TECHNICAL INFORMATION

Chapter Contents

Technical Information

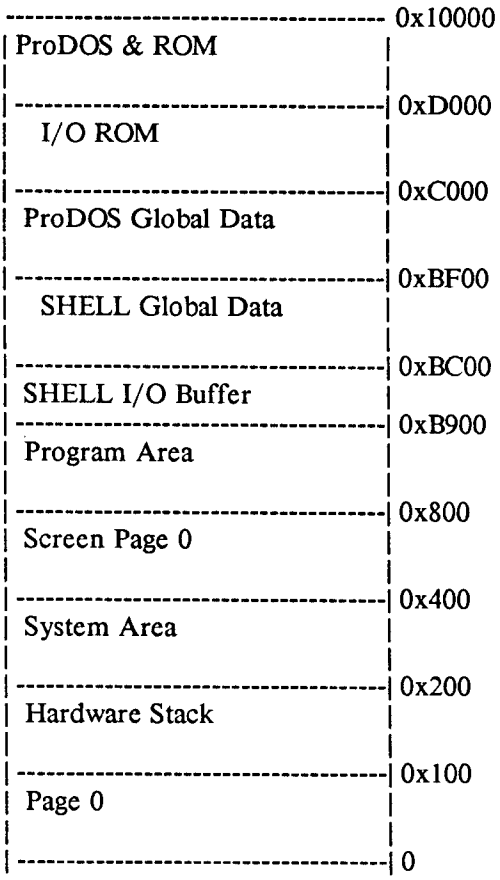
This chapter discusses technical topics, and topics that couldn't be conveniently discussed elsewhere.

It's divided into the following sections:

1. *Memory Organization.* Discusses the factors that affect the memory organization of a program.
2. *Command Programs.* Discusses the different types of programs that you can create using Aztec C65.
3. *Overlays.* Describes overlays: what they are, and how they're used.
4. *Libraries.* Discusses the object module libraries that are provided with Aztec C65.
5. *Mixing Assembler and C Routines.* Describes how to interface assembly language routines with C routines.
6. *Debugging Pseudo Code.* Describes how to debug *cci*-compiled functions using the monitor.
7. *Object Code Format.* Describes the format of object modules and libraries.
8. *DOS 3.3 Programs.* Discusses the creation of programs that run under DOS 3.3.
9. *The tmpdev console driver.* Describes a stripped-down console driver that can be included in programs in place of the standard console driver.

1. Memory Organization

For a PRG or BIN program running under ProDOS and the SHELL, memory is organized as follows:



For a BIN program running under ProDOS and the Basic Interpreter, memory is organized similarly, except for the following differences:

* The SHELL Areas are not present;
* The area between 0x9600 and 0xBF00 is occupied by the Basic Interpreter;
* The Program Area is between 0x800 and the value defined by HIMEM (which is somewhere below 0x9600).

For a SYS program running under ProDOS, memory is organized similarly, with the following differences:

* The SHELL Areas are not present;
* The Program Area is between 0x2000 (by default) and

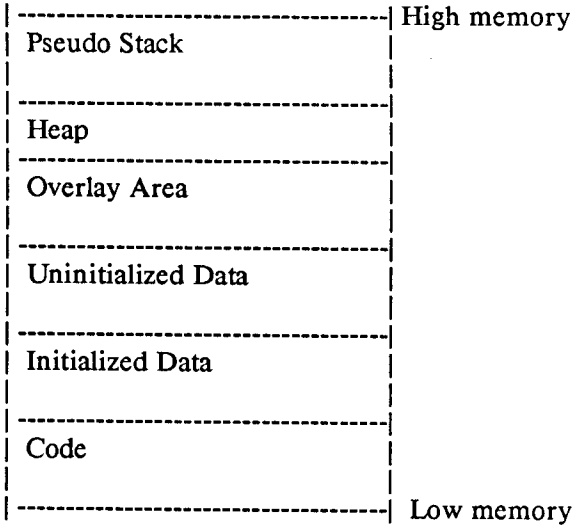
0xBF00

For a program running under DOS 3.3, the program area extends from 0x800 to the value defined by HIMEM; the areas above HIMEM are occupied by Basic and DOS, as defined in the DOS 3.3 documentation.

The following paragraphs discuss the program area and page 0 in more detail.

1.1 The Program Area

A program is loaded into the program area. By default, a program is organized into the following sections:



The following paragraphs discuss these areas.

1.1.1 The Code Area

The code area contains the executable code and the constants for a program's root segment (ie, for its non-overlay segment). The area is just large enough to contain the program's code and constants.

1.1.2 The Initialized and Uninitialized Data Areas

These two areas contain a program's global and static variables.

A variable is placed in the initialized data area if its definition assigns an initial value to the variable. For example, a variable that's defined outside all functions with the following statement would be placed in the initialized data area:

```
int i=1;
```

Global and static variables for which no initial value is specified are placed in the uninitialized data area.

When a program starts, the Manx-supplied startup code clears the uninitialized data area.

These areas are just large enough to contain a program's global and static data.

1.1.3 The Overlay Area

A program's overlays are loaded into the overlay area. The size of this area is set when you link the program's root segment, to the sum of the values specified in the +C and +D options. By default, these options are set to zero, resulting in an overlay area that is zero bytes long.

For more information on overlays, see the Overlay section of this chapter.

1.1.4 The Heap

The heap is the area of memory from which buffers are dynamically allocated.

A program doesn't have direct control over the size of this area. However, since this area is bounded below by the overlay area and above by the pseudo stack area, a program can indirectly control the size of this area by changing the size of the pseudo stack area, by calling the *rsvstk* function.

1.1.5 The Pseudo Stack Areas

The hardware stack is at most 256 bytes long. Since a C program makes heavy use of a stack, frequently requiring a stack that contains more than 256 bytes, Aztec C has implemented a pseudo stack, whose size is limited only by the amount of available memory.

The top of the pseudo stack area depends on the program and on the environment in which it is running:

* For a PRG program running under ProDOS and the SHELL, the pseudo stack top is just below the SHELL's I/O Buffer. If the program was activated using I/O redirection of its standard I/O devices and was passed command line arguments, the pseudo stack top will be pushed downward to make room for the the redirected devices' ProDOS I/O buffers and for the command line arguments.
* For a BIN program running under ProDOS and the SHELL, the pseudo stack top is always immediately below the SHELL's I/O buffer area. (The standard I/O devices of a BIN program can't be redirected, and it can't be passed command line arguments.)
* For a BIN program running under ProDOS and the Basic

Interpreter, or for a program running under DOS 3.3, the pseudo stack top is at the location defined by HIMEM.

* For a SYS program running under ProDOS, the pseudo stack top is 0xBF00.

The base of the pseudo stack area is by default 2K bytes below its top. A program can change the size of the pseudo stack area by calling the *rsvstk* function.

1.1.6 Symbols related to Program Organization

The following global symbols are related to program organization. The symbols are given in the form that an assembly language program would use to access them.

__Corg__ Name of the beginning of the program's code.

__Cend__ Name of the first byte beyond the program's executable code.

__Dorg__ Name of the beginning of the program's initialized data.

__Dend__ Name of the first byte beyond the program's initialized data.

__Uorg__ Name of the beginning of the program's uninitialized data.

__Uend__ Name of the first byte beyond the program's uninitialized data.

__mbot__ Name of a field containing a pointer to the beginning of the program's heap.

__Top__ Name of a field containing a pointer to the next byte to be allocated from the heap.

A C module can access all the above symbols by removing the appended underscore from the symbol name.

1.2 Page 0

An Aztec C-generated program makes the following use of page 0:

Location	*Use*
0- 1	Temporary storage
2- 3	Pointer to top of pseudo stack
4- 5	Pointer to current function's frame
6- 7	Pointer to current pseudo-code instruction
8- B	Temporary Register R0
C- F	Temporary Register R1
10-13	Temporary Register R2
14-17	Temporary Register R3
18-1B	Temporary Register R4
1C-1D	Pointer to floating point accumulator
1E-1F	Pointer to floating point secondary
...	
40-43	Temporary storage
...	
80-8F	Storage for program's register variables

2. Command Programs

Using the standard Aztec software, you can create programs of type PRG, BIN, and SYS that run on ProDOS; and binary programs that run on DOS 3.3. The main points of difference between the various types of ProDOS programs are:

* *Program activation.* A PRG program only be started from within the SHELL environment. A BIN or SYS program can be started either from within the SHELL environment or by the Basic Interpreter. A SYS program whose extension is ".system" can be started by ProDOS during system startup.
* *I/O redirection.* The standard i/o devices of PRG programs can be redirected from the console to another device or file. Those of BIN and SYS programs cannot be redirected.
* *Argument passing.* When a PRG program is started, values can be passed to it. The program sees the values as arguments of its *main* function. Values cannot be passed to BIN and SYS programs.
* *Starting other programs.* A PRG program can start other programs, by calling any of the the *exec* functions that are described in the *ProDOS Functions* chapter. A BIN program can use the *exec* functions to start other programs when it is active within the SHELL environment, but not when it's in the Basic Interpreter environment. A SYS program can never use the *exec* function to start another program.
* *Protection of the SHELL environment.* The SHELL keeps global information about its environment, which it needs to maintain even during execution of programs, in a section of memory just below the ProDOS global page. Execution of a PRG or BIN program within the SHELL environment will not modify this global information.

  When a SYS program is started from within the SHELL environment, it probably will destroy the SHELL global pages. This destruction doesn't necessarily occur, however: when you restart the SHELL with the SHELL already active, in order to redefine the file from which the SHELL will be loaded, the global information that was set up by the original SHELL is not modified.
* *Passing configuration information.* A PRG program knows about the Apple configuration that has been defined to the SHELL, and hence can make use of the configuration's special features without being explicitly told. A BIN or SYS program must be explicitly told about the configuration, even when the program is started by the SHELL.
* *Passing open files and devices.* When one PRG program starts another, any files or devices that were left open for unbuffered i/o in the calling program will still be open for

unbuffered i/o in the called program. The called program can access them using the same file descriptors as did the calling program.

A DOS 3.3 program runs in the Basic environment. Its standard I/O devices can't be redirected, it can't be passed command line arguments, and it can't start other programs.

2.1 Creating Programs

The following paragraphs discuss the procedure for creating the different types of programs.

2.1.1 Creating ProDOS PRG programs

If you don't do anything special when linking a program, the program will be a ProDOS program of type PRG. For example, linking the "hello, world" program using the following command will create a PRG program:

```
ln hello.o -lc
```

2.1.2 Creating ProDOS BIN programs

To create a ProDOS BIN program, you must specify the +B option and include the special startup routine *samain.o* when you link the program. For example, the following command creates a BIN "hello, world" program:

```
ln +b hello.o samain.o -lc
```

2.1.3 Creating ProDOS SYS programs

To create a ProDOS SYS program, you must specify the +S option and include the *samain* startup routine when you link the program. The +S option causes the base address for the program to default to 0x2000 and the extension of the file containing the program to default to ".system".

For example, the following command creates a SYS "hello, world" program that can be loaded by ProDOS when it starts:

```
ln +s hello.o samain.o -lc
```

2.1.4 Creating DOS 3.3 programs

To create a program that runs under DOS 3.3, link the program using the +B option and use one of the *d.lib* libraries instead of *c.lib*. Move the program onto an Apple DOS disk using the standard ProDOS conversion utility.

2.2 Creating Special Startup Routines

In this section we want to describe how you can substitute your own startup routines for ours in your command programs. We first describe the way that the linker decides what the startup routines are,

and describe the startup routines that are provided with Aztec C65. Next, we describe what the Aztec startup routines do. Finally, we discuss different ways that you can modify the standard startup routines.

2.2.1 How the linker finds the startup routines

If, among the modules that the linker includes in a command program, a module is found whose assembly language source contains a statement of the form

entry name

where *name* is a label within the module, then the linker makes *name* the entry point of the program. If no such module is found, the entry point is set to the first statement of the program's code segment.

Execution of a command program normally begins at the label *.begin*, which is in the module *crt0*. This module is in all the versions of *c.lib* (*c.lib*, *ci.lib*, *d.lib*, and *di.lib*). The following facts account for execution beginning at *.begin*:

* *crt0* contains the statement "entry .begin". No other Manx-supplied module contains an entry statement, and compiler-generated code doesn't contain an entry statement.
* When compiling a C source program, the compiler normally writes the statement "public .begin" to the assembly language source file.
* When the linker includes the object version of a C program containing "public .begin" in the program it's building, the statement causes the linker to look for a module containing the label *.begin*, and, when found, to include the module in the program that it's creating.

crt0 performs activities that are described below, and then calls the function *__main*.

Two modules are supplied that contain a *__main* function: *shmain* and *samain*. *__main* performs additional initialization activities as described below and then calls the program's function *main*.

shmain is in *c.lib* and *ci.lib*. *samain* is its own file (*samain.o*), and is also in the libraries *d.lib* and *di.lib*. When creating a ProDOS program, you will be linking with *c.lib* or *ci.lib* and will have their *shmain* module included in your program unless you explicitly tell the linker to include *samain.o* or your own *__main* module in the program.

When creating a DOS 3.3 program, you will be linking with *d.lib* or *di.lib*, and thus will have its *samain* included in your program unless you explicitly tell the linker to include your own *__main* module in the program.

2.2.2 What *crt0* Does

The *.begin* code in *crt0* performs the following actions:

* It sets up the field that points to the top of the pseudo stack. For a PRG or BIN program and for a DOS 3.3 program, this field is set to the contents of the Basic HIMEM field. For a SYS program, this field is set to the base of the ProDOS global page; that is, to 0xbf00.
* It clears the program's uninitialized data area.
* It initializes the field *__Top*, which points to the top of allocated heap space, to the end of the Uninitialized data area.
* It initializes pointers to the floating point accumulators.

The same *crt0* module is used for all types of programs, unless you replace it with your own.

When a PRG program is started from within the SHELL environment, the loader program sets up information for it just below the pages that define the SHELL's global variables, and sets the HIMEM field to the top of this stack. This information consists of the arguments that are being passing to the called program and the initial stack for the called program. This stack contains the *argc* and *argv* arguments for the called program's *main* function.

When a BIN program is started from within the SHELL environment, the loader program sets the HIMEM field to the base of the pages that contain the SHELL's global information.

2.2.3 The *shmain* Module

The version of *__main* that is in *shmain* simply sets up pointers to the versions of the device and unbuffered i/o tables that are in the SHELL's global information pages, and then calls the program's *main* function. When *main* returns, *__main* calls the *exit* function, passing to it the return code that was returned by *main*.

The device and unbuffered i/o tables are discussed in detail below.

2.2.4 The *samain* Module

The version of *__main* that is in *samain* first sets up pointers to the device and unbuffered i/o tables that are contained in the program itself. It then calls the program's *main* function, passing it 0 and a null pointer for its *argc* and *argv* parameters. When *main* returns, *__main* calls *exit*, passing the value 0 to it as the program's return code.

2.2.5 Customizing

Given this information, it should be clear how you can modify the startup procedure for a program. For example, a program could use our *crt0* and your own *__main* function.

Another possibility is to modify or replace our *crt0.* the new version could call *_main*, call *main* directly, and so on.

Another possibility is to remove *crt0* from the version of *c.lib* with which a program is linked, causing program execution to begin with the first statement of the program.

2.3 Passing Open Files and Devices to PRG Programs

A PRG program "inherits" the description of an Apple configuration as it has been defined to the SHELL, thus allowing it to take advantage of the special features of an Apple without having to be explicitly told about them. A BIN or SYS program, on the other hand, has to be explicitly told about the configuration in which it runs.

A PRG program also inherits the files and devices that were left open for unbuffered i/o by the calling program (SHELL or other program); that is, these files and devices are open for the called program, and it can access them using the same file descriptors as did the caller.

In this section we're going to describe how this is done.

2.3.1 The Device Table

The device table contains information about the devices that are connected to an Apple. A PRG program uses the same device table as the SHELL; thus, the configuration of an Apple need be defined just once, to the SHELL. PRG programs will then automatically know, and make use of, the configuration of the system on which they run.

The device table of BIN and SYS programs are contained in the programs themselves; even when started by the SHELL, they don't use the information that's in the SHELL's device table. Thus, before a BIN or SYS program can make full use of the devices that are on an Apple, the configuration must be explicitly defined to the program, using the *config* program.

2.3.2 The Unbuffered I/O Table

Associated with a program is an unbuffered i/o table, which contains an entry for each device or file opened for unbuffered i/o by the program. PRG programs use the same unbuffered i/o table as does the SHELL, which is located in the SHELL's environment pages. For a program of type BIN or SYS, the unbuffered i/o table is in the program's own space.

The unbuffered i/o table that is located in the SHELL's environment pages changes only when a program that uses this table opens or closes a file. Thus, if a program which uses this table leaves some open entries in it and calls another program which uses it, the same files and devices will be pre-opened for the called program, and can be accessed by it using the same file descriptors that the calling

program used.

The SHELL opens the standard i/o devices for a program before activating it: it simply closes its own standard i/o devices, opens them to the desired files or devices, and starts the program. Since the SHELL uses the unbuffered i/o table that is in the system area, as does the called program, the program's standard i/o devices will be open when it starts.

Since a BIN or SYS program uses its own unbuffered i/o table, it won't see any redirection of standard i/o that the SHELL performs on its own unbuffered i/o table.

2.3.3 The Standard I/O table

Contained within the program space of each program is a standard i/o table. This table contains an entry for each file or device opened by the program for standard i/o.

When a file or device is open for standard i/o, it's also open for unbuffered i/o, since the standard i/o functions use the unbuffered i/o functions to access a file or device.

Only a program's stdin, stdout, and stderr i/o devices are preopened for standard i/o. This pre-opening isn't done by the SHELL or by a startup routine: the entries in the standard i/o table for these devices are preinitialized by source code to be associated with the first three entries in the unbuffered i/o table.

Thus, redirection of a PRG program's stdin, stdout, and stderr standard i/o devices is accomplished by simply redirecting the first three entries in the unbuffered i/o table that is in the SHELL's environment pages.

Since the standard i/o table is located in a program's own space, files opened for standard i/o in one program aren't open for standard i/o in the called program.

3. Overlay Support

In order to allow you to run programs which are larger than the limited memory size of a microcomputer, Manx provides overlay support. This feature allows you 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.

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

Overlays can also 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.

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 and 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.

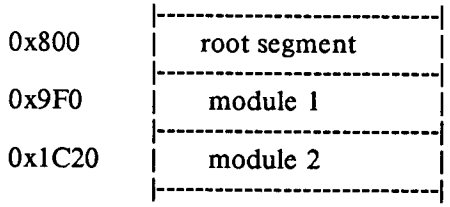


Figure 1

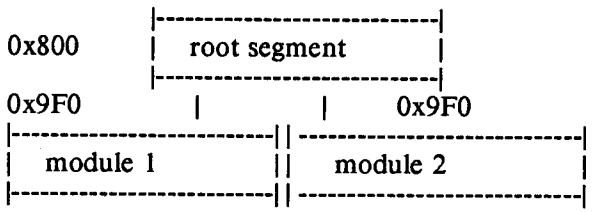


Figure 2

3.1 Calling an Overlay

A program segment (root or overlay) activates an overlay by calling the Manx-supplied function *ovloader*, which must reside in the root. The call has the form

ovloader(ovlyname, p1, p2, p3, ...)

where *ovlyname* is a pointer to a character string identifying the overlay name, and *p1*, *p2*, *p3*, ... are parameters that are to be passed to the overlay as its first, second, third, ... parameters.

ovloader derives the name of the file containing the overlay from the string pointed at by *ovlyname*, by appending the extension *.ovr* to it.

We provide you with the source to *ovloader*. When you compile it, you define the directories in which it will look for overlays: compiling it with the option *-DPATH* will cause it to search all directories specified in the *PATH* environment variable; compiling it without this option causes it to search just the current directory. If you create an overlaid program that will run under ProDOS outside of the SHELL environment or that will run under DOS 3.3, you must use a version of *ovloader* for it that looks for overlays in just the current directory, since environment variables are only available to programs running in the SHELL environment.

Each overlay must contain a function named *ovmain*, which you must write and which can be different for each overlay, and must also contain the Manx-supplied function named *ovbgn*. When an overlay is loaded, *ovloader* calls the overlay's *ovbgn* function, which in turn calls the overlay's *ovmain* function, passing to it the second, third, ... arguments that were passed to *ovloader*.

When *ovmain* completes its processing, it simply returns. *ovloader* then returns to the caller, returning as its value the value that was returned by *ovmain*.

An overlay can access any global functions and variables that are defined in the root segment and in the overlays that are currently active. For example, if the root calls overlay *ovly1*, which calls overlay *ovly11*, which calls overlay *ovly111*, then *ovly111* can access the global variables and functions that are defined in the root, in the overlays *ovly1* and *ovly11*, and in itself. But if the root also calls overlay *ovly2*, *ovly111* cannot access the global functions and variables that are in *ovly2*, since *ovly2* is not active when *ovly111* is.

3.2 Creating a root and and its overlays

To create a root and its overlays, the linker must be run several times, once to create the root, and once for each overlay. Each program segment (root or overlay) will be placed in a separate disk file.

The root must be created first. When overlays are nested, an overlay that itself calls overlays must be linked before the overlays that it calls.

When creating a program segment (root or overlay) which calls an overlay, the option *-R* must be specified; this causes the linker to generate a symbol table for use in linking the called overlay, placing it in a file whose filename is the same as that of the first file specified in the command line and whose extent is *.rsm*. When an overlay is linked, the symbol table file of the program segment that calls the overlay must be included in the linkage of the overlay.

When the root module is linked, the linker has to reserve some space into which the overlay can be loaded. This is done using the +C and +D linker options, which define the amount of space needed for the overlay code and data, respectively. If overlays are nested, a called overlay is located in memory immediately following the calling overlay. The amount of space reserved for the overlays must be enough to hold the longest 'thread' of overlays.

3.3 Example 1: Non-nested Overlays

This example demonstrates overlay usage when the overlays are not nested. The root segment, which consists of the function *main* and any neccesary run-time library routines, behaves as follows:

1. It calls the overlay *ovly1*, passing it as parameter a pointer to the string "first message".
2. It prints the integer value returned to it by ovly1;
3. It calls the overlay *ovly2*, passing it a pointer to the string "second message";
4. It prints the integer value returned to it by *ovly2*.

The overlay *ovly1* consists of the function *ovly1*, the Manx function *ovbgn*, and any neccesary run-time library routines. It prints the message "in ovly1" plus whatever character string was passed to it by *main*.

The overlay *ovly2* consists of the function *ovly2*, the function *ovbgn*, and any neccesary run-time library routines. It prints the message "in ovly2", plus whatever character string was passed to it by *main*.

Here then is the *main* function:

```
main() {
   int a;

   a = ovloader("ovly1","first message");
   printf("in main. ovly1 returned %d\n", a);
   a = ovloader("ovly2","second message");
   printf("in main. ovly2 returned %d\n",a);
}
```

Here is ovly1:

```
ovmain(a)
char *a;
{
   printf("in ovly1. %s\n",a);
   return 1;
}
```

Here is ovly2:

```
ovmain(a)
char *a;
{
   printf("in ovly2. %s\n",a);
   return 2;
}
```

The following commands link the root (which is in the file *root.c*) and the overlays:

```
ln -R +C 4000 +D 1000 root.o ovloader.o -lc
ln ovly1.o ovbgn.o root.rsm -lc
ln ovly2.o ovbgn.o root.rsm -lc
```

The command to link the root reserves 0x4000 bytes for the overlay's code and 0x1000 bytes for it's data. Techniques for determining this value are discussed below.

When the segments are generated and the root activated, the following messages appear on the console:

```
in ovly1. first message.
in main. ovly1 returned 1.
in ovly2. second message.
in main. ovly2 returned 2.
```

3.4 Example 2: Nested Overlays

In this example, there are three segments: a root segment, *root*, and two overlays segments, *ovly1* and *ovly2*. *root* calls *ovly1*, which calls *ovly2*. *ovly2* just returns.

Here is the root:

```
main()
{
   ovloader("ovly1","in ovly1");
}
```

Here is ovly1:

```
ovmain(a)
char * a;
{
   printf("%s\n",a);
   ovloader("ovly2", "in ovly2");
}
```

Here is ovly2:

```
ovmain(a)
char *a;
{
   printf("%s\n",a);
}
```

The following commands link the root and the two overlays:

```
ln -R root.o ovloader.o -lc
ln -R ovly1.o ovbgn.o root.rsm -lc
ln ovly2.o ovbgn.o ovly1.rsm -lc
```

When executed, the following messages appear on the console:

```
in ovly1
in ovly2
```

3.5 Determining the size of the overlay area

When you link the root module, you will have to know how much memory to reserve for the overlay, that is, you will have to know how large the overlay is. But since the overlays haven't been linked yet, how can you know how much space is needed for overlays?

The easiest way is to guess. That is, estimate the size and go ahead and link the root and the overlays, keeping track of the size of the code and data for the overlays as reported by the linker.

After all overlays have been linked, the size of the area needed for overlays is the size of the largest overlay (if overlays aren't nested) or the size of the longest 'thread' of overlays (if they are nested). You can then go back and relink the root, if necessary, with this value. You won't have to relink any overlays, since the +C and +D options don't affect the position of the overlays in memory.

3.6 Error messages from ovloader

If an error occurs while loading an overlay, *ovloader* will print a message of the form

Error %d loading overlay: %s

where %d is a number defining the error and %s is the name of the overlay. The error codes and their meanings are:

10	Can't open overlay file
20	Can't read overlay header record
30	Invalid header record
40	Overlay code & data overlaps with heap
50	Error reading overlay

3.7 Possible Problems

A possible source of difficulty in using overlays concerns initialized data. In the following program module, a global variable is initialized:

```
int i = 3;

function()
{
   return;
}
```

The initialization of "i" is performed by the linker, rather than at run time. In the same program, the following module is allowed:

```
int i;

main()
{
   function();
}
```

The global variables in each module refer to the same integer, "i". At link time, this variable is set to the value 3. Although this works when the two modules are linked together, a problem arises when the first module is linked as an overlay:

ln func.o ovbgn.o main.rsm -lc

From the *.rsm* file, the linker knows that "int i" has been declared in *main.o,* the root. But it tries to initialize "i" from the statement in the *func.o* module. This attempt fails because the variable "i" is part of *main.o,* a module which is not included in the linkage.

An attempt to initialize, in an overlay, a variable which has been declared in the root will produce an error:

attempt to initialize data in root

The simple solution is to change the statement, "int i = 3", to the following:

```
int i;
i = 3;
```

This assignment will be performed at run time, so that the linker does not try to perform an initialization.

3.8 Source

The source for the *ovloader* and *ovbgn* functions are in the files *ovloader.c* and *ovbgn.a65*. *ovloader* must be compiled by *cc*; as mentioned above, it can be compiled with or without the option -DPATH, as defined above. *ovbgn* must be assembled using *as*.

3.9 Cross-development and Overlays

3.9.1 Sending Overlaid programs to ProDOS

Overlaid programs for the Apple // can be created using the cross-development versions of Aztec C65. But there are some rules to follow when sending the root and overlay segments to ProDOS using the *xfer* program:

* When a root segment is sent to ProDOS, specify the *+P* option if the program is a PRG program that can only run under the SHELL, and specify the *+B* option when sending a BIN or SYS program to ProDOS that can run outside the SHELL environment.
* When an overlay segment is sent to ProDOS, don't specify the *+P* or *+B* options. That is, send an overlay segment to ProDOS as a pure binary file.

The reason for these rules is that the linker that's supplied with cross-development versions of Aztec C65 appends a four-byte header to an executable program, a header that is required by DOS 3.3. Since this header isn't used by ProDOS, *xfer* strips it off if you tell it, using the *+B* or *+P* options, that an executable program is being transferred.

When the cross-development linker creates an overlay, however, it doesn't append this header, since the overlay is loaded by the Aztec *ovloader* function and not by DOS 3.3 or ProDOS. So when an overlay segment is sent to ProDOS, it must be sent without specifying the *+P* or *+B* options, to prevent *xfer* from stripping off the first four bytes of the overlay.

3.9.2 Sending overlaid programs to DOS 3.3

The only rule when sending overlaid programs to DOS 3.3 using *xfer* is to send the root and overlay segments as binary files, that is, without specifying the *+A* option.

4. Libraries

Several libraries of object modules are provided with Aztec C65. There are two versions of each library: the native code compiler and assembler generated one version's modules; the pseudo code compiler and assembler generated the other's modules.

The native code libraries are:

c.lib	Most non-floating point functions: for ProDOS programs.
m.lib	Floating point functions.
g.lib	Graphics functions.
s.lib	Screen functions.
d.lib	DOS 3.3 version of *c.lib*.

The name of a pseudo code library is derived from its native code counterpart by adding the letter 'i' to the name:

ci.lib
mi.lib
gi.lib
si.lib
di.lib

5. Interfacing to Assembly Language

This section discusses assembly-language functions that can be called by, and themselves call, C-language functions.

5.1 Naming Convention

The compilers translate a global function or variable name into assembler by truncating it to contain no more than 31 characters, appending an underscore character '_' to the truncated name, and then generating a *public* directive for the resultant name.

For example, the following assembly language statements define the entry point to an assembly language function that would be referred to in a C language program using the name *sum*:

```
public  sum_
sum_            ;entry point to sum
```

5.2 Calling and Returning

On entry to a function, information about the call are on the tops of both the 6502 hardware stack and the pseudo stack.

At the top of the 6502 stack is the function's primary return address; this is the address to which the function should return by issuing an *rts* instruction. A non-reentrant function (ie, a function that doesn't call itself) can leave its return address on the 6502 stack and then return by issuing the 6502 *rts* instruction. For example, the very simplest assembly language function, which does nothing but return to the caller, would consist of just an *rts* instruction:

```
        public  nop_
nop_    rts
```

Because of limitations of the 6502 stack, a reentrant function should save its return address on the pseudo stack. When done, it should return by doing an indirect *jmp* to the location whose address is one greater than the saved address.

5.3 Returning a value

A function can return a *int* or *long* value by setting the value in pseudo register R0, which is located in memory page 0. (The *equ* statements that defines R0 and all the other 0 page locations used by Aztec C-generated programs are in the file *zpage.h*). The bytes of the value are stored in order, with the least significant byte at address 8 and the most significant byte at the highest addressed location.

For example, here's a function that always returns the *int* value 1:

```
        instxt  "zpage.h"
        public  one__
one__   lda     #1
        sta     R0
        lda     #0
        sta     R0+1
        rts
```

5.4 Passing parameters

On entry to a function, the parameters that are being passed to the function and a secondary return address are on the pseudo stack, and are accessed using the field named SP that is located in memory page 0 and that points to the top of the pseudo stack. (As with R0, the *equ* statement that defines SP is in the file *zpage.h*).

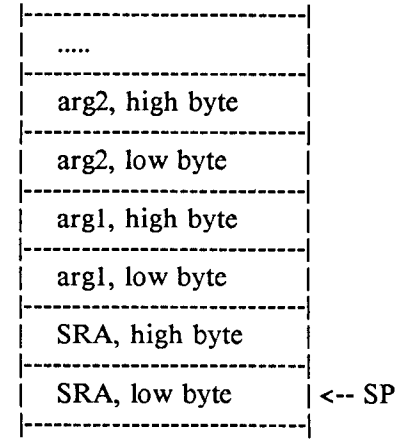
At the top of the pseudo stack is the two-byte secondary return address. This is a different address from the return address that is on the 6502 stack - a function should return using the address that's on the 6502 stack. The secondary return address is discussed in the section of the Tech Info chapter that discusses the pseudo stack.

Above the secondary return address on the pseudo stack are the parameters that are being passed to the function. The function parameters are in order on the pseudo stack, with the first parameter immediately following the secondary return address, the second parameter following the first, and so on. The bytes for a parameter are also on the pseudo stack in order, with a parameter's least significant byte at the lowest address and its most significant byte at the highest address.

For example, suppose the function *sum* is passed two parameters, as follows:

```
sum(arg1, arg2);
```

On entry to *sum__*, the pseudo stack will look like this (SRA means "secondary return address"):



5.5 An Example

The following assembly language function, named *sum*, is passed two *int*s as arguments. It returns their sum as its value.

```
        instxt  "zpage.h"
        public  sum__
sum__   clc
        ldy     #2
        lda     (SP),Y
        ldy     #4
        adc     (SP),Y
        sta     R0
        ldy     #3
        lda     (SP),Y
        ldy     #5
        adc     (SP),Y
        sta     R0+1
        rts
```

5.6 Page 0 Usage

A 6502 program makes extensive use of memory page 0. An assembly language 6502 function should obey the following restrictions on its usage of memory page 0 locations:

* It may use, without preserving, the two-byte-long VAL field and the following four-byte-long fields: VAL, R0, R1, R2, R3, R4, and TMP.
* It must preserve the contents of the SP, FRAME, and LFRAME (alias PC) fields and of the 16-byte REGS field.

These locations are defined in the file *zpage.h*.

5.7 Writing Programs that contain only Assembler

There are several topics concerning the linker which are important if the assembler and linker are to be used without any compiled code. The linker automatically creates several symbols that can be of use to an assembly language program, defining the beginning and end of the various program segments. These are described in the *Memory Organization* section of this chapter.

The entry point to a program is defined using the assembly language statement

```
entry loc
```

where *loc* is the name of the symbol where program execution is to begin. If a module containing an *entry* statement isn't encountered by the linker, it will set the program's entry point to the beginning of its code segment. For a discussion of the startup routines that are provided with Aztec C65, see the *Command Programs* section of this chapter.

5.8 Mixing C and Assembler in one Module

To include assembly language source in a C language module, surround the assembly language code with *#asm* and *#endasm* directives.

Finding a good example where this construct is necessary is very difficult, but here's a possible example:

```
rotate(arg)
{
   register int i;

   i = arg;
#asm
   lda  $81
   rol  A
   rol  $80
   rol  $$1
#endasm
   return(i);
}
```

This routine rotates a two byte quantity one bit to the left. This operation is messy in C and in a time critical application not feasible to make an assembly language subroutine. This routine is not a good example, since it would be better to write the entire thing in assembly. However, in the middle of a larger routine, it might conceivably be useful. This facility is provided as a last resort and is generally not recommended as it is completely non-portable.

6. Debugging Pseudo-code

In this section we want to provide information which will facilitate the debugging of *cci*-compiled modules using the monitor ROM.

You should debug programs using the monitor ROM only when all other techniques, such as those described in the *style* chapter, fail.

6.1 The Symbol Table

In order to debug a program using the monitor ROM, you will need the program's symbol table. To get it, specify the *-t* option when you link the program, and then either print it or copy the important addresses to paper.

6.2 Getting to the Monitor

When the SHELL is running, first tell the SHELL that you are going to debug a program by entering the command

debug

Then enter the command to start the program in the normal way (including arguments and i/o redirection, if desired). The SHELL will load the program, perform i/o redirection for it, and set up the stack as appropriate for the command line arguments. Normally, the SHELL will then transfer control of the processor to the program; however, when it sees the debug flag set, it will instead transfer control to the monitor.

When the Basic Interpreter is active, first load the program using the *bload* command. Then transfer control to the monitor by entering:

CALL -151

When you're done debugging the program from the monitor, you can return to the command processor program that was active (SHELL or Basic Interpreter) by entering

3D0G

Once in the monitor, you can set a breakpoint at the beginning of a function of interest and then issue the monitor's *G* command to transfer control to the program's starting address (which is usually 0x800).

6.3 Breakpointing the Entry to a *cci*-compiled Function

At the beginning of a *cci*-compiled function is a 6502 *jsr* instruction to the pseudo code interpreter. Following that is two bytes that specify the amount of stack space to reserve for the function's local variables. Following that is the function's pseudo code, which will be processed by the interpreter.

A function is called by issuing a 6502 *jsr* instruction to the function's entry point. This is true regardless of whether the calling

and called functions are compiled using *cc* or *cci*.

Thus, to set a breakpoint at the beginning of a *cci*-compiled function, replace the *jsr* opcode at its entry point (which is 0x20) with the opcode for the 6502 *brk* instruction (which is 0).

When the breakpoint is encountered, control will revert to the monitor. At any breakpoint, any global variables, whose addresses may be found in the program's symbol table, can be examined or modified using normal monitor commands.

When a breakpoint is taken at the entry point to a function, the following information exists:

* If the calling function was compiled with *cci*, locations 6 and 7 contain the called function's secondary return address; that is, the address of the byte in the calling function that follows the pseudo-code call instruction.
* A pointer to the top of the pseudo stack is in locations 2-3. At the top of the pseudo code stack is a two-byte field that also contains the function's secondary return address.
* Above the secondary return address on the pseudo stack are the parameters that are being passed to the function, with the first parameter immediately above the secondary return address, the second parameter above the second, and so on.

6.4 Determining a Function's Return Value

For the purposes of discussion, we shall assume that functions A and B were both compiled with *cci*, that function A calls function B, and that and we wish to determine the value returned by by B.

First, you must determine B's secondary return address. This is the address of the byte following the pseudo code call to B. This can be done in two ways:

* If B is called in just one place by A, you can set a breakpoint at the beginning of function B. When the breakpoint at the entry to B is hit, locations 6-7 will contain the secondary return address.
* If function B may be called from a number of places, you can't just set a breakpoint at the beginning of B, since the first call to B may not be the one from function A. In this case, you must look through the bytes in function A for a call to function B. This call, if direct, is three bytes long. It begins with the opcode for a direct function call (0xAC if no parameters are passed, 0xE9 if arguments are passed), and its last two bytes contain the address of function B. The byte following this instruction is B's secondary return address.

Once the secondary return address has been found, replace the byte of the pseudo code instruction that follows the pseudo code call to B

with FFH. This is the opcode for the pseudo code 'breakpoint' instruction. If no parameters were passed to B, this instruction begins at B's secondary return address. If parameters were passed, this instruction begins one byte after B's secondary return address.

Next, start the program.

When the interpreter encounters the pseudo breakpoint instruction, it will issue a 6502 *brk* instruction, which will display the address at which the *brk* was executed (ie, somewhere within the interpreter) and transfer control to the monitor. At this point, the value returned by function B will be in locations 8-9 (8-B if a long is returned).

Continuing from a pseudo-code breakpoint can't be easily done.

7. The Format of Aztec C65 Object Modules and Libraries

This section describes the format of object modules and libraries that have been generated by the Aztec C65 software, version 1.99. The symbols and structures referred to in this paper are defined in the header file *object.h*.

7.1 Object Module Format

An object module contains four sections: header, code, table of named symbols, and table of unnamed symbols. These sections are described in the following paragraphs.

7.1.1 The Header Section

The header section of an object module has the following structure:

```
struct module {
    int             m_magic;  /* type of object module */
    char            m_name[8];/* module name */
    unsigned short  m_code;   /* module's code size */
    unsigned short  m_data;   /* module's data size */
    unsigned short  m_static; /* module's bss data size */
    unsigned short  m_global; /*named sym tbl off.*/
    short           m_nglobal;/* # of named symbols */
    unsigned short  m_local;  /*unnamed sym tbl off.*/
    short           m_nlocal; /* # of unnamed symbols */
    unsigned short  m_end;    /* unnamed sym tbl end*/
    unsigned short  m_next;   /* offset to next module */
    unsigned short  m_nfix    /* # segment fixes required */
};
```

The following paragraphs discuss the fields within the *header* structure.

m_magic

Each of the different object module-related files created by the Aztec C software begins with the *m_magic* field, which contains a "signature" that identifies the file's contents. *m_magic* can have the following values:

M_MAGIC	Object module created by the assembler
M_OVROOT	Rsm file created by the linker
M_LIBRARY	Library of object modules

m_name

Contains the name of the object module. For object modules created by the assembler and for rsm files, this field normally contains null characters.

m_code, m_data, and m_static

Contain the size, in bytes, of an object module's code, data and uninitialized data segments, respectively.

m_global and m_nglobal

m_global contains the offset, in bytes, from the beginning of the module to the module's table of named symbols. *m_nglobal* contains the number of entries in this table.

m_local and m_nlocal

m_local contains the offset, in bytes, from the beginning of the module to the module's table of unnamed symbols. *m_nlocal* contains the number of entries in this table.

m_end

m_end contains the offset, in bytes, from the beginning of the module to the end of its table of unnamed symbols.

m_next

m_next contains the offset, in bytes, from the beginning of the module to the end of the module.

7.1.2 Symbols Tables

An object module contains two types of symbols: unnamed and named. An 'unnamed symbol' is a symbol whose name begins with a period followed by a digit. A 'named symbol' is any symbol that is not unnamed.

An object module has two symbol tables, one containing its named symbols, and the other its unnamed symbols. A symbol table contains entries, each of which describes one of the module's symbols. The entry for a symbol has the following structure:

```
struct symtab {
    char            s_type;   /* type of symbol */
    char            s_flags;  /* attributes of symbol */
    unsigned short  s_value;  /* another attr of symbol */
}
```

In addition, the entry for a named symbol is followed by a null-terminated string, which is the symbol's name.

The following paragraphs discuss the fields of the *symtab* structure.

s_type

The *s_type* field in a symbol's table entry defines the type of the symbol. Possible values:

S_ABS	Symbol was defined to be a constant value, using the assembler's *equ* directive.
S_CODE	Symbol was defined within the code segment.
S_DATA	Symbol was defined within the data

segment.

S_UND	Symbol was used but not defined within the program. Symbols that are defined using the assembler's *public* directive but that aren't defined in any statement's label field have this type, as do symbols defined using the assembler's *global* directive. The directive used to define a S_UND symbol can be determined from the symbol's *s_value* field, as defined below.
S_BSS	Symbol was defined using the assembler's *bss* directive.

s_flags

This field defines other attributes of a symbol. Possible values:

S_GLOBL	Set for symbols specified in *public* and *global* directives.
S_FIXED	Set for symbols defined in some statement's label field.

s_value

The meaning of this field depends on the type of the symbol. Symbol types and their associated values:

s_type	*Meaning of s_value*
S_ABS	Value specified for the symbol in the *equ* directive.
S_CODE	Offset of the symbol from the beginning of the module's code segment.
S_DATA	Offset of the symbol from the beginning of the module's data segment.
S_BSS	Size, in bytes, of the symbol as defined in the *bss* directive.
S_UND	For an S_UND symbol, *s_value* is zero if the symbol was defined in a *public* directive and non-zero if it was defined in a *global* directive. For a *global*-defined symbol, *s_value* contains the value specified in the directive's *size* operand.

7.1.3 The Code Section

The code section of an object module contains a translated version of the program. This format can be efficiently processed by the linker as it generates an executable version of the program. It contains a sequence of items, each of which directs the action of the linker. For

example, some items contain actual code and data, which the linker places in the output file, some cause the linker to reserve space in the output file, and some just pass information to the linker.

The linker builds several segments of a program simultaneously: a code segment, data segment, and an uninitialized data segment. Exactly one of these segments is said to be 'selected' at a time. There are loader items that select a segment.

The linker maintains a location counter for each of the segments that it is building. When a loader item requests that information be placed in the program or that space be reserved in it, the linker performs the requested operation in the current location of the currently-selected segment.

A loader item is a sequence of one or more bytes, with the first byte containing a code that identifies the item. Some codes are four bits long, and some are eight bits long; in the former case, the code occupies the most significant four bits of the byte.

Frequently, a loader item is two bytes long, with the item's code in the high order four bits of the item's first byte and a value in the other 12 bits. In this case, the value is stored with the least significant four bits in the first byte's least significant four bits, and the most significant eight bits in the second byte. We call this format "12-bit packed".

Descriptions of the loader items follow.

USECODE - Select code segment

The USECODE loader item selects the code segment. Data generated by following loader items will be placed in the code segment until another segment is selected.

The code for a USECODE loader item is 8 bits long: 0xf4.

USEDATA - Select initialed data segment

The USEDATA loader item selects the initialized data segment. Data generated by following loader items will be placed in the code segment until another segment is selected.

The code for the USEDATA loader item is 0xf5.

ABSDAT - Absolute data

The *ABSDAT* loader item defines a sequence of bytes that the linker is to output 'as is' to the current location in the currently-selected segment.

The loader item's first byte contains the code identifier, 1, in the most significant four bits, and the number of bytes to be output, less one, in the least significant four bits. Thus, this item can define one to sixteen bytes of absolute data.

The remaining bytes in the item are the absolute data.

For example, the following *ABSDATA* loader item defines the three bytes A1, B2, and C3:

12 A1 B2 C3

LCLSYM - local (ie, unnamed) symbol

The value of a LCLSYM loader item is the address at which an unnamed symbol is located in memory.

The item is two bytes long, with the item's code, 6, in the first byte's most significant four bits. The item's other twelve bits contain contain the number of the symbol's entry in the local symbol table, in 12-bit packed format.

For example, given the assembly language code

```
    dw      .98
.98 dw      12
```

with *.98* occupying the second entry in the table of unnamed symbols, the following code would be generated for the *dw .98*:

61 00

GBLSYM - Global Symbol

The GBLSYM loader item is just like LCLSYM except that it references an entry in the global symbol table rather than the local symbol table.

The code for GBLSYM is the four-bit value 7.

SPACE - Reserve space

The SPACE loader item reserves a specified amount of space at the current location in the currently-selected segment.

The item is two bytes long, with the item's code, 8, in the most significant four bits of the item's first byte. The other twelve bits contain the number of bytes to reserve, less one, in 12-bit packed format.

For example, the following loader item reserves 5 bytes:

84 00

CODEREF - Code segment offset

The CODEREF loader item defines an offset from the beginning of the module's code segment. The loader item has as its value the absolute address corresponding to that offset.

The CODEREF loader item is in two bytes, with the CODEREF code, 0xa, in the high-order four bits of the item's

first byte. The item's other 12 bits contain the offset, as a positive number, in 12-bit packed format.

DATAREF - Data segment offset

The DATAREF loader items is the same as the CODEREF loader item, except that the offset is relative to the beginning of the module's data segment.

The code for DATAREF is 0xb.

BSSREF - BSS segment offset

The BSSREF loader item is the same as the CODEREF loader item, except that the offset is relative to the beginning of the module's bss segment.

The code for BSSREF is 0xc.

LRGCODE - Code segment offset, large form

The LRGCODE loader item takes a 16-bit value that represents an offset from the beginning of its code segment, and generates as its value the absolute memory address of the location.

The loader item is in three bytes, with the first containing the item's 16-bit code, 0xf7. The other two bytes contain the positive offset, with the offset's most significant byte in the high-order byte.

LRGDATA - Data segment offset, large form

The LRGDATA loader item is the same as LRGCODE except that the offset is relative to the beginning of the module's data segment.

The code for the LRGDATA loader item is 0xf8.

LRGBSS - BSS segment offset, large form

The LRGBSS loader item is the same as LRGCODE except that the offset is relative to the beginning of the module's BSS segment.

The code for the LRGBSS loader item is 0xfb.

SMLINT - small integer

The SMLINT loader item defines an integer between 0 and 15, inclusive. This item can be used by itself or as an element of an EXPR loader item

The loader item consists of a single byte. Its most significant four bits are the item's code, 3; and the least significant four bits are the integer value.

corresponding values and operations are:

code	*value*	*operation*
ADD	1	Add the two loader items that follow
SUB	2	Subtract the following two loader items
MUL	3	Multiply the following two loader items
DIV	4	Divide the first item that follows by the second
MOD	5	Compute the modulus of the first item relative to the second.
AND	6	Logical AND of the following two items
OR	7	Logical OR of the following two items
XOR	8	Exclusive OR of the following two items
RSH	9	Right shift first item the number of bits defined by second item
LSH	10	Left shift first item the number of bits defined by the second
NOT	11	Logical NOT of item that follows
NEG	12	Compute two's complement of the item that follows

The items that can follow an EXPR item are SMLINT, MEDINT, LRGINT, LCLSYM, GBLSYM, CODEREF, DATAREF, BSSREF, LRDCODE, LRDDATA, LRDBSS, and another EXPR.

For example, given the assembly language code

```
dw a+4
```

with the entry for *a* being the fourth entry in the table of named symbols, the following loader items would be generated:

```
21 73 00 34
```

As mentioned above, an EXPR can have another EXPR as one of its loader items. In this case, the inner EXPR is evaluated, using the loader items that follow it, and then the outer EXPR is evaluated, using the resultant value of the inner EXPR as one value and whatever loader items are left for the other values. The loader items for the entire expression are thus in prefix-Polish notation. For example, the above expression, a+4, is represented by the loader items that correspond to

```
+a4
```

And the expression

```
(a+b)*c
```

would be represented by loader items that correspond to

*+abc

BEXPR - Evaluate byte expression

The BEXPR loader item has as its value the 8-bit value of the expression that follows it. BEXPR has an 8-bit code, 0xf1. BEXPR doesn't have an extra four bits in which an operation code can be placed; thus, to generate an 8-bit value from an expression, a BEXPR loader item will usually precede an EXPR loader item that is in turn followed by the loader items for the expression.

BREL - compute offset from location counter, byte form

The BREL loader item takes a relocatable value that represents a location in the module and generates the offset of the location from the current location counter.

The BREL loader item begins with a 8-bit code, 0xf2. It's followed by loader items representing the location.

For example, if the symbol *abc* is the fourth symbol in the global symbol table, then the loader items to generate the offset of the location that is four bytes beyond *abc* are

f2 21 73 00 34

WREL - compute offset from location counter, word form

The WREL loader item is the same as BREL except that it generates a 16-bit value instead of an 8-bit value.

STARTAD - Define program start address

The STARTAD loader item defines the address at which a program containing the module is to begin execution.

The item begins with the item's 8-bit code, 0xf6. It's followed by loader items identifying the starting address; these can be a GBLSYM, LCLSYM, EXPR, or any of the other "expression items" mentioned above.

INTJSR - Generate opcode for a subroutine call

The INTJSR loader item is translated by the linker into a machine-specific opcode that will cause a subroutine to be called. The loader item has the value 0xf9.

The instructions in a function that has been compiled with the interpretive compiler consist of a call to the Aztec interpreter routine followed by the function's other instructions. This first instruction is directly executed by the machine; the function's other instructions are in a pseudo code that is indirectly executed, by the Aztec interpreter.

It is desirable to allow the interpretive compiler to generate object modules that can be executed on different machines, and to allow a single object module generated using this compiler to be linked for execution on different processor chips. To support this, the interpretive compiler generates as a function's first instruction a special call instruction, in the pseudo code assembly language, to the interpreter. The pseudo code assembler translates this instruction into an INTJSR loader item followed by a GBLSYM loader item that references the interpreter routine. The machine-specific linker then translates this pair of loader items into a machine-specific call to the interpreter.

THEEND - End of code

The THEEND loader item identifies the end of the code section of the object file.

The code for the item is 00.

7.2 Object Library Format

A library of object modules consists of the object modules and a directory of symbols names.

7.2.1 Object Modules in a Library

When an object module is placed in a library its sections are reorganized but the contents of the module are left unchanged (with the exception of the module's header, whose fields are modified to reflect the reorganization). The module's header still is at the beginning of the module. This is followed by the table of named symbols, the table of unnamed symbols, and the code section.

The header is modified to define the positions of the tables in the reorganized module, and the module is given a name in its *m_name* field. The name is derived from the name of the file that contained the module by removing the file name's extension.

7.2.2 Library Dictionary

A library's dictionary consists of one or more blocks that are chained together. A block has the following structure:

```
struct newlib {
    short nl_magic;          /* magic number for libraries */
    unsigned short nl_next;  /* loc of next dir block */
    char nl_dict[LBSIZE];    /* dictionary for block */
}
```

nl_dict contains entries, each of which defines one symbol that is defined in a library module. The entry for a symbol consists of a short int that defines the position of the module that defines the symbol (the

absolute location at which the module begins, divided by 128), and a null-terminated string that is the symbol's name.

8. Making DOS 3.3 Programs

8.1 DOS 3.3 Files

The following files are provided for the creation of programs that can be run on DOS 3.3:

* *d.lib*, a DOS 3.3 version of *c.lib*. The modules in this library have been compiled with *cc*;
* *di.lib*, a DOS 3.3 version of *ci.lib*. The modules in this library have been compiled with *cci*.
* *d33roota.o*, a startup routine for DOS 3.3 programs that will prompt the operator for a command line, and call the program's *main* function with the function's *argc* and *argv* parameters set appropriately.
* *dos33.arc*, source archives for the DOS 3.3 functions.
* *convert*, an Apple utility for transferring programs from a ProDOS disk to a DOS 3.3 disk.

8.2 Creating DOS 3.3 Programs

To create a program that can be run on DOS 3.3, link the program using either *d.lib* or *di.lib* instead of *c.lib* or *ci.lib*; also, specify the *+b* option to the linker. If you want the program's startup code to read a line from the console and then pass the program's *main* function an *argc* and *argv* parameter list, include the *d33roota.o* in the command line to the linker. (if you don't specify this module, another startup routine, which is in the DOS 3.3 library, will be included in the program; this startup routine will pass 0 as *argc* and *argv* parameters to the *main* function.)

A DOS 3.3 program that you create can be transferred from a ProDOS disk to a DOS 3.3 disk using the standard ProDOS *convert* program. For a description of this program, see the ProDOS User's Manual.

8.3 Features of DOS 3.3 Programs

The following paragraphs discuss special features of the DOS 3.3 libraries.

8.3.1 DOS 3.3 Filenames

A DOS 3.3 filename has the following form:

name[,sxx][,dyy][,vzz]

where square brackets surround optional items. The items have the following meanings:

* *name* is the name of the file. It can contain up to 30 characters; all characters are valid, including spaces and

control characters.

* *sxx* specifies that *xx* is the number of the slot to which is connected the drive containing the disk that contains the file. If this item is not specified, the slot number is set to the value that's in the global *int* *__DSLOT*, which by default has the value 6. Thus, if you don't specify the slot on which a file is located, and if you don't change *__DSLOT*, the file is assumed to be in a drive that's connected to slot 6.

* *dyy* specifies that *yy* is the number of the drive containing the disk that contains the file. If this item is not specified, the drive number is set to the value that's in the global *int* *__DDRIVE*, which by default has the value 1.

* *vzz* specifies that *zz* is the number of the volume containing the file. If this item is not specified, the volume number is set to the value that's in the global *int* *__DVOLUME*, which by default has the value 0.

For example, the following filename defines a file named *hello.c* that is on slot 5, drive 2, volume 1:

```
hello.c,s5,d2,v1
```

The next filename defines a file named *hello.c* that is on the slot whose number is in *__DSLOT*, drive 2, and the volume whose number is in *__DVOLUME*:

```
hello,d2
```

8.3.2 File Type

There is a third parameter to the *open* and *creat* functions, which defines the type of the file. For a TEXT file, this parameter is 0; for a BINARY file, it's 4.

The *fopen* and *freopen* functions create a TEXT file. To create and open a file of another type for standard i/o, first create and open the file for unbuffered I/O, by calling *open* or *creat*; then make the file accessable by the standard i/o functions by calling *fdopen*.

8.3.3 Positioning a file

The *lseek* and *fseek* functions allow a file to be positioned relative to its beginning and to its current position, but not relative to its end. That is, the third parameter to these functions can be 0 or 1, but not 2.

Similarly, the unbuffered and standard i/o open functions do not support the "append" options. That is, *fopen*, *freopen*, and *fdopen* don't support the *a* and *a+* modes; and *open* doesn't support the *O__APPEND* mode.

8.3.4 Error codes

When the DOS file manager returns an error code, its ProDDOS equivalent is stored in the global integer *errno*. The original DOS error code is stored in the global integer *__d33errno*.

The *perror* function is supported, but not the global array of error messages *sys_errlist* or the global integer *sys_nerr*.

8.3.5 Temporary files

The function *tmpnam* creates a name that is unique on the default drive (that is, on the drive defined by *__DSLOT*, *__DDRIVE*, and *__DVOLUME*).

The *tmpfile* function always creates a temporary file on the default drive.

9. The TMPDEV Console Driver

tmpdev is a stripped-down console driver that supports line-oriented console I/O. It is designed to be used by programs that perform line-oriented console i/o and that also need to be as small as possible.

To include the *tmpdev* console driver in a program in place of the normal console driver, specify the *tmpdev* object module to the linker. For example:

```
ln prog.o tmpdev.o -lc
```

INDEX

Order of chapters in manual

System Dependent Chapters

System Independent Chapters

OVERVIEW OF LIBRARY FUNCTIONS

Chapter Contents

Overview of Library Functions

This chapter presents an overview of the functions that are provided with Aztec C. It's divided into the following sections:

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

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

3. *Unbuffered I/O:* Describes the other set of i/o functions, the unbuffered.

4. *Console I/O:* Describes special topics relating to console i/o.

5. *Dynamic Buffer Allocation:* Discusses topics related to dynamic memory allocation.

6. *Errors:* Presents an overview of error processing.

The overviews present information that is system independent. Overview information that is specific to your system is in the form of an appendix to this chapter; it accompanies the system dependent section of your manual.

1. Overview of I/O

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

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

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

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

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

There is a limit on the number of files and devices that can simultaneously be open; the limit on your system is defined in this chapter's system dependent appendix.

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

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

1.1 Pre-opened devices and command line arguments

There are three logical devices which are automatically opened when a program is started: standard input, standard output, and standard error. By default, these are associated with the console. The operator, as part of the command line which starts the program, can specify that these logical devices are to be "redirected" to another

device or file. Standard input is redirected by entering on the command line, after the program name, the name of the file or device, preceded by the character '<'. Standard output is redirected by entering the name of the file or device, preceded by '>'.

For example, suppose the executable program *cpy* reads standard input and writes it to standard output. Then the following command will read lines from the keyboard and write them to the display:

```
cpy
```

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

```
cpy >testfile
```

This will copy the file *exmplfil* to the console:

```
cpy <exmplfil
```

And this will copy *exmplfil* to *testfile*:

```
cpy <exmplfil >testfile
```

Aztec C will pass command line arguments to the user's program via the user's function *main(argc, argv)*. *argc* is an integer containing the number of arguments plus one; *argv* is a pointer to a an array of character pointers, each of which, except the first, points to a command line argument. On some systems, the first array element points to the command name; on others, it is a null pointer. Information on your system's treatment of this pointer is presented in this chapter's system dependent appendix.

For example, if the following command is entered:

```
prog arg1 arg2 arg3
```

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

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

```
prog arg1 arg2 arg3 >outfile
prog >outfile arg1 arg2 arg3
prog arg1 >outfile arg2 arg3
```

1.2 File I/O

A program can access files both sequentially and randomly, as discussed in the following paragraphs.

1.2.1 Sequential I/O

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

On systems which don't keep track of the last character written to a file, it isn't always possible to correctly position a file to which data is to be appended. If this is a problem on your system, it's discussed in the system dependent appendix to this chapter, which accompanies the system dependent section of your manual.

1.2.2 Random I/O

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

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

A file can be positioned relative to its beginning, current position, or end. Positioning relative to the beginning and current position is always correctly done. For systems which don't keep track of the last character written to a file, positioning relative to the end of a file can't always be correctly done. For information on this, see this chapter's system dependent appendix.

1.2.3 Opening files

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

For information on the opening of files on your system, see this chapter's system dependent appendix.

1.3 Device I/O

Aztec C allows programs to access devices as well as files. Each system has its own names for devices: for the names of devices on your system, see this chapter's system dependent appendix.

1.3.1 Console I/O

Console I/O can be performed in a variety of ways. There's a default mode, and other modes can be selected by calling the function *ioctl*. We'll briefly describe console I/O in this section; for more details, see the *Console I/O* section of this chapter and the system dependent appendix to this chapter.

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

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

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

Optionally, this translation can be disabled.

1.3.2 I/O to Other Devices

On most systems, few options are available when writing to devices other than the console. For a discussion of such options, if any, that are available on your system, see this chapter's system dependent appendix.

1.4 Mixing unbuffered and standard i/o calls

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

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

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

2. Overview of Standard I/O

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

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

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

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

2.1 Opening files and devices

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

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

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

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

2.2 Closing streams

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

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

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

2.3 Sequential I/O

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

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

2.4 Random I/O

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

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

2.5 Buffering

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

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

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

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

section *Dynamic Buffer Allocation.*

The size of an i/o buffer differs from system to system. See this chapter's system-dependent appendix for the size of this buffer on your system.

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

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

2.6 Errors

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

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

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

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

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

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

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

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

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

2.7 The standard i/o functions

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

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

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

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

agetc	Get an ASCII character
aputc	Send an ASCII character
fopen	Open a file or device
fdopen	Open as a stream a file or device already open for unbuffered i/o
freopen	Open an open stream to another file or device
fclose	Close an open stream
feof	Check for end of file on a stream
ferror	Check for error on a stream
fileno	Get file descriptor associated with stream
fflush	Write stream's buffer
fgets	Get a line of ASCII characters
fprintf	Format data and write it to a stream
fputs	Send a string of ASCII characters to a stream
fread	Read binary data
fscanf	Get data and convert it
fseek	Set current position within a file
ftell	Get current position
fwrite	Write binary data
getc	Get a binary character
getw	Get two binary characters
putc	Send a binary character
putw	Send two binary characters
setbuf	Specify buffer for stream
ungetc	Push character back into stream

3. Overview of Unbuffered I/O

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

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

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

Here are the unbuffered i/o functions:

open	Prepares a file or device for unbuffered i/o
creat	Creates a file and opens it
close	Concludes the i/o on an open file or device
read	Read data from an open file or device
write	Write data to an open file or device
lseek	Change the current position of an open file
rename	Renames a file
unlink	Deletes a file
ioctl	Change console i/o mode
isatty	Is an open file or device the console?

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

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

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

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

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

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

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

3.1 File I/O

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

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

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

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

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

3.2 Device I/O

3.2.1 Unbuffered I/O to the Console

There are several options available when accessing the console, which are discussed in detail in the Console I/O sections of this chapter and of the system-dependent appendix to this chapter. Here we just want to briefly discuss the line- or character-modes of console I/O as they relate to the unbuffered i/o functions.

Console input can be either line- or character-oriented. With line-oriented input, characters are read from the console into an internal buffer a line at a time, and returned to the program from this buffer. Line buffering of console input is available even when using the so-called "unbuffered" i/o functions.

With character-oriented input, characters are read and returned to the program when they are typed: no buffering of console input occurs.

3.2.2 Unbuffered I/O to Non-Console Devices

Unbuffered I/O to devices other than the console is truly unbuffered.

4. Overview of Console I/O

A program has control over several options relating to console i/o. The primary option allows console input to be either line- or character-oriented, as described below.

On most systems, a program can selectively enable and disable the echoing of typed characters to the screen; this is called the ECHO option. A program can also enable and disable the conversion of carriage return to newline on input and of newline to carriage return-linefeed on output; this is called the CRMOD option.

On some systems, additional options are available. If your system supports additional options, they are discussed in the system dependent appendix to this chapter.

All the console i/o options have default settings, which allow a program to easily access the console without having to set the options itself. In the default mode, console i/o is line-oriented, with ECHO and CRMOD enabled.

A program can easily change the console i/o options, by calling the function *ioctl*.

Console i/o behaves the same on all systems when the console options have their default settings. However, the behavior of console i/o differs from system to system when the options are changed from their default values. Thus, a program requiring machine independence should either use the console in its default mode or be careful how it sets the console options. In the paragraphs below, we will try to point out system dependencies.

4.1 Line-oriented input

With line-oriented input, a program issuing a read request to the console will wait until an entire line has been typed. On some systems a non-UNIX option (NODELAY) is available that will prevent this waiting. If this option is available on your system, it's discussed in the system-dependent appendix to this chapter.

The program need not read an entire line at once; the line will be internally buffered, and characters returned to the program from the buffer, as requested. When the program issues a read request to the console and the buffer is empty, the program will wait until an entire new line has been typed and stored in the internal buffer (again, on some systems programs can disable this wait by setting the non-UNIX NODELAY option).

A single unbuffered read operation can return at most one line.

On most systems , selecting line-oriented console input forces the ECHO option to be enabled. On such systems the program still has control over the CRMOD option. To find out if, on your system,

line-oriented mode always has ECHO enabled, see the system-dependent appendix to this chapter.

4.2 Character-oriented input

The basic idea of character-oriented console input is that a program can read characters from the console without having to wait for an entire line to be entered.

The behavior of character-oriented console input differs from system to system, so programs requiring both machine independence and character-oriented console input have to be careful in their use of the console. However, it is possible to write such programs, although they may not be able to take full advantage of the console i/o features available for a particular system.

There are two varieties of character-oriented console input, named CBREAK and RAW. Their primary difference is that with the console in CBREAK mode, a program still has control over the other console options, whereas with the console in RAW mode it doesn't. In RAW mode, all other console options are reset: ECHO and CRMOD are disabled.

Thus, to some extent RAW mode is simply an abbreviation for 'CBREAK on, all other options off'. However, there are some differences on some systems, as noted below and in this chapter's system-dependent appendix.

The system-dependent appendix to this chapter, which accompanies your manual, presents information about character-oriented console that is specific to your system.

4.2.1 Writing system-independent programs

To write system-independent programs that access the console in character-oriented input mode, the console should be set in RAW mode, and the program should read only a single character at a time from the console. All the non-UNIX options that are supported by some systems should be reset.

The standard i/o functions all read just one character at a time from the console, even when the calling program requests several characters. Thus, programs requiring system independence and character-oriented input can read the console using the standard i/o functions.

Some systems require a program that wants to set console option to first call *ioctl* to fetch the current console options, then modify them as desired, and finally call *ioctl* to reset the new console options. The systems that don't require this don't care if a program first fetches the console options and then modifies them. Thus, a program requiring system-independence and console i/o options other than the default should fetch the current console options before modifying them.

4.3 Using ioctl

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

```
#include <sgtty.h>

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

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

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

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

TIOCGETP	Fetch the console parameters and store them in the structure pointed at by *arg.*
TIOCSETP	Set the console parameters according to the structure pointed at by *arg.*
TIOCSETN	Equivalent to *TIOCSETP.*

The argument *arg* points to a structure named *sgttyb* that contains the following fields:

```
int sg_flags;
char sg_erase;
char sg_kill;
```

The order of these fields is system-dependent.

The *sg_flags* field is supported by all systems, while the other fields are not supported by some systems. If these fields are supported on your system, the system-dependent appendix to this chapter that accompanies your manual says so, and describes them.

To set console options, a program should fetch the current state of the sgtty fields, using ioctl's TIOCGETP option. Then it should modify the fields to the appropriate values and call ioctl again, using ioctl's TIOCSETP option.

4.4 The *sgtty* fields

4.4.1 The sg_flags field

sg_flags contains the following UNIX-compatible flags:

RAW	Set RAW mode (turns off other options). By default, RAW is disabled.
CBREAK	Return each character as soon as typed. By default, CBREAK is disabled.

ECHO	Echo input characters to the display. By default, ECHO is enabled.
CRMOD	Map CR to LF on input; convert LF to CR-LF on output. By default, CRMOD is enabled.

On some systems, other flags are contained in *sg_flags*. If your system supports other flags, they're described in the system-dependent appendix to this chapter that accompanies your manual.

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

When the console i/o options are set and RAW and CBREAK are reset, the console is set in line-oriented input mode.

4.5 Examples

4.5.1 Console input using default mode

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

```
#include <stdio.h>

main()
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
}
```

4.5.2 Console input - RAW mode

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

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

        if ((fp = fopen("con:", "r") == NULL){
                printf("can't open the console\n");
                exit();
        }

        ioctl(fileno(fp),TIOCGETP, &stty);

        stty.sg_flags |= RAW;
        ioctl(fileno(fp), TIOCSETP, &stty);
        for (;;){
                c = getc(fp);
                . . .
        }
}
```

4.5.3 Console input - console in CBREAK + ECHO mode

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

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

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

5. Overview of Dynamic Buffer Allocation

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

malloc	Allocates a buffer
calloc	Allocates a buffer and initializes it to zeroes
realloc	Allocates more space to a previously allocated buffer
free	Releases an allocated buffer for reuse

These standard UNIX functions are described in the System Independent Functions section of this chapter.

In addition, on some systems the UNIX-compatible functions *sbrk* and *brk* are provided that provide a more elementary means to allocate heap space. The *malloc*-type functions call *sbrk* to get heap space, which they then manage.

On some systems, non-UNIX memory allocation functions are also supported. If such functions are supported on your system, they are described in the system-dependent appendix to this chapter that accompanies your manual.

Dynamic allocation of standard i/o buffers

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

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

Where to go from here

For descriptions of the *sbrk* and *brk* functions and, when applicable, non-UNIX memory allocation functions see the System Dependent Functions chapter.

For a discussion of i/o buffer allocation, see the Standard I/O section of the Library Functions Overviews chapter.

For more information on the heap, see the Program Organization section of the Technical Information chapter.

6. Overview of Error Processing

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

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

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

Also, when an error occurs,

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

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

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

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

The following table lists the system-independent values which may be placed in *errno*. These symbolic values are defined in the file *errno.h*. Other, system-dependent, values may also be set in *errno* following an i/o operation; these are error codes returned by the operating system. System dependent error codes are described in the operating system manual for a particular system.

The system-independent error codes and their meanings are:

error code	*meaning*
ENOENT	File does not exist
E2BIG	Not used
EBADF	Bad file descriptor - file is not open or improper operation requested
ENOMEM	Insufficient memory for requested operation
EEXIST	File already exists on creat request
EINVAL	Invalid argument
ENFILE	Exceeded maximum number of open files
EMFILE	Exceeded maximum number of file descriptors
ENOTTY	Ioctl attempted on non-console
EACCES	Invalid access request
ERANGE	Math function value can't be computed
EDOM	Invalid argument to math function

SYSTEM-INDEPENDENT FUNCTIONS

Chapter Contents

System Independent Functions

This chapter describes in detail the functions which are UNIX-compatible and which are common to all Aztec C packages.

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

A section is organized into the following subsections:

TITLE

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

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

C	c.lib
M	m.lib

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

With *Apprentice C*, the functions are all in the run-time system, and not libraries.

SYNOPSIS

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

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

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

```
double atof();
```

The notation

```
#include "header.h"
```

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

On Radio Shack systems, a header file can use either a period or a slash to separate the filename from the extent. That is, the include statement can be as listed above, or

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

DESCRIPTION

Describes the section's functions.

SEE ALSO

Lists relevant sections. A letter in parentheses may follow a section name. This specifies where the section is located: no letter means that the section is in the current chapter; 'O' means that it's in the Functions Overview chapter; 'S' means that it's in the System Dependent Functions chapter.

DIAGNOSTICS

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

EXAMPLES

Gives examples on use of the section's functions.

Index to System Independent Functions

fwrite	FREAD	write binary data to i/o stream
getc	GETC	get binary char from i/o stream
getchar	GETC	get ASCII char from stdin
gets	GETS	get ASCII line from stdin
getw	GETW	get ASCII word from stdin
index	STRING	find char in string
ioctl	IOCTL	set mode of device
isalpha, etc.	CTYPE	char classification functions
isatty	IOCTL	is this a console?
ldexp	FREXP	build *double*
log	EXP	compute natural logarithm
log10	EXP	compute base-10 log
longjmp	SETJMP	non-local goto
lseek	LSEEK	position unbuffered i/o file
malloc	MALLOC	allocate buffer
movmem	MOVMEM	copy a block of memory
modf	FREXP	get components of *double*
open	OPEN	open file/device for unbuffered i/o
pow	EXP	compute x**y
printf	PRINTF	format data and print on stdout
putc	PUTC	put binary char to i/o stream
putchar	PUTC	put ASCII char to stdout
puterr	PUTC	put ASCII char to stderr
puts	PUTS	put ASCII string to stdout
putw	PUTC	put ASCII word to stdout
qsort	QSORT	Quick sort
ran	RAN	compute random number
read	READ	read unbuffered file/device
realloc	MALLOC	reallocate buffer
rename	RENAME	rename file
rindex	STRING	find char in string
scanf	SCANF	input string from stdin & convert
setbuf	SETBUF	set buffer for i/o stream
setjmp	SETJMP	*longjmp* partner
setmem	MOVMEM	set memory to specified byte
sin	SIN	compute sine
sinh	SINH	compute hyperbolic sine
sprintf	PRINTF	format string into buffer
sqrt	EXP	compute square root
sscanf	SCANF	convert string from buffer
strcat	STRING	concatenate two strings
strcmp	STRING	compare two strings
strcpy	STRING	copy char string
strlen	STRING	get length of char string
strncat	STRING	concatenate strings
strncmp	STRING	compare strings
strncpy	STRING	copy string
swapmem	MOVMEM	swap two blocks of memory

tan ..... SIN ..... compute tangent
tanh ..... SINH ..... compute hyperbolic tangent
tolower ..... TOUPPER ..... convert upper case char to lower
toupper ..... TOUPPER ..... convert lower case char to upper
ungetc ..... UNGETC ..... return char to i/o stream
unlink ..... UNLINK ..... delete file
write ..... WRITE ..... unbuffered write of binary data

NAME

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

SYNOPSIS

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

atoi(cp)
char *cp;

long atol(cp)
char *cp;

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

DESCRIPTION

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

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

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

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

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

NAME

close - close a device or file

SYNOPSIS

close(fd)
int fd;

DESCRIPTION

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

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

close returns 0 as its value if successful.

SEE ALSO

Unbuffered I/O (O), Errors (O)

DIAGNOSTICS

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

NAME

creat - create a new file

SYNOPSIS

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

DESCRIPTION

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

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

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

For most systems, *pmode* is optional: if specified, it's ignored. It should be included, however, for programs for which UNIX-compatibility is required, since the UNIX creat function requires it. In this case, *pmode* should have the octal value 0666.

For some systems, *pmode* is required and has a special meaning. If it is required for your system, the System Dependent Functions chapter will contain a description of the *creat* function, which will define the meaning.

SEE ALSO

Unbuffered I/O (O), Errors (O)

DIAGNOSTICS

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

NAME

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

SYNOPSIS

#include "ctype.h"

isalpha(c)

. . .

DESCRIPTION

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

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

NAME

exponential, logarithm, power, square root functions:
exp, log, log10, pow, sqrt

SYNOPSIS

```
#include <math.h>

double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

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

double sqrt(x)
double x;
```

DESCRIPTION

exp returns the exponential function of x.

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

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

sqrt returns the square root of x.

SEE ALSO

Errors (O)

DIAGNOSTICS

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

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

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

NAME

fclose, fflush - close or flush a stream

SYNOPSIS

```
#include "stdio.h"

fclose(stream)
FILE *stream;

fflush(stream)
FILE *stream;
```

DESCRIPTION

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

fclose is called automatically by *exit.*

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

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

SEE ALSO

Standard I/O (O)

DIAGNOSTICS

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

NAME

feof, ferror, clearerr, fileno - stream status inquiries

SYNOPSIS

```
#include "stdio.h"

feof(stream)
FILE *stream;

ferror(stream)
FILE *stream;

clearerr(stream)
FILE *stream;

fileno(stream)
FILE *stream;
```

DESCRIPTION

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

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

clearerr resets an error indication on the specified stream.

fileno returns the integer file descriptor associated with the stream.

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

SEE ALSO

Standard I/O (O)

NAME

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

SYNOPSIS

```
#include <math.h>

double floor(x)
double x;

double ceil(x)
double x;

double fabs(x)
double x;
```

DESCRIPTION

fabs returns the absolute value of *x*.

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

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

NAME

fopen, freopen, fdopen - open a stream

SYNOPSIS

```
#include "stdio.h"

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

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

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

SEE ALSO

I/O (O), Standard I/O (O)

DIAGNOSTICS

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

EXAMPLES

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

```
#include "stdio.h"

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

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

Here is a program which uses *freopen*:

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

Here is a program which uses *fdopen*:

```
#include "stdio.h"

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

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

NAME

fread, fwrite - buffered binary input/output

SYNOPSIS

```
#include "stdio.h"

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

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

DESCRIPTION

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

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

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

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

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

SEE ALSO

Standard I/O (O), Errors (O), fopen, ferror

DIAGNOSTICS

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

EXAMPLE

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

```
#include "stdio.h"

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

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

NAME

frexp, ldexp, modf - build and unbuild real numbers

SYNOPSIS

```
#include <math.h>

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

double ldexp(value, exp)
double value;

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

DESCRIPTION

Given *value*, *frexp* computes integers x and n such that value=x*2**n. x is returned as the value of *frexp*, and n is stored in the *int* field pointed at by *eptr*.

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

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

NAME

fseek, ftell - reposition a stream

SYNOPSIS

```
#include "stdio.h"

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

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

DESCRIPTION

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

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

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

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

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

If *fseek* is successful, it will return zero.

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

SEE ALSO

Standard I/O (O), I/O (O), lseek

DIAGNOSTICS

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

EXAMPLE

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

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

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

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

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

NAME

getc, agetc, getchar, getw

SYNOPSIS

```
#include "stdio.h"

int getc(stream)
FILE *stream;

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

int getchar()

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

DESCRIPTION

getc returns the next character from the specified input stream.

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

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

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

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

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

SEE ALSO

I/O (O), Standard I/O (O), fopen, fclose

DIAGNOSTICS

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

integer *errno*.

NAME

gets, fgets - get a string from a stream

SYNOPSIS

```
#include "stdio.h"

char *gets(s)
char *s;

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

DESCRIPTION

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

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

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

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

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

SEE ALSO

I/O (O), Standard I/O (O), ferror

DIAGNOSTICS

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

NAME

ioctl, isatty - device i/o utilities

SYNOPSIS

#include "sgtty.h"

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

isatty(fd)

DESCRIPTION

ioctl sets and determines the mode of the console.

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

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

SEE ALSO

Console I/O (O)

NAME

lseek - change current position within file

SYNOPSIS

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

DESCRIPTION

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

fd is the file descriptor associated with the file.

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

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

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

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

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

SEE ALSO

Unbuffered I/O (O), I/O (O)

DIAGNOSTICS

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

EXAMPLES

1. To seek to the beginning of a file:

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

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

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

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

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

3. To seek backward five bytes:

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

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

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

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

NAME

malloc, calloc, realloc, free - memory allocation

SYNOPSIS

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

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

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

free(ptr)
char *ptr;
```

DESCRIPTION

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

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

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

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

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

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

SEE ALSO

Memory Usage (O), break (S)

DIAGNOSTICS

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

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

NAME

movmem, setmem, swapmem

SYNOPSIS

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

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

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

DESCRIPTION

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

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

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

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

NAME

open

SYNOPSIS

```
#include "fcntl.h"

open(name, mode)    /* calling sequence on most systems */
char *name;

/* calling sequence on some systems (see below): */
open(name, mode, param3)
char *name;
```

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

param3 is not needed or used on many systems. If it is needed for your system, the System Dependent Library Functions chapter will contain a description of the *open* function, which will define this parameter.

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

SEE ALSO

I/O (O), Unbuffered I/O (O), Errors (O)

DIAGNOSTICS

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

EXAMPLES

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

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

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

2. To open the file, sub1, for read-write access:

```
fd = open("sub1", O_RDWR+O_CREAT);
```

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

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

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

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

NAME

printf, fprintf, sprintf, format
- formatted output conversion functions

SYNOPSIS

```
#include "stdio.h"

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

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

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

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

DESCRIPTION

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

printf outputs the result to the standard output stream, *stdout*;

fprintf outputs the result to the stream specified in its first argument, *stream*;

sprintf places the result in the buffer pointed at by its first argument, *buffer*, and terminates the result with the null character, '\0'.

format calls the function *func* with each character of the result. In fact, *printf*, *fprintf*, and *sprintf* call *format* with each character that they generate.

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

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

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

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

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

The conversion characters are:

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

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

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

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

f The float or double *arg* is converted to decimal notation in the style '[-]ddd.ddd'. The number

of d's after the decimal point is equal to the precision given in the conversion specification. If the precision is missing, it defaults to six digits. If the precision is explicitly 0, the decimal point is also not printed.

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

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

% Output a %. No argument is converted.

SEE ALSO

Standard I/O (O)

EXAMPLES

1. The following program fragment:

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

   will print a message of the form

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

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

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

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

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

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

4. This example demonstrates how to use the *format* function by listing *printf*, which calls *format* with each character that it generates.

```
printf(fmt,args)
char *fmt; unsigned args;
{
    extern int putchar();
    format(putchar,fmt,&args);
}
```

NAME

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

SYNOPSIS

```
#include "stdio.h"

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

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

putchar(c)
char c;

putw(w,stream)
FILE *stream;

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

DESCRIPTION

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

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

* When a newline character is passed to *aputc*, an end- of-line sequence (eg, carriage return followed by line feed on CP/M, and carriage return only on Apple DOS) is written to the stream;
* The most significant bit of a character is set to zero before being written to the stream.
* *aputc* is not a UNIX function. It is, however, supported on all Aztec C systems, and provides a convenient, system-independent way for a program to write text.
* *putchar* writes the character *c* to the standard output stream, stdout. It's identical to *aputc(c, stdout)*.
* *putw* writes the word *w* to the specified stream. It returns *w* as its value. *putw* neither requires nor causes special alignment in the file.
* *puterr* writes the character *c* to the standard error stream, stderr. It's identical to *aputc(c, stderr)*. It is not a UNIX function.

SEE ALSO

Standard I/O

DIAGNOSTICS

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

NAME

puts, fputs - put a character string on a stream

SYNOPSIS

```
#include "stdio.h"

puts(s)
char *s;

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

DESCRIPTION

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

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

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

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

SEE ALSO

Standard I/O (O), putc

DIAGNOSTICS

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

NAME

qsort - sort an array of records in memory

SYNOPSIS

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

DESCRIPTION

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

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

EXAMPLE

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

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

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

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

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

NAME

ran - random number generator

SYNOPSIS

double ran()

DESCRIPTION

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

NAME

read - read from device or file without buffering

SYNOPSIS

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

DESCRIPTION

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

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

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

bufsize is the number of characters to be transferred.

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

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

SEE ALSO

Unbuffered I/O (O), open, close

DIAGNOSTICS

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

NAME

rename - rename a disk file

SYNOPSIS

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

DESCRIPTION

rename changes the name of a file.

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

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

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

NAME

scanf, fscanf, sscanf - formatted input conversion

SYNOPSIS

```
#include "stdio.h"

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

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

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

DESCRIPTION

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

The functions get the text from different places:

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

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

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

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

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

* Conversion specifications;
* 'White space' characters (space, tab newline);
* Ordinary characters; that is, characters which aren't part of a conversion specification and which aren't white space.

A scan function works its way through a control string, trying to match each control item to a portion of the input stream or buffer. During the matching process, it fetches characters one at a time from the input. When a character is fetched which isn't appropriate for the control item being matched, the scan function pushes it back into the input stream or buffer and

finishes processing the current control item. This pushing back frequently gives unexpected results when a stream is being accessed by other i/o functions, such as *getc*, as well as the scan function. The examples below demonstrate some of the problems that can occur.

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

Matching 'white space' characters

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

Matching ordinary characters

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

Matching conversion specifications

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

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

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

Details of input conversion

A conversion specification consists of:

* The character '%', which tells the scan function that it

has encountered a conversion specification;
* Optionally, the assignment suppression character '*';
* Optionally, a 'field width'; that is, a number specifying the maximum number of characters to be fetched for the conversion;
* A conversion character, specifying the type of conversion to be performed.

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

The following conversion characters are supported:

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

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

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

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

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

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

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

* Optionally, a '^' or '~' character;
* A set of characters;
* A right bracket, ']'.

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

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

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

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

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

SEE ALSO

Standard I/O (O)

EXAMPLES

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

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

The input line

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

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

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

lines, with the white space character newline used to separate the values.

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

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

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

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

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

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

   The programmer might place a newline character at the end of a control string, mistakenly thinking that this will circumvent the problem discussed in example 2. One might think that *scanf* will treat the newline as it would an

ordinary character in the control string; that is, that *scanf* will search for, and remove, the terminating newline character from the input stream after it has matched the z variable. However, this is incorrect, and should be remembered as a common misinterpretation.

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

   32 rufus 75.36e-1

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

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

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

   or

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

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

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

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

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

7. Consider the following program fragment:

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

   When given the following input:

   12345 678 90a65

   scanf will assign 12 to x, 345.0 to y, skip '678', and place

the string '90 ' in z. The next call to *getchar* will return 'a'.

NAME

setbuf - assign buffer to a stream

SYNOPSIS

```
#include "stdio.h"

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

DESCRIPTION

setbuf defines the buffer that's to be used for the i/o stream *stream*. If *buf* is not a NULL pointer, the buffer that it points at will be used for the stream instead of an automatically allocated buffer. If *buf* is a NULL pointer, the stream will be completely unbuffered.

When *buf* is not NULL, the buffer it points at must contain BUFSIZ bytes, where BUFSIZ is defined in *stdio.h*.

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

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

SEE ALSO

Standard I/O (O), malloc

NAME

setjmp, longjmp - non-local goto

SYNOPSIS

```
#include "setjmp.h"

setjmp(env)
jmp_buf env;

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

DESCRIPTION

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

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

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

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

WARNING

longjmp must not be called without *env* having been initialized by a call to *setjmp.* It also must not be called if the function that called *setjmp* has since returned.

EXAMPLE

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

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

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

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

```
#include "setjmp.h"

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

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

Here's one of the low level functions:

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

NAME

trigonometric functions:
sin, cos, tan, cotan, asin, acos, atan, atan2

SYNOPSIS

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

double tan(x)
double x;

double cotan(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

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

DESCRIPTION

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

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

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

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

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

SEE ALSO

Errors (O)

DIAGNOSTICS

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

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

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

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

NAME

sinh, cosh, tanh

SYNOPSIS

```
#include <math.h>

double sinh(x)
double x;

double cosh(x)
double x;

double tanh(x)
double x;
```

DESCRIPTION

These functions compute the hyperbolic functions of their arguments.

SEE ALSO

Errors (O)

DIAGNOSTICS

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

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

NAME

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

SYNOPSIS

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

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

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

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

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

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

strlen(s)
char *s;

char *index(s, c)
char *s;

char *rindex(s, c)
char *s;
```

DESCRIPTION

These functions operate on null-terminated strings, as follows:

strcat appends a copy of string *s2* to string *s1*. *strncat* copies at most *n* characters. Both terminate the resulting string with the null character (\0) and return a pointer to the first character of the resulting string.

strcmp compares its two arguments and returns an integer greater than, equal, or less than zero, according as *s1* is lexicographically greater than, equal to, or less than *s2*. *strncmp* makes the same comparison but looks at *n* characters at most.

strcpy copies string *s2* to *s1* stopping after the null character has been moved. *strncpy* copies exactly *n* characters: if *s2* contains less than *n* characters, null characters will be appended to the resulting string until *n* characters have been moved; if *s2* contains *n* or more characters, only the first *n* will be moved, and the resulting string will not be null terminated.

strlen returns the number of characters which occur in *s* up to the first null character.

index returns a pointer to the first occurrance of the character *c* in string *s*, or zero if *c* isn't in the string.

rindex returns a pointer to the last occurrance of the character *c* in string *s*, or zero if *c* isn't in the string.

NAME

toupper, tolower

SYNOPSIS

toupper(c)

tolower(c)

#include "ctype.h"

_toupper(c)

_tolower(c)

DESCRIPTION

toupper converts a lower case character to upper case: if *c* is a lower case character, *toupper* returns its upper case equivalent as its value, otherwise *c* is returned.

tolower converts an upper case character to lowr case: if *c* is an upper case character *tolower* returns its lower case equivalent, otherwise *c* is returned.

toupper and *tolower* do not require the header file *ctype.h.*

_toupper and *_tolower* are macro versions of *toupper* and *tolower*, respectively. They are defined in *ctype.h.* The difference between the two sets of functions is that the macro versions will sometimes translate non-alphabetic characters, whereas the function versions don't.

NAME

ungetc - push a character back into input stream

SYNOPSIS

```
#include "stdio.h"

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

DESCRIPTION

ungetc pushes the character *c* back on an input stream. That character will be returned by the next *getc* call on that stream. *ungetc* returns *c* as its value.

Only one character of pushback is guaranteed. EOF cannot be pushed back.

SEE ALSO

Standard I/O (O)

DIAGNOSTICS

ungetc returns EOF (-1) if the character can't be pushed back.

NAME

unlink

SYNOPSIS

unlink(name)
char *name;

DESCRIPTION

unlink erases a file.

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

unlink returns 0 if successful.

DIAGNOSTICS

unlink returns -1 if it couldn't erase the file and places a code in the global integer *errno* describing the error.

NAME

write

SYNOPSIS

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

DESCRIPTION

write writes characters to a device or disk which has been previously opened by a call to *open* or *creat.* The characters are written to the device or file directly from the caller's buffer.

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

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

bufsize is the number of characters to be written.

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

SEE ALSO

Unbuffered I/O (O) , open, close, read

DIAGNOSTICS

If the operation is unsuccessful, *write* returns -1 and places a code in the global integer *errno.*

STYLE

Chapter Contents

Style

This section was written for the programmer who is new to the C language, to communicate the special character of C and the programming practices for which it is best suited. This material will ease the new user's entry into C. It gives meaning to the peculiarities of C syntax, in order to avoid the errors which will otherwise disappear only with experience.

1. Introduction

what's in it for me?

These are the benefits to be reaped by following the methods presented here:

* Reduced debugging times;
* Increased program efficiency;
* Reduced software maintenance burden.

The aim of the responsible programmer is to write straightforward code, which makes his programs more accessible to others. This section on style is meant to point out which programming habits are conducive to successful C programs and which are especially prone to cause trouble.

The many advantages of C can be abused. Since C is a terse, subtle language, it is easy to write code which is unclear. This is contrary to the "philosophy" of C and other structured programming languages, according to which the structure of a program should be clearly defined and easily recognizable.

keep it simple

There are several elements of programming style which make C easier to use. One of these is *simplicity*. Simplicity means *keep it simple*. You should be able to see exactly what your code will do, so that when it doesn't you can figure out why.

A little suspicion can also be useful. The particular "problem areas" which are discussed later in this section are points to check when code "looks right" but does not work. A small omission can cause many errors.

learn the C idioms

C becomes more valuable and more flexible with time. Obviously, elementary problems with syntax will disappear. But more importantly,

C can be described as "idiomatic." This means that certain expressions become part of a standard vocabulary used over and over.

For example,

```
while ((c = getchar()) != EOF)
```

is readily recognized and written by any C programmer. This is often used as the beginning of a loop which gets a character at a time from a source of input. Moreover, the inside set of parentheses, often omitted by a new C programmer, is rarely forgotten after this construct has been used a few times.

be flexible in using the library

The standard library contains a choice of functions for performing the same task. Certain combinations offer advantages, so that they are used routinely. For instance, the standard library contains a function, *scanf*, which can be used to input data of a given format. In this example, the function "scans" input for a floating point number:

```
scanf("%f", &flt_num);
```

There are several disadvantages to this function. An important debit is that it requires a lot of code. Also, it is not always clear how this function handles certain strings of input. Much time could be spent researching the behavior of this function. However, the equivalent to the above is done by the following:

```
flt_num = atof(gets(inp_buf));
```

This requires considerably less code, and is somewhat more straightforward. *gets* puts a line of input into the buffer, "inp_buf," and *atof* converts it to a floating point value. There is no question about what the input function is "looking for" and what it should find.

Furthermore, there is greater flexibility in the second method of getting input. For instance, if the user of the program could enter either a special command ("e" for exit) or a floating point value, the following is possible:

```
gets(inp_buf);
if (inp_buf[0] == 'e')
        exit(0);
flt_num = atof(inp_buf);
```

Here, the first character of input is checked for an "e", before the input is converted to a float.

The relative length of the library description of the *scanf* function is an indication of the problems that can arise with that and related functions.

write readable code

Readability can be greatly enhanced by adhering to what common sense dictates. For instance, most lines can easily accommodate more than one statement. Although the compiler will accept statements which are packed together indiscriminately, the logic behind the code will be lost. Therefore, it makes sense to write no more than one statement per line.

In a similar vein, it is desirable to be generous with whitespace. A blank space should separate the arithmetic and assignment operators from other symbols, such as variable names. And when parentheses are nested, dividing them with spaces is not being too prudent. For example,

```
if((fp=fopen("filename","r")==NULL))
```

is not the same as

```
if ( (fp = fopen("filename", "r")) == NULL )
```

The first line contains a misplaced parenthesis which changes the meaning of the statement entirely. (A file is opened but the file pointer will be null.) If the statement was expanded, as in the second line, the problem could be easily spotted, if not avoided altogther.

use straightforward logical expressions

Conditionals are apt to grow into long expressions. They should be kept short. Conditionals which extend into the next line should be divided so that the logic of the statement can be visualized at a glance. Another solution might be to reconsider the logic of the code itself.

learn the rules for expression evaluation

Keep in mind that the evaluation of an expression depends upon the order in which the operators are evaluated. This is determined from their relative precedence.

Item 7 in the list of "things to watch out for", below, gives an example of what may happen when the evaluation of a boolean expression stops "in the middle". The rule in C is that a boolean will be evaluated only until the value of the expression can be determined.

Item 8 gives a well known example of an "undefined" expression, one whose value is not strictly determined.

In general, if an expression depends upon the order in which it is evaluated, the results may be dubious. Though the result may be strictly defined, you must be certain you know what that definition is.

a matter of taste

There are several popular styles of indentation and placement of the braces enclosing compound statements. Whichever format you

adopt, it is important to be consistent. Indentation is the accepted way of conveying the intended nesting of program statements to other programmers. However, the compiler understands only braces. Making them as visible as possible will help in tracking down nesting errors later.

However much time is devoted to writing readible code, C is low-level enough to permit some very peculiar expressions.

```
/* It is important to insert comments on a regular basis! */
```

Comments are especially useful as brief introductions to function definitions.

In general, moderate observance of these suggestions will lessen the number of "tricks" C will play on you-- even after you have mastered its syntax.

2. Structured Programming

"Structured programming" is an attempt to encourage programming characterized by method and clarity. It stems from the theory that any programming task can be broken into simpler components. The three basic parts are statements, loops, and conditionals. In C, these parts are, respectively, anything enclosed by braces or ending with a semicolon; *for*, *while* and *do-while*; *if-else*.

modularity and block structure

Central to structured programming is the concept of modularity. In one sense, any source file compiled by itself is a module. However, the term is used here with a more specific meaning. In this context, modularity refers to the independence or isolation of one routine from another. For example, a routine such as *main()* can call a function to do a given task even though it does not know how the task is accomplished or what intermediate values are used to reach the final result.

Sections of a program set aside by braces are called "blocks". The "privacy" of C's block structure ensures that the variables of each block are not inadvertently shared by other blocks. Any left brace ({) signals the beginning of a block, such as the body of a function or a *for* loop. Since each block can have its own set of variables, a left brace marks an opportunity to declare a temporary variable.

A function in C is a special block because it is called and is passed control of execution. A function is called, executes and returns. Essentially, a C program is just such a routine, namely, *main*.

A function call represents a task to be accomplished. Program statements which might otherwise appear as several obscure lines can be set aside in a function which satisfies a desired purpose. For instance, *getchar* is used to get a single character from standard input.

When a section of code must be modified, it is simpler to replace a single modular block than it is to delete a section of an unstructured program whose boundaries may be unclear at best. In general, the more precisely a block of program is defined, the more easily it can be changed.

3. Top-down Programming

"Top-down" programming is one method that takes advantage of structured programming features like those discussed above. It is a method of designing, writing, and testing a program from the most general function (i.e., (main()) to the most specific functions (such as getchar()).

All C programs begin with a function called main(). main() can be thought of as a supervisor or manager which calls upon other functions to perform specific tasks, doing little of the work itself. If the overall goal of the program can be considered in four parts (for instance, input, processing, error checking and output), then main() should call at least four other functions.

step one

The first step in the design of a program is to identify what is to be done and how it can be accomplished in a "programmable" way. The *main* routine should be greatly simplified. It needs to call a function to perform the crucial steps in the program. For example, it may call a function, init(), which takes care of all necessary startup initializations. At this point, the programmer does not even need to be certain of all the initializations that will take place in init().

All functions consist of three parts: a parameter list, body, and return value. The design of a function must focus on each of these three elements.

During this first stage of design, each function can be considered a black box. We are concerned only with what goes in and what comes out, not with what goes on inside.

Do not allow yourself to be distracted by the details of the implementation at this point. Flowcharts, pseudocode, decision tables and the like are useful at this stage of the implementation.

A detailed list of the data which is passed back and forth between functions is important and should not be neglected. The interface between functions is crucial.

Although all functions are written with a purpose in mind, it is easy to unwittingly merge two tasks into one. Sometimes, this may be done in the interests of producing a compact and efficient program function. However, the usual result is a bulky, unmanageable function. If a function grows very large or if its logic becomes difficult to comprehend, it should be reduced by introducing additional function calls.

step two

There comes a time when a program must pass from the design stage into the coding stage. You may find the top-down approach to

coding too restrictive. According to this scheme, the smallest and most specific functions would be coded last. It is our nature to tackle the most daunting problems first, which usually means coding the low-level functions.

Whereas the top-down approach is the preferred method for designing software, the bottom-up approach is often the most practical method for writing software. Given a good design, either method of implementation should produce equally good results.

One asset of top-down writing is the ability to provide immediate tests on upper level routines. Unresolved function calls can be satisfied by "dummy" functions which return a range of test values. When new functions are added, they can operate in an environment that has already been tested.

C functions are most effective when they are as mutually independent as is possible. This independence is encouraged by the fact that there is normally only one way into and one way out of a function: by calling it with specific arguments and returning a meaningful value. Any function can be modified or replaced so long as its entry and exit points are consistent with the calling function.

4. Defensive Programming and Debugging

"Defensive programming" obeys the same edict as defensive driving: trust no one to do what you expect. There are two sides to this rule of thumb. Defend against both the possibility of bad data or misuse of the program by the user, and the possibility of bad data generated by bad code.

Pointers, for example, are a prime source of variables gone astray. Even though the "theory" of pointers may be well understood, using them in new ways (or for the first time) requires careful consideration at each step. Pointers present the fewest problems when they appear in familiar settings.

faced with the unknown

When trying something new, first write a few test programs to make sure the syntax you are using is correct. For example, consider a buffer, *str_buf*, filled with null-terminated strings. Suppose we want to print the string which begins at offset *begin* in the buffer. Is this the way to do it?

```
printf("%s", str_buf[begin]);
```

A little investigation shows that str_buf[begin] is a character, not a pointer to a string, which is what is called for. The correct statement is

```
printf("%s", str_buf + begin);
```

This kind of error may not be obvious when you first see it. There are other topics which can be troublesome at first exposure. The promotion of data types within expressions is an example. Even if you are sure how a new construct behaves, it never hurts to doublecheck with a test program.

Certain programming habits will ease the bite of syntax. Foremost among these is simplicity of style. Top-down programming is aimed at producing brief and consequently simple functions. This simplicity should not disappear when the design is coded.

Code should appear as "idiomatic" as possible. Pointers can again provide an example: it is a fact of C syntax that arrays and pointers are one and the same. That is,

```
array[offset]
```

is the same as

```
*(array + offset)
```

The only difference is that an array name is not an lvalue; it is fixed. But mixing the two ways of referencing an object can cause confusion, such as in the last example. Choosing a certain idiom, which is known to behave a certain way, can help avoid many errors in usage.

when bugs strike

The assumption must be that you will have to return to the source code to make changes, probably due to what is called a bug. Bugs are characterized by their persistence and their tendency to multiply rapidly.

Errors can occur at either compile-time or run-time. Compile-time errors are somewhat easier to resolve since they are usually errors in syntax which the compiler will point out.

from the compiler

If the compiler does pick up an error in the source code, it will send an error code to the screen and try to specify where the error occurred. There are several peculiarities about error reporting which should be brought up right away.

The most noticeable of these peculiarities is the number of spurious errors which the compiler may report. This interval of inconsistency is referred to as the compiler's recovery. The safest way to deal with an unusually long list of errors is to correct the first error and then recompile before proceeding.

The compiler will specify where it "noticed" something was wrong. This does not necessarily indicate where you must make a change in the code. The error number is a more accurate clue, since it shows what the compiler was looking for when the error occurred.

if this ever happens to you

A common example of this is error 69: "missing semicolon." This error code will be put out if the compiler is expecting a semicolon when it finds some other character. Since this error most often occurs at the end of a line, it may not be reported until the first character of the following line-- recall that whitespace, such as a newline character, is ignored.

Such an error can be especially treacherous in certain situations. For example, a missing semicolon at the end of a #*include*'d file may be reported when the compiler returns to read input in the original file.

In general, it is helpful to look at a syntax error from the compiler's point of view.

Consider this error:

```
struct structag {
        char c;
        int i;
}

int j;
```

This should generate an error 16: "data type conflict". The arrow in the error message should show that the error was detected right after the "int" in the declaration of *j*. This means that the error has to do with something before that line, since there is nothing illegal about the *int* keyword.

By inspection, we may see that the semicolon is missing from the preceding line. If this fact escapes our notice, we still know that error 16 means this: the compiler found a declaration of the form

[data type] [data type] [symbol name]

where the two data types were incompatible. So while *shortint* is a good data type, *double int* is not. A small intuitive leap leads us to assume that the compiler has read our source as a kind of "struct int" declaration; *struct* is the only keyword preceding the *int* which could have caused this error. Since the compiler is reading the two declarations as a single statement, we must be missing a delimiter.

run-time errors

It takes a bit more ingenuity to locate errors which occur at run-time. In numerical calculations, only the most anomalous results will draw attention to themselves. Other bugs will generate output which will appear to have come from an entirely different program.

A bug is most useful when it is repeatable. Bugs which show up only "sometimes" are merely vexing. They can be caused by a corrupted disk file or a bad command from the user.

When an error can be consistently produced, its source can be more easily located. The nature of an error is a good clue as to its source. Much of your time and sanity will be preserved by setting aside a few minutes to reflect upon the problem.

Which modules are involved in the computation or process? Many possibilities can be eliminated from the start, such as pieces of code which are unrelated to the error.

The first goal is to determine, from a number of possibilities, which module might be the source of the bug.

checking input data

Input to the program can be checked at a low cost. Error checking of this sort should be included on a "routine" basis. For instance, "if ((fp=fopen("file","r"))==NULL)" should be reflex when a file is

opened. Any useful error handling can follow in the body of the *if*.

It is easy to check your data when you first get your hands on it. If an error occurs after that, you have a bug in your program.

printf it

It is useful to know where the data goes awry. One brute force way of tracking down the bug is to insert *printf* statements wherever the data is referenced. When an unexpected value comes up, a single module can be chosen for further investigation.

The printf search will be most effective when done with more refinement. Choose a suspect module. There are only two keys points to check: the entry and return of the function. *printf* the data in question just as soon as the function is entered. If the values are already incorrect, then you will want to make sure the correct data was passed in the function call.

If an incorrect value is returned, then the search is confined to the guilty function. Even if the function returns a good value, you may want to make sure it is handled correctly by the calling function.

If everything seems to be working, jump to the next tricky module and perform another check. When you find a bad result, you will still have to backtrack to discover precisely where the data was spoiled.

function calls

Be aware that data can be garbled in a funtion call. Function parameters must be declared when they are not two byte integers. For instance, if a function is called:

```
fseek(fp, 0, 0);
```

in order to "seek" to the beginning of a file, but the function is defined this way:

```
fseek(fp, offset, origin)
FILE *fp;
long offset;
int origin;
```

there will be unfortunate consequences.

The second parameter is expected to be a *long* integer (four bytes), but what is being passed is a *short* integer (two bytes). In a function call, the arguments are just being pushed onto the stack; when the function is entered, they are pulled off again. In the example, two bytes are being pushed on, but four bytes (whatever four bytes are there) are being pulled off.

The solution is just to make the second parameter a long, with a suffix (0L) or by the cast operator (as in (long)i).

A similar problem occurs when a non-integer return value is not declared in the calling function. For example, if *sqrt* is being called, it must be declared as returning a *double*:

```
double sqrt();
```

This method of debugging demonstrates the usefulness of having a solid design before a function is coded. If you know what should be going into a function and what should be coming out, the process of checking that data is made much simpler.

found it

When the guilty function is isolated, the difficulty of finding the bug is proportional to the simplicity of the code. However, the search can continue in a similar way. You should have a good notion of the purpose of each block, such as a loop. By inserting a *printf* in a loop, you can observe the effect of each pass on the data.

printf's can also point out which blocks are actually being executed. "Falling through" a test, such as an *if* or a *switch*, can be a subtle source of problems. Conditionals should not leave cases untested. An *else*, or a *default* in a *switch*, can rescue the code from unexpected input.

And if you are uncertain how a piece of code will work, it is usually worthwhile to set up small test programs and observe what happens. This is instructional and may reveal a bug or two.

5. Things to Watch Out for

Some errors arise again and again. Not all of them go away with experience. The following list will give you an idea of the kinds of things that can go wrong.

* **missing semicolon or brace**

The compiler will tell you when a missing semicolon or brace has introduced bad syntax into the code. However, often such an error will affect only the logical structure of the program; the code may compile and even execute. When this error is not revealed by inspection, it is usually brought out by a test *printf* which is executed too often or not enough. See compiler error 69.

* **assignment (=) vs comparison (==)**

Since variables are assigned values more often than they are tested for equality, the former operator was given the single keystroke: =. Notice that all the comparison tests with equality are two characters: <=, >= and ==.

* **misplaced semicolon**

When typing in a program, keep in mind that all source lines do not automatically end with a semicolon. Control lines are especially susceptible to an unwanted semicolon:

```
for (i=0; i<100; i++);
        printf("%d",i);
```

This example prints the single number 100.

* **division (/) vs escape sequence (\)**

C definitely distinguishes between these characters. The division sign resides below the question mark on a standard console; the backslash is generally harder to find.

* **character constant vs character string**

Character constants are actually integers equal to the ASCII values of the respective character. A character string is a series of characters terminated by a null character (\0). The appropriate delimiter is the single quote and double quote, respectively.

* **uninitialized variable**

At some point, all variables must be given values before they are used. The compiler will set global and static variables to zero, but automatic variables are guaranteed to contain garbage every time they are created.

* **evaluation of expressions**

For most operations in C, the order of evaluation is rigidly defined; thus, many expressions can be written without lots of parentheses.

However, the order in which unparenthesized expressions are evaluated are not always what you would expect; therefore, it's usually a good idea to use parentheses liberally in expressions where there may be doubt about the order of evaluation (in your mind or in the mind of someone who may later read your program).

For example, the result of the following example is 6:

```
int a = 2, b = 3, c = 4, d;
d = a + b / a * c;
```

The above expression is equivalent to the parenthesized expression *d = a + ((b / a) * c);*. You should probably use some parentheses in this expression, to make its effect clear to yourself and to others.

Consider this example:

```
if ( (c = 0) || (c = 1) )
        printf("%d", c);
```

"1" will be printed; since the first half of the conditional evaluates to zero, the second half must be also evaluated. But in this example:

```
if ( (c = 0) && (c = 1) )
        ;
printf("%d", c);
```

a "0" is printed. Since the first half evaluates to zero, the value of the conditional must be zero, or false, and evaluation stops. This is a property of the logical operators.

* **undefined order of evaluation**

Unfortunately, not all operators were given a complete set of instructions as to how they should be evaluated. A good example is the increment (or decrement) operator. For instance, the following is undefined:

```
i = ++i + --i/++i - i++;
```

How such an expression is evaluated by a particular implementation is called a "side effect." In general, side effects are to be avoided.

* **evaluation of boolean expressions**

Ands, ors and nots invite the programmer to write long conditionals whose very purpose is lost in the code. Booleans should be brief and to the point. Also, the bitwise logical operators must be fully parenthesized. The table in sections 2.12 and 18.1 of *The C Programming Language*, by Kernighan and Ritchie, shows their precedence in relation to other operators.

Here is an extreme example of how a lengthy boolean can be reduced:

```
if ((c = getchar()) != EOF && c >= 'a' && c <= 'z' &&
(c = getchar()) >= '1' && c <= '9')
   printf("good input\n");

if ((c = getchar()) != EOF)
   if (c >= 'a' && c <= 'z')
      if ((c = getchar()) >= '0' && c <= '9')
         printf("good input\n");
```

* **badly formed comments**

The theory of comment syntax is simply that everything occurring between a left /* and a right */ is ignored by the compiler. Nonetheless, a missing */ should not be overlooked as a possible error.

Note that comments cannot be nested, that is

```
/* /*  this will cause an error  */  */
```

And this could happen to you too:

```
/* the rest of this file is ignored until another comment /*
```

* **nesting error**

Remember that nesting is determined by braces and not by indentations in the text of the source. Nested *if* statements merit particular care since they are often paired with an *else*.

* **usage of else**

Every else must pair up with an *if*. When an *else* has inexplicably remained unpaired, the cause is often related to the first error in this list.

* **falling through the cases in a switch**

To maintain the most control over the *cases* in a *switch* statement, it is advisable to end each *case* with a *break*, including the last *case* in the *switch*.

* **strange loops**

The behavior of loops can be explored by inserting *printf* statements in the body of the loop. Obviously, this will indicate if the loop has even been entered at all in course of a run. A counter will show just how many times the loop was executed; a small slip-up will cause a loop to be run through once too often or seldom. The condition for leaving the loop should be doublechecked for accuracy.

* **use of strings**

All strings must be terminated by a null character in memory. Thus, the string, "hello", will occupy a six-element array; the sixth element is '\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.

* **pointer vs object of a pointer**

The greatest difficulty in using pointers is being sure of what is needed and what is being used. Functions which take a pointer argument require an address in memory. The best way to ensure that the correct value is being passed is to keep track of what is being pointed to by which pointer.

* **array subscripting**

The first element in a C array has a subscript of zero. The array name without a subscript is actually a pointer to this element. Obviously, many problems can develop from an incorrect subscript. The most damaging can be subscripting out of bounds, since this will access memory above the array and overwrite any data there. If array elements or data stored with arrays are being lost, this error is a good candidate.

* **function interface**

During the design stage, the components of a program should be associated with functions. It is important that the data which is passed among or shared by these functions be explicitly defined in the preliminary design of the program. This will greatly facilitate the coding of the program since the interface between functions must be precise in several respects.

First of all, if the parameters of a function are established, a call can be made without the reservation that it will be changed later. There is less chance that the arguments will be of the wrong type or specified in the wrong order.

A function is given only a private copy of the variables it is passed. This is a good reason to decide while designing the program how functions should access the data they require. You will be able to detail the arguments to be passed in a function call, the global data which the function will alter, the value which the function will return and what declarations will be appropriate-- all without concern for how the function will be coded.

Argument declarations should be a fairly simple matter once these things are known. Note that this declaration list must stand before the left brace of the function body.

The type of the function is the same as the type of the value it returns. Functions must be declared just like any variable. And just like variables, functions will default to type int, that is, the compiler will assume that a function returns an integer if you do not tell it otherwise with a declaration. Thus if function f calls function g which returns a variable of type double, the following declaration is needed:

```
function f()
{
        double g(), bigfloat;

        g(bigfloat);
}
double g(arg)
double arg;
{
        return(arg);
}
```

* **be sure of what a function returns**

You will probably know very well what is returned by a function you have written yourself. But care should be taken when using functions coded by someone else. This is especially true of the standard library functions. Most of the supplied library functions will return an int or a char pointer where you might expect a char. For instance, getchar() returns an int, not a char. The functions supplied by Manx adhere to the UNIX model in all but a few cases.

Of course, the above applies to a function's arguments as well.

* **shared data**

Variables that are declared globally can be accessed by all functions in the file. This is not a very safe way to pass data to functions since once a global variable is altered, there is no returning it to its former state without an elaborate method of saving data. Moreover, global data must be carefully managed; a function may process the wrong variable and consequently inhibit any other function which depends on that data.

Since C provides for and even encourages private data, this definitely should not be a common bug.

COMPILER ERROR MESSAGES

Chapter Contents

Compiler Error Messages

This chapter discusses error messages that can be generated by the compiler. It is divided into three sections: the first summarizes the messages, the second explains them, and the third discusses fatal compiler error messsages.

1. Summary of error codes

No. Interpretation

1: bad digit in octal constant
2: string space exhausted
3: unterminated string
4: internal error
5: illegal type for function
6: inappropriate arguments
7: bad declaration syntax
8: syntax error in typecast
9: array dimension must be constant
10: array size must be positive integer
11: data type too complex
12: illegal pointer reference
13: unimplemented type
14: internal
15: internal
16: data type conflict
17: unsupported data type
18: data type conflict
19: obsolete
20: structure redeclaration
21: missing }
22: syntax error in structure declaration
23: incorrect type for library function (Apprentice C only)
obsolete (other Aztec C compilers)
24: need right parenthesis or comma in arg list
25: structure member name expected here
26: must be structure/union member
27: illegal typecast
28: incompatible structures
29: illegal use of structure
30: missing : in ? conditional expression
31: call of non-function
32: illegal pointer calculation
33: illegal type
34: undefined symbol
35: typedef not allowed here
36: no more expression space
37: invalid expression for unary operator
38: no auto. aggregate initialization allowed
39: obsolete
40: internal
41: initializer not a constant
42: too many initializers

43: initialization of undefined structure
44: obsolete
45: bad declaration syntax
46: missing closing brace
47: open failure on include file
48: illegal symbol name
49: multiply defined symbol
50: missing bracket
51: lvalue required
52: obsolete
53: multiply defined label
54: too many labels
55: missing quote
56: missing apostrophe
57: line too long
58: illegal # encountered
59: macro too long
60: obsolete
61: reference of member of undefined structure
62: function body must be compound statement
63: undefined label
64: inappropriate arguments
65: illegal argument name
66: expected comma
67: invalid else
68: syntax error
69: missing semicolon
70: goto needs a label
71: statement syntax error in do-while
72: 'for' syntax error: missing first semicolon
73: 'for' syntax error: missing second semicolon
74: case value must be an integer constant
75: missing colon on case
76: too many cases in switch
77: case outside of switch
78: missing colon on default
79: duplicate default
80: default outside of switch
81: break/continue error
82: illegal character
83: too many nested includes
84: too many array dimensions
85: not an argument
86: null dimension in array
87: invalid character constant
88: not a structure
89: invalid use of register storage class
90: symbol redeclared

91: illegal use of floating point type
92: illegal type conversion
93: illegal expression type for switch
94: invalid identifier in macro definition
95: macro needs argument list
96: missing argument to macro
97: obsolete
98: not enough arguments in macro reference
99: internal
100: internal
101: missing close parenthesis on macro reference
102: macro arguments too long
103: #else with no #if
104: #endif with no #if
105: #endasm with no #asm
106: #asm within #asm block
107: missing #endif
108: missing #endasm
109: #if value must be integer constant
110: invalid use of : operator
111: invalid use of void expression
112: invalid use function pointer
113: duplicate case in switch
114: macro redefined
115: keyword redefined
116: field width must be > 0
117: invalid 0 length field
118: field is too wide
119: field not allowed here
120: invalid type for field
121: ptr to int conversion
122: ptr & int not same size
123: function ptr & ptr not same size
124: invalid ptr/ptr assignment
125: too many subscripts or indirection on integer

Error codes between 116 and 125 will not occur on Aztec C compilers whose version number is less than 3.

Error codes greater than 200 will occur only if there's something wrong with the compiler. If you get such an error, please send us the program that generated the error.

2. Explanations

1: bad digit in octal constant

The only numerals permitted in the base 8 (octal) counting system are zero through seven. In order to distinguish between octal, hexadecimal, and decimal constants, octal constants are preceded by a zero. Any number beginning with a zero must not contain a digit greater than seven. Octal constants look like this: 01, 027, 003. Hexadecimal constants begin with 0x (e.g., 0x1, 0xAA0, 0xFFF).

2: string space exhausted

The compiler maintains an internal table of the strings appearing in the source code. Since this table has a finite size, it may overflow during compilation and cause this error code. The table default size is about one or two thousand characters depending on the operating system. The size can be changed using the compiler option -Z. Through simple guesswork, it is possible to arrive at a table size sufficient for compiling your program.

3: unterminated string

All strings must begin and end with double quotes ("). This message indicates that a double quote has remained unpaired.

4: internal error

This error message should not occur. It is a check on the internal workings of the compiler and is not known to be caused by any particular piece of code. However, if this error code appears, please bring it to the attention of MANX. It could be a bug in the compiler. The release documentation enclosed with the product contains further information.

5: illegal type for function

The type of a function refers to the type of the value which it returns. Functions return an *int* by default unless they are declared otherwise. However, functions are not allowed to return aggregates (arrays or structures). An attempt to write a function such as *struct sam func()* will generate this error code. The legal function types are *char*, *int*, *float*, *double*, *unsigned*, *long*, *void* and a pointer to any type (including structures).

6: error in argument declaration

The declaration list for the formal parameters of a function stands immediately before the left brace of the function body, as shown below. Undeclared arguments default to *int*, though it is usually better practice to declare everything. Naturally, this declaration list may be empty, whether or not the function takes any arguments at all.

No other inappropriate symbols should appear before the left (open) brace.

```
badfunction(arg1, arg2)
shrt arg 1;    /* misspelled or invalid keyword */
double arg 2;
{ /* function body */
}

goodfunction(arg1,arg2)
float arg1;
int arg2;   /* this line is not required */
{ /* function body */
}
```

7: bad declaration syntax

A common cause of this error is the absence of a semicolon at the end of a declaration. The compiler expects a semicolon to follow a variable declaration unless commas appear between variable names in multiple declarations.

```
int i, j;          /* correct */
char c d;          /* error 7 */
char *s1, *s2
float k;           /* error 7 detected here */
```

Sometimes the compiler may not detect the error until the next program line. A missing semicolon at the end of a *#include*'d file will be detected back in the file being compiled or in another *#include* file. This is a good example of why it is important to examine the context of the error rather than to rely solely on the information provided by the compiler error message(s).

8: syntax error in type cast

The syntax of the cast operator must be carefully observed. A common error is to omit a parenthesis:

```
i = 3 * (int number);          /* incorrect usage */
i = 3 * ((int)number);         /* correct usage */
```

9: array dimension must be constant

The dimension given an array must be a constant of type *char*, *int*, or *unsigned*. This value is specified in the declaration of the array. See error 10.

10: array size must be positive integer

The dimension of an array is required to be greater than zero. A dimension less than or equal to zero becomes 1 by default. As can be seen from the following example, specifying a dimension of zero is not the same as leaving the brackets empty.

```
char badarray[0];              /* meaningless */
extern char goodarray[];       /* good */
```

Empty brackets are used when declaring an array that has been defined (given a size and storage in memory) somewhere else (that is, outside the current function or file). In the above example, *goodarray* is external. Function arguments should be declared with a null dimension:

```
func(s1,s2)
char s1[], s2[];
{
    ...
}
```

11: data type too complex

This message is best explained by example:

```
char *******foo;
```

The form of this declaration implies six pointers-to-pointers. The seventh asterisk indicates a pointer to a *char*. The compiler is unable to keep track of so many "levels". Removing just one of the asterisks will cure the error; all that is being declared in any case is a single two-byte pointer. However it is to be hoped that such a construct will never be needed.

12: illegal pointer reference

The type of a pointer must be either *int* or *unsigned*. This is why you might get away with not declaring pointer arguments in functions like *fopen* which return a pointer; they default to *int*. When this error is generated, an expression used as a pointer is of an invalid type:

```
char c;
int var;                       /* any variable */
int varaddress;
varaddress = &var;             /* valid since addresses */
*(varaddress) = 'c';           /* can fit in an int */
*(expression) = 10;            /* in general, expression
                                  must be an int or unsigned */
*c = 'c';                      /* error 12 */
```

13: internal [see error 4]

14: internal [see error 4]

15: storage class conflict

Only automatic variables and function parameters can be specified as *register*.

This error can be caused by declaring a *static register* variable. While structure members cannot be given a storage class at all, function

arguments can be specified only as *register.*

A *register int i* declaration is not allowed outside a function--it will generate error 89 (see below).

16: data type conflict

The basic data types are not numerous, and there are not many ways to use them in declarations. The possibilities are listed below.

This error code indicates that two incompatible data types were used in conjunction with one another. For example, while it is valid to say *long int i*, and *unsigned int j*, it is meaningless to use *double int k* or *float char c.* In this respect, the compiler checks to make sure that *int, char, float* and *double* are used correctly.

data type	*interpretation*	*size(bytes)*
char	character	1
int	integer	2
unsigned/unsigned int	unsigned integer	2
short	integer	2
long/long integer	long integer	4
float	floating point number	4
long float/double	double precision float	8

17: Unsupported data type

This message occurs only when data types are used which are supported by the extended C language, such as the *enum* data type.

18: data type conflict

This message indicates an error in the use of the *long* or *unsigned* data type. *long* can be applied as a qualifier to *int* and *float. unsigned* can be used with *char, int* and *long.*

```
long i;                 /* a long int */
long float d;           /* a double */
unsigned u;             /* an unsigned int */
unsigned char c;
unsigned long l;
unsigned float f;       /* error 18 */
```

19: obsolete

Error codes interpreted as obsolete do not occur in the current version of the compiler. Some simply no longer apply due to the increased adaptability of the compiler. Other error codes have been translated into full messages sent directly to the screen. If you are using an older version of the product and have need of these codes, please contact Manx for information.

20: structure redeclaration

The compiler is able to tell you if a *struct*ure has already been defined. This message informs you that you have tried to redefine a *struct*ure.

21: missing }

The compiler expects to find a comma after each member in the list of fields for a *struct*ure initialization. After the last field, it expects a right (close) brace.

For example, the following program fragment will generate error 21, since the initialization of the structure named 'harry' doesn't have a closing brace:

```
struct sam {
    int bone;
    char license[10];
} harry = {
    1,
    "23-4-1984";
```

22: syntax error in structure declaration

The compiler was unable to find the left (open) brace which follows the tag in a *struct*ure declaration. In the example for error 21, "sam" is the structure tag. A left brace must follow the keyword *struct* if no structure tag is specified.

23: incorrect type for library function (Apprentice C only)

For Apprentice C, this error means that your program has either explicitly or implicitly incorrectly declared the type of a function that's in the run-time system. For example, you will get this error if you call the run-time system function *sqrt* without declaring that it returns a *double*.

23: obsolete (Other Aztec C Compilers)

For Compilers other than Apprentice C, this error should not occur.

24: need right parenthesis or comma

The right parenthesis is missing from a function call. Every function call must have an argument list enclosed by parentheses even if the list is empty. A right parenthesis is required to terminate the argument list.

In the following example, the parentheses indicate that *getchar* is a function rather than a variable.

```
getchar();
```

This is the equivalent of

```
CALL getchar
```

which might be found in a more explicit programming language. In general, a function is recognized as a name followed by a left parenthesis.

With the exception of reserved words, any name can be made a function by the addition of parentheses. However, if a previously defined variable is used as a function name, a compilation error will result.

Moreover, a comma must separate each argument in the list. For example, error 24 will also result from this statement:

```
funccall(arg1, arg2 arg3);
```

25: structure member name expected here

The symbol name following the dot operator or the arrow must be valid. A valid name is a string of alphanumerics and underscores. It must begin with an alphabetic (a letter of the alphabet or an underscore). In the last line of the following example, "(salary)" is not valid because '(' is not an alphanumeric.

```
empptr = &anderson;
empptr->salary = 12000;          /* these three lines */
(*empptr).salary = 12000;        /* are */
anderson.salary = 12000;         /* equivalent */
empptr = &anderson.;             /* error 25 */
empptr-> = 12000;                /* error 25 */
anderson.(salary) = 12000;       /* error 25 */
```

26: must be structure/union member

The defined structure or union has no member with the name specified. If the -S option was specified, no previously defined structure or union has such a member either.

Structure members cannot be created at will during a program. Like other variables, they must be fully defined in the appropriate declaration list. Unions provide for variably typed fields, but the full range of desired types must be anticipated in the union declaration.

27: illegal type cast

It is not possible to cast an expression to a function, a structure, or an array. This message may also appear if a syntax error occurs in the expression to be cast.

```
structure sam { ... } thom;
thom = (struct sam)(expression);          /* error 27 */
```

28: incompatible structures

C permits the assignment of one structure to another. The compiler will ensure that the two structures are identical. Both structures must have the same structure tag. For example:

```
struct sam harry;
struct sam thom;
    ...
harry = thom;
```

29: illegal use of structure

Not all operators can accept a structure as an operand. Also, structures cannot be passed as arguments. However, it is possible to take the address of a structure using the ampersand (&), to assign structures, and to reference a member of a structure using the dot operator.

30: missing : in ? conditional expression

The standard syntax for this operator is:

```
expression ? statement1 : statement2
```

It is not desirable to use ?: for extremely complicated expressions; its purpose lies in brevity and clarity.

31: call of non-function

The following represents a function call:

```
symbol(arg1, arg2, ..., argn);
```

where "symbol" is not a reserved word and the expression stands in the body of a function. Error 31, in reference to the expression above, indicates that "symbol" has been previously declared as something other than a function.

A missing operator may also cause this error:

```
a(b + c);                    /* error 31 */
a * (b + c);                 /* intended */
```

The missing '*' makes the compiler view "a()" as a function call.

32: illegal pointer calculation

Pointers may be involved in three calculations. An integral value can be added to or subtracted from a pointer. Pointers to objects of the same type can be subtracted from one another and compared to one another. (For a formal definition, see Kernighan and Ritchie pp. 188-189.) Since the comparison and subtraction of two pointers is dependent upon pointer size, both operands must be the same size.

33: illegal type

The unary minus (-) and bit complement (~) operators cannot be applied to structures, pointers, arrays and functions. There is no reasonable interpretation for the following:

```
int function();
char array[12];
struct sam { ... } harry;
a = -array;              /* ? */
b = -harry;
c = ~function & WRONG;
```

34: undefined symbol

The compiler will recognize only reserved words and names which have been previously defined. This error is often the result of a typographical error or due to an omitted declaration.

35: typedef not allowed here

Symbols which have been defined as types are not allowed within expressions. The exception to this rule is the use of *sizeof(expression)* and the cast operator. Compare the accompanying examples:

```
struct sam {
    int i;
} harry;
typedef double bigfloat;
typedef struct sam foo;

j = 4 * bigfloat f;      /* error 35 */
k = &foo;                /* error 35 */
x = sizeof(bigfloat);
y = sizeof(foo);         /* good */
```

The compiler will detect two errors in this code. In the first assignment, a typecast was probably intended; compare error 8. The second assignment makes reference to the address of a structure type. However, the structure type is just a template for instances of the structure (such as "harry"). It is no more meaningful to take the address of a structure type than any other data type, as in *&int.*

36: no more expression space

This message indicates that the expression table is not large enough for the compiler to process the source code. It is necessary to recompile the file using the -E option to increase the number of available entries in the expression table. See the description of the compiler in the manual.

37: invalid expression

This error occurs in the evaluation of an expression containing a unary operator. The operand either is not given or is itself an invalid expression.

Unary operators take just one operand; they work on just one variable or expression. If the operand is not simply missing, as in the example below, it fails to evaluate to anything its operator can accept. The unary operators are logical not (!), bit complement (~), increment (++), decrement (--), unary minus (-), typecast, pointer-to (*), address-of (&), and sizeof.

```
if (!) ;
```

38: no auto. aggregate initialization

It is not permitted to initialize automatic arrays and structures. Static and external aggregates may be initialized, but by default their members are set to zero.

```
char array[5] = { 'a', 'b', 'c', 'd' };
function()
{
    static struct sam {
        int bone;
        char license[10];
    } harry = {
        1,
        "123-4-1984"
    };
    char autoarray[2] = { 'f', 'g' };    /* no good */
    extern char array[];
}
```

There are three variables in the above example, only two of which are correctly initialized. The variable "array" may be initialized because it is external. Its first four members will be given the characters as shown. The fifth member will be set to zero.

The structure "harry" is static and may be initialized. Notice that "license" cannot be initialized without first giving a value to "bone". There are no provisions in C for setting a value in the middle of an aggregate.

The variable "autoarray" is an automatic array. That is, it is local to a function and it is not declared to be static. Automatic variables reappear automatically every time a function is called, and they are guaranteed to contain garbage. Automatic aggregates cannot be initialized.

39: obsolete [see error 19]

40: internal [see error 4]

41: initializer not a constant

In certain initializations, the expression to the right of the equals sign (=) must be a constant. Indeed, only automatic and register variables may be initialized to an expression. Such initializations are meant as a convenient shorthand to eliminate assignment statements. The initialization of statics and globals actually occurs at link-time, and not at run-time.

```
{
    int i = 3;
    static int j = (2 + i);        /* illegal */
}
```

42: too many initializers

There were more values found in an initialization than array or structure members exist to hold them. Either too many values were specified or there should have been more members declared in the aggregate definition.

In the initialization of a complex data structure, it is possible to enclose the initializer in a single set of braces and simply list the members, separated by commas. If more than one set of braces is used, as in the case of a structure within a structure, the initializer must be entirely braced.

```
struct {
    struct {
        char array[];
    } substruct;
} superstruct =
```

version 1:

```
{
    "abcdefghij"
};
```

version 2:

```
{
    {
        { 'a','b','c',...,'i','j'}
    }
};
```

In version 1, the initializers are copied byte-for-byte onto the structure, *superstruct.*

Another likely source of this error is in the initialization of arrays with strings, as in:

```
char array[10] = "abcdefghij";
```

This will generate error 42 because the string constant on the right is null-terminated. The null terminator (' ' or 0x00) brings the size of the initializer to 11 bytes, which overflows the ten-byte array.

43: undefined structure initialization

An attempt has been made to assign values to a structure which has not yet been defined.

```
struct sam {...};
struct dog sam = { 1, 2, 3};  /* error 43 */
```

44: obsolete [see error 19]

45: bad declaration syntax

This error code is an all purpose means for catching errors in declaration statements. It indicates that the compiler is unable to interpret a word in an external declaration list.

46: missing closing brace

All the braces did not pair up at the end of compilation. If all the preceding code is correct, this message indicates that the final closing brace to a function is missing. However, it can also result from a brace missing from an inner block.

Keep in mind that the compiler accepts or rejects code on the basis of syntax, so that an error is detected only when the rules of grammar are violated. This can be misleading. For example, the program below will generate error 46 at the end even though the human error probably occurred in the *while* loop several lines earlier.

As the code appears here, every statement after the left brace in line 6 belongs to the body of the *while* loop. The compilation error vanishes when a right brace is appended to the end of the program, but the results during run time will be indecipherable because the brace should be placed at the end of the loop.

It is usually best to match braces visually before running the compiler. A C-oriented text editor makes this task easier.

```
main()
{
    int i, j;
    char array[80];

    gets(array);
    i = 0;
    while (array[i])  {
        putchar(array[i]);
        i++;
    for ( i=0; array[i];i++) {
        for (j=i + 1; array[j]; j++)  {
            printf("elements %d and %d are ", i, j);
            if (array[i] == array[j])
                printf("the same\n");
            else
                printf("different\n");
        }
        putchar('\n');
    }
}
```

47: open failure on include file

When a file is *#included*, the compiler will look for it in a default area (see the manual description of the compiler). This message will be generated if the file could not be opened. An open failure usually occurs when the included file does not exist where the compiler is searching for it. Note that a drive specification is allowed in an include statement, but this diminishes flexibility somewhat.

48: illegal symbol name

This message is produced by the preprocessor, which is that part of the compiler which handles lines which begin with a pound sign (#). The source for the error is on such a line. A legal name is a string whose first character is an alphabetic (a letter of the alphabet or an underscore). The succeeding characters may be any combination of alphanumerics (alphabetics and numerals). The following symbols will produce this error code:

```
2nd_time,
dont_do_this!
```

49: multiply defined symbol

This message warns that a symbol has already been declared and that it is illegal to redeclare it. The following is a representative example:

```
int i, j, k, i;          /* illegal */
```

50: missing bracket

This error code is used to indicate the need for a parenthesis, bracket or brace in a variety of circumstances.

51: lvalue required

Only *lvalues* are are allowed to stand on the left-hand side of an assignment. For example:

```
int num;
num = 7;
```

They are distinguished from *rvalues*, which can never stand on the left of an assignment, by the fact that they refer to a unique location in memory where a value can be stored. An *lvalue* may be thought of as a bucket into which an *rvalue* can be dropped. Just as the contents of one bucket can be passed to another, so can an lvalue *y* be assigned to another lvalue, *x*:

```
#define NUMBER  512
x = y;
1024 = z;              /* wrong; l/rvalues are reversed */
NUMBER = x;            /* wrong; NUMBER is still an rvalue */
```

Some operators which require *lvalues* as operands are increment (++), decrement (--), and address-of (&). It is not possible to take the address of a register variable as was attempted in the following example:

```
register int i, j;
i = 3;
j = &i;
```

52: obsolete [see error 19]

53: multiply defined label

On occasions when the goto statement is used, it is important that the specified label be unique. There is no criterion by which the computer can choose between identical labels. If you have trouble finding the duplicate label, use your text editor to search for all occurrences of the string.

54: too many labels

The compiler maintains an internal table of labels which will support up to several dozen labels. Although this table is fixed in size, it should satisfy the requirements of any reasonable C program. C was structured to discourage extravagance in the use of goto's. Strictly speaking, goto statements are not required by any procedure in C; they are primarily recommended as a quick and simple means of exiting from a nested structure.

This error indicates that you should significantly reduce the number of goto's in your program.

55: missing quote

The compiler found a mismatched double quote (") in a *#define* preprocessor command. Unlike brackets, quotes are not paired innermost to outermost, but sequentially. So the first quote is associated with the second, the third with the fourth, and so on. Single quotes (') and double quotes (") are entirely different characters and should not be confused. The latter are used to delimit string constants. A double quote can be included in a string by use of a backslash, as in this example:

```
"this is a string"
"this is a string with an embedded quote: \". "
```

56: missing apostrophe

The compiler found a mismatched single quote or apostrophe (') in a *#define* preprocessor command. Single quotes are paired sequentially (see error 55). Although quotes can not be nested, a quote can be represented in a character constant with a backslash:

```
char c = '\'';          /* c is initialized to
                           single quote  */
```

57: line too long

Lines are restricted in length by the size of the buffer used to hold them. This restriction varies from system to system. However, logical lines can be infinitely long by continuing a line with a backslash-newline sequence. These characters will be ignored.

58: illegal # encountered

The pound sign (#) begins each command for the preprocessor: *#include, #define, #if, #ifdef, #ifndef, #else, #endif, #asm, #endasm, #line and #undef.* These symbols are strictly defined. The pound sign (#) must be in column one and lower case letters are required.

59: macro too long

Macros can be defined with a preprocessor command of the following form:

```
#define [identifier] [substitution text]
```

The compiler then proceeds to replace all instances of "identifier" with the substitution text that was specified by the *#define.*

This error code refers to the substitution text of a macro. Whereas ideally a macro definition may be extended for an arbitrary number of lines by ending each line with a backslash (), for practical purposes the size of a macro has been limited to 255 characters.

60: obsolete [see error 19]

61: reference of member of undefined structure

Occurs only under compilation without the -S option. Consider the following example:

```
int bone;
struct cat {
   int toy;
} manx;
struct dog *samptr;
manx.toy = 1;
bone = samptr->toy;      /* error 61 */
```

This error code appears most often in conjunction with this kind of mistake. It is possible to define a pointer to a structure without having already defined the structure itself. In the example, *samptr* is a structure pointer, but what form that structure ("dog") may take is still unknown. So when reference is made to a member of the structure to which *samptr* points, the compiler replies that it does not even known what the structure looks like.

The *-S* compiler option is provided to duplicate the manner in which earlier versions of UNIX treated structures. Given the example above, it would make the compiler search all previously defined structures for the member in question. In particular, the value of the member "toy" found in the structure "manx" would be assigned to the variable "bone". The -S option is not recommended as a short cut for defining structures.

62: function body must be compound statement

The body of a function must be enclosed by braces, even though it may consist of only one statement:

```
function()
{
   return 1;
}
```

This error can also be caused by an error inside a function declaration list, as in:

```
func(a, b)
int a; chr b;
{
   ...
```

63: undefined label

A *goto* statement is meaningless if the corresponding label does not appear somewhere in the code. The compiler disallows this since it must be able to specify a destination to the computer.

It is not possible to goto a label outside the present function (labels are local to the function in which they appear). Thus, if a label does not exist in the same procedure as its corresponding goto, this message will be generated.

64: inappropriate arguments

When a function is declared (as opposed to defined), it is poor syntax to specify an argument list:

```
function(string)
char *string;
{
    char *func1();              /* correct */
    double func2(x,y);          /* wrong */
    ...
}
```

In this example, function() is being defined, but func1() and func2() are being declared.

65: illegal or missing argument name

The compiler has found an illegal name in a function argument list. An argument name must conform to the same rules as variable names, beginning with an alphabetic (letter or underscore) and continuing with any sequence of alphanumerics and underscores. Names must not coincide with reserved words.

66: expected comma

In an argument list, arguments must be separated by commas.

67: invalid else

An *else* was found which is not associated with an *if* statement. *else* is bound to the nearest *if* at its own level of nesting. So if-else pairings are determined by their relative placement in the code and their grouping by braces.

```
if(...) {
    ...
    if (...) {
        ...
    } else if (...)
        ...
    } else {
        ...
    }
```

The indentation of the source text should indicate the intended structure of the code. Note that the indentation of the if and else-if means only that the programmer wanted both conditionals to be nested at the same level, in particular one step down from the presiding if

statement. But it is the placement of braces that determines this for the compiler. The example above is correct, but probably does not conform to the expectations revealed by the indentation of the else statement. As shown here, the else is paired with the first if, not the second.

68: syntax error

The keywords used in declaring a variable, which specify storage class and data type, must not appear in an executable statement. In particular, all local declarations must appear at the beginning of a block, that is, directly following the left brace which delimits the body of a loop, conditional or function. Once the compiler has reached a non-declaration, a keyword such as *char* or *int* must not lead a statement; compare the use of the casting operator:

```
func()
{
    int i;
    char array[12];
    float k = 2.03;

    i = 0;
    int m;                          /* error 68 */
    j = i + 5;
    i = (int) k;                    /* correct */
    if (i) {
        int i = 3;
        j = i;
        printf("%d",i);
    }
    printf("%d%d\n",i,j);
}
```

This trivial function prints the values 3, 2 and 3. The variable i which is declared in the body of the conditional (if) lives only until the next right brace; then it dies, and the original i regains its identity.

69: missing semicolon

A semicolon is missing from the end of an executable statement. This error code is subject to the same vagaries as its cousin, error 7. It will remain undetected until the following line and is often spuriously caused by a previous error.

70: bad goto syntax

Compare your use of goto with an example. This message says that you did not specify where you wanted to goto with a label:

```
    goto label;
    ...
label:
    ...
```

It is not possible to goto just any identifier in the source code; labels are special because they are followed by a colon.

71: statement syntax error in do-while

The body of a *do-while* may consist of one statement or several statements enclosed in braces. A *while* conditional is required after the body of the loop. This is true even if the loop is infinite, as it is required by the rules of syntax. After typing in a long body, don't forget the *while* conditional.

72: 'for' syntax error: missing first semicolon

This error focuses on another control flow statement, the *for*. The keyword, *for*, must be followed by parentheses. In the parentheses belong three expressions, any or all of which may be null. For the sake of clarity, C requires that the two semicolons which separate the expressions be retained, even if all three expressions are empty.

```
for (;                  /* an infinite loop which does */
    ;                   /* absolutely nothing */
```

Error 72 signifies that the compiler didn't find the first semicolon within the parentheses.

73: 'for' syntax error: missing second semicolon

This error is similar to error 72; it means that the compiler didn't find the second semicolon within the parenthesized expression following the 'for'.

74: case value must be integer constant

Strictly speaking, each value in a *case* statement must be a constant of one of three types: *char*, *int* or *unsigned*. This is similar to the rule for a *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.

77: case outside of switch

The keyword, *case*, belongs to just one syntactic structure, the *switch*. If "case" appears outside the braces which contain a switch statement, this error is generated. Remember that all keywords are reserved, so that they cannot be used as variable names.

78: missing colon

This message indicates that a colon is missing after the keyword, *default*. Compare error 75.

79: duplicate default

The compiler has found more than one *default* in a *switch*. Switch will compare a variable to a given list of values. But it is not always possible to anticipate the full range of values which the variable may take. Nor is it feasible to specify a large number of cases in which the program is not particularly interested.

So C provides for a default case. The default will handle all those values not specified by a case statement. It is analogous to the else companion to the conditional, if. Just as there is one else for every if, only one default case is allowed in a switch statement. However, unlike the else statement, the position of a default is not crucial; a default can appear anywhere in a list of cases.

80: default outside of switch

The keyword, *default*, is used just like *case*. It must appear within the brackets which delimit the switch statement.

81: break/continue error

Break and continue are used to skip the remainder of a loop in order to exit or repeat the loop. Break will also end a switch statement. But when the keywords, *break* or *continue*, are used outside of these contexts, this message results.

82: illegal character

Some characters simply do not make sense in a C program, such as '$' and '@'. Others, for instance the pound sign (#), may be valid only in particular contexts.

83: too many nested includes

#includes can be nested, but this capacity is limited. The compiler will balk if required to descend more than three levels into a nest. In the example given, file D is not allowed to have a #include in the compilation of file A.

```
file A          file B          file C          file D
#include "B"    #include "C"    #include "D"
```

84: too many array dimensions

An array is declared with too many dimensions. This error should appear in conjunction with error 11.

85: not an argument

The compiler has found a name in the declaration list that was not in the argument list. Only the converse case is valid, i.e., an argument can be passed and not subsequently declared.

86: null dimension in array

In certain cases, the compiler knows how to treat multidimensional arrays whose left-most dimensions are not given in its declaration. Specifically, this is true for an extern declaration and an array initialization. The value of any dimension which is not the left-most must be given.

```
extern char array[][12];        /* correct */
extern char badarray[5][];      /* wrong */
```

87: invalid character constant

Character constants may consist of one or two characters enclosed in single quotes, as 'a' or 'ab'. There is no analog to a null string, so '' (two single quotes with no intervening white space) is not allowed. Recall that the special backslash characters (\b, \n, \t etc.) are singular,

so that the following are valid: '\n', '\na', 'a\n'; 'aaa' is invalid.

88: not a structure

Occurs only under compilation without the -S option. A name used as a structure does not refer to a structure, but to some other data type.

```
int i;
i.member = 3;          /* error 88 */
```

89: invalid storage class

A globally defined variable cannot be specified as register. Register variables are required to be local.

90: symbol redeclared

A function argument has been declared more than once.

91: illegal use of floating point type

Floating point numbers can be negated (unary minus), added, subtracted, multiplied, divided and compared; any other operator will produce this error message.

92: illegal type conversion

This error code indicates that a data type conversion, implicit in the code, is not allowed, as in the following piece of code:

```
int i;
float j;
char *ptr;
...
i = j + ptr;
```

The diagram shows how variables are converted to different types in the evaluation of expressions. Initially, variables of type *char* and *short* become *int*, and *float* becomes *double*. Then all variables are promoted to the highest type present in the expression. The result of the expression will have this type also. Thus, an expression containing a *float* will evaluate to a *double*.

hierarchy of types:

```
double <-- float
long
unsigned
int <-- short, char
```

This error can also be caused by an attempt to return a structure, since the structure is being cast to the type of the function, as in:

```
int func()
{
   struct tag sam;
   return sam;
}
```

93: illegal expression type for switch

Only a *char*, *int* or *unsigned* variable can be switched. See the example for error 74.

94: bad argument to define

An illegal name was used for an argument in the definition of a macro. For a description of legal names, see error 65.

95: no argument list

When a macro is defined with arguments, any invocation of that macro is expected to have arguments of corresponding form. This error code is generated when no parenthesized argument list was found in a macro reference.

```
#define  getchar()  getc(stdin)
...
c = getchar;                    /* error 95 */
```

96: missing argument to macro

Not enough arguments were found in an invocation of a macro. Specifically, a "double comma" will produce this error:

```
#define reverse(x,y,z)  (z,y,x)
func(reverse(i,,k));
```

97: obsolete [see error 19]

98: not enough args in macro reference

The incorrect number of arguments was found in an invocation of a previously defined macro. As the examples show, this error is not identical to error 96.

```
#define  exchange(x,y)  (y,x)
func(exchange(i));              /* error 98 */
```

99: internal [see error 4]

100: internal [see error 4]

101: missing close parenthesis on macro reference

A right (closing) parenthesis is expected in a macro reference with arguments. In a sense, this is the complement of error 95; a macro argument list is checked for both a beginning and an ending.

102: macro arguments too long

The combined length of a macro's arguments is limited. This error can be resolved by simply shortening the arguments with which the macro is invoked.

103: #else with no #if

Correspondence between #if and #else is analogous to that which exists between the control flow statements, if and else. Obviously, much depends upon the relative placement of the statements in the code. However, #if blocks must always be terminated by #endif, and the #else statement must be included in the block of the #if with which it is associated. For example:

```
#if ERROR > 0
   printf("there was an error\n");
#else
   printf("no error this time\n");
#endif
```

#if statements can be nested, as below. The range of each #if is determined by a #endif. This also excludes #else from #if blocks to which it does not belong:

```
#ifdef   JAN1
   printf("happy new year!\n");
#if      sick
   printf("i think i'll go home now\n");
#else
   printf("i think i'll have another\n");
#endif
#else
   printf("i wonder what day it is\n");
#endif
```

If the first #endif was missing, error 103 would result. And without the second #endif, the compiler would generate error 107.

104: #endif with no #if

#endif is paired with the nearest #if, #ifdef or #ifndef which precedes it. (See error 103.)

105: #endasm with no #asm

#endasm must appear after an associated #asm. These compiler-control lines are used to begin and end embedded assembly code. This error code indicates that the compiler has reached a #endasm without having found a previous #asm. If the #asm was simply missing, the error list should begin with the assembly code (which are undefined symbols to the compiler).

106: #asm within #asm block

There is no meaningful sense in which in-line assembly code can be nested, so the #asm keyword must not appear between a paired #asm/#endasm. When a piece of in-line assembly is augmented for temporary purposes, the old #asm and #endasm can be enclosed in comments as place-holders.

```
#asm
   /* temporary asm code */
/*  #asm          old beginning  */
   /* more asm code */
#endasm
```

107: missing #endif

A #endif is required for every #if, #ifdef and #ifndef, even if the entire source file is subject to a single conditional compilation. Try to assign pairs beginning with the first #endif. Backtrack to the previous #if and form the pair. Assign the next #endif with the nearest unpaired #if. When this process becomes greatly complicated, you might consider rethinking the logic of your program.

108: missing #endasm

In-line assembly code must be terminated by a #endasm in all cases. #asm must always be paired with a #endasm.

109: #if value must be integer constant

#if requires an integral constant expression. This allows both integer and character constants, the arithmetic operators, bitwise operators, the unary minus (-) and bit complement, and comparison tests.

Assuming all the macro constants (in capitals) are integers,

```
#if DIFF >= 'A'-'a'
#if (WORD &= ~MASK) >> 8
#if MAR | APR | MAY
```

are all legal expressions for use with #if.

110: invalid use of colon operator

The colon operator occurs in two places: 1. following a question mark as part of a conditional, as in (flag ? 1 : 0); 2. following a label inserted by the programmer or following one of the reserved labels, *case* and *default.*

111: illegal use of a void expression

This error can be caused by assigning a *void* expression to a variable, as in this example:

```
void func();
int h;

h = func(arg);
```

112: illegal use of function pointer

For example,

```
int (*funcptr) ();
   ...
funcptr++;
```

funcptr is a pointer to a function which returns an integer. Although it is like other pointers in that it contains the address of its object, it is not suject to the rules of pointer arithmetic. Otherwise, the offending statement in the example would be interpreted as adding to the pointer the size of the function, which is not a defined value.

113: duplicate case in switch

This simply means that, in a *switch* statement, there are two *case* values which are the same. Either the two *case*s must be combined into one, or one of them must be discarded. For instance:

```
switch (c) {
case NOOP:
   return (0);
case MULT:
   return (x * y);
case DIV:
   return (x / y);
case ADD:
   return (x + y);
case NOOP:
default:
   return;
}
```

The case of NOOP is duplicated, and will generate an error.

114: macro redefined

For example,

```
#define   islow(n)  (n>=0&&n<5)
   ...
#define   islow(n)  (n>=0&&n<=5)
```

The macro, *islow*, is being used to classify a numerical value. When a second definition of it is found, the compiler will compare the new substitution string with the previous one. If they are found to be different, the second definition will become current, and this error code will be produced.

In the example, the second definition differs from the first in a single character, '='. The second definition is also different from this one:

```
#define  islow(n)  n>=0&&n<=5
```

since the parentheses are missing.

The following lines will not generate this error:

```
#define  NULL 0
  ...
#define  NULL 0
```

But these are different from:

```
#define  NULL ' '
```

In practice, this error message does not affect the compilation of the source code. The most recent "revision" of the substitution string is used for the macro. But relying upon this fact may not be a wise habit.

115: keyword redefined

Keywords cannot be defined as macros, as in:

```
#define  int  foo
```

If you have a variable which may be either, for instance, a short or a long integer, there are alternative methods for switching between the two. If you want to compile the variable as either type of integer, consider the following:

```
#ifdef   LONGINT
   long i;
#else
   short i;
#endif
```

Another possibility is through a *typedef*:

```
#ifdef   LONGINT
   typedef  long    VARTYPE;
#else
   typedef  short   VARTYPE;
#endif

VARTYPE i;
```

116: field width must be > 0

A field in a bit field structure can't have a negative number of bits.

117: invalid 0 length field

A field in a bit field structure can't have zero bits.

118: field is too wide

A field in a bit field structure can't have more than 16 bits.

119: field not allowed here

A bit field definition can only be contained in a structure.

120: invalid type for field

The type of a bit field can only be of type *int* of *unsigned int.*

121: ptr/int conversion

The compiler issues this warning message if it must implicitly convert the type of an expression from pointer to *int* or *long*, or vice versa.

If the program explicitly casts a pointer to an *int* this message won't be issued. However, in this case, error 122 may occur.

For example, the following will generate warning 121:

```
char *cp;
int i;
...
i = cp; /* implicit conversion of char * to int */
```

When the compiler issues warning 121, it will generate correct code if the sizes of the two items are the same.

122: ptr & int not same size

If a program explicitly casts a pointer to an *int*, and the sizes of the two items differ, the compiler will issue this warning message. The code that's generated when the converted pointer is used in an expression will use only as much of the least significant part of the pointer as will fit in an *int.*

123: function ptr & ptr not same size

If a program explicitly casts a pointer to a data item to be a pointer to a function, or vice versa, and the sizes of the two pointers differ, the compiler issues this warning message.

If the program doesn't explicitly request the conversion, warning 124 will be issued instead of warning 123.

124: invalid ptr/ptr assignment

If a program attempts to assign one pointer to another without explicitly casting the two pointers to be of the same type, and the types of the two pointers are in fact different, the compiler will issue this warning message.

The compiler will generate code for the assignment, and if the sizes of the two pointers are the same, the code will be correct. But if the

sizes differ, the code may not be correct.

125: too many subscripts or indirection on integer

This warning message is issued if a program attempts to use an integer as a pointer; that is, as the operand of a star operator.

If the sizes of a pointer and an *int* are the same, the generated code will access the correct memory location, but if they don't, it won't.

For example,

```
char c;
long g;
*0x5c=0; /* warning 125, because 0x5c is an int */
c[i]=0;  /* warning 125, because c+i is an int */
g[i]=0;  /* error 12, because g+i is a long */
```

3. Fatal Compiler Error Messages

If the compiler encouters a "fatal" error, one which makes further operation impossible, it will send a message to the screen and end the compilation immediately.

Out of disk space!

There is no room on the disk for the output file of the compiler. Previous disk files will not be overwritten by the compiler's assembly language output. To make room on the disk, it is usually sufficient to remove unneeded files from the disk.

unknown option:

The compiler has been invoked with an option letter which it does not recognize. The manual explicitly states which options the compiler will accept. The compiler will specify the invalid option letter.

duplicate output file

If an output file name has been specified with the -o option and that file already exists on the disk, the compiler will not overwrite it. -O must specify a new file.

too few arguments for -o option

The compiler expected to find the output filename following the "-o", but didn't find it. The output file name must follow the option letter and the name of the file to be compiled must occur last in the command line.

Open failure on input

The input file specified in the command line does not exist on the disk or cannot be opened. A path or drive specification can be included with a filename according to the operating system in use.

No input!

While the compiler was able to open the input file given in the command line, that file was found to be empty.

Open failure on output

The compiler was unable to create an output file. On some systems, this error could occur if a disk's directory is full.

Local table full! (use -L)

The compiler maintains an internal table of the local variables in the source code. If the number of local symbols in use exceeds the available entries in the table at any time during compilation, the compiler will print this message and quit. The default size of the local symbol table (40 entries) can be changed with the -L option for the

compiler. Local variables are those defined within braces, i.e., in a function body or in a compound statement. The scope of a local variable is the body in which it is defined, that is, it is defined until the next right brace at its own nesting level.

Out of memory!

Since the compiler must maintain various tables in memory as well as manipulate source code, it may run out of memory during operation. The more immediate solution is to vary the sizes of the internal tables using the appropriate compiler options. Often, a compilation will require fewer than the default number of entries in a particular table. By reducing the size of that table, memory space is freed up during compile time. The amount of memory used while compiling does not affect the size or content of the assembly or object file output. If this stategy fails to squeeze the compilation into the available memory, the only solution is to divide the source file into modules which can be compiled separately. These modules can then be linked together to produce a single executable file.

Aztec C65 for the Apple // ProDOS
Version 3.2b
Release Document, ed 2
15 July 1986

This release document introduces the features of Aztec C65, v 3.2b, for the Apple //. It's divided into the following sections:

1. *Machine Requirements*;
2. *Known Bugs*;
3. *Packaging*
4. *Technical Support*

If changes have been made to Aztec C65 since this release document was printed, information about the changes will be in a file named *read.me* on the first distribution disk.

1. Machine Requirements

Using Aztec C65, programs can be developed on the Apple // machines whose configurations are listed below.

Created programs will run on machines having these configurations. They may also run, with some limitations, on machines having other configurations; for example, on an Apple // Plus in 80 column mode, a program may be able to write simple text to the console, but may not be able to use the Aztec C65 screen functions.

Supported configurations:

* Apple //c, in either 40- or 80-column mode.
* Apple //e, in 40-column mode or, if Apple's 80-column card is installed, in 80-column mode.
* Apple // or // Plus in 40-column mode that meets the following requirements: (1) it must support keyboard entry of lower case characters (this can be provided by installing the single wire shift key mod); (2) it must support console display of the full set of displayable ASCII characters (this can be provided by installing a lower-case adaptor).

Development must be done under ProDOS on an Apple that has at least two disk drives.

2. Known bugs

The following bugs are known to exist in Aztec C65, v3.2b:

* The *convert* utility doesn't correctly transfer an overlay file from ProDOS to DOS 3.3. If the type of the overlay file on ProDOS is TXT, which is the type given to it by the linker, the file can't be read under DOS 3.3 by the overlay loader. And if the file is of another type, *convert* adds four bytes onto the front of the file, as if it was a standard program.
* On old Apple 2e's, typing the ESC key will put the console in escape mode, rather than sending an escape character to the program.
* There are some bugs involving bit fields. In particular,
    + Arithmetic operations on bit fields don't work, except for operations of the form *op*=;
    + Assignment of a conditional expression's value to a bit field gives error 202. For example:

```
struct unit {
        unsigned a1:1;
        unsigned a2:1;
} *ur;
int a;

main()
{
        ur->a1 = ((a == 1) ? 0:1);
}
```

3. Product Packaging

The following paragraphs describe the contents of the Aztec C65 disks.

The /SYSTEM Disk

PRODOS	ProDOS
FILER	ProDOS Filer
SHELL.SYSTEM	the SHELL
VED	Text Editor
VED.ARC	Text Editor: source archive
EXMPL.C	Sample C source program
CRC	File checking utility

The /CC Disk

CC	6502/65C02 Compiler
AS	6502/65C02 Assembler
CCI	Pseudo Code Compiler
ASI	Pseudo Code Assembler
-INCLUDE	Directory of #include files

The /LN Disk

LN	The linker
C.LIB	Library of non-floating point functions (*cc* compiled)
CI.LIB	Same as C.LIB, but compiled with *cci*
SAMAIN.O	Standalone program's startup routine
TMPDEV.O	Stripped-down console driver

The /UTIL Disk

CNM	Object module lister
LB	Object module librarian
OBD	Object module dumper
ORD	Object module sorter
SQZ	Object module squeezer
TTY	Terminal emulater
TTY.ARC	TTY source archive

The /UTIL2 Disk

ARCV	Source dearchiver
CMP	File comparator
CONFIG	Device configurator
DIFF	Source file comparator
GREP	Pattern finder
HD	Hex dump utility
MKARCV	Source archive generator

The /LIB Disk

G.LIB	Graphics functions (*cc*-compiled)
GI.LIB	Graphics functions (*cci*-compiled)
M.LIB	Floating point functions (*cc*-compiled)
MI.LIB	Floating point functions (*cci*-compiled)
S.LIB	Screen functions (*cc*-compiled)
SI.LIB	Screen functions (*cci*-compiled)
FLT.ARC	Floating point functions source archive
G.ARC	Graphics functions source archive
S.ARC	Screen functions source archive
OVLY.ARC	Overlay functions

The /DOS33 Disk

D.LIB	DOS 3.3 functions (*cc*-compiled)
DI.LIB	DOS 3.3 functions (*cci*-compiled)
D33ROOTA.O	Special startup routine for DOS 3.3 programs
DOS33.ARC	DOS 3.3 functions source archive
CONVERT	ProDOS-to-DOS 3.3 conversion utility

The /ARCV Disk

CONFIG.ARC	*config* source archive
DEV.ARC	Device driver source
MCH65.ARC	6502 functions
MISC.ARC	Miscellaneous functions
PRODOS.ARC	ProDOS interface functions
STDIO.ARC	Standard I/O functions
TIME.ARC	Time functions

4. Technical support information

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

Technical Support Information

Dear User of Aztec C,

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

These are the guidelines...

Have everything with you.

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

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

Know what questions you wish to ask.

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

Isolate the code that caused the problem.

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

Use your C language book and technical manuals first.

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

When to expect an answer.

A normal turn around time for a question is anywhere from the 2 minutes to 24 hours, depending on the nature of the question. A few questions like tracing compiler bugs may take a little longer. If you can call us back the next day, or when the person you talk to in technical support recommends, we will have an indepth answer for you. But normally we can answer your questions immediately.

Utilize our mail-in service.

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

Updates, Availability, Prices.

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

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

Bulletin board system.

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

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

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

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

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

Phone support, number and hours.

And finally, technical support for Aztec C is available between 9:00 am and 6:00 pm eastern standard time at 1-(201)-542-1795. Phone support is available to registered users of Aztec C with the exception of the Apprentice C and C Prime products. For those products, please use the mail-in support service and send questions/problems to P.O. Box 8, Shrewsbury, N.J. 07701.

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

Manx Software Systems
Technical Support Dept.

Date:_____/_____/_____

Name:_____

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

Company :_____

Address :_____

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

VERSION #:_____ Serial #:_____

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

Send this form to :

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

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

or call tech support at 1-201-542-1795 between 9am - 6pm EST.
(Sorry, phone support not available for the C Prime/Apprentice C product.)

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