this conversion can be used only to read a single character.

[a sequence of characters, optionally preceded by white space but not terminated by white space is expected. The input characters, plus a terminating null character, are placed in the character array pointed at by the current **pointer** argument. The left bracket is followed by:

- o optionally, a '^' or '~' character;
- o a set of characters;
- o a right bracket, ']'.

If the first character in the set isn't ^ or ~, the set specifies characters which are allowed; characters are fetched from the input until one is read which isn't in the set.

If the first character in the set is ^ or ~, the set specifies characters which aren't allowed; characters are fetched from the input until one is read which is in the set.

e
f a floating point number is expected. The input string is converted to floating point format and stored in the **float** field pointed at by the current **pointer** argument. The input format for floating point numbers consists of an optionally signed string of digits, possibly containing a decimal point, optionally followed by an exponent field consisting of an E or e followed by an optionally signed digit.

The conversion characters d, o, and x can be capitalized or preceded by **l** to indicate that the corresponding **pointer** is to a **long** rather than an **int.** Similarly, the conversion characters e and f can be capitalized or preceeded by **l** to indicate that the corresponding **pointer** is to a **double** rather than a **float.**

The conversion characters o, x, and d can be optionally preceded by **h** to indicate that the corresponding **pointer** is to a **short** rather than an **int.** Since **short** and **int** fields are the same in Aztec C, this option has no effect.

SEE ALSO

STANDARD I/O (O)

EXAMPLES

1. In this program fragment, **scanf** is used to read values for the int x, the float y, and a character string into the char array z:

```
int x; float y; char z[50];
scanf("%d%f%s", &x, &y, z);
```

The input line

```
32  75.36e-1 rufus
```

will assign 32 to x, 7.536 to y, and "rufus\0" to z. **scanf** will return 3 as its value, signifying that three conversion specifications were matched.

The three input strings must be delimited by 'white space' characters; that is, by blank, tab, and newline characters. Thus, the three values could also be entered on separate lines, with the white space character newline used to separate the values.

2. This example discusses the problems which may arise when mixing **scanf** and other input operations on the same stream.

   In the previous example, the character string entered for the third variable, z, must also be delimited by white space characters. In particular, it must be terminated by a space, tab, or newline character. The first such character read by **scanf** while getting characters for z will be 'pushed back' into the standard input stream. When another read of stdin is made later, the first character returned will be the white space character which was pushed back.

   This 'pushing back' can lead to unexpected results for programs that read stdin with functions in addition to **scanf**. Suppose that the program in the first example wants to issue a **gets** call to read a line from stdin, following the **scanf** to stdin. **scanf** will have left on the input stream the white space character which terminated the third value read by **scanf**. If this character is a newline, then **gets** will return a null string, because the first character it reads is the pushed back newline, the character which terminates **gets**. This is most likely not what the program had in mind when it called **gets**.

   It is usually unadvisable to mix **scanf** and other input operations on a single stream.

3. This example discusses the behavior of **scanf** when there are white space characters in the control string.

   The control string in the first example was "%d%f%s". It doesn't contain or need any white space, since **scanf**, when attempting to match a conversion specification, will skip leading white space. There's no harm in having white space before the %d, between the %d and %f, or between the %f and %s. However, placing a white space character after the %s can have unexpected results. In this case, **scanf** will, after having read a character string for z, keep reading characters until a non-white-space character is read. This forces the operator to enter, after the three values for x, y, and z, a non-white space character; until this is done, **scanf** will not terminate.

   The programmer might place a newline character at the end of a control string, mistakenly thinking that this will circumvent the problem discussed in example 2. One might think that **scanf** will treat the newline as it would an ordinary character in the control string; that is, that **scanf** will search for, and remove, the terminating newline character from the input stream after it has matched the z variable. However, this is incorrect, and should be remembered as a common misinterpretation.

4. **scanf** only reads input it can match. If, for the first example, the input line had been

```
32  rufus  75.36e-1
```

scanf would have returned with value 1, signifying that only one conversion specification had been matched. x would have the value 32, y and z would be unchanged. All characters in the input stream following the 32 would still be in the input stream, waiting to be read.

5. One common problem in using **scanf** involves mismatching conversion specifications and their corresponding arguments. If the first example had declared y to be a double, then one of the following statements would have been required:

```
scanf("%d%lf%s", &x, &y, z);
```
or
```
scanf("%d%F%s", &x, &y, z);
```

to tell **scanf** that the floating point variable was a double rather than a float.

6. Another common problem in using **scanf** involves passing **scanf** the value of a variable rather than its address. The following call to **scanf** is incorrrect:

```
int x; float y; char z[50];
scanf("%d%f%s", x, y, z);
```

scanf has been passed the value contained in x and y, and the address of z, but it requires the address of all three variables. The "address of" operator, &, is required as a prefix to x and y. Since z is an array, its address is automatically passed to **scanf**, so z doesn't need the & prefix, although it won't hurt if it is given.

7. Consider the following program fragment:

```
int x; float y; char z[50];
scanf("%2d%f%*d%[1234567890]", &x, &y, z);
```

When given the following input:

```
12345 678 90a65
```

scanf will assign 12 to x, 345.0 to y, skip '678', and place the string '90\0' in z. The next call to **getchar** will return 'a'.

NAME

setbuf - assign buffer to a stream

SYNOPSIS

```
#include "stdio.h"

setbuf(stream, buf)
FILE *stream;
char *buf;
```

DESCRIPTION

setbuf defines the buffer **buf** to be used for the specified **stream.**

The buffer must be BUFSIZ bytes, where BUFSIZ is defined in stdio.h.

setbuf must be called after the stream has been opened, but before any read or write operations to it are made.

If the user's program doesn't specify the buffer to be used for a stream, the standard i/o functions will dynamically allocate a buffer for the stream, by calling the function **malloc**, when the first read or write operation is made on the stream. Then, when the stream is closed, the dynamically allocated buffer is freed by calling **free**.

SEE ALSO

STANDARD I/O (O), MALLOC

NAME

setjmp, longjmp - non-local goto

SYNOPSIS

```
#include "setbuf.h"

setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;
```

DESCRIPTION

These functions are useful for dealing with errors encountered by the low-level functions of a program.

setjmp saves its stack environment in the memory block pointed at by **env** and returns 0 as its value.

longjmp causes execution to continue as if the last call to **setjmp** was just terminating with value **val. val** cannot be zero.

The parameter **env** is a pointer to a block of memory which can be used by **setjmp** and **longjmp**. The block must be defined using the typedef jmp_buf.

EXAMPLE

In the following example, the function **getall** builds a record pertaining to a customer and returns the pointer to the record if no errors were encountered and 0 otherwise.

getall calls other functions which actually build the record. These functions in turn call other functions, which in turn ...

getall defines, by calling **setjmp**, a point to which these functions can branch if an unrecoverable error occurs. The low level functions abort by calling **longjmp** with a non-zero value.

If a low level function aborts, execution continues in **getall** as if its call to **setjmp** had just terminated with a non-zero value. Thus by testing the value returned by **setjmp** **getall** can determine whether **setjmp** is terminating because a low level function aborted.

```
#include "jmpbuf.h"

jmp_buf envbuf; /* environment saved here by setjmp */

getall(ptr)
char *ptr; /* ptr to record to be built */
{
     if (setjmp(envbuf))
          /* a low level function has aborted */
          return 0;
     getfield1(ptr);
     getfield2(ptr);
     getfield3(ptr);
     return ptr;
}
```

Here's one of the low level functions:

```
getsubfld21(ptr)
char *ptr;
{
     ...
     if (error)
          longjmp(envbuf, -1);
     ...
}
```

NAME

trigonometric functions:

sin, cos, tan, asin, acos, atan, atan10

SYNOPSIS

```
double sin(x)
double x;

double cos(x)
double x;

double tan(x)
double x;

double cotan(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

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

DESCRIPTION

sin, cos, tan, and **cotan** return trigonometric functions of radian arguments.

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

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

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

atan2 returns the arc tangent of x/y in the range -pi to pi.

SEE ALSO

ERRORS

DIAGNOSTICS

If a trig function can't perform the computation, it returns an arbitrary value and sets a code in the global integer **errno**; otherwise, it returns the computed number, without modifying **errno**.

A function will return the symbolic value EDOM if the argument is invalid, and the value ERANGE if the function value can't be computed. EDOM and ERANGE are defined in the file errno.h.

The values returned by the trig functions when the computation can't be performed are listed below. The symbolic values are defined in math.h.

function	error	f(x)	meaning
sin	ERANGE	0.0	abs(x) > XMAX
cos	ERANGE	0.0	abs(x) > XMAX
tan	ERANGE	0.0	abs(x) > XMAX
cotan	ERANGE	HUGE	0<x< XMIN
cotan	ERANGE	-HUGE	-XMIN <x <0
cotan	ERANGE	0.0	abs(x) >= XMAX
asin	EDOM	0.0	abs(x) > 1.0
acos	EDOM	0.0	abs(x) > 1.0
atan2	EDOM	0.0	x = y = 0

NAME

sinh, cosh, tanh

SYNOPSIS

```
double sinh(x)
double x;

double cosh(x)
double x;

double tanh(x)
double x;
```

DESCRIPTION

These functions compute the hyperbolic functions of their arguments.

SEE ALSO

ERRORS

DIAGNOSTICS

If the absolute value of the argument to **sinh** or **cosh**is to greater than 348.6, the function sets the symbolic value ERANGE in the global integer **errno** and returns a huge value. This code is defined in the file **errno.h.**

If no error occurs, the function returns the computed value without modifying **errno**.

NAME

strcat, strncat, strcmp, strncmp, strcpy, strlen, index, rindex - string operations

SYNOPSIS

```
char *strcat(s1, s2)
char *s1, *s2;

char *strncat(s1, s2)
char *s1, *s2;

strcmp(s1, s2)
char *s1, *s2;

strncmp(s1, s2, n)
char *s1, s2;

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;

strlen(s)
char *s;

char *index(s, c)
char *s;

char *rindex(s, c)
char *s;
```

DESCRIPTION

These functions operate on null-terminated strings, as follows:

strcat appends a copy of string **s2** to string **s1. strncat** copies at most **n** characters. Both terminate the resulting string with the null character (\0) and return a pointer to the first character of the resulting string.

strcmp compares its two arguments and returns an integer greater than, equal, or less than zero, according as **s1** is lexicographically greater than, equal to, or less than **s2. strncmp** makes the same comparison but looks at **n** characters at most.

strcpy copies string **s2** to **s1** stopping after the null character has been moved. **strncpy** copies exactly **n** characters: if **s2** contains less than **n** characters, null characters will be appended to the resulting string until **n** characters have been moved; if **s2** contains **n** or more characters, only the first **n** will be moved, and the resulting string will not be null terminated.

strlen returns the number of characters which occur in **s** up to the first null character.

index returns a pointer to the first occurrance of the character **c** in string **s**, or zero if **c** isn't in the string.

rindex returns a pointer to the last occurrance of the character **c** in string **s**, or zero if **c** isn't in the string.

NAME

toupper, tolower

SYNOPSIS

toupper(c)

tolower(c)

#include "ctype.h"

_toupper(c)

_tolower(c)

DESCRIPTION

toupper converts a lower case character to upper case: if **c** is a lower case character, **toupper** returns its upper case equivalent as its value, otherwise **c** is returned.

tolower converts an upper case character to lowr case: if **c** is an upper case character **tolower** returns its lower case equivalent, otherwise **c** is returned.

toupper and **tolower** do not require the header file **ctype.h.**

_toupper and **_tolower** are macro versions of **toupper** and **tolower,** respectively. They are defined in ctype.h. The difference between the two sets of functions is that the macro versions will sometimes translate non-alphabetic characters, whereas the function versions don't.

NAME

ungetc - push a character back into input stream

SYNOPSIS

```
#include "stdio.h"

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

DESCRIPTION

ungetc pushes the character **c** back on an input stream. That character will be returned by the next **getc** call on that stream. **ungetc** returns **c** as its value.

Only one character of pushback is guaranteed. EOF cannot be pushed back.

SEE ALSO

STANDARD I/O

DIAGNOSTICS

ungetc returns EOF (-1) if the character can't be pushed back.

NAME

unlink

SYNOPSIS

```
unlink(name)
char *name;
```

DESCRIPTION

unlink erases a file.

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

unlink returns 0 if successful.

DIAGNOSTICS

unlink returns -1 if it couldn't erase the file and places a code in the global integer **errno** describing the error.

NAME

write

SYNOPSIS

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

DESCRIPTION

write writes characters to a device or disk which has been previously opened by a call to **open** or **creat.** The characters are written to the device or file directly from the caller's buffer.

fd is the file descriptor which was returned to the caller when the device or file was opened.

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

bufsize is the number of characters to be written.

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

SEE ALSO

UNBUFFERED I/O(O) , OPEN, CLOSE, READ

DIAGNOSTICS

If the operation is unsuccessful, **write** returns -1 and places a code in the global integer **errno.**

CP/M Functions

This subchapter describes functions which are available only to programs which are running on CP/M.

As with the subchapter describing the system indendent functions, this subchapter is divided into sections, with each section describing a group of related functions. A section is divided into subsections; for a discussion of these subsections, see the introduction to the system independent functions.

Following this introduction is an index to the CP/M functions, and then detailed descriptions of the functions themselves.

Index to CP/M Functions

functions	section
bdos	BDOS
bdoshl	BDOS
bios	BIOS
bioshl	BIOS
exit	EXIT
_exit	EXIT
fcbinit	FCBINIT
getusr	USER
in	PORT
out	PORT
rstusr	USER
rsvstk	BREAK
sbrk	BREAK
setusr	USER

NAME

bdos, bdoshl - issue call to CP/M bdos

SYNOPSIS

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

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

DESCRIPTION

These functions call the CP/M bdos with register pair BC set to **bc** and DE set to **de**.

bdos returns as its value the value which the CP/M bdos returned in register A.

bdoshl returns as its value the value which the CP/M bdos returned in register pair HL.

NAME

bios, bioshl - issue call to CP/M bios

SYNOPSIS

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

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

DESCRIPTION

These functions call the **n**'th entry into the CP/M bios with register pair BC set to **bc** and DE set to **de**. For example,

```
bios(0)
```

is a cold boot.

bios returns as its value the value which the CP/ bios returned in register A.

bioshl returns as its value the value which the CP/M bios returned in register pair HL.

NAME

sbrk, rsvstk

SYNOPSIS

sbrk(size)

rsvstk(size)

DESCRIPTION

sbrk increments an internal pointer by **size** bytes, and returns the value the pointer had on entry. The pointer initially points to the base of the heap.

rsvstk sets the heap-stack boundary **size** bytes below the current top of stack. The stack begins at the base of the CPM bdos. Unless **rsvstk** is called, the heap-stack boundary is 2048 bytes below the bdos base.

SEE ALSO

DYNAMIC BUFFER ALLOCATION (O), MALLOC

DIAGNOSTICS

sbrk returns -1 if the updated internal pointer would be above the heap-stack boundary. In this case, the internal pointer isn't modified.

NAME

execl, execv, execlp, execvp - execute a program

SYNOPSIS

```
execl(name, arg0, arg1, ..., argn, 0)
char *name, *arg0, *arg1, ..., *argn;

execv(name, argv)
char *name, *argv[];

execlp(name, arg0, arg1, ..., argn, 0)
char *name, *arg0, *arg1, ...*argn;

execvp(name, argv)
char *name, *argv[];
```

DESCRIPTION

The exec functions load another program from a disk file and transfer control to it. The new program is loaded over the currently executing program, so the exec functions never return to the caller.

name specifies the file from which the program is to be loaded. It has the same format as that of a file being opened for i/o; see the I/O overview section for details. If the extent for the filename isn't specified, it's assumed to be that of an executable file; for example, on CPM it is assumed to be ".COM" and on TRSDOS it's "/CMD".

The exec functions allow parameters to be passed to the called program. They do this by building a command line from the arguments passed to the exec function and then passing the command line to the new program. For **execl** and **execlp** the arguments are **arg1, arg1, ..., argn.** For **execv** and **execvp** the arguments are **argv[1], argv[2], ..., argv[n].** Note that **arg0** and **argv[0]** aren't passed to the new program; on Unix these strings are conventionally the name of the program being executed.

There are really only two different exec functions: **execl** and **execv. execlp** and **execvp** simply call **execl** and **execv,** respectively, and are provided for Unix compatibility.

execl and **execlp** are useful when a program knows exactly how many arguments are to be passed to the new program. **execv** and **execvp** are useful when the number of arguments to be passed aren't known until the call is made.

NAME

exit, _exit

SYNOPSIS

exit(n)

_exit(n)

DESCRIPTION

exit returns to the operating system, after closing any open files or devices. Buffered data for files and devices opened for standard i/o is written to disk, if necessary.

If **n** is non-zero, the file A:$$$.SUB is deleted, thus preventing the continuation of the submit file, if any, which initiated the program.

_exit is the same as **exit**, except that devices and files opened for standard i/o are never flushed.

NAME

fcbinit - initialize file control block

SYNOPSIS

```
#include "io.h"

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

DESCRIPTION

The file control block pointed at by **fcbptr** is initialized to zeroes and the file name pointed at by **name** is placed in it.

If the file name specifies a user number, **fcbinit** returns it as its value; otherwise it returns 255.

The file control block area pointed at by **fcbptr** must be at least 36 bytes long.

The parameter **name** is a pointer to a character string which is the name of the file. The format of the name is described in the I/O overview section.

NAME

in, out - port access functions

SYNOPSIS

in(port)

out(port, val)

DESCRIPTION

in reads a character from **port** and returns it as its value.

out writes the character in the least significant byte of **val** to **port**.

NAME

getusr, setusr, rstusr

SYNOPSIS

getusr()

setusr(user)

rstusr()

DESCRIPTION

getusr returns the current user number as its value.

setusr sets the user number to **user**. The user number which was active on entry to **setusr** is saved for subsequent use by **rstusr**.

rstusr resets the user number to the value which was saved during the last call to **setusr**.

SEE ALSO

I/O (O)

SECTION VIII

Technical Information

Data Formats

1. character

Characters are 8 bit ASCII.

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

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

2. pointer

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

location	contents in hex format
100	AB
101	F0

3. int, short

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

location	contents in hex format
100	FE
101	FF

4. long

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

5. float and double

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

The hexadecimal representation for a float with a value of 1.0 is:

```
41 01 00 00
```

A 255.0 would be:

```
41 FF 00 00
```

A 256.0 would be:

```
42 01 00 00
```

Floating Point Support

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

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

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

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

Floating Point Exceptions

When a C program requests that a floating point arithmetic operation be performed, a call will be made to functions in the floating point support software.

While performing the operation, these functions check for the occurence of the floating point exception conditions; namely, overflow, underflow, and division by zero. On return to the caller, the global integer **flterr** indicates whether an exception has occurred.

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

If the value of this integer is zero, no error occurred, and the value returned is the computed value of the operation. Otherwise, an error has occurred, and the value returned is arbitrary. The table lists the possible settings of flterr, and for each setting, the associated value returned and the meaning.

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

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

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

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

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

The example below demonstrates how floating point errors can be trapped and reported. In **main**, a pointer in the **Sysvec** array is set to the routine, **usertrap**. If a floating point exception occurs during the execution of the program, this routine is called with the arguments described above. The error handling routine prints the appropriate error message, and returns to the floating point support routines.

Internal Representation of Floating Point Numbers

Floats

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

```
#include "libc.lib"

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

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

     switch (errcode)  {
          case '1':
               printf("floating point underflow at %x\n",buff);
               break;
          case '2':
               printf("floating point overflow at %x\n",buff);
               break;
          case '3':
               printf("floating point division by zero at %x\n",buff);
               break;
          default:
               printf("invalid code %d passed to usertrap\n",errcode);
               break;
     }
```

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

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

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

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

Doubles

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

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

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

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

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

Floating Point Operations

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

Assembly Language Interface

A. Embedded Assembler Source

Assembly language statements can be embedded in a "C" program between an **#asm** and an **#endasm** statement. The pound sign (#) must stand in column one of the line, and the letters must be lower case.

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.

In general, it is safest to contain assembly code in a subroutine rather than embedding it in C source. This is the recommended approach where feasible.

B. Assembler Subroutines

The calling conventions used by the Aztec C II compiler are very simple. The arguments to a function are pushed onto the stack in reverse order, i.e. the first argument is pushed last and the last argument is pushed first.

The function is then called using the 8080 CALL instruction. When the function returns, the arguments are removed from the stack. A function is required to return with the arguments still on the stack unless something is pushed back in place of them.

Registers BC, IX, and IY must be preserved by routines called from C. The function's return value should be in HL and the Z flag set according to the value in HL.

Example:

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

SECTION IX

Software Extensions

The Tiny Library

This library reduces the overhead in code size when a program is linked with the standard library. This is accomplished through some basic tradeoffs, as detailed below.

The linkage command when using the tiny library has this form:

```
ln module.o [program modules] t.lib c.lib
```

what makes it so small?

I/O redirection is not supported by the tiny library. Furthermore, I/O is restricted to files and the console. No output to the printer is allowed. Thus, **fopen** can open only files. The console must be accessed with the functions: getchar, putchar, gets, puts, printf. Since **stdin** and **stdout** are not supported (see below), **agetc(stdin)** and **aputc(stdout)** cannot be used in place of **getchar** and **putchar**.

The tiny library does not allow the use of any low-level I/O routines such as **open** and **read.** Only buffered I/O is supported, with the limitations noted.

A maximum of four files can be open simultaneously. One can be open for writing, three for reading. A file cannot be open for both reading and writing. Specifically, the tiny **fopen** supports the "r" option for read only; any other option is construed as a "w" for write only.

Two buffers are maintained for disk file I/O to all the open files. One is used for writing and the other for reading. As long as input requests are being made to only one file, the buffer is refilled only when necessary. However, when reading from a new file, the buffer is refilled. This can be inefficient when two or three files are being read alternately, since the buffer is refilled every time the input file changes.

The formatting performed by **printf** and related functions is also restricted. The special format characters supported are **%c** for a character, **%d** for a decimal integer, **%x** for hexadecimal, and **%s** for a string. One exception to this rule is when **sscanf** is called but **printf** is not. This will cause formatting to be done the standard way.

An examination of the header file, "stdio.h", indicates some of the internal changes that have been made. A source module to be linked with the tiny library should be compiled with the -D option to define the symbol, TINY, as in the command:

```
cc -DTINY prog.c
```

This excludes the definitions of stdin, stdout, stderr, getchar, putchar, feof, ferror, clearerr, fileno, and fflush. The functions, **getchar** and **putchar**, are provided as library functions rather than as macros. The remaining functions are not available with the tiny library. Note that since stdin and stdout do not exist with the tiny library, I/O to the console cannot use **agetc** or **aputc**.

The following functions are not available when linking with the tiny library:

clearerr	**freopen**
fdopen	**fseek**
feof	**ftell**
ferror	**scanf**
fflush	**setbuf**
fileno	**puterr**
flush	**ungetc**

A Fast Linker

This feature is provided to shorten the time required to link a module for testing. The basic idea is that the module is linked as an overlay of the program, **r.com.** For this reason, there is no need at all to search an object file library. Essentially, the standard library is incorporated into the program, r.com.

Given a program consisting of a single module, a fast link command would have this form:

```
ln prog.o r.lib
```

or more conveniently,

```
ln prog.o -lr
```

This creates a disk file called "prog.ovr". This is just an overlay file of r.com, which loads the program into memory and calls the **main** routine. The command to r.com has this form:

```
r [program name] [args]
```

If "prog" required one command line argument, the command to run it would be:

```
r prog one_arg
```

The program name specifies which overlay file is to be loaded in memory. The arguments which follow are passed directly to the program just as if it had been linked as usual.

This fast linkage is generally intended for testing purposes. With it, changes to a test program can be made quickly. When a module is fully debugged, it should be linked with **c.lib** as is normally the case.

MS and DRI Compatibility

The Aztec C II compiler can generate assembly code for the M80 and RMAC assemblers from Microsoft and Digital Research, respectively. In each case an option must be specified at compile time.

The **-M** option will generate code for Microsoft's M80 assembler. In general, there are no special source level restrictions in using M80. Labels should not begin with an underscore, '_'. Under the -M option, the compiler generates code to ensure that statics and globals are initialized to zero, so that it is not necessary to use a switch to do this with M80.

A module that is assembled with M80 must be linked using L80. The run-time system necessary to perform this linkage is available as a **\PRO** extension to the standard package. Included in the **\PRO** extended software package are two files, **libc.rel** and **math.rel**, which provide the same run-time support as **c.lib** and **m.lib**. **c.lib** and **m.lib** cannot be linked with the Microsoft linker.

Note that some early versions of L80 are not supported by this interface.

The **-R** option produces code for the RMAC assembler by Digital Research. This switch has no effect at the C source code level. Modules assembled with RMAC must be linked with LINK-80. The files, **libc.rel** and **math.rel**, are the standard libraries which must be included in a linkage with LINK-80. They provide the same run-time support as **c.lib** and **m.lib**.

Manx Overlay Support

In order to allow users to run programs which are larger than the limited memory size of a microcomputer, Manx provides overlay support. This feature allows a user to divide a program into several segments. One of the segments, called the root segment, is always in memory. The other segments, called overlays, reside on disk and are only brought into memory when requested by the root segment. There is only one area of memory into which the overlays are loaded.

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

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

How to Make an Overlay File

What is an Overlay?

An overlay is one or more sections of executable code that run in the same area of memory. The advantage of an overlay, therefore, is that it allows the user to run programs of unlimited size in a machine which has a limited memory capacity.

How do I Call an Overlay From a Program?

The following is the format for calling an overlay:

```
ovloader(overlay name,p1, p2, p3...)
```

The ovloader function's first parameter must be the name of the overlay file. The parameters p1, p2, p3 are passed directly to the overlay. The overlay is loaded from a file whose name is overlay name and whose extent is ".ovr". ovloader returns as its value the value which was returned by the overlay.

How Do I Make a Function an Overlay?

The function, **ovloader**, loads the overlay and then passes control to **ovbgn**, a function which is linked with every overlay. **ovbgn** in turn calls **ovmain**. "ovmain" must be the name of your function which takes control when the overlay is loaded. This function can then call any other

function which is in memory.

So other than the naming of "ovmain", the overlay does not have to know that it is an overlay. "ovmain" executes and returns just like any other function.

What Files are Created on the Disk?

.com	The file which contains the root has the extent of .com
.ovr	There is one file for each overlay, the extent of which is .ovr
.rsm	There is a file containing the relocatable symbol table with the extent .rsm for the root and for any overlay that invokes another overlay.

Sample Run:

```
1)    ln +c 1020 +d a0 -r myroot.o ovloader.o libc.lib math.lib

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

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

In this example, there are three modules which comprise the program, namely, "myroot.o", "mysub1.o" and "mysub2.o". The first step serves two purposes, to create the executable file, "myroot.com", and to generate a file, "myroot.rsm".

This second file is called the relocatable symbol table. It contains information about the contents of the root module which is needed when an overlay is linked.

The **+C** and **+D** options are explained below. The **-R** option specifies that the following module is a root. An ".rsm" file for that module will be created.

The module, "ovloader.o" is just the routine which loads the overlay into memory.

The second step links the first overlay, "mysub1.o". The ".rsm" file for the root must be included in the linkage. The module, "ovbgn.o", is the startup routine which calls **ovmain**.

Step three performs the linkage of the second overlay in a manner identical to the first.

Figure 1 shows a program, run as a single module, that can be logically divided into three segments. Figure 2 shows the same program run as an overlay. In figure 2, module 1 and module 2 occupy the same memory locations. A possible flow of control would be for the base routine to call module 1, module 1 then returns to the root and the root calls module 2, module 2 returns to the root and the root calls module 1 again. Then module 1 returns to the root the root exits to the operating system.

Notice that all overlay segments must return to their caller and that overlays at the same level cannot directly invoke each other.

```
     address

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

Figure 1

A single binary image with 3 segments

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

Figure 2

Layout of the Program in Figure 1 as an Overlay

Programmer Information

The root loads an overlay by calling the MANX-supplied function "ovloader", which must reside in the root segment. The parameters to ovloader are a character string, giving the name of the overlay to be loaded, followed by the parameters which are to be passed to the overlay. **ovloader** is of the same type as **ovmain.**

When the overlay is loaded, control passes to the MANX-supplied function **ovbgn,** which must be linked with the overlay. In turn, **ovbgn** transfers control to the function in the overlay whose name is "ovmain". **ovmain** receives the arguments passed to ovloader.

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

Overlays can be nested; that is, an overlay can call a second overlay, provided that they are not at the same nesting level. Also, an overlay can access any global functions and variables which are defined in the calling segment.

Using Overlays

There are a few caveats for overlay usage. When the root module is linked (with the -r option), the linker has to reserve some of the memory used by the program for the overlay. This memory which is set aside is where the overlay will be loaded at run time.

There are two linker options which are relevant here. They are **+C** and **+D.** These options will increase the code and data areas of a program by a specified amount. Since these areas actually comprise a single segment under CP\M, both these options have the same effect.

When you link the root module, you will have to know how much memory to reserve for the overlay, that is, you will have to know how large the overlay is. This can be determined by linking the overlay as in the following command:

```
ln mysub.o ovbgn.o -lm -lc
```

Notice that this does not correctly link an overlay, since the ".rsm" file from the root is missing. But the linker will return a message giving the size of the overlay. Specifically, it will return the size of the code, initialized and uninitialized data, and the total of all three. Let's suppose the linker message looks like this:

```
Base: 0100   Code: 24a0   Data: 02c1   Udata: 0101   Total: 2862
```

In general, the total is the sum of the code, data and udata values. This is the total amount of memory that will be needed to load the overlay.

For this example, the root, myroot.o, could be linked with this command:

```
ln +c 24a0 +d 3c2 -r myroot.o ovloader.o -lm -lc
```

This will reserve an extra 24a0 bytes for the code of the overlay and 3c2 bytes for all the data, both initialized and otherwise. Since these bytes are part of the same area of memory, this linkage is equivalent to the following:

```
ln +d 2862 -r myroot.o ovloader.o -lm -lc
```

In this case, all the memory was set aside with a single option, +D. Notice that this is due to the fact that both code and data are part of the same segment on the 8080 and related microprocessors.

A Method

When the root is linked with the -r option, a ".rsm" file is created on the disk. This file must be included in the linkage of each overlay called by the root. In the example above, this file was left out when mysub.o was linked, because the ".rsm" file is produced when the root is linked; and to link the root, we needed to know how large the mysub.ovr overlay would be. This means that we have to relink mysub.o with the ".rsm" file.

A more convenient method is to link the root first. The ".rsm" file produced is not affected by the +D or +C option. If you can estimate the size of the overlay, specify it now. With the ".rsm", link the overlay:

```
ln mysub.o ovbgn.o root.rsm -lm -lc
```

If the linker returns with a total size greater than your previous estimate, you can relink the root with the appropriate value for the +D option. This method minimizes the number of linkages you will have to run.

The root will usually call more than one overlay. In this case, it necessary to reserve enough of memory for the largest overlay. You can first link the root, with an estimate of this largest size for the +D option. Then you can proceed to link each of the overlays individually with the ".rsm" file.

If one of the overlays should happen to turn out larger than your initial estimate, the root can be relinked without relinking every overlay.

Nested Overlays

Overlays can also be nested. Essentially, a module can load into memory any overlay for which it is the root. This is best illustrated by an example. In the situation depicted below, the root loads up ovly1, which in turn loads up ovly2. At this point, all three modules are in memory at the same. For this reason, when the root is linked, enough memory has to be reserved for both ovly1 and ovly2.

The following commands link the root and the two overlays:

```
ln +d 4000 -r root.o ovloader.o -lc
ln -r ovly1.o ovbgn.o root.rsm -lc
ln ovly2.o ovbgn.o ovly1.rsm -lc
```

Here, it is assumed that hex 4000 bytes are sufficient for loading in both ovly1 and ovly2. If the combined totals of these overlays is larger than this, it is necessary to relink the root.

Note that ovloader.o does not have to be included in the linkage of ovly1, since it is already linked in with the root. Also note that the ".rsm" file included in the linkage of ovly2 is that generated by the linkage of ovly1.

root.c:

```
main()
{
     ovloader("ovly1", "in ovly1");
     printf("just returned from ovly1\n");
}
```

ovly1.c:

```
ovmain(str)
char *str;
{
     printf("%s\n", str);
     ovloader("ovly2", "in ovly2");
     printf("just returned from ovly2\n");
}
```

ovly2.c:

```
ovmain(str)
char *str;
{
     printf("%s\n", str);
}
```

Possible Problems

A possible source of difficulty in using overlays concerns initialized data. In the following program module, a global variable is initialized:

```
int i = 3;

function()
{
        return;
}
```

The initialization of "i" is performed by the linker, rather than at run time. In the same program, the following module is allowed:

```
int i;

main()
{
        function();
}
```

The global variables in each module refer to the same integer, "i". At link time, this variable is set to the value 3. Although this works when the two modules are linked together, a problem arises when the first module is linked as an overlay:

```
ln func.o ovbgn.o main.rsm -lc
```

From the ".rsm" file, the linker knows that "int i" has been declared in main.o, the root. But it tries to initialize "i" from the statement in the func.o module. This attempt fails because the variable "i" is part of main.o, a module which is not included in the linkage.

An attempt to initialize, in an overlay, a variable which has been declared in the root will produce an error: "org out of range".

The simple solution is to change the statement, "int i = 3", to the following:

```
int i;
i = 3;
```

This assignment will be performed at run time, so that the linker does not try to perform an initialization.

Generating ROMable Code

Anatomy of a Program

A linked program can be visualized as having three major components: the code, the initialized data and the uninitialized data. Normally, a program will look like this in memory:

```
|_______________|
|               |
|_______________|    uninit'd data  (above init'd)
|               |
|               |
|_______________|    init'd data  (above code)
|               |
|               |
|_______________|    code  (hex address 103)
|_______________|    base  (hex address 100)
```

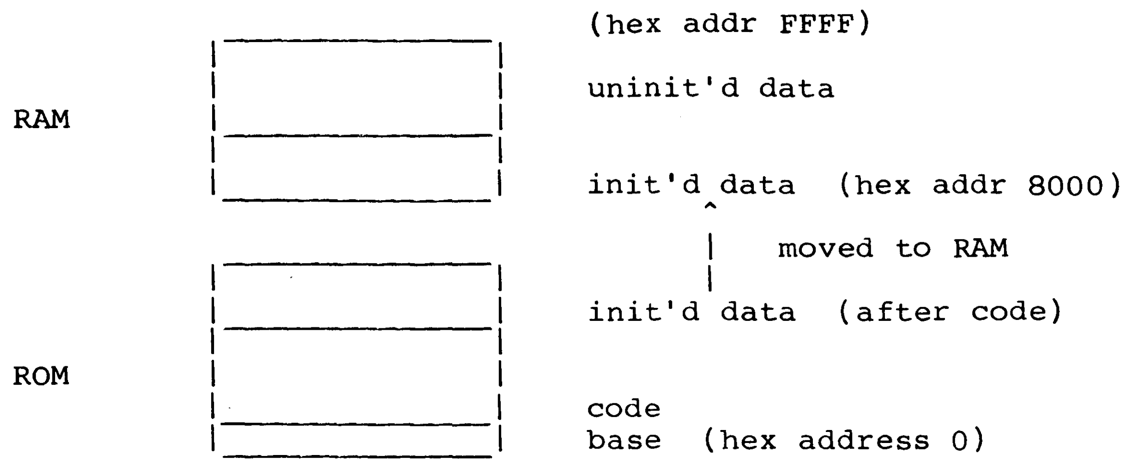
The linker has options for specifying each of the addresses depicted. They are as follows:

-B	base address
-C	code segment
-D	initialized data area
-U	uninitialized data area

By default, uninitialized data is placed at the end of the initialized data, and initialized data is placed at the end of the code.

The Problem with Initialized Data

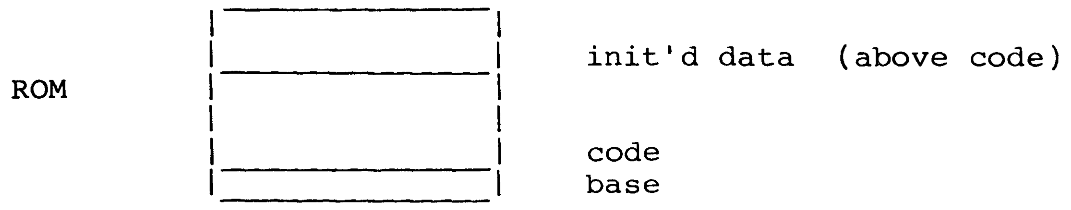
Initialized data poses a problem when putting a program in ROM, since data in ROM can be read but not altered. Data which starts out in ROM must be moved into RAM, where it then can be manipulated by the program. The program must look like this when it is run:



In the particular configuration shown above, the data is placed at the available RAM beginning at hex address 8000. The base is specified as the beginning of ROM, hex address 0. These values are given at link time, as in

ln -b 0 -d 8000 program.o ROM.lib

When converted to hex format for burning into ROM, the program looks like this:



The ROM Library

What is needed is a routine to copy the initialized data from ROM into RAM. This is provided as part of the startup routine which is automatically linked in from the library, **ROM.lib.** The startup routine will move the initialized data to the address specified with the -D option, and clear the uninitialized data area to zeroes (nulls). So it is not necessary to initialize data to zero, since this is guaranteed for uninitialized global data. Under certain conditions, data might best be initialized by the program itself, as with a loop which sets an array to {1, 2, 3, ...}.

Leaving Data in ROM

If initialized data is "read only", that is, is never changed by the program, then there is no reason why it cannot be left in ROM. The following link command will set up this situation:

```
ln -b 0 -u 8000 ROM.lib
```

The initialized data defaults to being placed right after the code, i.e, in ROM. The -U option to the linker specifies the starting address of the uninitialized data area. The program will look like this:

```
                 _________________      (hex addr FFFF)
   RAM          |                 |
                |_________________|     uninit'd data  (hex addr 8000)


                |‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾|
                |_________________|     initialized data
   ROM          |                 |
                |_________________|     code
                |_________________|     base  (hex address 0)
```

Leaving Part of the Data in ROM

If some initialized data does not need to be changed but other data does, it is possible to modify the assembly language output of the compiler to separate the two types of data. The goal here is to have the startup routine move into RAM only that initialized data which needs to be altered at run time.

This is done by putting the read-only data in the same area as the code of the program. The data in the code area will remain in ROM, while the data in the data area will be moved into RAM.

To force data to remain in the code segment, it is necessary to strip out the "dseg" pseudo-op generated by the compiler for each data definition.

This must not be done for data defined within a function; the program will not run with data definitions treated as executable instructions.

Since only global data can be stripped of the dseg pseudo-op, it is usually a good idea to put all this "code area" data into a single file by itself; this avoids the danger of stripping dsegs within a procedure.

With initialized data "mixed" in this way, you will have to specify values for the -B and -D options, so that the startup routine will load the data area properly into RAM.

Why Two Libraries

The standard library, **c.lib**, is not directly suited for generating ROMable code. The changes which are needed in c.lib have been incorporated into a separate library, **ROM.lib.** This library includes the startup routine which will move initialized data into RAM. It does not support all the functions available in the standard library. It contains no routines for memory allocation and no i/o except for the function, **sprintf.** It does contain the function, **format**, so that the user can write functions for formatted i/o.

There is no initialized data in ROM.lib which must be moved to RAM, as implied by the linkage example above. However, there is initialized data in the floating point package in **m.lib** which must be moved by the startup routine in ROM.lib.

Aztec Utility Programs

The following descriptions explain in detail the use of special utility programs provided with the Aztec development system. The format of each explanation is similar to those of the library functions.

NAME

cnm - display object file info

SYNOPSIS

cnm [-s] file [file ...]

DESCRIPTION

cnm displays the size and symbols of its object file arguments. The files can be object modules created by AS, libraries of object modules created by LN, and 'rsm' files created by LN during the linking of an overlay root.

The **-s** option tells **cnm** to just display the size information for the files.

For example, the following displays the size and symbols for the object module subl.o, the library c.lib, and the rsm file root.rsm:

```
cnm subl.o c.lib root.rsm
```

By default, the information is sent to the display. It can be redirected to a file or device in the normal way. For example, the following three commands send information about subl.o to the display, the file dispfile, and the printer, respectively:

```
cnm subl.o
cnm subl.o > dispfile
cnm >lst: subl.o
```

A filename can optionally specify multiple files, using the "wildcard" characters **?** and *****. These have their standard CP/M meanings: **?** matches a single character; ***** matches zero or more characters. For example

*.o	specifies all files with extent '.o'
a??.lib	specifies all files whose filename has three characters, the first of which is 'a', and whose extent is '.lib'

cnm displays information about an program's 'named' symbols; that is, about the symbols whose first two characters are other than a period followed by a digit. For example, the symbol **quad** is named, so information about it would be displayed; the symbol **.0123** is unnamed, so information about it would not be displayed.

For each named symbol in a program, **cnm** displays its name, a code specifying its type, and an associated value. The value displayed depends on the type of the symbol.

A type code is a single character, and can be either upper or lower case, specifying that the symbol is global or local to the program, respectively. The types codes are:

a The symbol was defined using the assembler's EQUATE directive. The value listed is the equated value of its symbol.

The compiler doesn't generate symbols of this type.

t The symbol is in the code segment. The value is the offset of the symbol within the code segment.

The compiler generates this type symbol for function names; static functions are local to the function, and so have type **t**; all other functions are global, that is, callable from other programs, and hence have type **T**.

d The symbol is in the data segment. The value is the offset of the symbol from the start of the data segment.

The compiler generates symbols of this type for initialized variables which are declared outside any function. Static variables are local to the program and so have type **d**; all other variables are global, that is, accessable from other programs, and hence have type **D**.

C The symbol is the name of a common block. The value is the size of the common block, in bytes. **C** is in upper case because common block names are always global.

The compiler doesn't generate this type symbol.

r The symbol is defined within a common block. The value is the offset of the symbol from the beginning of the common block.

The compiler doesn't generate this type symbol.

u The symbol is used but not defined within the program. The value has no meaning.

The compiler generates **U** symbols for functions that are called but not defined within the program, for variables that are declared to be **extern** and which are actually used within the program, and for unitialized, global dimensionless arrays. Variables which are declared to be **extern** but which are not used within the program don't make it to the object file.

The compiler generates **u** symbols for variables which are used but not defined within the program.

b The symbol is in the uninitalized data segment. The value is the space reserved for the symbol.

The compiler generates **b** symbols for static, uninitialized variables which are declared outside all functions and which aren't dimensionless arrays.

The assembler generates **b** symbols for symbols defined using the **bss** assembler directive. If the symbol also appears in the **public** directive, it's type is **B** instead of **b.**

G The symbol is in the uninitialized data segment. The value is the space reserved for the symbol.

The compiler generates **G** symbols for non-static, uninitialized variables which are declared outside all functions and which aren't dimensionless arrays.

The assembler generates **G** symbols for variables declared using the **global** directive which have a non-zero size.

NAME

sqz - squeeze an object library

SYNOPSIS

sqz file [outfile]

DESCRIPTION

sqz compresses an object library which was created by **LIBUTIL.**

The first parameter is the name of the file containing the library to be compressed. The second parameter, which is optional, is the name of the file to which the compressed library will be written.

If the output file is specified, the original file isn't modified or erased.

If the output file isn't specified, **sqz** creates the compressed library in a file having a temporary name, erases the original library file, and renames the output file to the name of the original file. The temporary name is derived from the input library name by changing it's extent to '.sqz'.

If the output file isn't specified and an error occurs during the creation of the compressed library, the original library isn't erased or modified.

SECTION X

Style and Debugging

Style

This section was written for the programmer who is new to the C language, to communicate the special character of C and the programming practices for which it is best suited. This material will ease the new user's entry into C. It gives meaning to the peculiarities of C syntax, in order to avoid the errors which will otherwise disappear only with experience.

what's in it for me?

These are the benefits to be reaped by following the methods presented here:

- o reduced debugging times
- o increased program efficiency
- o reduced software maintenance burden

The aim of the responsible programmer is to write straightforward code, which makes his programs more accessible to others. This section on style is meant to point out which programming habits are conducive to successful C programs and which are especially prone to cause trouble.

The many advantages of C can be abused. Since C is a terse, subtle language, it is easy to write code which is unclear. This is contrary to the "philosophy" of C and other structured programming languages, according to which the structure of a program should be clearly defined and easily recognizable.

keep it simple

There are several elements of programming style which make C easier to use. One of these is **simplicity.** Simplicity means **keep it simple.** You should be able to see exactly what your code will do, so that when it doesn't you can figure out why.

A little suspicion can also be useful. The particular "problem areas" which are discussed later in this section are points to check when code "looks right" but does not work. A small omission can cause many errors.

learn the C idioms

C becomes more valuable and more flexible with time. Obviously, elementary problems with syntax will disappear. But more importantly, C can be described as "idiomatic." This means that certain expressions become part of a standard vocabulary used over and over. For example,

```
                while ((c = getchar()) != EOF)
```

is readily recognized and written by any C programmer. This is often used as the beginning of a loop which gets a character at a time from a source of input. Moreover, the inside set of parentheses, often omitted by a new C programmer, is rarely forgotten after this construct has been used a few times.

be flexible in using the library

The standard library contains a choice of functions for performing the same task. Certain combinations offer advantages, so that they are used routinely. For instance, the standard library contains a function, **scanf**, which can be used to input data of a given format. In this example, the function "scans" input for a floating point number:

```
                scanf("%f", &flt_num);
```

There are several disadvantages to this function. An important debit is that it requires a lot of code. Also, it is not always clear how this function handles certain strings of input. Much time could be spent researching the behavior of this function. However, the equivalent to the above is done by the following:

```
                flt_num = atof(gets(inp_buf));
```

This requires considerably less code, and is somewhat more straightforward. **gets** puts a line of input into the buffer, "inp_buf," and **atof** converts it to a floating point value. There is no question about what the input function is "looking for" and what it should find.

Furthermore, there is greater flexibility in the second method of getting input. For instance, if the user of the program could enter either a special command ("e" for exit) or a floating point value, the following is possible:

```
                gets(inp_buf);
                if (inp_buf[0] == 'e')
                     exit(0);
                flt_num = atof(inp_buf);
```

Here, the first character of input is checked for an "e", before the input is converted to a float.

The relative length of the library description of the **scanf** function is an indication of the problems that can arise with that and related functions.

write readible code

Readibility can be greatly enhanced by adhering to what common sense dictates. For instance, most lines can easily accommodate only one statement. Although the compiler will accept statements which are packed together indiscriminately, the logic behind the code will be lost. Therefore, it makes sense to write no more than one statement per line.

In a similar vein, it is desirable to be generous with whitespace. A blank space should separate the arithmetic and assignment operators from other symbols, such as variable names. And when parentheses are nested, dividing them with spaces is not being too prudent. For example,

```
if((fp=fopen("filename","r")==NULL))
```

is not the same as

```
if ( (fp = fopen("filename", "r")) == NULL )
```

The first line contains a misplaced parenthesis which changes the meaning of the statement entirely. (A file is opened but the file pointer will be null.) If the statement was expanded, as in the second line, the problem could be easily spotted, if not avoided altogther.

use straightforward logical expressions

Conditionals are apt to grow into long expressions. They should be kept short. Conditionals which extend into the next line should be divided so that the logic of the statement can be visualized at a glance. Another solution might be to reconsider the logic of the code itself.

learn the rules for expression evaluation

Keep in mind that the evaluation of an expression depends upon the order in which the operators are evaluated. This is determined from their relative precedence.

Item 7 in the list of "things to watch out for", below, gives an example of what may happen when the evaluation of a boolean expression stops "in the middle". The rule in C is that a boolean will be evaluated only until the value of the expression can be determined.

Item 8 gives a well known example of an "undefined" expression, one whose value is not strictly determined.

In general, if an expression depends upon the order in which it is evaluated, the results may be dubious. Though the result may be strictly defined, you must be certain you know what that definition is.

a matter of taste

There are several popular styles of indentation and placement of the braces enclosing compound statements. Whichever format you adopt, it is important to be consistent. Indentation is the accepted way of conveying the intended nesting of program statements to other programmers. However, the compiler understands only braces. Making them as visible as possible will help in tracking down nesting errors later.

However much time is devoted to writing readible code, C is low-level enough to permit some very peculiar expressions.

```
/* It is important to insert comments on a regular basis! */
```

Comments are especially useful as brief introductions to function definitions.

In general, moderate observance of these suggestions will lessen the number of "tricks" C will play on you-- even after you have mastered its syntax.

Structured Programming

"Structured programming" is an attempt to encourage programming characterized by method and clarity. It stems from the theory that any programming task can be broken into simpler components. The three basic parts are statements, loops, and conditionals. In C, these parts are, respectively, anything enclosed by braces or ending with a semicolon; **for**, **while** and **do-while**; **if-else**.

modularity and block structure

Central to structured programming is the concept of modularity. In one sense, any source file compiled by itself is a module. However, the term is used here with a more specific meaning. In this context, modularity refers to the independence or isolation of one routine from another. For example, a routine such as **main()** can call a function to do a given task even though it does not know how the task is accomplished or what intermediate values are used to reach the final result.

Sections of a program set aside by braces are called "blocks". The "privacy" of C's block structure ensures that the variables of each block are not inadvertently shared by other blocks. Any left brace ({) signals the beginning of a block, such as the body of a function or a **for** loop. Since each block can have its own set of variables, a left brace marks an opportunity to declare a temporary variable.

A function in C is a special block because it is called and is passed control of execution. A function is called, executes and returns. Essentially, a C program is just such a routine, namely, **main**.

A function call represents a task to be accomplished. Program statements which might otherwise appear as several obscure lines can be set aside in a function which satisfies a desired purpose. For instance, **getchar** is used to get a single character from standard input.

When a section of code must be modified, it is simpler to replace a single modular block than it is to delete a section of an unstructured program whose boundaries may be unclear at best. In general, the more precisely a block of program is defined, the more easily it can be changed.

Top-down Programming

"Top-down" programming is one method that takes advantage of structured programming features like those discussed above. It is a method of designing, writing, and testing a program from the most general function (i.e., (main()) to the most specific functions (such as getchar()).

All C programs begin with a function called main(). main() can be thought of as a supervisor or manager which calls upon other functions to perform specific tasks, doing little of the work itself. If the overall goal of the program can be considered in four parts (for instance, input, processing, error checking and output), then main() should call at least four other functions.

step one

The first step in the design of a program is to identify what is to be done and how it can be accomplished in a "programmable" way. The **main** routine should be greatly simplified. It needs to call a function to perform the crucial steps in the program. For example, it may call a function, init(), which takes care of all necessary startup initializations. At this point, the programmer does not even need to be certain of all the initializations that will take place in init().

All functions consist of three parts: a parameter list, body, and return value. The design of a function must focus on each of these three elements.

During this first stage of design, each function can be considered a black box. We are concerned only with what goes in and what comes out, not with what goes on inside.

Do not allow yourself to be distracted by the details of the implementation at this point. Flowcharts, pseudocode, decision tables and the like are useful at this stage of the implementation.

A detailed list of the data which is passed back and forth between functions is important and should not be neglected. The interface between functions is crucial.

Although all functions are written with a purpose in mind, it is easy to unwittingly merge two tasks into one. Sometimes, this may be done in the interests of producing a compact and efficient program function. However, the usual result is a bulky, unmanageable function. If a function grows very large or if its logic becomes difficult to comprehend, it should be reduced by introducing additional function calls.

step two

There comes a time when a program must pass from the design stage into the coding stage. You may find the top-down approach to coding too restrictive. According to this scheme, the smallest and most specific functions would be coded last. It is our nature to tackle the most daunting problems first, which usually means coding the low-level functions.

Whereas the top-down approach is the preferred method for designing software, the bottom-up approach is often the most practical method for writing software. Given a good design, either method of implementation should produce equally good results.

One asset of top-down writing is the ability to provide immediate tests on upper level routines. Unresolved function calls can be satisfied by "dummy" functions which return a range of test values. When new functions are added, they can operate in an environment that has already been tested.

C functions are most effective when they are as mutually independent as is possible. This independence is encouraged by the fact that there is normally only one way into and one way out of a function: by calling it with specific arguments and returning a meaningful value. Any function can be modified or replaced so long as its entry and exit points are consistent with the calling function.

Defensive Programming

"Defensive programming" obeys the same edict as defensive driving: trust no one to do what you expect. There are two sides to this rule of thumb. Defend against both the possibility of bad data or misuse of the program by the user, and the possibility of bad data generated by bad code.

Pointers, for example, are a prime source of variables gone astray. Even though the "theory" of pointers may be well understood, using them in new ways (or for the first time) requires careful consideration at each step. Pointers present the fewest problems when they appear in familiar settings.

faced with the unknown

When trying something new, first write a few test programs to make sure the syntax you are using is correct. For example, consider a buffer, **str_buf**, filled with null-terminated strings. Suppose we want to print the string which begins at offset **begin** in the buffer. Is this the way to do it?

```
printf("%s", str_buf[begin]);
```

A little investigation shows that str_buf[begin] is a character, not a pointer to a string, which is what is called for. The correct statement is

```
printf("%s", str_buf + begin);
```

This kind of error may not be obvious when you first see it. There are other topics which can be troublesome at first exposure. The promotion of data types within expressions is an example. Even if you are sure how a new construct behaves, it never hurts to doublecheck with a test program.

Certain programming habits will ease the bite of syntax. Foremost among these is simplicity of style. Top-down programming is aimed at producing brief and consequently simple functions. This simplicity should not disappear when the design is coded.

Code should appear as "idiomatic" as possible. Pointers can again provide an example: it is a fact of C syntax that arrays and pointers are one and the same. That is,

```
array[offset]
```

is the same as

```
*(array + offset)
```

The only difference is that an array name is not an lvalue; it is fixed. But mixing the two ways of referencing an object can cause confusion, such as in the last example. Choosing a certain idiom, which is known to behave a certain way, can help avoid many errors in usage.

when bugs strike

The assumption must be that you will have to return to the source code to make changes, probably due to what is called a bug. Bugs are characterized by their persistence and their tendency to multiply rapidly.

Errors can occur at either compile-time or run-time. Compile-time errors are somewhat easier to resolve since they are usually errors in syntax which the compiler will point out.

from the compiler

If the compiler does pick up an error in the source code, it will send an error code to the screen and try to specify where the error occurred. There are several peculiarities about error reporting which should be brought up right away.

The most noticeable of these peculiarities is the number of spurious errors which the compiler may report. This interval of inconsistency is referred to as the compiler's recovery. The safest way to deal with an unusually long list of errors is to correct the first error and then recompile before proceeding.

The compiler will specify where it "noticed" something was wrong. This does not necessarily indicate where you must make a change in the code. The error number is a more accurate clue, since it shows what the compiler was looking for when the error occurred.

if this ever happens to you

A common example of this is error 69: "missing semicolon." This error code will be put out if the compiler is expecting a semicolon when it finds some other character. Since this error most often occurs at the end of a line, it may not be reported until the first character of the following line-- recall that whitespace, such as a newline character, is ignored.

Such an error can be especially treacherous in certain situations. For example, a missing semicolon at the end of a **#include**'d file may be reported when the compiler returns to read input in the original file.

In general, it is helpful to look at a syntax error from the compiler's point of view.

Consider this error:

```
          struct structag {
               char c;
               int i;
          }

          int j;
```

This should generate an error 16: "data type conflict". The arrow in the error message should show that the error was detected right after the "int" in the declaration of **j**. This means that the error has to do with something before that line, since there is nothing illegal about the **int** keyword.

By inspection, we may see that the semicolon is missing from the preceding line. If this fact escapes our notice, we still know that error 16 means this: the compiler found a declaration of the form

```
          [data type] [data type] [symbol name]
```

where the two data types were incompatible. So while **short int** is a good data type, **double int** is not. A small intuitive leap leads us to assume that the compiler has read our source as a kind of "struct int" declaration; **struct** is the only keyword preceding the **int** which could have caused this error. Since the compiler is reading the two declarations as a single statement, we must be missing a delimiter.

run-time errors

It takes a bit more ingenuity to locate errors which occur at run-time. In numerical calculations, only the most anomalous results will draw attention to themselves. Other bugs will generate output which will appear to have come from an entirely different program.

A bug is most useful when it is repeatable. Bugs which show up only "sometimes" are merely vexing. They can be caused by a corrupted disk file or a bad command from the user.

When an error can be consistently produced, its source can be more easily located. The nature of an error is a good clue as to its source. Much of your time and sanity will be preserved by setting aside a few minutes to reflect upon the problem.

Which modules are involved in the computation or process? Many possibilities can be eliminated from the start, such as pieces of code which are unrelated to the error.

The first goal is to determine, from a number of possibilities, which module might be the source of the bug.

checking input data

Input to the program can be checked at a low cost. Error checking of this sort should be included on a "routine" basis. For instance, "if ((fp=fopen("file","r"))==NULL)" should be reflex when a file is opened. Any useful error handling can follow in the body of the **if**.

It is easy to check your data when you first get your hands on it. If an error occurs after that, you have a bug in your program.

printf it

It is useful to know where the data goes awry. One brute force way of tracking down the bug is to insert **printf** statements wherever the data is referenced. When an unexpected value comes up, a single module can be chosen for further investigation.

The printf search will be most effective when done with more refinement. Choose a suspect module. There are only two keys points to check: the entry and return of the function. **printf** the data in question just as soon as the function is entered. If the values are already incorrect, then you will want to make sure the correct data was passed in the function call.

If an incorrect value is returned, then the search is confined to the guilty function. Even if the function returns a good value, you may want to make sure it is handled correctly by the calling function.

If everything seems to be working, jump to the next tricky module and perform another check. When you find a bad result, you will still have to backtrack to discover precisely where the data was spoiled.

function calls

Be aware that data can be garbled in a funtion call. Function parameters must be declared when they are not two byte integers. For instance, if a function is called:

```
                fseek(fp, 0, 0);
```

in order to "seek" to the beginning of a file, but the function is defined this way:

```
                fseek(fp, offset, origin)
                FILE *fp;
                long offset;
```

```
            int origin;
```

there will be unfortunate consequences.

The second parameter is expected to be a **long** integer (four bytes), but what is being passed is a **short** integer (two bytes). In a function call, the arguments are just being pushed onto the stack; when the function is entered, they are pulled off again. In the example, two bytes are being pushed on, but four bytes (whatever four bytes are there) are being pulled off.

The solution is just to make the second parameter a long, with a suffix (0L) or by the cast operator (as in (long)i).

A similar problem occurs when a non-integer return value is not declared in the calling function. For example, if **sqrt** is being called, it must be declared as returning a **double**:

```
            double sqrt();
```

This method of debugging demonstrates the usefulness of having a solid design before a function is coded. If you know what should be going into a function and what should be coming out, the process of checking that data is made much simpler.

found it

When the guilty function is isolated, the difficulty of finding the bug is proportional to the simplicity of the code. However, the search can continue in a similar way. You should have a good notion of the purpose of each block, such as a loop. By inserting a **printf** in a loop, you can observe the effect of each pass on the data.

printf's can also point out which blocks are actually being executed. "Falling through" a test, such as an **if** or a **switch**, can be a subtle source of problems. Conditionals should not leave cases untested. An **else**, or a **default** in a **switch**, can rescue the code from unexpected input.

And if you are uncertain how a piece of code will work, it is usually worthwhile to set up small test programs and observe what happens. This is instructional and may reveal a bug or two.

Things to Watch Out for

Some errors arise again and again. Not all of them go away with experience. The following list will give you an idea of the kinds of things that can go wrong.

1. missing semicolon or brace

The compiler will tell you when a missing semicolon or brace has introduced bad syntax into the code. However, often such an error will affect only the logical structure of the program; the code may compile and even execute. When this error is not revealed by inspection, it is usually brought out by a test **printf** which is executed too often or not enough. See compiler error 69.

2. assignment (=) vs comparison (==)

Since variables are assigned values more often than they are tested for equality, the former operator was given the single keystroke: =. Notice that all the comparison tests with equality are two characters: <=, >= and ==.

3. misplaced semicolon

When typing in a program, keep in mind that all source lines do not automatically end with a semicolon. Control lines are especially susceptible to an unwanted semicolon:

```
for (i=0; i<100; i++);
    printf("%d",i);
```

This example prints the single number 100.

4. division (/) vs escape sequence (\)

C definitely distinguishes between these characters. The division sign resides below the question mark on a standard console; the backslash is generally harder to find.

5. character constant (') vs character string (")

Character constants are actually integers equal to the ASCII values of the respective character. A character string is a series of characters terminated by a null character (\0). The appropriate delimiter is the single quote and double quote, respectively.

6. uninitialized variable

At some point, all variables must be given values before they are used. The compiler will set global and static variables to zero, but automatic variables are guaranteed to contain garbage every time they are created.

7. evaluation of expressions

For most operations in C, the order of evaluation is rigidly defined. For example, the result of the following example is 8. Notice that evaluation does not quite obey "My Dear Aunt Sally;" operators of equal precedence are evaluated left-to-right or right-to-left, as they are defined.

```
int a = 2, b = 3, c = 4, d;
d = a + b / a * c;
    /* is evaluated */
d = a + ((b / a) * c);
```

Consider this example:

```
if ( (c = 0) || (c = 1) )
    printf("%d", c);
```

"1" will be printed; since the first half of the conditional evaluates to zero, the second half must be also evaluated. But in this example:

```
if ( (c = 0) && (c = 1) )
    ;
printf("%d", c);
```

a "0" is printed. Since the first half evaluates to zero, the value of the conditional must be zero, or false, and evaluation stops. This is a property of the logical operators.

8. undefined order of evaluation

Unfortunately, not all operators were given a complete set of instructions as to how they should be evaluated. A good example is the increment (or decrement) operator. For instance, the following is undefined:

```
i = ++i + --i/++i - i++;
```

How such an expression is evaluated by a particular implementation is called a "side effect." In general, side effects are to be avoided.

9. evaluation of boolean expressions

Ands, ors and nots invite the programmer to write long conditionals whose very purpose is lost in the code. Booleans should be brief and to the point. Also, the bitwise logical operators must be fully parenthesized. The table on page [] shows their precedence in relation to other operators.

Here is an extreme example of how a lengthy boolean can be reduced:

```
if ((c = getchar()) != EOF && c >= 'a' && c <= 'z' &&
              (c = getchar()) >= '1' && c <= '9')
     printf("good input\n");

if ((c = getchar()) != EOF)
     if (c >= 'a' && c <= 'z')
          if ((c = getchar()) >= '0' && c <= '9')
               printf("good input\n");
```

10. badly formed comments

The theory of comment syntax is simply that everything occurring between a left /* and a right */ is ignored by the compiler. Nonetheless, a missing */ should not be overlooked as a possible error.

Note that comments cannot be nested, that is

```
/*  /*   this will cause an error   */    */
```

And this could happen to you too:

```
/* the rest of this file is ignored until another comment /*
```

11. nesting error

Remember that nesting is determined by braces and not by indentations in the text of the source. Nested **if** statements merit particular care since they are often paired with an **else.**

12. usage of else

Every else must pair up with an **if.** When an **else** has inexplicably remained unpaired, the cause is often related to the first error in this list.

13. falling through the cases in a switch

To maintain the most control over the **cases** in a **switch** statement, it is advisable to end each **case** with a **break,** including the last **case** in the **switch**.

14. strange loops

The behavior of loops can be explored by inserting **printf** statements in the body of the loop. Obviously, this will indicate if the loop has even been entered at all in course of a run. A counter will show just how many times the loop was executed; a small slip-up will cause a loop to be run through once too often or seldom. The condition for leaving the loop should be doublechecked for accuracy.

15. use of strings

All strings must be terminated by a null character in memory. Thus, the string, "hello", will occupy a six-element array; the sixth element is '\0'. This convention is essential when passing a string to a standard library function. The compiler will append the null character to string constants automatically.

16. pointer vs object of a pointer

The greatest difficulty in using pointers is being sure of what is needed and what is being used. Functions which take a pointer argument require an address in memory. The best way to ensure that the correct value is being passed is to keep track of what is being pointed to by which pointer.

17. array subscripting

The first element in a C array has a subscript of zero. The array name without a subscript is actually a pointer to this element. Obviously, many problems can develop from an incorrect subscript. The most damaging can be subscripting out of bounds, since this will access memory above the array and overwrite any data there. If array elements or data stored with arrays are being lost, this error is a good candidate.

18. function interface

During the design stage, the components of a program should be associated with functions. It is important that the data which is passed among or shared by these functions be explicitly defined in the preliminary design of the program. This will greatly facilitate the coding of the program since the interface between functions must be precise in several respects.

First of all, if the parameters of a function are established, a call can be made without the reservation that it will be changed later. There is less chance that the arguments will be of the wrong type or specified in the wrong order.

A function is given only a private copy of the variables it is passed. This is a good reason to decide while designing the program how functions should access the data they require. You will be able to detail the arguments to be passed in a function call, the global data which the function will alter, the value which the function will return and what declarations will be appropriate-- all without concern for how the function will be coded.

Argument declarations should be a fairly simple matter once these things are known. Note that this declaration list must stand before the left brace of the function body.

The type of the function is the same as the type of the value it returns. Functions must be declared just like any variable. And just like variables, functions will default to type int, that is, the compiler will assume that a function returns an integer if you do not tell it otherwise with a declaration. Thus if function f calls function g which returns a variable of type double, the following declaration is needed:

```
function f()
{
     double g(), bigfloat;

     g(bigfloat);
}
double g(arg)
double arg;
{
     return(arg);
}
```

19. be sure of what a function returns

You will probably know very well what is returned by a function you have written yourself. But care should be taken when using functions coded by someone else. This is especially true of the standard library functions. Most of the supplied library functions will return an int or a char pointer where you might expect a char. For instance, getchar() returns an int, not a char. The functions supplied by Manx adhere to the UNIX model in all but a few cases.

Of course, the above applies to a function's arguments as well.

20. shared data

Variables that are declared globally can be accessed by all functions in the file. This is not a very safe way to pass data to functions since once a global variable is altered, there is no returning it to its former state without an elaborate method of saving data. Moreover, global data must be carefully managed; a function may process the wrong variable and consequently inhibit any other function which depends on that data.

Since C provides for and even encourages private data, this definitely should not be a common bug.

ERROR MESSAGES

COMPILER ERROR CODES

No.	Interpretation
1:	bad digit in octal constant
2:	string space exhausted
3:	unterminated string
4:	internal error
5:	illegal type for function
6:	inappropriate arguments
7:	bad declaration syntax
8:	syntax error in typecast
9:	array dimension must be constant
10:	array size must be positive integer
11:	data type too complex
12:	illegal pointer reference
13:	unimplemented type
14:	internal
15:	internal
16:	data type conflict
17:	internal
18:	data type conflict
19:	obsolete
20:	structure redeclaration
21:	missing }
22:	syntax error in structure declaration
23:	obsolete
24:	need right parenthesis or comma in arg list
25:	structure member name expected here
26:	must be structure/union member
27:	illegal typecast
28:	incompatible structures
29:	illegal use of structure
30:	missing : in ? conditional expression
31:	call of non-function
32:	illegal pointer calculation
33:	illegal type
34:	undefined symbol
35:	typedef not allowed here
36:	no more expression space
37:	invalid expression for unary operator
38:	no auto. aggregate initialization allowed
39:	obsolete
40:	internal
41:	initializer not a constant
42:	too many initializers
43:	initialization of undefined structure
44:	obsolete
45:	bad declaration syntax
46:	missing closing brace
47:	open failure on include file
48:	illegal symbol name

```
49:  multiply defined symbol
50:  missing bracket
51:  lvalue required
52:  obsolete
53:  multiply defined label
54:  too many labels
55:  missing quote
56:  missing apostrophe
57:  line too long
58:  illegal # encountered
59:  macro too long
60:  obsolete
61:  reference of member of undefined structure
62:  function body must be compound statement
63:  undefined label
64:  inappropriate arguments
65:  illegal argument name
66:  expected comma
67:  invalid else
68:  syntax error
69:  missing semicolon
70:  goto needs a label
71:  statement syntax error in do-while
72:  statement syntax error in for
73:  statement syntax error in for body
74:  case value must integer constant
75:  missing colon on case
76:  too many cases in switch
77:  case outside of switch
78:  missing colon on default
79:  duplicate default
80:  default outside of switch
81:  break/continue error
82:  illegal character
83:  too many nested includes
84:  too many array dimensions
85:  not an argument
86:  null dimension in array
87:  invalid character constant
88:  not a structure
89:  invalid use of register storage class
90:  symbol redeclared
91:  illegal use of floating point type
92:  illegal type conversion
93:  illegal expression type for switch
94:  invalid identifier in macro definition
95:  macro needs argument list
96:  missing argument to macro
97:  obsolete
98:  not enough arguments in macro reference
99:  internal
100:  internal
101:  missing close parenthesis on macro reference
102:  macro arguments too long
103:  #else with no #if
```

104:	#endif with no #if
105:	#endasm with no #asm
106:	#asm within #asm block
107:	missing #endif
108:	missing #endasm
109:	#if value must be integer constant
110:	invalid use of : operator
111:	invalid use of void expression
112:	invalid use function pointer
113:	duplicate case in switch
114:	macro redefined
115:	keyword redefined

Explanations

1: bad digit in octal constant

The only numerals permitted in the base 8 (octal) counting system are zero through seven. In order to distinguish between octal, hexadecimal, and decimal constants, octal constants are preceded by a zero. Any number beginning with a zero must not contain a digit greater than seven. Octal constants look like this: 01, 027, 003. Hexadecimal constants begin with 0x (e.g., 0x1, 0xAA0, 0xFFF).

2: string space exhausted

The compiler maintains an internal table of the strings appearing in the source code. Since this table has a finite size, it may overflow during compilation and cause this error code. The table default size is about one or two thousand characters depending on the operating system. The size can be changed using the compiler option **-Z**. Through simple guesswork, it is possible to arrive at a table size sufficient for compiling your program. The following example illustrates the use of this option:

```
cc -Z3000 bigexmpl.c
```

The new table size allows the strings in the file to total 3000 bytes in length. This is equal to 3000 characters.

3: unterminated string

All strings must begin and end with double quotes ("). This message indicates that a double quote has remained unpaired.

4: internal error

This error message should not occur. It is a check on the internal workings of the compiler and is not known to be caused by any particular piece of code. However, if this error code appears, please bring it to the attention of MANX. It could be a bug in the compiler. The release documentation enclosed with the product contains further information.

5: illegal type for function

The type of a function refers to the type of the value which it returns. Functions return an **int** by default unless they are declared otherwise. However, functions are not allowed to return aggregates (arrays or structures). An attempt to write a function such as **struct sam func()** will generate this error code. The legal function types are **char, int, float, double, unsigned, long, void** and a pointer to any type (including structures).

6: error in argument declaration

The declaration list for the formal parameters of a function stands immediately before the left brace of the function body, as shown below. Undeclared arguments default to **int,** though it is usually better practice to declare everything. Naturally, this declaration list may be empty, whether or not the function takes any arguments at all.

No other inappropriate symbols should appear before the left (open) brace.

```
badfunction(arg1, arg2)
shrt arg 1;    /* misspelled or invalid keyword */
double arg 2;
{ /* function body */
}

goodfunction(arg1,arg2)
float arg1;
int arg2;   /* this line is not required */
{ /* function body */
}
```

7: bad declaration syntax

A common cause of this error is the absence of a semicolon at the end of a declaration. The compiler expects a semicolon to follow a variable declaration unless commas appear between variable names in multiple declarations.

```
int i, j;           /* correct */
char c d;             /* error 7 */
char *s1, *s2
float k;            /* error 7 detected here */
```

Sometimes the compiler may not detect the error until the next program line. A missing semicolon at the end of a **#include**'d file will be detected back in the file being compiled or in another **#include** file. This is a good example of why it is important to examine the context of the error rather than to rely solely on the information provided by the compiler error message(s).

8: syntax error in type cast

The syntax of the cast operator must be carefully observed. A common error is to omit a parenthesis:

```
i = 3 * (int number);          /* incorrect usage */
i = 3 * ((int)number);         /* correct usage */
```

9: array dimension must be constant

The dimension given an array must be a constant fo type **char, int,** or **unsigned.** This value is specified in the declaration of the array. See error 10.

10: array size must be positive integer

The dimension of an array is required to be greater than zero. A dimension less than or equal to zero becomes 1 by default. As can be seen from the following example, specifying a dimension of zero is not the same as leaving the brackets empty.

```
char badarray[0];       /* meaningless */
extern char goodarray[];       /* good */
```

Empty brackets are used when declaring an array that has been defined (given a size and storage in memory) somewhere else (that is, outside the current function or file). In the above example, **goodarray** is external. Function arguments should be declared with a null dimension:

```
func(s1,s2)
char s1[], s2[];
{
     ...
}
```

11: data type too complex

This message is best explained by example:

```
char ******foo;
```

The form of this declaration implies five pointers-to-pointers. The sixth asterisk indicates a pointer to a **char.** The compiler is unable to keep track of so many "levels". Removing just one of the asterisks will cure the error; all that is being declared in any case is a single two-byte pointer. However it is to be hoped that such a construct will never be needed.

12: illegal pointer reference

The type of a pointer must be either **int** or **unsigned.** This is why you might get away with not declaring pointer arguments and functions like **fopen** which return a pointer; they default to **int.** When this error is generated, an expression used as a pointer is of an invalid type:

```
char c;
int var;              /* any variable */
int varaddress;
varaddress = &var;        /* valid since addresses */
*(varaddress) = 'c';        /* can fit in an int; */
*(expression) = 10;     /* in general, expression
                   must be an int or unsigned */
*c = 'c';      /* error 12 */
```

13: internal [see error 4]

14: internal [see error 4]

15: storage class conflict

Only automatic variables and function parameters can be specified as **register.**

This error can be caused by declaring a **static register** variable. While structure members cannot be given a storage class at all, function arguments can be specified only as **register.**

A **register int i** declaration is not allowed outside a function--it will generate error 89 (see below).

16: data type conflict

The basic data types are not numerous, and there are not many ways to use them in declarations. The possibilities are listed below.

This error code indicates that two incompatible data types were used in conjunction with one another. For example, while it is valid to say **long int i**, and **unsigned int j**, it is meaningless to use **double int k** or **float char c**. In this respect, the compiler checks to make sure that **int, char, float** and **double** are used correctly.

data type	interpretation	size(bytes)
char	character	1
int	integer	2
unsigned/unsigned int	unsigned integer	2
short	integer	2
long/long integer	long integer	4
float	floating point number	4
long float/double	double precision float	8

17: internal error [see error 4]

18: data type conflict

This message indicates an error in the use of the **long** or **unsigned** data type. **long** can be applied as a qualifier to **int** and **float. unsigned** can be used with **char, int** and **long.**

```
long i;                         /* a long int */
long float d;                   /* a double */
unsigned u;                 /* an unsigned int */
unsigned char c;
unsigned long l;
unsigned float f;               /* error 18 */
```

19: obsolete

Error codes interpreted as obsolete do not occur in release 1.06 of the compiler. Some simply no longer apply due to the increased adaptibility of the compiler. Other error codes have been translated into full messages sent directly to the screen. If you are using an older version of the product and have need of these codes, please contact MANX for information.

20: structure redeclaration

The compiler is able to tell you if a **struct**ure has already been defined. This message informs you that you have tried to redefine a **struct**ure.

21: missing }

The compiler expects to find a comma after each member in the list of fields for a **struct**ure initialization. After the last field, it expects a right (close) brace.

```
struct sam {
     int bone;
     char license[10];
```

```
                } harry = {
                    1,
                    123-4-1984;
```

22: syntax error in structure declaration

The compiler was unable to find the left (open) brace which follows the tag in a **structure** declaration. In the example for error 21, "sam" is the structure tag. A left brace must follow the keyword **struct** if no structure tag is specified.

23: obsolete [see error 19]

24: need right parenthesis or comma

The right parenthesis is missing from a function call. Every function call must have an argument list enclosed by parentheses even if the list is empty. A right parenthesis is required to terminate the argument list.

In the following example, the parentheses indicate that **getchar** is a function rather than a variable.

```
getchar();
```

This is the equivalent of

```
CALL getchar
```

which might be found in a more explicit programming language. In general, a function is recognized as a name followed by a left parenthesis.

With the exception of reserved words, any name can be made a function by the addition of parentheses. However, if a previously defined variable is used as a function name, a compilation error will result.

Moreover, a comma must separate each argument in the list. For example, error 24 will also result from this statement:

```
funccall(arg1, arg2 arg3);
```

25: structure member name expected here

The symbol name following the dot operator or the arrow must be valid. A valid name is a string of alphanumerics and underscores. It must begin with an alphabetic (a letter of the alphabet or an underscore). In the last line of the following example, "(salary)" is not valid because '(' is not an alphanumeric.

```
empptr = &anderson;
empptr->salary = 12000;      /* these three lines */
(*empptr).salary = 12000;          /* are */
anderson.salary = 12000;       /* equivalent */
empptr = &anderson.;         /* error 25 */
empptr-> = 12000;            /* error 25 */
anderson.(salary) = 12000;   /* error 25 */
```

26: must be structure/union member

The defined structure or union has no member with the name specified. If the -S option was specified, no previously defined structure or union has such a member either.

Structure members cannot be created at will during a program. Like other variables, they must be fully defined in the appropriate declaration list. Unions provide for variably typed fields, but the full range of desired types must be anticipated in the union declaration.

27: illegal type cast

It is not possible to cast an expression to a function, a structure, or an array. This message may also appear if a syntax error occurs in the expression to be cast.

```
structure sam { ... } thom;
thom = (struct sam)(expression);    /* error 27 */
```

28: incompatible structures

C permits the assignment of one structure to another. The compiler will ensure that the two structures are identical. However,that both structures must have the same structure tag. For example:

```
struct sam harry;
struct sam thom;
     ...
harry = thom;
```

29: illegal use of structure

Not all operators can accept a structure as an operand. Also, structures cannot be passed as arguments. However, it is possible to take the address of a structure using the ampersand (&), to assign structures, and to reference a member of a structure using the dot operator.

30: missing : in ? conditional expression

The standard syntax for this operator is:

```
expression ? statement1 : statement2
```

It is not desirable to use ?: for extremely complicated expressions; its purpose lies in brevity and clarity.

31: call of non-function

The following represents a function call:

```
symbol(arg1, arg2, ..., argn);
```

where "symbol" is not a reserved word and the expression stands in the body of a function. Error 31, in reference to the expression above, indicates that "symbol" has been previously declared as something other than a function.

A missing operator may also cause this error:

```
a(b + c);                    /* error 31 */
a * (b + c);                 /* intended */
```

The missing '*' makes the compiler view "a()" as a function call.

32: illegal pointer calculation

Pointers may be involved in three calculations. An integral value can be added to or subtracted from a pointer. Pointers to objects of the same type can be subtracted from one another and compared to one another. (For a formal definition, see Kernighan and Ritchie pp. 188-189.) Since the comparison and subtraction of two pointers is dependent upon pointer size, both operands must be the same size.

33: illegal type

The unary minus (-) and bit complement (~) operators cannot be applied to structures, pointers, arrays and functions. There is no reasonable interpretation for the following:

```
int function();
char array[12];
struct sam { ... } harry;
a = -array;                  /* ? */
b = -harry;
c = ~function & WRONG;
```

34: undefined symbol

The compiler will recognize only reserved words and names which have been previously defined. This error is often the result of a typographical error or due to an omitted declaration.

35: typedef not allowed here

Symbols which have been defined as types are not allowed within expressions. The exception to this rule is the use of **sizeof(expression)** and the cast operator. Compare the accompanying examples:

```
struct sam {
     int i;
     } harry;
typedef double bigfloat;
typedef struct sam foo;

j = 4 * bigfloat f;           /* error 35 */
k = &foo;                     /* error 35 */
x = sizeof(bigfloat);
y = sizeof(foo);                /* good */
```

The compiler will detect two errors in this code. In the first assignment, a typecast was probably intended; compare error 8. The second assignment makes reference to the address of a structure type. However, the structure type is just a template for instances of the structure (such as "harry"). It is no more meaningful to take the address of a structure type than any other data type, as in **&int**

36: no more expression space

This message indicates that the expression table is not large enough for the compiler to process the source code. It is necessary to recompile the file using the -E option to increase the number of available entries in the expression table. See the description of the compiler in the manual.

The command sequence should look like this:

```
cc -E500 filename.c
```

37: invalid expression

This error occurs in the evaluation of an expression containing a unary operator. The operand either is not given or is itself an invalid expression.

Unary operators take just one operand; they work on just one variable or expression. If the operand is not simply missing, as in the example below, it fails to evaluate to anything its operator can accept. The unary operators are logical not (!), bit complement (~), increment (++), decrement (--), unary minus (-), typecast, pointer-to (*), address-of (&), and sizeof.

```
               if (!)  ;
```

38: no auto. aggregate initialization

It is not permitted to initialize automatic arrays and structures. Static and external aggregates may be initialized, but by default their members are set to zero.

```
          char array[5] = { 'a', 'b', 'c', 'd' };
          function()
          {
               static struct sam {
                    int bone;
                    char license[10];
               } harry = {
                    1,
                    "123-4-1984"
               };
               char autoarray[2] = { 'f', 'g' };     /* no good */
               extern char array[];
          }
```

There are three variables in the above example, only two of which are correctly initialized. The variable "array" may be initialized because it is external. Its first four members will be given the characters as shown. The fifth member will be set to zero.

The structure "harry" is static and may be initialized. Notice that "license" cannot be initialized without first giving a value to "bone". There are no provisions in C for setting a value in the middle of an aggregate.

The variable "autoarray" is an automatic array. That is, it is local to a function and it is not declared to be static. Automatic variables reappear automatically every time a function is called, and they are guaranteed to contain garbage. Automatic aggregates cannot be initialized.

39: obsolete [see error 19]

40: internal [see error 4]

41: initializer not a constant

In certain initializations, the expression to the right of the equals sign (=) must be a constant. Indeed, only automatic and register variables may be initialized to an expression. Such initializations are meant as a convenient shorthand to eliminate assignment statements. The initialization of statics and globals actiually occurs at link-time, and not at run-time.

```
{
int i = 3;
static int j = (2 + i);        /* illegal */
}
```

42: too many initializers

There were more values found in an initialization than array or structure members exist to hold them. Either too many values were specified or there should have been more members declared in the aggregate definition.

In the initialization of a complex data structure, it is possible to enclose the initializer in a single set of braces and simply list the members, separated by commas. If more than one set of braces is used, as in the case of a structure within a structure, the initializer must be entirely braced.

```
struct {
     struct {
          char array[];
     } substruct;
} superstruct =

version 1:
                   {
                        "abcdefghij"
                   };
version 2:
                    {
                        {
                          { 'a','b','c',...,'i','j'}
                        }
                    };
```

In version 1, the initializers are copied byte-for-byte onto the structure, **superstruct**.

Another likely source of this error is in the initialization of arrays with strings, as in:

```
char array[10]= "abcdefghij";
```

This will generate error 42 because the string constant on the right is null-terminated. The null terminator ('\0' or 0x00) brings the size of the initializer to 11 bytes, which overflows the ten-byte array.

43: undefined structure initialization

An attempt has been made to assign values to a structure which has not yet been defined.

```
struct sam {...};
struct dog sam = { 1, 2, 3};  /* error 43 */
```

44: obsolete [see error 19]

45: bad declaration syntax

This error code is an all purpose means for catching errors in declaration statements. It indicates that the compiler is unable to interpret a word in an external declaration list.

46: missing closing brace

All the braces did not pair up at the end of compilation. If all the preceding code is correct, this message indicates that the final closing brace to a function is missing. However, it can also result from a brace missing from an inner block.

Keep in mind that the compiler accepts or rejects code on the basis of syntax, so that an error is detected only when the rules of grammar are violated. This can be misleading. For example, the program below will generate error 46 at the end even though the human error probably occurred in the **while** loop several lines earlier.

As the code appears here, every statement after the left brace in line 6 belongs to the body of the **while** loop. The compilation error vanishes when a right brace is appended to the end of the program, but the results during run time will be indecipherable because the brace should be placed at the end of the loop.

It is usually best to match braces visually before running the compiler. A C-oriented text editor makes this task easier.

```
main()
{
     int i, j;
     char array[80];

     gets(array);
     i = 0;
     while (array[i])  {
          putchar(array[i]);
          i++;
     for ( i=0; array[i];i++) {
          for (j=i + 1; array[j]; j++)  {
```

```
                    printf("elements %d and %d are ", i,
                    if (array[i] == array[j])
                        printf("the same\n");
                    else printf("different\n");
                }
                putchar('\n');
            }
        }
```

47: open failure on include file

When a file is **#include**d, the compiler will look for it in a default area (see the manual description of the compiler). This message will be generated if the file could not be opened. An open failure usually occurs when the included file does not exist where the compiler is searching for it. Note that a drive specification is allowed in an include statement, but this diminishes flexibility somewhat.

48: illegal symbol name

This message is produced by the preprocessor, which is that part of the compiler which handles lines which begin with a pound sign (#). The source for the error is on such a line. A legal name is a string whose first character is an alphabetic (a letter of the alphabet or an underscore). The succeeding characters may be any combination of alphanumerics (alphabetics and numerals). The following symbols will produce this error code:

```
2nd_time,
dont_do_this!
```

49: multiply defined symbol

This message warns that a symbol has already been declared and that it is illegal to redeclare it. The following is a representative example:

```
int i, j, k, i;          /* illegal */
```

50: missing bracket

This error code is used to indicate the need for a parenthesis, bracket or brace in a variety of circumstances.

51: lvalue required

Only **lvalues** are are allowed to stand on the left-hand side of an assignment. For example:

```
                int num;
                num = 7;
```

They are distinguished from **rvalues**, which can never stand on the left of an assignment, by the fact that they refer to a unique location in memory where a value can be stored. An **lvalue** may be thought of as a bucket into which an **rvalue** can be dropped. Just as the contents of one bucket can be passed to another, so can an lvalue **y** be assigned to another lvalue, **x**:

```
        #define NUMBER   512
        x = y;
        1024 = z;     /* wrong; l/rvalues are reversed */
        NUMBER = x;   /* wrong; NUMBER is still an rvalue */
```

Some operators which require **lvalues** as operands are increment (++), decrement (--), and address-of (&). It is not possible to take the address of a register variable as was attempted in the following example:

```
                register int i, j;
                i = 3;
                j = &i;
```

52: obsolete [see error 16]

53: multiply defined label

On occasions when the goto statement is used, it is important that the specified label be unique. There is no criterion by which the computer can choose between identical labels. If you have trouble finding the duplicate label, use your text editor to search for all occurrences of the string.

54: too many labels

The compiler maintains an internal table of labels which will support up to several dozen labels. Although this table is fixed in size, it should satisfy the requirements of any reasonable C program. C was structured to discourage extravagance in the use of goto's. Strictly speaking, goto statements are not required by any procedure in C; they are primarily recommended as a quick and simple means of exiting from a nested structure.

This error indicates that you should significantly reduce the number of goto's in your program.

55: missing quote

The compiler found a mismatched double quote (") in a **#define** preprocessor command. Unlike brackets, quotes are not paired innermost to outermost, but sequentially. So the first quote is associated with the second, the third with the fourth, and so on. Single quotes (') and double quotes (") are entirely different characters and should not be confused. The latter are used to delimit string constants. A double quote can be included in a string by use of a backslash, as in this example:

```
"this is a string"
"this is a string with an embedded quote: \". "
```

56: missing apostrophe

The compiler found a mismatched single quote or apostrophe (') in a **#define** preprocessor command. Single quotes are paired sequentially (see error 55). Although quotes can not be nested, a quote can be represented in a character constant with a backslash:

```
char c = '\'';        /* c is initialized to
                            single quote  */
```

57: line too long

Lines are restricted in length by the size of the buffer used to hold them. This restriction varies from system to system. However, logical lines can be infinitely long by continuing a line with a backslash-newline sequence. These characters will be ignored.

58: illegal # encountered

The pound sign (#) begins each command for the preprocessor: **#include, #define, #if, #ifdef, #ifndef, #else, #endif, #asm, #endasm, #line and #undef.** These symbols are strictly defined. The pound sign (#) must be in column one and lower case letters are required.

59: macro too long

Macros can be defined with a preprocessor command of the following form:

#define [identifier] [substitution text]

The compiler then proceeds to replace all instances of "identifier" with the substitution text that was specified by the **#define**.

This error code refers to the substitution text of a macro. Whereas ideally a macro definition may be extended for an arbitrary number of lines by ending each line with a backslash (\), for practical purposes the size of a macro has been limited to 255 characters.

60: obsolete [see error 19]

61: reference of member of undefined structure

Occurs only under compilation without the -S option. Consider the following example:

```
int bone;
struct cat {
     int toy;
} manx;
struct dog *samptr;
manx.toy = 1;
bone = samptr->toy;        /* error 61 */
```

This error code appears most often in conjunction with this kind of mistake. It is possible to define a pointer to a structure without having already defined the structure itself. In the example, **samptr** is a structure pointer, but what form that structure ("dog") may take is still unknown. So when reference is made to a member of the structure to which **samptr** points, the compiler replies that it does not even known what the structure looks like.

The **-S** compiler option is provided to duplicate the manner in which earlier versions of UNIX treated structures. Given the example above, it would make the compiler search all previously defined structures for the member in question. In particular, the value of the member "toy" found in the structure "manx" would be assigned to the variable "bone". The -S option is not recommended as a short cut for defining structures.

62: function body must be compound statement

The body of a function must be enclosed by braces, even though it may consist of only one statement:

```
function()
{
     return 1;
}
```

63: undefined label

A **goto** statement is meaningless if the corresponding label does not appear somewhere in the code. The compiler disallows this since it must be able to specify a destination to the computer.

It is not possible to goto a label outside the present function (labels are local to the function in which they appear). Thus, if a label does not exist in the same procedure as its corresponding goto, this message will be generated.

64: inappropriate arguments

When a function is declared (as opposed to defined), it is poor syntax to specify an argument list:

```
function(string)
char *string;
{
     char *func1();               /* correct */
     double func2(x,y);           /* wrong */
          ...
}
```

In this example, function() is being defined, but func1() and func2() are being declared.

65: illegal or missing argument name

The compiler has found an illegal name in a function argument list. An argument name must conform to the same rules as variable names, beginning with an alphabetic (letter or underscore) and continuing with any sequence of alphanumerics and underscores. Names must not coincide with reserved words.

66: expected comma

In an argument list, arguments must be separated by commas.

67: invalid else

An **else** was found which is not associated with an **if** statement. **else** is bound to the nearest **if** at its own level of nesting. So if-else pairings are determined by their relative placement in the code and their grouping by braces.

```
if(...)  {
     ...
     if (...)  {
          ...
     } else if (...)
          ...
     } else {
     ...
     }
```

The indentation of the source text should indicate the intended structure of the code. Note that the indentation of the if and else-if means only that the programmer wanted both conditionals to be nested at the same level, in particular one step down from the presiding if statement. But it is the placement of braces that determines this for the compiler. The example above is correct, but probably does not conform to the expectations revealed by the indentation of the else statement. As shown here, the else is paired with the first if, not the second.

68: syntax error

The keywords used in declaring a variable, which specify storage class and data type, must not appear in an executable statement. In particular, all local declarations must appear at the beginning of a block, that is, directly following the left brace which delimits the body of a loop, conditional or function. Once the compiler has reached a non-declaration, a keyword such as **char** or **int** must not lead a statement; compare the use of the casting operator:

```
func()
{
     int i;
     char array[12];
     float k = 2.03;

     i = 0;
     int m;                        /* error 68 */
     j = i + 5;
     i = (int)k;                   /* correct */
     if (i) {
          int i = 3;
          j = i;
          printf("%d",i);
     }
     printf("%d%d\n",i,j);
}
```

This trivial function prints the values 3, 2 and 3. The variable i which is declared in the body of the conditional (if) lives only until the next right brace; then it dies, and the original i regains its identity.

69: missing semicolon

A semicolon is missing from the end of an executable statement. This error code is subject to the same vagaries as its cousin, error 7. It will remain undetected until the following line and is often spuriously caused by a previous error.

70: bad goto syntax

Compare your use of goto with an example. This message says that you did not specify where you wanted to goto with a label:

```
                goto label;
                    ...
            label:
                    ...
```

It is not possible to goto just any identifier in the source code; labels are special because they are followed by a colon.

71: statement syntax error in do-while

The body of a **do-while** may consist of one statement or several statements enclosed in braces. A **while** conditional is required after the body of the loop. This is true even if the loop is infinite, as it is required by the rules of syntax. After typing in a long body, don't forget the **while** conditional.

72: statement syntax error in for

This error focuses on another control flow statement, the **for**. The keyword, **for**, must be followed by parentheses. In the parentheses belong three expressions, any or all of which may be null. For the sake of clarity, C requires that the two semicolons which separate the expressions be retained, even if all three expressions are empty.

```
for (;;)        /* an infinite loop which does */
    ;          /* absolutely nothing */
```

73: statement syntax error in for

See error 72.

74: case value must be integer constant

Strictly speaking, each value in a **case** statement must be a constant of one of three types: **char, int** or **unsigned.** This is similar to the rule for a **switch**ed variable. In the following example, a float must be cast to an int in order to be switched; however, notice that the programmer did not check his case statements. The second case value is invalid, and the code will not compile.

```
float k = 5.0;
switch((int)k)  {
case 4:
     printf("good case value\n");
     break;
case 5.0:
     printf("bad case value\n");
     break;
}
```

The programmer must replace "case 5.0:" with "case 5".

75: missing colon on case

This should be straightforward. If the compiler accepts a case value, a colon should follow it. A semi-colon must not be accidently entered in its place.

76: too many cases in switch

The compiler reserves a limited number of spaces in an internal table for **case** statements. If a program requires more cases than the table initially allows, it becomes necessary to tell the compiler what the table value should be changed to. It is not necessary to know exactly how many are needed; an approximation is sufficient, depending on the requirements of the situation.

The following command line will change the size of the case table to 200 entries for the compilation of the file, switchit.c:

```
cc -Y200 switchit.c
```

77: case outside of switch

The keyword, **case,** belongs to just one syntactic structure, the **switch.** If "case" appears outside the braces which contain a switch statement, this error is generated. Remember that all keywords are reserved, so that they cannot be used as variable names.

78: missing colon

This message indicates that a colon is missing after the keyword, **default.** Compare error 75.

79: duplicate default

The compiler has found more than one **default** in a **switch.** Switch will compare a variable to a given list of values. But it is not always possible to anticipate the full range of values which the variable may take. Nor is it feasible to specify a large number of cases in which the program is not particularly interested.

So C provides for a default case. The default will handle all those values not specified by a case statement. It is analogous to the else companion to the conditional, if. Just as there is one else for every if, only one default case is allowed in a switch statement. However, unlike the else statement, the position of a default is not crucial; a default can appear anywhere in a list of cases.

80: default outside of switch

The keyword, "default", is used just like "case". It must appear within the brackets which delimit the switch statement.

81: break/continue error

Break and continue are used to skip the remainder of a loop in order to exit or repeat the loop. Break will also end a switch statement. But when the keywords, **break** or **continue,** are used outside of these contexts, this message results.

82: illegal character

Some characters simply do not make sense in a C program, such as '$' and '@'. Others, for instance the pound sign (#), may be valid only in particular contexts.

83: too many nested includes

#includes can be nested, but this capacity is limited. The compiler will balk if required to descend more than three levels into a nest. In the example given, file D is not allowed to have a #include in the compilation of file A.

```
    file A        file B          file C        file D
  #include "B"  #include "C"    #include "D"
```

84: too many array dimensions

An array is declared with too many dimensions. This error should appear in conjunction with error 11.

85: not an argument

The compiler has found a name in the declaration list that was not in the argument list. Only the converse case is valid, i.e., an argument can be passed and not subsequently declared.

86: null dimension in array

In certain cases, the compiler knows how to treat multidimensional arrays whose left-most dimensions are not given in its declaration. Specifically, this is true for an extern declaration and an array initialization. The value of any dimension which is not the left-most must be given.

```
extern char array[][12];       /* correct */
extern char badarray[5][];     /* wrong */
```

87: invalid character constant

Character constants may consist of one or two characters enclosed in single quotes, as 'a' or 'ab'. There is no analog to a null string, so '' (two single quotes with no intervening white space) is not allowed. Recall that the special backslash characters (\b, \n, \t etc.) are singular, so that the following are valid: '\n', '\na', 'a\n'; 'aaa' is invalid.

88: not a structure

Occurs only under compilation without the -S option. A name used as a structure does not refer to a structure, but to some other data type.

```
int i;
i.member = 3;             /* error 88 */
```

89: invalid storage class

A globally defined variable cannot be specified as register. Register variables are required to be local.

90: symbol redeclared

A function argument has been declared more than once.

91: illegal use of floating point type

Floating point numbers can be negated (unary minus), added, subtracted, multiplied, divided and compared; any other operator will produce this error message.

92: illegal type conversion

This error code indicates that a data type conversion, implicit in the code, is not allowed, as in the following piece of code:

```
int i;
float j;
char *ptr;
...
i = j + ptr;
```

The diagram shows how variables are converted to different types in the evaluation of expressions. Initially, variables of type **char** and **short** become **int**, and **float** becomes **double.** Then all variables are promoted to the highest type present in the expression. The result of the expression will have this type also. Thus, an expression containing a **float** will evaluate to a **double.**

```
hierarchy of types:

double <-- float
long
unsigned
int <-- short, char
```

93: illegal expression type for switch

Only a **char, int** or **unsigned** variable can be switched. See the example for error 74.

94: bad argument to define

An illegal name was used for an argument in the definition of a macro. For a description of legal names, see error 65.

95: no argument list

When a macro is defined with arguments, any invocation of that macro is expected to have arguments of corresponding form. This error code is generated when no parenthesized argument list was found in a macro reference.

```
#define   getchar()   getc(stdin)
    ...
    c = getchar;         /* error 95 */
```

96: missing argument to macro

Not enough arguments were found in an invocation of a macro. Specifically, a "double comma" will produce this error:

```
#define reverse(x,y,z)   (z,y,x)

func(reverse(i,,k));
```

97: obsolete [see error 19]

98: not enough args in macro reference

The incorrect number of arguments was found in an invocation of a previously defined macro. As the examples show, this error is not identical to error 96.

```
#define   exchange(x,y)    (y,x)

func(exchange(i));       /* error 98 */
```

99: internal [see error 4]

100: internal [see error 4]

101: missing close parenthesis on macro reference

A right (close) parenthesis is expected in a macro reference with arguments. In a sense, this is the complement of error 95; a macro argument list is checked for both a beginning and an ending.

102: macro arguments too long

The combined length of a macro's arguments is limited. This error can be resolved by simply shortening the arguments with which the macro is invoked.

103: #else with no #if

Correspondence between #if and #else is analogous to that which exists between the control flow statements, if and else.

Obviously, much depends upon the relative placement of the statements in the code. However, #if blocks must always be terminated by #endif, and the #else statement must be included in the block of the #if with which it is associated. For example:

```
#if ERROR > 0
        printf("there was an error\n");
#else
        printf("no error this time\n");
#endif
```

#if statements can be nested, as below. The range of each #if is determined by a #endif. This also excludes #else from #if blocks to which it does not belong:

```
#ifdef  JAN1
        printf("happy new year!\n");
#if     sick
        printf("i think i'll go home now\n");
#else
        printf("i think i'll have another\n");
#endif
#else
        printf("i wonder what day it is\n");
#endif
```

If the first #endif was missing, error 103 would result. And without the second #endif, the compiler would generate error 107.

104: #endif with no #if

#endif is paired with the nearest #if, #ifdef or #ifndef which precedes it. (See error 103.)

105: #endasm with no #asm

#endasm must appear after an associated #asm. These compiler-control lines are used to begin and end embedded assembly code. This error code indicates that the compiler has reached a #endasm without having found a previous #asm. If the #asm was simply missing, the error list should begin with the assembly code (which are undefined symbols to the compiler).

106: #asm within #asm block

There is no meaningful sense in which in-line assembly code can be nested, so the #asm keyword must not appear between a paired #asm/#endasm. When a piece of in-line assembly is augmented for temporary purposes, the old #asm and #endasm can be enclosed in comments as place-holders.

```
#asm
```

```
                    /* temporary asm code */
        /*   #asm            old beginning   */
                    /* more asm code */
             #endasm
```

107: missing #endif

A #endif is required for every #if, #ifdef and #ifndef, even if the entire source file is subject to a single conditional compilation. Try to assign pairs beginning with the first #endif. Backtrack to the previous #if and form the pair. Assign the next #endif with the nearest unpaired #if. When this process becomes greatly complicated, you might consider rethinking the logic of your program.

108: missing #endasm

In-line assembly code must be terminated by a #endasm in all cases. #asm must always be paired with a #endasm.

109: #if value must be integer constant

#if requires an integral constant expression. This allows both integer and character constants, the arithmetic operators, bitwise operators, the unary minus (-) and bit complement, and comparison tests.

Assuming all the macro constants (in capitals) are integers,

```
#if DIFF >= 'A'-'a'
#if (WORD &= ~MASK) >> 8
#if MAR | APR | MAY
```

are all legal expressions for use with #if.

110: invalid use of colon operator

The colon operator occurs in two places: 1. following a question mark as part of a conditional, as in (flag ? 1 : 0); 2. following a label inserted by the programmer or following one of the reserved labels, **case** and **default.**

111: illegal use of a void expression

This error can be caused by assigning a **void** expression to a variable, as in this example:

```
void func();
int h;

h = func(arg);
```

112: illegal use of function pointer

For example,

```
int (*funcptr) ();
     ...
funcptr++;
```

funcptr is a pointer to a function which returns an integer. Although it is like other pointers in that it contains the address of its object, it is not suject to the rules of pointer arithmetic. Otherwise, the offending statement in the example would be interpreted as adding to the pointer the size of the function, which is not a defined value.

113: duplicate case in switch

This simply means that, in a **switch** statement, there are two **case** values which are the same. Either the two **cases** must be combined into one, or one of them must be discarded. For instance:

```
switch (c)  {
     case NOOP:
          return (0);
     case MULT:
          return (x * y);
     case DIV:
          return (x / y);
     case ADD:
          return (x + y);
     case NOOP:
     default:
          return;
}
```

The case of NOOP is duplicated, and will generate an error.

114: macro redefined

For example,

```
#define   islow(n)  (n>=0&&n<5)
     ...
#define   islow(n)  (n>=0&&n<=5)
```

The macro, **islow**, is being used to classify a numerical value. When a second definition of it is found, the compiler will compare the new substitution string with the previous one. If they are found to be different, the second definition will become current, and this error code will be produced.

In the example, the second definition differs from the first in a single character, '='. The second definition is also different from this one:

```
#define   islow(n)  n>=0&&n<=5
```

since the parentheses are missing.

The following lines will not generate this error:

```
#define   NULL 0
     ...
#define   NULL 0
```

But these are different from:

```
#define   NULL '\0'
```

In practice, this error message does not affect the compilation of the source code. The most recent "revision" of the substitution string is used for the macro. But relying upon this fact may not be a wise habit.

115: keyword redefined

Keywords cannot be defined as macros, as in:

```
#define   int   foo
```

If you have a variable which may be either, for instance, a short or a long integer, there are alternative methods for switching between the two. If you want to compile the variable as either type of integer, consider the following:

```
#ifdef    LONGINT
     long i;
#else
     short i;
#endif
```

Another possibility is through a **typedef**:

```
#ifdef    LONGINT
     typedef   long      VARTYPE;
#else
     typedef   short     VARTYPE;
#endif

VARTYPE i;
```

Fatal Error Messages

If the compiler encouters a "fatal" error, one which makes further operation impossible, it will send a message to the screen and end the compilation immediately.

1. Out of disk space!

   There is no room on the disk for the output file of the compiler. Previous disk files will not be overwritten by the compiler's assembly language output. To make room on the disk, it is usually sufficient to remove unneeded files from the disk.

2. unknown option:

   The compiler has been invoked with an option letter which it does not recognize. The manual explicitly states which options the compiler will accept. The compiler will specify the invalid option letter.

3. duplicate output file

   If an output file name has been specified with the -o option and that file already exists on the disk, the compiler will not overwrite it. -O must specify a new file.

4. too few arguments for -o option

   Usage of the -o option is as follows:

```
cii -o newfile.asm oldfile.c
```

   The new file name must follow the option letter and the name of the file to be compiled must occur last in the command line.

5. Open failure on input

   The input file specified in the command line does not exist on the disk or cannot be opened. A path or drive specification can be included with a filename according to the operating system in use.

6. No input!

   While the compiler was able to open the input file given in the command line, that file was found to be empty.

7. Open failure on output

8. Local table full! (use -L)

   The compiler maintains an internal table of the local variables in the source code. If the number of local symbols in use exceeds the available entries in the table at any time during compilation, the compiler will print this message and quit. The default size of the local symbol table (40 entries) can be changed with the -L option for the compiler:

```
        cii -L100 filename.c
```

Local variables are those defined within braces, i.e., in a function body or in a compound statement. The scope of a local variable is the body in which it is defined, that is, it is defined until the next right brace at its own nesting level.

9. Out of memory!

Since the compiler must maintain various tables in memory as well as manipulate source code, it may run out of memory during operation. The more immediate solution is to vary the sizes of the internal tables using the appropriate compiler options. Often, a compilation will require fewer than the default number of entries in a particular table. By reducing the size of that table, memory space is freed up during compile time. The amount of memory used while compiling does not affect the size or content of the assembly or object file output.

If this stategy fails to squeeze the compilation into the available memory, the only solution is to divide the source file into modules which can be compiled separately. These modules can then be linked together to produce a single executable file.

ASSEMBLER ERROR CODES

Most errors during assembly are caused by assembly language code written by the programmer. The assembler supplied by Manx is tailored to accept code generated by the Aztec compiler. However, the programmer is free to write his own assembler code either in-line or as separate modules, and to modify the output of the compiler.

1. No space for expression work area!!

or

No symbol table space left!!!

Both of these messages essentially mean "out of memory."

2. <symbol name> is undefined

The message will specify an undefined symbol name.

3. invalid character in number

A digit is not within the range of valid numerals for the particular base. For example, 'F' is invalid in an octal or decimal number.

4. Invalid expression for block allocation

The operand given a define storage opcode must be an absolute expression, i.e., all symbols in it must have been previously defined.

5. Multiply defined symbol

A symbol has been defined more than once.

6. Cannot redefine symbol

The left hand side of an "equ" is a symbol which cannot be redefined, such as a code label.

7. Cannot open file

An included file cannot be opened.

8. Too many nested includes

Included files have been nested too deeply.

9. Operand out of range

The operand does not fall within the possible range of values for the given opcode, e.g., RST 8.

10. Operand must be even register

For example, L was referenced instead of H in the HL register pair.

11. Global size must be absolute

or

BSS size must be absolute

The second operand given the **global** or **BSS** opcode must be an absolute expression.

12. Location counter must always increase

I.e., the location counter cannot be reset backward.

13. can't change type of location counter

For example,

```
        . equ  dseg_symbol
```

14. Cannot equ a common block name

The left-hand symbol of an **equ** cannot be the name of a common block.

15. Null string not allowed here

A null string ('') is not allowed in an expression.

16. Too many chars in CHAR constant

A character constant may consist of up to two characters. Any more will be ignored by the assembler.

17. invalid operator in evaluate

This is an internal assembler error. Try reassembling with new copies of the assembler and your file to ensure that neither is bad. If this error is produced consistently, please contact Manx with the problem.

Linker Errors

Command line errors:

1. unknown option '<bad option letter>'
2. too few arguments in command line.
3. No input given!
4. Cannot have nested -f options.
5. too few arguments in -f file: <filename>
6. multiple <origin> declarations, last one used.

I/O errors:

7. can't open <filename>, err=<error number>
8. Cannot open -f file: <filename>
9. I/O error (<error number>) reading/writing output file
10. Cannot write output file
11. Cannot create output file: <filename>
12. Cannot create symbol table output
13. Cannot create overlay symbol table output

Corrupted object files:

14. object file is bad!
15. invalid operator in evaluate <hex value>
16. library format is invalid!
17. Cannot read module from <input> on pass2
    can't find symbol, <symbol name>, on pass two
18. <filename> is not a rel file!

Errors in use of Memory:

19. Insufficient memory!
20. Too many symbols!
21. -C or -D value less than base address
22. Code and data regions overlap

Errors arising from source code:

23. Undefined symbol: <symbol name>
24. <symbol name> multiply defined
25. pass1(<hex value>) and pass2(<hex value>) values differ:
    symbol type differs on pass two: <symbol name>
26. Org out of range in object file
27. undefined COMMON <symbol name>

LINKER ERROR CODES

When invoked by a command line beginning with LN, the linker will process the arguments given it, perform the linkage requested and generate an executable output file on the disk. The first line to appear on the screen is a banner which indicates that the linker has been loaded and is running. If everything goes well, the base address message will follow and the linker will finish. The linker may encounter an error while it is running, in which case it will send a verbal message to the screen.

Errors may be reported at a variety of points during the linking process. LN consists of two stages, known as pass 1 and pass 2. The base address message is printed at the end of pass 1, so any errors occurring after that have been detected during pass 2 of the linker.

Following is a list of the messages which the linker will generate in response to an error. The messages are grouped according to the source of the errors which cause them. Elements which are variable are enclosed by angled brackets: <>.

Command line errors:

1. unknown option '<bad option letter>'

An option letter has been specified which the linker does not recognize. Only the letter will be ignored; everything else on the command line has been preserved, and the linker will try to execute what it has interpreted. See page [] for a list of options which are supported.

2. too few arguments in command line.

Several of the linker options have an associated value or name, such as -B 2000. If a needed value is missing, the linker will give this message and die.

3. No input given!

The linker will quit immediately if not given any input to process.

4. Cannot have nested -f options.

A file which is given as a -f argument can contain any option letter except -f itself. However, more than one -f is allowed on a command line.

5. too few arguments in -f file: <filename>

An option letter specified in the file, "filename," requires a value or name to follow it. If an option appears at the end of the file, its associated value may not appear back on the command line.

6. multiple <origin> declarations, last one used.

The message will specify that one of the following option letters was specified more than once in the command line:

-C	code org	origin of code in memory
-D	data org	origin of data
-U	udata org	origin of uninitialized global/static data
-B	base	base address of memory

The linker will consider only the last use of an option letter when it runs.

I/O errors:

7. can't open <filename>, err=<errno>

If any file in the command line cannot be opened, this message will be sent to the screen, specifying the filename and the current value of errno.

8. Cannot open -f file: <filename>

A file given with the -f option cannot be opened.

9. I/O error (<errno>) reading/writing output file

An error reading or writing the output file probably means there is no more disk space available. In particular, a block of the output file was written to disk and then could not be read back. The current value of errno is given in these messages.

10. Cannot write output file

See error 9.

11. Cannot create output file: <filename>

This message usually indicates that all available directory space on the disk has been exhausted.

12. Cannot create symbol table output

The -T option was given in the command line, but the file containing the linkage symbol table cannot be written to disk. It is possible that there is no more space on the disk.

13. Cannot create overlay symbol table output

Occurs when using the -R option. The file containing the overlay symbol table cannot be written to disk.

Corrupted object files:

14. object file is bad!

This is the most explicit indication that an object file in the linkage has been corrupted. The solution is simply to recompile and assemble the source file. A bad object file will not be discovered until the second pass of the linker.

15. invalid operator in evaluate <hex value>

This is really the same as error 14. Unless you have changed the object code by hand, the file has been corrupted.

16. library format is invalid!

A library in the linkage has been corrupted.

17. Cannot read module from <input> on pass2

or

can't find symbol, <symbol name>, on pass two

Either message indicates that a module has been corrupted between pass 1 and pass 2. On a multiuser system, it is possible that another user changed the file while the linker was running. Otherwise, the error was probably due to a hardware failure.

18. <filename> is not a rel file!

A file given to the linker does not contain relocatable object code which LN can process. For instance, a source file may have been included in the link.

Errors in use of memory:

19. Insufficient memory!

The linkage process needs memory space for LN, global and local symbol tables, and approximately 5K for buffers. Just as with compilation, most memory use is devoted to the program software and symbol tables. Since LN is not especially large, only an extremely complicated linkage might run out of memory.

20. Too many symbols!

This is another way of saying that not enough memory was available for the symbol tables needed for the linkage.

21. -C or -D value less than base address

It is not possible for the starting address of the code or data to be less than the base address of the program, which is specified by the option, -B.

22.

By default, data resides above the code area in memory. The starting addresses of both areas must be spaced far enough apart to accommodate all the code. If the -D option is used to begin the data area in the middle of the code, this error message will be put out.

Errors arising from source code:

23. Undefined symbol: <symbol name>

A global symbol name has remained undefined. This is commonly a function which has been referenced in the source code but not included anywhere in the link.

24. <symbol name> multiply defined

A global symbol has been defined more than once. For instance, if two functions are accidentally given the same name, this message will be generated.

25. pass1(<hex value>) and pass2(<hex value>) values differ:

or

symbol type differs on pass two: <symbol name>

Either of these errors may be generated during pass 2 when error 24 appeared in pass 1. They may be considered a confirmation of what was discovered in pass 1 of the linker.

26. Org out of range in object file

On the source code level, this means that a global symbol was defined in the root of an overlay and then initialized in an overlay module. For example,

```
root:                              overlay:

int i;                             extern int i = 3;
```

The problem arises because the initialization is performed by the linker, but the variable to be initialized is in an entirely different file.

The situation which follows is valid because the assignment statement is evaluated at run time:

```
          root:                                  overlay:

          int i;                                 extern int i;
                                                 function()
                                                 {
                                                      extern int i;
                                                      i = 3;
                                                 }
```

27. undefined COMMON <symbol name>

This error now occurs only in reference to the user's own assembly language routines. It is generated by a COMMON block of size zero.