Aztec C68K, Version 3.4 for the Macintosh Release Document

May 1987

This release document introduces the features of Aztec C68K, version 3.4, for the Macintosh and is divided into the following sections:

1. Product Description
2. New Users
3. Features
4. Changes (All Releases)
5. Packaging
6. Additional Documentation

1. Product Description

Aztec C68K, version 3.4, consists of software and a manual for developing programs in the C language that will run on the Macintosh.

To acquaint yourself with Aztec C68K, we recommend that you finish reading this release document and then read the Overview and Tutorial chapters of the Aztec C68K manual.

There are three Aztec C68K systems for the Macintosh: *Commercial*, *Developer*, and *Professional*. Each system's features are a subset of the next higher system's features.

The manual and documentation that is appended to this release document describes the *Commercial* system's features. If you have the *Professional* or *Developer* system and decide later to upgrade to the *Commercial* system, we'll just send you disks and you'll be ready to go!

1.1.1 THREE SYSTEMS

1.1.1 The *Professional* System

The *Professional* system contains the basics needed to develop C and/or assembly language programs for the Macintosh. It consists of the compiler, assembler, linker, libraries, and header files. In addition, there are a number of example programs.

1.1.2 The *Developer* System

The *Developer* system contains everything found in the *Professional* system with the following additions:

* Utility programs *make*, *grep*, *diff*, *obd*, and *ord*
* Special math support libraries for the 68881, the Manx IEEE emulation, and Standard Apple Numeric Environment (SANE)
* powerful and symbolic debugger, DB
* Z program editor and ctags

1.1.3 The *Commercial* System

The *Commercial* system contains everything found in the *Developer* system with the following additions:

* source to all library functions provided with the Aztec C68K
* one year of *free* updates.

1.2 README

Please check the disks to see if there is a *README* on them. This file (if there is one) contains important information that was added after the manual was printed.

2. New Users

The best way to acquaint yourself with our package is to go through the tutorial on the SHELL by walking through some of the commands. The next sections you should read in the manual are the ones on the SHELL, compiler, assembler, and the linker that describe in more detail what these programs do and what options are available. You should also read the section on style to help you with C programming.

This release document serves several purposes. The Features section describes the latest C68K system features and the Changes section describes the enhancements made to the system since the last release, including bug fixes. The Packaging section lists the contents of the disks that are included with this release. Finally, the Additional Documentation section contains the new or changed documents included with the release, briefly describes their contents, and suggests where you should file them in your manual.

3. Features

The following is a brief summary of the new features and changes found in the 3.4 version. Full details are contained in an appendix to the appropriate section of the manual in the "Additional Documentation" section of this release document.

3.1 NEW COMPILER FEATURES

The following enhancements and changes are made to the compiler:

* The compiler implements and supports the *32 bit int* option. Code generated using 32 bit ints is now much smaller.
* The compiler supports three different floating point formats: Standard Apple Numeric Environment (SANE), IEEE double precision emulation, and Motorola 68881 coprocessor support. Use of floating point numbers requires linking in one of the math libraries, therefore, this release adds new options to accomplish this.
* There are some new preprocessor manifests. In particular, the compiler always defines AZTEC_C while it defines the names __LARGE_CODE and __LARGE_DATA when the appropriate

option is given. (This release includes a detailed description of memory modules, the differences between large and small data and code, and how to generate libraries using the *make* file in the "Additional Documentation" section. This description is in an appendix to the Technical Information section and should be filed at the end of that section in your manual.)

* The compiler correctly handles structure arguments and return values.
* The compiler supports enumerated data types (enum).
* The compiler supports bit fields.
* The INCLUDE environment variable now supports multiple directories by separating names with ';'.
* The compiler attaches *leading* underscores to filenames, rather than *appending* underscores.
* The compiler adds a *Void* data type.
* Variable name length increases from 8 to 31 significant characters.

3.2 NEW ASSEMBLER FEATURES

The following enhancements and changes are made to the assembler.

* After partial redesign, the assembler also provides full support for the 68010, 68020, and 68881.
* The assembler squeeze algorithm is rewritten and is now much faster on large files. The new algorithm is not recursive, so less stack is required.
* The 3.4 version adds several new directives. These reflect the assembler redesign and the change in the way that floating point numbers are returned.

3.3 NEW LINKER FEATURES

The following enhancements and changes are made to the linker:

* System Dependent and System Independent options require prefixes of "+" and "-", respectively.
* The object format change necessitates recompiling and reassembling ALL object modules that are used with the version 3.4 linker and the version 3.4 libraries.
* The linker automatically adds an ".o" extension to files that have no extension. It also checks the current directory and all

directories defined in the CLIB environment variable. Therefore, if you wish to link with *mixcroot.o*, give the name, and the linker checks the current directory and all the CLIB directories.

* The CLIB environment variable supports multiple entries separated by ';'.
* A new linking process decreases link time significantly.

3.4 NEW Z FEATURES

The following enhancements and changes are made to the Z text editor.

* When Z is started, users may give a tag name or line number as an argument.
* A new command, *:fn*, searches a *funclist* file and displays the line containing the keyword.
* A new flag, *ak*, allows users to move the cursor via the keyboard arrow keys.
* A new flag, *sm*, indicates whether or not macros should perform their operation silently. If nonzero, a macro performs all iterations and redisplays the screen.
* During insert, ^*W* deletes the previous word.
* When activating files from the shell, users may specify up to 30 files instead of 10.

3.5 NEW UTILITY PROGRAMS

This section describes additions, enhancements, replacements to, and removals from, the Utilities supplied with Aztec C68K. See "Additional Documentation" in this release for details.

The following utility functions are new:

prof — The new profiler report program (*prof*) is an optimizing tool that determines the percentage of program run time spent in a function. *prof* is described in detail in the appendix contained in the "Additional Documentation" section of this release.

mon — Because it plays a significant role in successfully running the *prof* program, the monitor function is also described in detail in the appropriate appendix contained in the "Additional Documentation" of this release.

obd This new function lists the loader items in an object file. See "Additional Documentation" in this release letter for a copy of the *obd* command page.

The following utility replaces an existing one:

lb The object module librarian *lb* replaces the *libutil* utility. Manual pages describing *lb* are included in the appropriate appendix contained in the "Additional Documentation" section of this release.

The following utilities are changed:

diff *diff* now handles boundary conditions correctly. Therefore, the message that *diff* cannot synchronize the change record should not occur as often.

grep The -f options work correctly.

ord Problem no longer exists when *ord* is given object code that does not have global references.

cnm Because of the new object format, *cnm* now displays the size and symbols of its object file arguments. See "Additional Documentation" in this release letter for a copy of the *cnm* command page.

hd *hd* and *cmp* each check the alternate of the data and resource forks if the first fork checked is empty.

The following utilities are removed:

hfs convert Disks shipped are now double sided HFS format.

FixAttr Use *ResEdit* if you wish to set the console driver and Monaco 9- and 12-point fonts so they would be loaded into the system heap at boot time.

SetStartup To make the shell the startup application instead of the finder, click on the shell from the finder and select the SetStartup item from the special menu.

fldr Under *hfs* real folders exist and can be created from within the shell by using *mkdir*.

MountRam The optional argument (size) of the MountRam command represents the size in kilobytes of the desired RamDisk. If no argument is given, the default size is all the available RAM minus 128K. However, be sure to allow 150K for *db* plus the size of the program to be free when setting up the RamDisk. Also, modifications require that MountRam be mounted as drive number 21 instead

of 5. If you do not wish 21 as the drive number for the RamDisk, modify the variable DRVNUM in ramdisk.asm and use *make* to recompile it.

3.6 NEW LIBRARY FEATURES

This release contains two sets of libraries--one, the new MPW compatible libraries; and the other, the existing Aztec C compatible libraries. The "Additional Documentation" section of this release contains two appendices to the "Technical Information" section: One appendix discusses the differences that now exist and how to work with the libraries. The second describes the memory modules and their organization in detail and also describes how to generate libraries using the *make* file.

Note: Because of object format changes since the 106i release, you must recompile all your source to link with the current libraries and compiler generated code.

In the Apple-compatible MPW libraries, Macintosh Toolbox routines are converted to Pascal strings for you, thus eliminating the dilemma of deciding when to pass Pascal or C strings to Toolbox calls.

The following items distinguish these libraries from previous releases:

* Object module format changed.
* Variable name length increased.
* Underscores appended at beginning of identifiers instead of at end.

Differences also exist between the Aztec compatible header files and the MPW compatible header files. Therefore, the libraries and their respective header files are on separate disks. As stated, this release contains appendices that describe how to work with the libraries.

In this release, example programs show both old and new formats. However, we are considering supporting one or the other of the formats in future releases. Please let us know whether you have a preference and what it would be!

3.6.1 Using the MPW Compatible Libraries.

To use old source with the new libraries, users should:

* Include the relevant .h files for the initialization calls (i.e., #include <Quickdraw.h> when using InitGraf, InitFont, or InitCursor)

* Change calling conventions to several routines, as follows:

Old	New
InitGraf(&thePort);	InitGraf(&qd.thePort);
SetArrow(&arrow);	SetArrow(&qd.arrow);
SetCursor(&arrow);	SetCursor(&qd.arrow);

* Use the address of type Point variables to pass them, rather than *pass()*. For example, in the new libraries the call would be:

```
FindWindow( &mvEvent.where, &whichWindow);
```

instead of:

```
FindWindow( pass(mvEvent.where), &whichWindow);
```

Note: If you are using *pass()* for other than Toolbox calls, using the & operator will give the address, NOT the value as *pass()* did.

* In the SFReply structure .fName is no longer declared char *, but is a structure. Therefore, prefix it with &, such as &variable.fName.

* Most new glue routines expect C strings of type char *, while the old routines expect Pascal strings of type Str255. Therefore, remove unnecessary ctop() and/or ptoc() calls and remove the leading \P from string constants.

* Several header files are renamed, as follows:

Old	New
control.h	Controls.h
dialog.h	Dialogs.h
disk.h	Disks.h
event.h	Events.h
font.h	Fonts.h
inits.h	appropriate .h for init routines
list.h	Lists.h
menu.h	Menus.h
osutil.h	OSUtils.h
pb.h	Files.h, Devices.h, and OsEvents.h
print.h	Printing.h
resource.h	Resources.h
segment.h	SegLoad.h
syserr.h	Errors.h
toolutil.h	ToolUtils.h and FixMath.h
window.h	Windows.h

* Structure definitions in pb.h in many cases don't match the new structure definitions given in Files.h.

* The header file Graf3D.h is supplied without Graf3D routines in the library. Users who have a Graf3D library in the MDS object format may access the routines from within a C program.

3.6.2 Compiling Using the New Libraries.

In MPW C source, ints are 32 bits; in Aztec C they are 16 bits. Users may compile with the +L option to generate code for 32-bit ints, but no libraries currently support Macintosh Toolbox calls with 32-bit integers. The 32-bit libraries are only useable for standard I/O calls but not for Toolbox calls.

3.6.3 Source Recompilation Required.

Note: Because of object format changes since the 106i release, you must recompile all your source to link with current libraries and compiler generated code.

3.6.4 SANE Library Support

Support for SANE is provided with ms.lib and is available in both library formats.

The Manx-supplied IEEE library in m.lib is faster than the SANE equivalents, but makes your program slightly larger because Manx uses RAM while SANE is in ROM.

With this release, type extended is not supported directly by the compiler but is defined in Sane.h to be type double, which is 64 bits long. This means that the SANE routines, although using the 80-bit extended type, convert the results back to type double and, therefore, are no more accurate than the Manx-supplied IEEE implementation. Direct compiler support for the 80-bit extended type won't be added until the next release.

3.7 BUGS

Several bugs were fixed in the libraries.

3.8 DEBUGGER

This release contains full documentation on *db*. The following features are included in the 3.4 release of *db* (Note: At the present time *db* does not work on the Mac II):

* Leading underscores and 31 character flexnames supported.
* Breakpoints corrected on *trapname* for *g* and *bs*.
* Problem corrected that prevented screen output when loading with *lp* after running the first program loaded to completion.
* OpenResFile correctly executed for a filename on the current directory without requiring the full pathname.

* *vc* and *vd* SYMBOL = ADDR works. Users must specify type of symbol (e.g. *c* = code or *d* data).

  Note: Two problems still exist--at present, users may not use *db* on the new Mac II, and may not single step through ROM traps.

3.9 MACSBUG OR TMON

If you use MacsBug or TMON to debug your code, you may enter the debugger in two ways. As mentioned in the MacsBug command page in your manual, a call to TickCount has been placed at the beginning of the various Croot() routines to start an application. To gain access immediately upon entering the application, enter the debugger from the shell and set a breakpoint for calls to the trap for TickCount as follows:

In MacsBug, to set a breakpoint call to the trap for TickCount, type:

```
AB 175
```

In TMON, enter the user area and set the following trap intercept:

```
Trap intercept (t0 [t1 [PC0 Pc1]]):_TickCount
```

To enter either debugger from within program, use the following example to execute the Debugger trap:

```
pascal void dbg() = 0xa9ff;

main()
{
   dbg();
}
```

You may wish to use the *Compiler* option *+N* to embed MacsBug-readable function names in your code, or the *Linker* option *+T* option to create a TMON-readable .map file.

3.10 TOOLBOX CALLS

We are unable at this time to include documentation on complete Toolbox calls and their calling conventions. We plan to include up-to-date Toolbox calls in the next release. However, in the meantime check your disk for most current titles. For the most part, the calls that are currently listed in the Toolbox section of the manual may still be used.

4. Changes from Previous Releases

4.1 CHANGES SINCE RELEASE 1.06i

Both bug fixes and enhancements will be listed by program file.

4.1.1. SHELL.

a) Supports the new MPW compatible libraries, in addition to supporting the the existing Aztec C compatible libraries.

Note: Because of object format changes since the 106i release, you must recompile all your source to link with the current libraries and compiler generated code.

4.1.2 CC.

a) Supports three different floating point formats: Standard Apple Numeric Environment (SANE), IEEE double precision emulation, and Motorola 68881 coprocessor support.

4.1.3. AS.

a) Changes squeeze algorithm to be nonrecursive and to dynamically allocate the squeeze table. Change also makes algorithm much faster on large files.

4.1.4. LN.

a) The CLIB environment variable supports multiple entries separated by ';'.

b) The *+T* option generates a .map file that may be read and used by TMON to view symbols as code resource relative.

4.1.5. Z.

a) Added *:fn* which searches a *funclist* file and displays the line containing the keyword.

b) Added a new flag, *sm*, which enables users to disable Display during macro execution.

c) Added a new flag, *ak*, which allows users to move the cursor via the keyboard arrow keys.

4.1.6. C.LIB.

a) Added the following new glue routines to *c.lib*:

```
pascal void PStr2Dec(s,index,d,validPrefix)
  Str255 *s; int *index; decimal *d; Boolean *validPrefix;

pascal void CStr2Dec(s,index,d,validPrefix)
  char *s; int *index; decimal *d; Boolean *validPrefix;
```

```
pascal void Dec2Str(f,d,s)
  decform *f; decimal *d; Str255*s; /* char *s in MPW
compatible version */

pascal void DrawlControl (theControl)
  ControlHandle theControl;

pascal void ScreenRes (scrnHRes, scrnVRes)
  short *screnHRes, *scrnVRes;

extended Fix2X (x)
  Fixed X;

extended Frac2X (x)
  Fract X; OSErr GetVRefNum (pathRefNum, vRefNum)
  short pathRefNum; short *vRefNum;

OSErr OpenRF (fileName, vRefNum, refNum)
  OSStrPtr fileName; short vRefNum; short *refNum;

OSErr PBSetVInfo (hparamBlock,async)
  HPrmBlkPtr hparamBlock; Boolean async;

Fixed X2Fix (X)
extended x;

Fract X2Frac (X)
extended x;
```

b) CLIB environment variable supports multiple entries of specifying where libraries may be found using ';' as a delimiter.

4.1.7 M.LIB.

a) Added the following routine to *m.lib*:

```
abs(i)
int i;
```

b) Added *m8.lib* for the 68881 coprocessor.

4.1.8 INCLUDE.

a) Added *GetNodeAddress( )*, *ATPRequest( )*, *ATPResponse( )*, *ATPReqCancel( )*, *ATPRspCancel( )*; *MPPClose( )*, *IsMPPOpen( )*, and *IsATPOpen( )* to *appletalk.h*.

b) Added *GetApplLimit( )* to *memory.h*.

c) Added new header--*sane.h*.

d) Changed *ioFIStBlk* and *ioFIRStBlk* from short to unsigned short in *pb.h*.

e) Corrected names in lines 208 through 216 to include the prefix "extern" in *quickdraw.h.*

f) Added #defines (-128) to (-145) to *syserr.h.*

4.1.9 Grep.

a) Corrected *-F* option.

4.1.10 Diff.

a) Handles boundary conditions correctly now. The message that *diff* cannot synchronize the change record should not occur as often.

4.2 CHANGES SINCE VERSION 1.06h

Both bug fixes and enhancements will be listed by program/file.

4.2.1 SHELL.

a) Added *Quit* command to file menu in SHELL.

4.2.2 CC.

a) Fixed a bug where register arguments to a pascal type C function generated bad assembly language.

b) Fixed a bug where ? : with no assignments did not work.

c) Options 'b', 'u', and 'q' must now be specified with a '+' instead of a '-'.

d) Compiler now has the error messages built in so there is no need for the file *cc.msg* anymore.

e) New options are +I and +H for pre-compiled header files.

4.2.3 LN.

a) New option '-w' creates code resource 'SYMS' needed for the symbolic debugger.

4.2.4 Z.

a) Added tilde ( ) command that toggles the case of a character.

4.2.5 C.LIB.

a) Updated *fclose( )* to clear several additional fields before returning.

b) Updated *scanf( )*.

c) Added functions *strchr( ), strrchr( ), asctime( ), ctime( ), localtime( ), time( ), mktemp( )*, and *tmpnam( )* to *c.lib*.

d) Added *Environs( )* and *Restart( )* to *c.lib*.

e) *MoveHHi( )* is a trap call if using a 128K ROM and is in *c.lib*.

f) Added *_newrom( )* call to *c.lib*.

g) Added List manager calls to *c.lib*.

h) Added the following HFS calls to *c.lib*: *PBHGetVInfo( ), PBHSetVInfo( ), PBHGetVol( ), PBHSetVol( ), PBHOpen( ), PBHOpenRF( ), PBLockRange( ), PBUnlockRange( ), PBCloseWD( ), PBOpenWD( ), PBCatMove( ), PBDirCreate( ), PBGetWDInfo( ), PBGetFCBInfo( ), PBGetCatInfo( ), PBSetCatInfo( ), PBSetFMSP( ), PBHCreate( ), PBHDelete( ), PBHRename( ), PBHRstFLock( ), PBHGetFInfo( ), PBHSetFInfo( ), PBHSetFLock( ), PBSetEOF( ), PBHAllocContig( )*.

i) Changed the name of VCB structure in close.c to be C_VCB so not to conflict with VCB structure in *pb.h*.

j) Added *stat( )* and *access( )* functions to *c.lib*.

k) Added new *malloc( )*, *calloc( )*, *lmalloc( )*, *free( )*, *realloc( )* to *c.lib*.

l) Removed INIT'd bit from *creat( )* and *open( )* in *c.lib*.

m) Corrected names *RAMSDOpen( )* and *RAMSDClose( )* in *c.lib*.

4.2.6 S.LIB.

a) Corrected *scr_home( )* so that it does home the cursor.

b) Fixed *scr_insert( )*.

c) Added *scr_echo( )* and *scr_getc( )*.

4.2.7 SCSI.LIB

a) Created new library *scsi.lib* that contains the following calls: *SCSICmd( )*, *SCSIComplete( )*, *SCSIGet( )*, *SCSIInstall( )*, *SCSIRBlind( )*, *SCSIRead( )*, *SCSIReset( )*, *SCSISelect( )*, *SCSIStat( )*, *SCSIWBlind( )*, and *SCSIWrite( )*.

4.2.8 A.LIB.

a) Created new library *a.lib* that contains the appletalk calls.

4.2.9 RGen.

a) Fixed it to allow INCLUDE files to be greater than 64K.

b) Fixed optional name.

4.2.10 INCLUDE.

a) Added new HFS structures and calls to *pb.h*.

b) Added #defines P_tmpdir and L_tmpname to *stdio.h* for the *tmpnam( )* function.

c) Added new headers *time.h*, *stat.h*, and *scsi.h*.

d) Added #defines bDevCItoh, bDevLaser, lPrLFSizth, lPrPageOpen, lPrPageClose, lPrLFStd, lPrDocOpen, and lPrDocClose to *print.h*.

e) Added #defines iPrSavPFil, controlErr, and abortErr to *syserr.h*.

f) Added *GetEvQHdr( )* to *pb.h*.

g) Added *CopyMask( )*, *GetMaskTable( )*, *MeasureText( )*, *CalcMask( )*, and *SeedFill( )* to *quickdraw.h*.

h) Made *MaxApplZone( )* a trap call if using a 128K ROM.

i) Added *HSetRBit( ), HClrRBit( ), HGetState( ), HSetState( ), MaxBlock( ), PurgeSpace( ), MoveHHi( ), StackSpace( )*, and *NewEmptyHandle( )* to *memory.h.*

j) Added *Count1Resource( ), Get1IndResource( ), Count1Types( ), Get1IndType( ), Unique1ID( ), Get1Resource( ), MaxSizeRsrc( ), Get1NamedResource( ), RsrcMapEntry( )*, and *OpenRFPerm( )* to *memory.h.*

k) Added *SetFScaleDisable( )* and *FontMetrics( )* to *font.h.*

l) Added *TrackBox( )* and *ZoomWindow( )* to *window.h.*

m) Added *UpdtControls( )* to *control.h.*

n) Added *InsMenuItem( )* and *DelMenuItem( )* to *menu.h.*

o) Added *TESelView( ), TEPinScroll( )* and *TEAutoView( )* to *textedit.h.*

p) Added *HideDItem( ), ShowDItem( ), UpdtDialog( )*, and *FindDItem( )* to *dialog.h.*

q) Added *Long2Fix( ), Fix2Long( ), Fix2Frac( ), Frac2Fix( ), FracCos( ), FracSin( ), FracSqrt( ), FracMul( ), FracDiv( ), FixAtan2( )*, and *FixDiv( )* to *toolutil.h*

r) Added *Pack8( ), Pack9( ), Pack10( ), Pack11( ), Pack12( ), Pack13( ), Pack14( ), Pack15( )* to *packages.h.*

s) Added *RelString( )* to *osutil.h.*

t) Added type Fract and Fixed to *types.h.*

u) Added *Environs( )* and *Restart( )* to *osutil.h.*

v) Changed prInfoPt to be prInfoPT in *print.h.*

w) Corrected return of *PostEvent( )* to be OSErr in *event.h.*

x) Added #define TIOCNTLC to *sgtty.h* to check for Clover ".".

y) Corrected names *RAMSDOpen( )* and *RAMSDClose( )* in *serial.h.*

z) Removed *cc.msg* as it is included in the new compiler (*cc*).

4.2.11 Make.a

a) Supports file dependencies in other directories.

4.2.12 Diff.

a) Corrected it so that it would find all the differences.

4.2.13 Ctags.

a) Updated it to handle comments better.

4.2.14 EDIT.

a) New EDIT version 2.0D1 that runs with HFS. This is licensed from Apple.

4.2.15 RMaker.

a) New RMaker that runs with HFS.

4.2.16 Mixcroot.o.

a) New *mixcroot.o* eliminates need to run InstallConsole.

4.2.17 Ramdisk.

a) In *ramdisk.asm*, added 16K for larger compiler.

4.3 CHANGES SINCE VERSION 1.06g

Both bug fixes and enhancements will be listed by program/file.

4.3.1 SHELL.

a) Fixed *mkdir* so that it gives an error message when attempting to create a directory on an MFS volume.

b) When the menu bar is turned off and a blank disk is inserted, the mouse cursor will now appear.

c) Fixed the *ls* command so that the correct contents of a disk will be listed after a disk has been ejected and another has been inserted.

d) Set *AppParmHandle* location to zero after the *DisposeHdl()* call on that location has been made.

e) Made a fix to not eject the external disk upon startup.

f) Fixed the *cd* command to create a directory only on an MFS volume if the directory does not already exist.

4.3.2 CC.

a) Fixed an initializing problem in "for" loops.

4.3.3 LN.

a) Fixed a bug to recognize mixmode MDS function names when linking in MDS .rel files.

b) Corrected segment numbers in the *.sym* file.

4.3.4 C.LIB.

a) Updated *malloc()* to return an error when allocation request for memory fails.

b) Added *ScreenRes()*, *SetUpA5()*, and *RestoreA5()* functions.

4.3.5 M.LIB.

a) Updated *atan()*, *tan()*, *exp()*, and *pow()* functions.

4.3.6 A.LIB.

a) Updated the AppleTalk interface routines.

4.3.7 ABPackage.

a) Added the AppleTalk resource (*ABPackage*) to be included in AppleTalk programs.

4.3.8 RGen.

a) Corrected the PROC resource to strip off the first 4 bytes from CODE 1 resource before copying.

4.3.9 INCLUDE.

a) Added *ScreenRes( )* declaration to *toolutil.h.*

b) Added *SetUpA5( )* and *RestoreA5( )* declarations to *osutil.h.*

c) Added #defines *inZoomIn* and *inZoomOut* to *window.h.*

d) Added #define __MEMORY to *memory.h.*

e) Added #defines (-120) to (-127) to *syserr.h.*

4.3.10 Make.

a) Corrected case sensitivity. Before it did not recognize that FOO.c needed to be compiled if its date was later than that of foo.o.

4.3.11 Grep.

a) Fixed *-f* option to work when pattern has upper case in it.

4.3.12 DB.

a) Fixed the *x?* command.

b) Fixed the skip count command to default to 1 when breakpoints are set for traps.

c) Fixed the dot (.) command.

d) Fixed the characters going out the serial port to be no parity so that commands like *d?* would work.

5. Packaging

This section describes the files that are provided with each version of the Aztec C68K for the Macintosh. The files for the *Professional* version are contained on three disks named *sys:*, *sys2:*, and *sys3:*. The files for the *Developer* version are contained on a different set of three disks named *sys:*, *sys2:*, and *sys3:*. The *Commercial* version has the *Developer* three disks plus an additional disk named *sys4:*.

Professional Version

5.1 Contents of *sys:*

The root directory of *sys:* contains the following files and directories:

.profile	the startup file of SHELL commands
hello.c	example program
SHELL	SHELL command parser
System	System file minus some fonts and desk accessories

5.1.1 Contents of *sys:bin*

as	assembler
cc	compiler
Edit	text editor
ln	linker

5.1.2 Contents of *sys2:lib/*

a.lib	Library of appletalk routines
c.lib	Portable C and Macintosh Toolbox
m.lib	Library of Manx Aztec C (IEEE Double Precision Floating point emulation) functions
m8.lib	Library of 68881 floating point functions
mixcroot.o	Croot() for stand-alone programs that want UNIX-style console I/O
ms.lib	Standard Apple Numeric Environment functions Portable C and Macintosh Toolbox
prscreen.o	printer glue for printing to screen
s.lib	Libary of screen functions
sacroot.o	Croot() for stand-alone programs
scsi.lib	Library of SCSI functions

5.1.3 Contents of *sys2:include*

appletalk.h	Appletalk Manager declarations
control.h	Control Manager declarations
ctype.h	macro definitions for the 'is...' functions
desk.h	Desk Manager declarations
dialog.h	Dialog Manager declarations
disk.h	Disk Manager declarations
errno.h	system independent error codes

event.h	Event Manager declarations
fcntl.h	unbuffered I/O symbol definitions
font.h	Font Manager declarations
inits.h	initialization functions
list.h	List Manager declarations
math.h	Math Manager declarations
memory.h	Memory Manager declarations
menu.h	Menu Manager declarations
monitor.h	profiler declarations
obj68k.h	Aztec object file format
osutil.h	OS utility declarations
packages.h	Package declarations
pb.h	File and Device Manager declarations
pb.inc	File and Device Manager declarations
print.h	Print Manager declarations
quickdraw.h	Quickdraw declarations
resource.h	Resource Manager declarations
retrace.h	Vertical Retrace Manager declarations
sane.h	SANE Manager functions
scrap.h	Scrap Manager declarations
scsi.h	SCSI Manager declarations
segment.h	Segment Loader declarations
serial.h	Serial Manager declarations
setjmp.h	Set Jump declarations
sgtty.h	console I/O declarations
sound.h	Sound Manager declarations
stat.h	stat function declarations
stdio.h	buffered I/O declarations
syserr.h	Macintosh system error codes
textedit.h	Textedit Manager declarations
time.h	time function declarations
toolutil.h	Toolbox utilities
types.h	common declarations
window.h	Window Manager declarations

5.1.4 Contents of *sys2:bin*

arcv	source dearchiver
cmp	binary file compare
cnm	object file utility
cprsrc	resource copy utility
hd	hex dump utility
lb	object file librarian
mkarcv	source archiver
prsetup	printer setup utility
RGen	resource generator
term	terminal communications utility

5.2 Contents of *sys3:*

InstallConsole stand-alone program for installing the console driver

5.2.1 Contents of *sys3:Apple/*

freeterm	terminal emulation program
RMaker	resource compiler
RMaker 2.0 doc--TEXT	
SERD	RAM serial driver resources

5.2.2 Contents of *sys3:debug/*

db	debugger
MacsBug	full-screen debugger licensed from Apple

5.2.3 Contents of *sys3:example/*

explor.c	C desk accessory
grow.c	C version of Pascal Grow example
grow.r	RGen input file for Grow example
makefile	for building examples
medit.c	"mini-edit" source
medit.res	RGen source input for mini-edit
modal.c	modal dialog example
modal.r	RGen input file for modal dialog example
print.c	print
print.res	print demonstration
procptr.c	RGen output for print demo
pset.c	PRSetup source
pset.res	RGen setup for PRSetup
qdsample.c	a quickdraw example
qdsample.r	RGen input for quickdraw example
retrace.c	example of vertical retrace manager
textbox.c	drawing a box and text

5.2.4 Contents of *sys3:macintalk/*

file	sample input for speak example
macintalk.h	header file for MacinTalk
macintfo.o	interface object module
macintalk.h	header file for MacinTalk
macintalk.info	desc. of files in directory
mkall	input to make program to example speak
speak.c	example program of Macintalk

5.2.5 Contents of *sys3:mdef/*

edit.c	modified EDIT program with MDEF procedure as part of program
grow.c	resource compiler input for MDEF as a resource
mkall	input to make program to example speak
mymenu.c	MDEF procedure to be made into a resource

5.2.6 Contents of *sys3:term/*

makefile	input to make program to produce term
menu.c	menu handler for term
screen.c	screen functions for term
term.c	main module for term

5.2.7 Contents of *sys3:util/*

cmp.c	source to cmp utility
cnm.c	source to cnm utility
con	compiled console driver
cprsrc.c	source to cprsrc utility
hd.c	source to hd utility
install.c	InstallConsole source
makefile	program maintenance utility
obj68k.h	object file utility
prsetup.c	source to presetup utility

Developer Version

5.3 Contents of *sys:*

The root directory of *sys:* contains the following files and directories:

System	System file minus some fonts and desk accessories
SHELL	SHELL command parser
README	changes made since manual was printed
.profile	the startup file of SHELL commands
InstallConsole	stand-alone program for installing the console driver

5.3.1 Contents of *sys:bin/*

arcv	source dearchiver
as	assembler
cc	Aztec C68K compiler
cmp	binary file compare
cnm	object file utility
cprsrc	resource copy utility
ctags	program used to create Z tags file
db	debugger
diff	text file difference reporter
grep	regular expression search program
hd	hex dump utility
lb	object file librarian
ln	overlay linker

make	automatic program generation utility
mkarcv	source archiver
obd	list object code
ord	sort object module list
presetup	printer setup utility
RGen	resource generator
su	trailing-to-leading underscores utility
term	terminal communications utility
z	text editor

5.4 Contents of *sys2:*

hello.c	sample program

5.4.1 Contents of *sys2:/util*

cmp.c	source to cmp utility
cnm.c	source to cnm utility
con	compiled console driver
cprsrc.c	source to cprsrc utility
hd.c	source to hd utility
install.c	InstallConsole source
makefile	program maintenance utility
prsetup.c	source to prsetup utility

5.4.2 Contents of *sys2:term/*

makefile	input to make program to produce term
menu.c	menu handler for term
screen.c	screen functions for term
term.c	main module for term

5.4.3 Contents of *sys2:ram/*

mkram	SHELL exec file to make the parts of the Ram Disk
mountram.c	C program to start the Ram Disk
ramdisk.asm	assembly language source to .Ram driver

5.4.4 Contents of *sys2:mdef/*

edit.c	modified EDIT program with MDEF procedure as part of program
grow.c	resource compiler input for MDEF as a resource
mkall	input to make program to example speak
mymenu.c	MDEF procedure to be made into a resource

5.4.5 Contents of *sys2:macintalk/*

file	sample input for speak example
macintalk.h	header file for MacinTalk
macintf.o	interface object module
macintalk.info	files in directory
mkall	input to make program to example speak
speak.c	example program of MacinTalk

5.4.6 Contents of *sys2:lib/*

a.lib	appletalk routines
c.lib	MPW functions
c32.lib	MPW function integers (32 bit)
m.lib	Manx Aztec C (IEEE Double Precision Floating point emulation) functions
m8.lib	Library of 68881 floating point functions
mixcroot.o	Croot() for stand-alone programs that want UNIX-style console I/O
ms.lib	Standard Apple Numeric Environment functions
prscreen.o	printer glue for printing to screen
s.lib	Library of screen functions
sacroot.o	Croot() for stand-alone programs
scsi.lib	Library of SCSI functions

5.4.7 Contents of *sys2:include/*

Appletalk.h	Appletalk Manager declarations
Controls.h	Control Manager declarations
CType.h	macro definitions for the 'is...' functions
DeclROMDefs.h	ROM definition interfaces
Desk.h	Desk Manager declarations
Devices.h	Devices Manager declarations
Dialogs.h	Dialog/Alert Manager declarations
Diskinit.h	Disk Initialization declarations
Disks.h	Disk Driver declarations
ErrNo.h	system independent error codes
Errors.h	System Error Handler declarations
Events.h	Event Manager declarations
FCntl.h	unbuffered I/O symbol definitions
Files.h	File Manager declarations
FixMath.h	Fixed-Point Math declarations
Fonts.h	Font Manager declarations
Graf3D.h	3-D Graphics routines
IndVideoIntf.h	video interface
IOCtl.h	device control values
Lists.h	List Manager declarations
Math.h	transcendental functions declarations
Memory.h	Memory Manager declarations
Menus.h	Menu Manager declarations
monitor.h	profiler declarations
ncOSIntf.h	new (SE and II) OS interfaces
obj68k.h	Aztec object file format
OSEvents.h	OS events declarations
OSUtils.h	OS utility declarations
Packages.h	Package declarations
pb.inc	File and Device Manager declarations
Printing.h	Print Manager declarations
PrintTraps.h	new Printing Manager interface

Quickdraw.h	Quickdraw declarations
Resources.h	Resource Manager declarations
Retrace.h	Vertical Retrace Manager declarations
SANE.h	SANE Manager functions
Scrap.h	Scrap Manager declarations
SCSIIntf.h	SCSI Manager declarations
SegLoad.h	Segment Loader declarations
Serial.h	Serial Manager declarations
SetJmp.h	Set Jump declarations
sgtty.h	console I/O declarations
Signal.h	Signal Manager declarations
Sound.h	Sound Manager declarations
stat.h	stat function declarations
StdIO.h	buffered I/O declarations
Strings.h	String Conversions declarations
TxtEdit.h	Textedit Manager declarations
Time.h	Time Manager Interface
utime.h	time function declarations
ToolUtils.h	Toolbox utilities
Types.h	common declarations
Values.h	values declarations
Windows.h	Window Manager declarations

5.4.8 Contents of *sys2:example/*

explor.c	C desk accessory
grow.c	C version of Pascal Grow example
grow.r	RGen input file for Grow example
makefile	for building examples
medit.c	"mini-edit" source
medit.res	RGen source input for mini-edit
modal.c	model dialog example
modal.r	RGen input for modal
print.c	print
print.res	print demonstration
procptr.c	RGen output for print demo
pset.c	PRSetup source
pset.res	RGen setup for PRSetup
qdsample.c	a quickdraw example
qdsample.r	RGen input for quickdraw example
retrace.c	example of vertical retrace manager
textbox.c	drawing a box and text

5.4.9 Contents of *sys2:apple/*

MacsBug	full-screen debugger licensed from Apple
ResEdit	a resource editor
RMaker	a resource compiler
SERD	RAM serial driver resources

5.5 Contents of *sys3:*

freeterm	terminal emulation program
mdef/	
lib/	
include/	
example/	
profiler/	

5.5.1 Contents of *sys3:mdef/*

edit.c
grow.c
mkall
mymenu.c

5.5.2 Contents of *sys3:lib/*

a.lib	mixcroot.o
c.lib	ms.lib
c32.lib	prescreen.o
m.lib	s.lib
m8.lib	sacroot.o
	scsi.lib

5.5.3 Contents of *sys3:include/*

appletalk.h	print.h
control.h	quickdraw.h
ctype.h	resource.h
desk.h	retrace.h
dialog.h	sane.h
disk.h	scrap.h
errno.h	scsi.h
event.h	segment.h
fcntl.h	serial.h
font.h	setjmp.h
inits.h	sgtty.h
list.h	sound.h
math.h	stat.h
memory.h	stdio.h
menu.h	syserr.h
monitor.h	textedit.h
obj68k.h	time.h
osutil.h	toolutil.h
packages.h	types.h
pb.h	window.h
pb.inc	

5.5.4 Contents of *sys3:example/*

explor.c	print.c
grow.c	print.res
grow.r	procptr.c

makefile	pset.c
medit.c	pset.res
medit.res	qdsample.c
modal.c	qdsample.r
modal.r	retrace.c
	textbox.c

5.5.5 Contents of *sys:profiler/*

monitor.c	performs runtime analysis
monitor.o	compiled monitor.c
prof	reports on the execution of *monitor* program
test.c	demonstration of *prof* use

Commercial Version

5.6 Contents of *sys4:*

The root directory of *sys:4* contains the following files and directories:

sys2_arc	source for MPW-compatible libraries
sys3_arc	source for Aztec libraries

5.6.1 Contents of *sys4:sys2_arc/*

inp.arc	master build archive (Open Me First!)
atalk.arc	appletalk routines
cntrl.arc	control manager routines
con.arc	console driver
csu.arc	crt0, Croots and exit functions
dialog.arc	dialog routines
disk.arc	disk routines
fs.arc	fs routines
m881.arc	68881 interface
math.arc	floating point and transcendental functions
mch68.arc	some low-level 68K functions
mem.arc	memory manager assembly language routines
menu.arc	menu manager toolbox interface
misc.arc	miscellaneous system-independent functions
newglue.arc	MPW-compatible glue
osmisc.arc	miscellaneous operating system interface routines
osutil.arc	operating system utility routines
pack.arc	packages assembly language routines
pb.arc	low-level file and device manager routines
print.arc	print routines
qd.arc	quickdraw interface routines
rsc.arc	resource manager
sane.arc	SANE routines
screen.arc	screen routines
scsi.arc	scsi routines
serial.arc	serial interface routines

sound.arc	sound interface routines
stdio.arc	standard I/O routines
sysio.arc	system I/O
tool.arc	miscellaneous toolbox interface routines

5.6.2 Contents of *sys4:sys3_arc/*

atalk.arc	osutil.arc
con.arc	pack.arc
csu.arc	pb.arc
disk.arc	print.arc
fs.arc	sane.arc
inp.arc	screen.arc
math.arc	scsi.arc
m881.arc	serial.arc
mch68.arc	sound.arc
mem.arc	stdio.arc
misc.arc	sysio.arc
osmisc.arc	tool.arc

6. Additional Documentation

This part of the release document contains two sections of information:

1. Common Problems
2. Documentation Updates

Start with the common problems chapter when you are first having a problem. File the updates and additions where suggested so you may reference them easily.

6.1 COMMON PROBLEMS

6.1.1 Can't Find Finder.

Symptom:
A disk being booted displays the bomb box and the message "Can't find Finder". If the disk is the distribution disk, then it must have been "fixed" by an older version of the Finder. If the disk is a copy of the distribution disk, then it must have been copied with the Finder, and not the *cp* command or the Single Disk Copy program.
Solution:
Get a new version of the Finder from your Apple dealer. This version has 4 selections under the Special menu while the older Finder had only 3 selections. Select the SHELL by clicking it once with the mouse. Go to the Special menu and pick the last option which should be "Set Startup". Now you can boot the disk.

This is a Macintosh application which comes up with a window displaying instructions and will set the startup program on any disk that you wish.

6.1.2. SHELL bombs with ID = 99.

Symptom:
After the SHELL has read the key disk, it bombs with an ID = 99. This will most likely happen when the SHELL is clicked from the Finder. This usually means that the SHELL cannot find the console driver.
Solution:
One solution is to boot the distribution disk directly. This will cause the System file on the distribution disk to be used, which contains the console driver. The second solution is to use the *InstallConsole* program to install the console driver into the System file currently being used. Then clicking the SHELL should work correctly.

6.1.3 Printer Doesn't Flow Control

Symptom:
When printing a long file by redirecting output from the SHELL, after a page or so, large sections of text are simply lost and the output

appears garbled. This usually means that the printer is not set up to do flow control correctly.
Solution:
The serial ports default to doing hardware handshake with the output device. The Apple imagewriter printer must have Switch 2-3 set to OPEN to enable hardware handshake.

6.1.4 Can I Use MacWrite?

Symptom:
The programmer misses using the mouse.
Solution:
Yes, you can use MacWrite from the SHELL. The only thing that is important, is that when the document is saved, that it be saved as TEXT-only. You can also use the mouse-based EDIT from Apple which is supplied with this release.

6.1.5 Printing With %f Doesn't Work.

Symptom:
Using *printf()* to print floating point numbers with the %f, %e, or %g formats just prints the letter after the %.
Solution:
This means that the program was incorrectly linked. There are two versions of *printf()* and *scanf()* in the libraries. One version knows about floating point numbers, while the other does not. This is done since most programs don't use floating point. It seemed wasteful to force them all to carry the extra overhead of the *atof()* and *ftoa()* routines. These routines would have been included every time the functions *printf()* or *scanf()* were called.

So, the floating point versions of these two routines are kept in the *m.lib* library. When you link, you should type:

```
ln file.o -lm -lc
```

so that the *m.lib* library is searched before the *c.lib* and the correct routine is loaded.

6.1.6 Pointers Don't Print Correctly.

Symptom:
A *printf()* statement that works correctly on other machines doesn't work right on the Macintosh.
Solution:
On most machines, printing an address or the value of a pointer can be done using a "%d" or a "%x". However, on the Macintosh, a pointer is a long value and must be displayed using the "%ld" or "%lx" format.

6.2 DOCUMENTATION UPDATES

6.2.1 Compiler, Assembler, Linker, and Z.

Several pages describe the options and changes in release 3.4. File these pages at the end of the appropriate sections in the Aztec C68K manual.

6.2.2 Debugger.

The *db* document with this release replaces the section previously sent to you. Add the new section at the back of your Aztec C68K manual.

6.2.3 Profiler Report.

The *prof* command page describes the profiler report function that is in *c.lib*. Add it to the *util* section of your Aztec C68K manual.

6.2.4 Monitor Function.

The *monitor* command page describes the function that is used in conjunction with the profiler utility. Add this page to the *libmac* section of your Aztec C68K manual.

6.2.5 lb Utility.

The *lb* command page describes the library utility function that creates and maintains user libraries. *lb* replaces *libutil* in the *util* section of your Aztec C68K manual.

6.2.6 obd Utility.

The *obd* command page describes the list loader file function. Add this page to the *util* section in your Aztec C68K manual.

6.2.6 Miscellaneous Command Pages.

The *make*, *mktemp*, *newrom*, *time*, *screen*, and *tmpnam* command pages describe enhancements and changes from release 1.06h. The *hfsconvert* command page describes changes from release 1.06i. Copies of these documents are included again here so that version 3.4 is a complete package containing all files and all documentation to keep you completely up to date.

6.2.7 Modified Include Files.

The documents in this section describe the additional functions and calling sequences added with this, and previous, releases. These documents should be added to the back of the Toolbox section of your manual. The Sane Manager description is new; the Memory Manager included here is changed and, therefore, replaces the one in your manual; and the Apple Talk Manager, List Manager, and SCSI Manager functions are duplicates of copies issued with release 1.06h.

Aztec C
for the Macintosh

version 1.06
March 1986

Distributed by:
Manx Software Systems, Inc.
P.O. Box 55
Shrewsbury, N.J. 07701
201-542-2121

USE RESTRICTIONS

The components of the Aztec C68K software development system are licensed software products. Manx Software Systems reserves all distribution rights to these products. Use of these products is prohibited without a valid license agreement. The license agreement is provided with each package. Before using any of these products the license agreement must be signed and mailed to:

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

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

Software that is developed with Aztec C68K software development system can be run on machines that are not licensed for these products as long as no part of the Aztec C software, libraries, supporting files, or documentation is distributed with or required by the software. In the latter case a licensed copy of the appropriate Aztec C software is required for each machine utilizing the software. There is no licensing required for executable modules that include runtime library routines.

RESTRICTED RIGHTS LEGEND

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

COPYRIGHT

DISCLAIMER

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

TRADEMARKS

Aztec C68K, Manx AS, Manx LN, Aztec SHELL, and Z are trademarks of Manx Software Systems. CP/M-86 is a tradmark of Digital Research. MSDOS is a trademark of Microsoft. PCDOS is a trademark of IBM. UNIX is a trademark of Bell Laboratories. Macintosh is a trademark of Apple Computer.

Manual Revision History

March 1986 ..... Third Edition
March 1985 ..... Second Edition
Oct 1984 ..... First Edition

Summary of Contents

Macintosh Chapters

System Independent Chapters

Index

Contents

OVERVIEW

Overview

The Aztec C68K Software Development Package is a set of programs for developing programs in the C programming language; the resulting programs run on a Macintosh. The development can be done on a Macintosh; it can also be done on several other type systems, as described below, and the executable code downloaded to the Macintosh.

Some of the features of Aztec C68K are:

* The full C language, as defined in the book *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, is supported, with the exception of the bit- field data type;
* On the Macintosh, development is done using a program called the SHELL, which replaces the Macintosh Finder program and provides a UNIX-like environment;
* With some versions of Aztec C68K, several utility programs are provided that are similar to UNIX programs: *Z*, a text editor, which is like the UNIX *vi* editor; *make*, which automates some of the steps in program development and maintainance; *grep*, a pattern-matcher; *diff*, a program that determines the difference in source files;
* An extensive set of user-callable functions is provided;
* Features and functions are provided that allow programs to call the Macintosh Toolbox and OS routines;
* Code can be segmented, allowing programs to be created and executed that are larger than available memory;
* Modular programming is supported, allowing the components of a program to be compiled separately, and then linked together;
* Programs can be developed that can only be activated from within the SHELL environment. Such programs have many UNIX features.
* Programs can also be developed that can be activated from within the SHELL or Finder environments. Such programs have fewer UNIX features but can be created to support UNIX-style console i/o, if desired. When development is done on the Macintosh, these type programs can only be created if you have the commercial version of the Development

Package.

* Assembly language code can either be combined in-line with C source code, or placed in separate modules which are then linked with C modules. This feature is not available with the Personal version of the Aztec Development Software Package that runs on the Macintosh.

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

The system-dependent functions allow programs to take advantage of special features of the Macintosh. Some of these system dependent functions act as an interface to the Macintosh toolbox and OS routines.

Several extensions to the C language are supported by the Aztec C compiler that allow C programs to directly call Toolbox and OS routines, thus making programs smaller and more efficient.

Several header files are included with the Development Package that facilitate the accessing of toolbox and OS routines by C programs. Using the constants, data structures, and routines defined in these files, a C program can access Toolbox and OS routines in a manner similar to a Pascal program. Thus, a programmer can decide how a Pascal program would call toolbox or OS routines and then easily translate this to C.

Versions

Several versions of the Aztec C68K Software Development System are available, for use in different environments. Some, called "native development systems", allow development to be done on the Macintosh; the others, called "cross development systems", allow development to be done on other machines, with the resulting programs downloaded to the Macintosh.

For a description of the native development systems, and for the names of systems on which cross development can be done, see the Aztec C68K product bulletins.

Cross development, with the Macintosh as host

Manx has compilers for developing C programs in which the resulting programs run on systems other than the Macintosh. Cross-development versions of many of these other compilers are available which use the Macintosh as the host system. For more information about cross development, with the Macintosh as the host, see the Aztec C68K product bulletins.

Components

Aztec C68K contains the following components:

* The compiler, assembler, linker, and object file librarian;
* The SHELL, for native-Macintosh versions;
* Object libraries containing user-callable functions and support functions;
* Several utility programs, including, with some versions of Aztec C68K, programs similar in function to the UNIX utilities *make*, *grep*, *diff*, and *vi*.

Preview

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

The Macintosh-specific chapters and their identifying codes are:

tut describes how to get started with Aztec C68K: it discusses the installation of Aztec C68K, presents an introduction to the SHELL, and gives an overview of the process for turning a C source program into an executable form;

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

z describes the text editor Z;

utility describes the utility programs that are provided with Aztec C68K;

libovmac presents Macintosh-specific information about the library functions;

libmac describes the special, Macintosh-specific functions provided with Aztec C68K;

tools describes how C programs can access toolbox and OS routines;

tech discusses several miscellaneous topics, including memory organization, creation of command programs, drivers, and desktop accessories, and the representation of data;

examples discusses sample programs;

debug describes the debugging utilities that are provided with Aztec C68k.

The System-independent chapters and their codes are:

libov presents an overview of the functions provided with Aztec C68K;

lib describes the system-independent functions provided with Aztec C68K;

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

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

TUTORIAL INTRODUCTION

Chapter Contents

Tutorial Introduction

This chapter describes how to quickly start using Aztec C68K.

We first describe how to make backup copies of the distribution disks. Then we introduce the SHELL, the command processor program to which you'll enter commands while developing programs. Finally, we go through the steps you can follow to create and execute the a program.

1. Getting Started

The first thing you should do with your Aztec C68K software is to make a copy of the distribution disks. Use the copies for doing development, not the distribution disks.

1.1 Copying disks on a Macintosh having two drives

If your Macintosh has two drives, the disks are most easily copied using the *cp* copy command that is built into the SHELL. So the first thing to do is get the SHELL started. To do this, put the first distribution disk in the Macintosh's "internal" drive and turn on the Macintosh. The SHELL will automatically be loaded and started; it will display a title message and then wait. Type any key to get it going. The SHELL will erase the screen, issue its prompt, '-? ', and wait for you to enter a command.

Next, put a blank disk in the "external" drive. If the disk is uninitialized, the Macintosh will say so and ask if you want to initialize it. Click 'yes', using the mouse. When it asks for a name, leave it untitled. When the initialization is done, the SHELL will still be waiting for you to enter a command.

Now, you can enter the *cp* command to backup the distribution disk. Enter:

```
cp 1: 2:
```

cp will ask

```
Are you sure?
```

If you are, type *y* followed by a carriage return. *cp* will procede to copy the contents of the disk in drive 1: to the disk in 2:. When *cp* is done, it will eject the newly created disk, and return to the SHELL. The SHELL will display its '-?' prompt, and wait for another command to be entered.

If there's more than one distribution disk, eject the disk in the internal drive by holding down the key with the cloverleaf symbol and

then pressing the '1' key, and then copy another disk, using the *cp* command again.

Continuing the discussion of ejecting disks, you tell the SHELL to eject a disk by holding down the cloverleaf key and then typing the number of the drive whose disk is to be ejected. Thus, holding down the control key and typing the '1' key ejects the disk in the internal drive, and holding down the cloverleaf key and typing the '2' key ejects the disk in the external drive.

1.2 Copying disks on a Macintosh having just one drive

If your Macintosh has only one disk drive, you can copy the distribution disks by activating the Macintosh Finder program and then copying the disks using the standard single-disk copy utility.

1.3 Disk usage during development

While developing programs, you will normally use two disks: one of the disks, a system disk, contains the SHELL, the system file, and frequently used programs, libraries, and header files. Initially, you can probably just use a copy of the first distribution disk as your system disk.

The other disk, the working disk, contains your own files: C source, object modules, executable programs, and so on.

A working disk is prepared by simply initializing it. When an uninitialized disk is inserted in a drive, a message will be displayed saying so, and you will be asked whether you want to initialize it. You should click 'yes'; when the initialization is done, you should then give the volume a name.

If you have a Macintosh with a single drive, you can still develop programs using two disks. When an attempt is made to access the disk which isn't in the drive, the disk in the drive will be ejected and a message displayed prompting you to insert it. When this is done, execution automatically continues.

The use of Aztec C68K on a single-drive Macintosh is discussed in more detail in the Technical Information chapter.

2. Using the SHELL

In this section, we want to introduce you to some of the features of the SHELL and to some frequently used commands.

The first thing you need to do is to start the SHELL, if it's not still running from the disk copying that you did. To boot the SHELL from the copy of the first distribution disk, put the disk in the internal drive and then turn on the Macintosh or press the reset button. When you do this, the SHELL will clear the screen, display its '-?' prompt, and wait for you to enter a command.

2.1 Looking around

The function of the SHELL is to execute commands that you enter. There are two ways of entering commands: by typing them on the keyboard and by selecting them, using the mouse, from one of the SHELL's menus.

Select the *Commands* menu and look at the items that appear. These are the names of all the SHELL's 'builtin' commands; that is, commands whose code is contained in the SHELL itself. The commands whose names are in the top part of the menu require no arguments and can be executed by selecting them with the mouse. The commands whose names are in the bottom part of the menu can't be executed via the mouse; they're listed in the menu as a handy reference, allowing you to see at a glance the names of all the SHELL's builtin commands. The two groups of names are separated by a horizontal line.

So now, let's try some commands. Execute the *mount* builtin command by either typing its name, followed by typing the return key, or by selecting it from the *Commands* menu.

The SHELL will display the names of all mounted volumes along with information about them. You should see a volume called *sys:*. The information displayed before the name and after is discussed in more detail in the SHELL reference section. In general, the information discussed in this tutorial will use commands without any detailed discussion. More information can be found in other sections of this manual.

Next, execute the *ls* builtin command by either typing its name followed by hitting the return key or by selecting it from the *Commands* menu. This command displays the names of all files in the 'current directory'. The current directory is analogous to the top level window of the Finder.

To see more detailed information about the files, type or select from the *Commands* menu:

```
ls -l
```

Notice that certain names end with a '/' character. This means that this is not a file. Instead, it is a subdirectory similar to a folder under the Finder. The detailed listing also identifies it by the <DIR> associated with these names.

You can pass arguments to commands. For example, you can tell *ls* the name of a directory whose contents you want to examine. To see the contents of the *bin/* directory, type:

```
ls bin/
```

You can't execute the above command by selecting it from the *Commands* menu: it requires an argument (*/bin*), and commands activated from the *Commands* menu cannot be passed arguments.

Just as the Finder has the concept of a front or current window, the SHELL has the current directory. If you select:

```
pwd
```

the SHELL will display what it is using as the current directory. The SHELL remembers a volume name as part of the current directory. To change the current directory, type

```
cd include
```

This moves us from the top level directory to the *include* directory. (Note that you can't execute this command by selecting it from the *Commands* menu, since it needs an argument.) Try the *pwd* command again now. To get back to the top level, simply type:

```
cd /
```

Once you are in the *include* directory, use the *ls* command to see what files are in the directory. There are a lot of files there, so to see detailed information on a few of the files, try typing:

```
ls -l s*
```

to get information on all files beginning with 's'.

To look at the contents of a file, use the *cat* command. Try typing:

```
cat errno.h
```

This will display the information on the Macintosh screen.

The *cd*, *pwd*, *mount*, *cat*, and *ls* commands can be used to move around on the disk and look at volume and file information. Try looking in other directories as well.

2.2 Working with Files

Now let's try and do something useful. First, create a new directory by using the *cd* command as follows:

```
cd /test
```

Since the *test* directory does not yet exist, the SHELL displays the "Empty directory" message. Now let's try something. Type the *cat* command on a line with no arguments. Now try typing some lines followed by carriage returns. Notice that each line is redisplayed when the return is pressed. The SHELL supports the concept of standard input and output devices. The *cat* command copies the contents of any files specified as arguments to the standard output. The standard output defaults to the Macintosh screen. Thus, the previous use of *cat* displayed the contents of the file on the screen.

If no arguments are specified, then *cat* takes its input from the standard input which defaults to the Macintosh keyboard. Thus, when no arguments were given, it copied the input from the keyboard to the screen a line at a time. It will keep doing this until it reaches the end of the file. This can be simulated on the keyboard by holding down the cloverleaf key and 'd' key simultaneously. The cloverleaf key is used as a CONTROL key by the SHELL, so the end of file is more traditionally known as ^D.

A useful feature of the SHELL which we can use is the ability to change where the standard input and output are connected. This is done using the characters '<' for input and '>' for output. For example, in the previous case where we looked at the contents of the *errno.h* file, we could have said:

```
cat < errno.h
```

which would have changed the standard input from the keyboard to the file *errno.h*. We could also use the *cat* command to copy a file by typing something like:

```
cat < inputfile > outputfile
```

We'll see an easier way of doing this later.

We can use this to create a file by typing:

```
cat > greet
```

The disk will whir and the cursor will wait for input. Anything that you type will be placed in the file *greet*. Try typing:

```
main()
{
  printf("greetings!!\n");
}
^D
```

To make sure the file is correct, use the *cat* command again to display the file to the screen. If you now do a directory listing you will see the file *greet*. Since most C programs usually are distinguished by a file name extension of ".c", use the following command to rename the

file:

```
mv greet greet.c
```

This renames the first file name to the second.

A better way to create and modify files is to use a text editor. The Aztec C68K Commercial package contains two editors you can use for entering programs. One of them, named 'z', is similar to the UNIX editor *vi*. The other, named *edit*, is mouse-based. *z* is described in its own chapter of this manual, and *edit* is described in the Utility Programs chapter. The Personal and Developer versions contain only *edit*.

You can also use MacWrite for entering programs, if you want.

2.3 Builtin commands, command programs, and exec files

A builtin command is a command whose code is contained in the SHELL. All the commands used up to this point have been builtin commands.

The SHELL can also execute two other types of commands: command programs and exec files.

A command program is another type of command that the SHELL can execute; its code is in a disk file. The compiler, assembler, linker, and the programs that you create are all command programs.

An exec file is the last type of command that the SHELL can execute. It is a file containing a list of commands. We're not going to discuss exec files any further in this section. For more information on them, see the chapter on the SHELL.

When a command is given to the SHELL it checks its built-in list first. If it's not found there, the SHELL then looks for a file that has the same name. This file can contain either a command program, or a sequence of commands that the SHELL is to execute.

When looking for a file containing a command program or exec file, the SHELL will look in a definable set of directories. For example, the compiler, *cc*, is in the directory */bin*. Thus, to run the compiler, we could say:

```
/bin/cc
```

This directory is by default automatically searched by the SHELL when it is looking for a file to execute. Thus, we could also say:

```
cc
```

This is explained more in the SHELL section, but for a peek, select from the Commands menu:

```
set
```

which will display a list of names followed by strings. The name *PATH* specifies the search path for program loading.

3. Compiling, Assembling and Linking

Now that you're somewhat familar with the SHELL, let's create an executable version of your 'greetings' program. You can do this by either selecting items from the *Source* and *Compile* menus, or by typing commands. We'll present both techniques below.

First, be sure that you are in the *test* directory by typing:

```
cd /test
```

3.1 Using the *Source* and *Compile* menus

The menus *Source* and *Compile* appear on the menu bar when the directory you are currently in contains *.asm* and *.c* files. The *Source* menu contains the names of all the *.asm* and *.c* files in the current directory; the *Compile* menu defines operations that can be performed on files listed in the *Source* menu, and options that will be used when performing the operations.

One of the files in the *Source* menu is defined to be the 'current' file; it is on this file that *Compile* menu operations are performed. You want to perform the operations on the *greet.c* file, so activate the *Source* menu and then select *greet.c* using the mouse. A selected file in the *Source* menu remains current until you explicitly select another one. The current file has a check mark beside it, as you can see by selecting the *Source* menu again.

Now select the *Compile* menu. Notice that the options *Auto Assemble*, *Auto Link*, and *Auto Run* are checked, indicating that after a program is compiled it will be automatically assembled, linked, and executed. Select the *Compile* item in this menu. It will be compiled, assembled, linked, and started, resulting in the message

```
greetings!!
```

appearing on the screen.

To illustrate the use of the options in the *Compile* menu, deselect the *Auto Link* and *Auto Run* options. Now when you select the *Compile* item in the *Compile* menu, *greet.c* will be just be compiled and assembled. To link the object module version of the current file, select the *Link* item in the *Compile* menu. Then to run the executable version of the current file, select the *Run* item.

3.2 Using the keyboard

You can also compile, assemble, link, and execute programs by typing commands. To compile and assemble *greet.c*, type

```
cc greet.c
```

This invokes the compile, and then the assembler. If you type the *ls* command, you will see that a new file, *greet.o* has been created. This is the object file created by the assembler.

To create an executable program, type:

```
ln greet.o -lc
```

This invokes the linker to link the *greet.o* module with the standard C library, and places the executable program in the file *greet.*

To execute the program, type:

```
greet
```

3.3 Cleaning up

To get rid of the files *greet.o* and *greet*, which you no longer need, type

```
rm greet.o greet
```

4. More menus

4.1 The *Apple* and *File* menus

The *Apple* menu lets you activate desk accessories such as the alarm clock and the puzzle. The menu has one special item, *About the SHELL*. Clicking it causes information about the SHELL to be displayed, such as its version number. To resume, once this information has been displayed, click the mouse.

When you select items from the *Apple* menu that generate their own window, their window appears in the foreground and the SHELL window is in the background. For example, select the Alarm Clock item. Notice that you now can't enter commands to the SHELL: the *Commands* menu is dimmed; you can look at items in the menu, but you can't select any of them. To bring the SHELL window into the foreground, click the mouse anywhere in the screen OUTSIDE the alarm clock box. The alarm clock program is still active, but its window has moved to the background and is invisible, since it is covered by the SHELL's window.

To bring windows in the background to the foreground, select the *File* menu and click the *See Windows* item.

To halt a program that was activated from the *Apple* menu and whose window is in the foreground, select the *File* menu and click the *Close* item.

4.2 The *Programs* menu

The *Commands* menu allows you to activate builtin commands whose names are listed in the menu, and the *Compile* command allows you to perform operations on the current file. The SHELL supports another menu, whose items you define, each of which is a command that will be executed when its item is clicked.

For example, suppose you frequently want to execute the command

```
ls -lt
```

which lists more information about the files in that directory in order of the time that they were last modified. To enter this command into the *Programs* menu, type:

```
set PRG_LIST='ls -lt'
```

Notice that the *Programs* menu has appeared. Select the *Programs* menu: it has a single item, the *ls -lt* command. If you click this item, the command will be executed.

You can enter multiple items in the *Programs* menu by typing the *set PRG_LIST* command, with the items' commands separated by semicolons. For example, typing:

```
set PRG_LIST='ls -lt;cd /'
```

will create a *Programs* menu containing the items

```
ls -lt
cd /
```

5. Where to go from here

In this chapter, we've just begun to describe the features of Aztec C68K. You should know enough now to create some simple programs, which you can do while continuing to read the rest of this manual.

In your reading, be sure to read the sections on the SHELL, compiler and linker. You should scan through the Utility Programs chapter, which describes in detail each of the builtin commands and command programs that are provided with Aztec C68K.

The Technical Information chapter also discusses several topics which might be of interest to you. For example, it talks about using Aztec C68K on systems that have a hard disk, on systems having a single drive, and on systems having 128K and 512K bytes of RAM.

Once you're accustomed to writing C programs with Aztec C68K, you can start writing C programs that access the special features of the Macintosh. For this, read the Toolbox chapter of this manual. This chapter is designed to accompany *Inside Macintosh*, and shows how your C programs can call the Macintosh toolbox and OS functions. Also, you can look at the source for the sample programs provided with Aztec C68K, which use the special features of the Macintosh.

THE SHELL

Chapter Contents

The SHELL

The SHELL is a program which provides an efficient and convenient environment in which to develop programs.

The basic function of the SHELL is to execute commands. You can enter commands by typing on the keyboard, or by selecting items from one of the SHELL's menus. When it finishes executing a command, the SHELL writes a prompt to the screen and waits for another command to be entered.

There are three types of commands: builtins, programs, and exec files. The operator doesn't have to specify the type of an entered command, just its name. When a command is entered, the SHELL first searches for a builtin command, and then for a program or exec file.

Builtins are commands whose code is built into the SHELL. To execute a builtin command, the SHELL simply transfers control of the processor to the command's code. When done, the command's code returns control of the processor to the main body of the SHELL.

Programs are commands whose code resides in a disk file. The name of a command is the name of the file containing its code. The SHELL executes a program by loading its code into memory, overlaying the SHELL, and then transfering control of the processor to the loaded code. When the program is done, the SHELL is automatically reloaded into memory and regains control of the processor.

Exec files are disk files containing text for a sequence of commands. The SHELL executes an exec file by executing each of the file's commands

This chapter is divided into two sections: the first discusses the file system that is implemented by the SHELL. The second discusses the features of the SHELL and shows you how to use the SHELL.

The *utilities* chapter describes the SHELL's builtin commands and the program commands that are provided with the Aztec C package.

1. The file system

Programs can access information contained on one or more disks, or 'volumes', as they're called in the rest of this manual. The information is contained in logical entities called 'files', each of which has a name. A single file is contained within one volume; that is, a file can't span several volumes.

The SHELL creates the illusion that the file system is a UNIX-type file system, in which each volume contains a hierarchy of directories: a root directory and, optionally, subdirectories, each of which has a name. A directory contains a number of entries, each of which describes a file or points to another directory. Files having entries in a particular directory are said to be contained in the directory, and the directories pointed at by entries within a directory are said to be subdirectories of that directory. A file is contained in exactly one directory, and a directory other than the root directory is a subdirectory of exactly one directory.

The name of a file or directory must be unique within the directory that contains it, but two files or directories that are in different directories can have the same name.

An example

For example, figure 1 depicts the organization of the volume named *sys:*. This volume contains the following directories:

* the root, which doesn't have a name;
* *include*, a subdirectory of the root;
* *subs*, a subdirectory of the *include* directory;
* *work*, a subdirectory of the root;
* *subs*, a subdirectory of the *work* directory.

Notice that there are two directories named *subs*. We'll describe below the naming convention for directories, which will make clear how a directory is uniquely identified.

The root directory contains the files *SHELL* and *finder*; that is, contains entries describing these two files. It also contains pointers to the *include* and *work* subdirectories.

The *include* directory contains the files *stdio.h* and *ctype.h* and a pointer to one of the *subs* directories.

The *subs* directory which is a subdirectory of the *include* directory contains just the file *in.c*.

The *work* directory contains the files *hello.c* and *hello.o* and a pointer to the other *subs* directory.

The *subs* directory which is a subdirectory of the *work* directory contains two files: *in.c* and *out.c*. The *in.c* file in this directory is different from the *in.c* which is in the other *subs* directory.

Advantages

The advantages of a hierarchical file structure such as the one simulated by the SHELL are:

* It allows the files on a volume to be partitioned into related groups, thus making it possible for a volume to contain many files without becoming chaotic;
* Related files can be easily examined and worked on together, thus allowing the operator to more efficiently and effectively manipulate and manage the information on a volume.

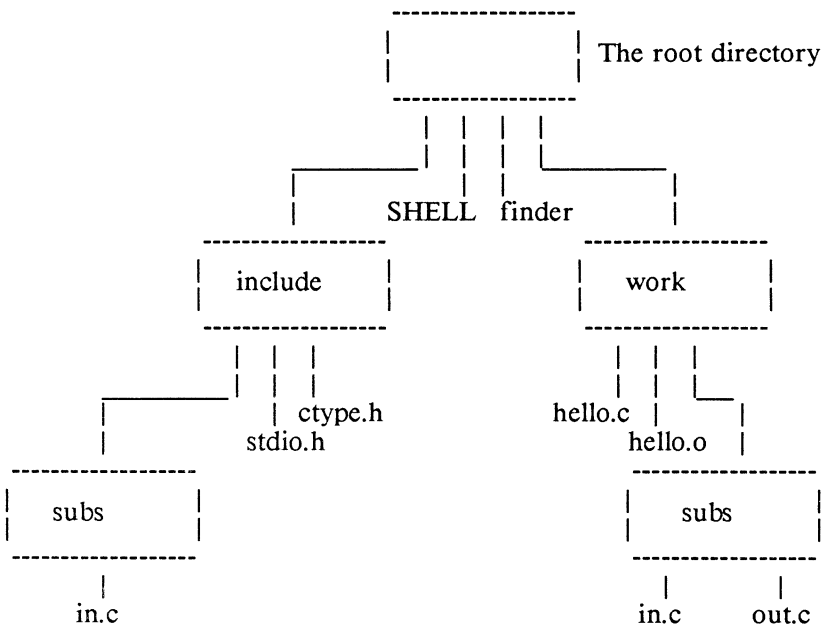


Figure 1: a sample volume, named *sys:*

1.1 File names

There are three parts to a file name which is accessed by the SHELL or a SHELL-activated program:

* The name of the volume on which it's contained;
* The path to the directory containing it;
* The file name itself.

For example, the complete file name of the file *in.c* in figure 1, which is in the *subs* directory, which is a subdirectory of the *work* directory, which is a subdirectory of the root directory, is:

sys:/work/subs/in.c

sys: is the volume name, */work/subs/* is the path identifier, and *in.c* is the file name.

The following paragraphs describe the naming convention in detail.

Volume names

The name of a volume is assigned by the operator, using the Macintosh Finder program.

A volume which is in a drive can also be referred to using the number of the drive instead of the volume name. The internal drive is 1 and the external 2. For example, if the *sys:* volume is in the internal drive, the file used in the above example could also be referred to as

1:/work/subs/in.c

Path identifiers

The path component of a file name specifies the directories which must be passed through to get to the directory containing the file. It is a list of the directory names, with each pair separated by a forward slash character, /. The root directory doesn't have a name, and is represented by a null string.

For example, the paths to the directories used in figure 1 are:

(null) path to the root directory;

/include path to the *include* subdirectory of the root directory;

/include/subs path to one of the *subs* directory, which is a subdirectory of the *include* directory;

/work path to the *work* directory, which is a subdirectory of the root directory;

/work/subs path to the other *subs* directory, which is a subdirectory of the 'work' directory.

Each directory can be reached from the root directory by passing through a unique path of directories. This is why two directories which are subdirectories of two different directories can have the same name and still be uniquely identified: the path to each one is different.

Filenames

A filename can contain any printable ASCII characters. By convention, the Manx programs assume that a file name contains a main part, usually called the "filename", optionally followed by a period and an extension. With this convention, related files can have the same basic filename, and different extensions. Extensions used by the Manx software are:

extension	*file contents*
.c	C source
.asm	assembler source
.o	relocatable object code
.sym	symbol table for an executable file
.lst	assembler listing

By default, the file created by the linker which contains executable code has no extension.

For example, the C source code for the "hello, world" program might be put in a file named *hello.c*. The file containing the relocatable object code for this program would by default be named *hello.o*, and the file containing the executable code for the program would be named *hello*.

Complete Filenames

A complete file name has the form

volume:path/filename

That is, the volume name comes first, followed by a colon; then comes the path, followed by a slash; then comes the file name.

Thus, the complete names of some of the files in figure 1 are:

sys:/Finder
sys:/include/stdio.h
sys:/include/subs/in.c
sys:/work/hello.c
sys:/work/subs/in.c

The SHELL and SHELL-activated programs don't distinguish between upper and lower case letters in volume, directory, and file names. When a volume, directory, or file is created, the name is recorded exactly as entered, but to refer to it, the operator or program doesn't have to worry about the case of its letters. For example, if the complete name of a file is

Sys:/Progs/Finder

then it could be referenced by names such as

sys:/progs/finder
SYS:/PROGS/FINDER

Volume, directory, and file names can contain any printable ascii characters, including spaces. If a name contains spaces, references to it must surround the name of which it is a part with either single or double quotes. For example, if a volume is named *data disk:* then the file *hello.c* in the directory */source* would be referenced by the quoted string

"data disk:/source/hello.c"

Using the number of the drive containing a volume instead of its name can be convenient in cases where the volume name has spaces. For example, if "data disk:" is in the external drive, then the file referred to above could also be referred to as

2:/source/hello.c

The length of a file name plus the path to it must be less than 64 characters.

Frequently, the complete file name needn't be given to identify a file. The file can be located relative to a directory called the 'current directory', thus allowing the volume and/or the path to be omitted from the file name. This is discussed below.

1.2 The current directory

Having to specify the complete name of each file you want to access would be very cumbersome. Also, when developing programs, at any time, you are generally interested in the files on a single directory. For these reasons, the SHELL allows one directory, called the 'current directory', to be singled out.

When the SHELL is first started, the root directory on the volume containing the SHELL is the current directory; there is also a command, *cd*, which allows the operator to make another directory the current directory.

A file on or near the current directory can be specified by the operator or program without having to list the complete name of the file:

* If the name doesn't specify the volume or the path, the file is assumed to be in the current directory.
* If the name specifies a path, but not a volume, the file is assumed to be in the specified directory on the current directory's volume.

* If the name doesn't specify a volume, and doesn't specify a path which begins at the root, the path is assumed to begin with the current directory.

For example, suppose that the current directory on the volume depicted in figure 1 is *work*. The complete name of the file *hello.c* in this directory is

sys:/work/hello.c

Since this file is in the current directory, the operator or a program can refer to it without the volume or path; that is, simply as

hello.c

Since the directory */include/subs* is on the same volume as the current directory, the file *in.c* within this directory can be identified without a volume name; that is, as

/include/subs/in.c

Since the directory */work/subs* is a subdirectory of the current directory, the file *out.c* within this directory can be identified without a volume name, and with only a partial path name; that is, as

subs/out.c

As a further abbreviation, if a file name specifies a volume and a path, the path is assumed to begin with the root directory on the specified drive. Thus, the leading '/' in the path, which normally separates the null root directory name from the next directory name or filename, is optional. For example, the following two file names both identify the *Finder* file on the root directory of the volume *sys:*.

sys:/Finder
sys:Finder

And the following two names both identify the file *stdio.h* in the *include* directory:

sys:/include/stdio.h
sys:include/stdio.h

1.2.1 The '.' directory

The current directory can be referred to using the character '.'. For example, the following command will copy the file *hello.c* in the directory *sys:/source* to the current directory:

cp sys:/source/hello.c .

1.2.2 The '..' directory

The parent directory of the current directory can be specified using two periods as the path name. For example, in figure 1, with the *work* directory as the current directory, the file *Finder* could be referred to

as

../Finder

and the file *ctype.h* in the directory *include* could be identified as:

../include/ctype.h

1.3 Directory-related builtin commands

The SHELL has several builtin commands for examining and manipulating directories: pwd, cd, and ls. We want to introduce these commands in this section; complete descriptions are presented in another section of the manual.

pwd

This command, whose name is a mnemonic for 'print working directory', displays the name of the current directory.

cd

This command makes another directory the current directory. If the new directory doesn't exist, it is created; in this case, *cd* prints the message 'Empty Directory'.

The command has one argument, which specifies the volume on which the directory is located and the path to it.

The volume name is optional; if not specified, the directory is assumed to be on the current directory's volume.

The path has the same format as the path component of a file name.

For example, considering the volume *sys:* in figure 1, with *work* being the current directory, the following *cd* commands change the current directory as indicated:

command	*new current directory*
cd /include	/include
cd subs	/work/subs
cd ..	/ (the root directory)
cd objects	/work/objects (created by cd)

A directory exists only when files are created in it. Hence, if you 'create' the *objects* subdirectory of the *work* directory, as shown in the last example above, and then move to another directory without creating any files in *objects*, the *objects* directory would cease to exist.

ls

ls displays the names of files and the contents of the directories whose names are passed to it.

The format is:

ls [-l] [name] [name] ...

where square brackets indicate that the enclosed field is optional.

-l causes **ls** to display information about the files or directories in addition to their names.

The *name* arguments are the names of the files and directories of interest. If no 'name' arguments are specified, the command displays information about the current directory.

To specify a directory, include a slash character at the end of its name.

For example, the following displays the names of the files and directories in the current directory:

ls

The following displays information about the files and directories in the current directory:

ls -l

The following displays the names of the files and directories contained in the /include directory:

ls /include/

The following displays information about the file *in.c* in the directory *vol:/john/progs*:

ls -l vol:john/progs/in.c

For more information about the *ls* command, particularly about the information displayed when the '-l' option is used, see the description of *ls* in the utilities chapter.

1.4 Accessing files on several volumes

The SHELL allows multiple volumes and disk drives to be accessed in the development and execution of programs.

It's even possible to develop and run programs when more volumes are required than the number of available disk drives. When a request is made to access a file contained on a volume which isn't in a disk drive, the disk in the internal drive is ejected and a message is displayed on the screen, prompting the operator to insert the required disk in the drive. When this is done, the program continues automatically.

For example, if you have a single drive on the Macintosh, you will probably develop programs using at least two volumes: the first could contain the development software, such as the SHELL, the compiler, assembler, linker, libraries, and text editor. The second could contain

your own files: source files, executable files, data files, and so on.

Continuing with this example, suppose you want to create a source program on the data volume, and that this volume is in the internal drive. Type 'z' to activate the text editor of that name. Since the file containing the editor is on the system volume, the data volume is ejected from the internal drive, and you will be prompted to insert the system volume. When this is done, Z is loaded into memory and activated. When Z attempts to access the data volume, the system volume will be ejected and you'll be prompted to insert the data volume. When this is done, Z let's you create and edit the source program and access the data volume. When you exit Z, the SHELL needs to be reloaded into memory from the system volume, so the data volume is ejected and you are prompted to insert the system volume. When this is done, the SHELL is loaded and activated, and prompts you for another command.

1.4.1 Mounted volumes

The only volumes that can be accessed by the SHELL or other programs are those that are 'mounted'; that is, those that have an entry in the Macintosh's mounted volume table.

A volume doesn't have to be in a drive to have an entry in this table: once in the table, an entry remains there, independent of the presence or absence of the volume in a drive.

An entry in the table is made for a volume when the volume is inserted for the first time in a drive. An entry is removed from the table using the SHELL command *umount*.

1.4.2 Volumes having the same name

It's possible to have several volumes having the same name contained in drives. In this case, each volume would be identified using the number of the drive containing the volume rather than its name.

For example, if two volumes are in the internal and external drives, and both have the name *sys:*, the file */work/hello.c* in the volume contained in the internal drive would be referred to as

1:/work/hello.c

and the same file in the external drive would be referred to as

2:/work/hello.c

1.4.3 Commands for multi-volume use

The SHELL has several commands useful for multi-volume development: *mount*, *umount*, and 'eject'. See the descriptions of these functions in the Utility programs chapter for complete information.

mount

This command displays information about the volumes in the mounted volume table.

umount

This command removes an entry from the mounted volume table. It has the format:

umount vol:

where *vol:* is either the name of the volume to be ejected or the number of the drive. The internal drive is 1 and the external 2.

'eject'

This command ejects the disk that's in a specified drive. The command is activated by pressing the key with the clover symbol and then the number of the drive.

This command doesn't affect the mounted volume table.

1.5 Miscellaneous file-related commands

In this section we want to list the rest of the file-related commands that are built into the SHELL. For complete descriptions, see the utilities chapter.

rm	-	Remove files
cp	-	Copy files
mv	-	Move files. This will either rename the files or copy them and erase the originals, depending on whether the old and new files are on the same volume.
cat	-	Display text files.
lock/unlock	-	Lock/unlock files.
flock/funlock	-	Lock/unlock files for the Finder.

1.6 Implementaion of the SHELL's file system

The SHELL presents the illusion that the file system is hierarchical; that is, that the file system has one or more directories having the following properties:

* One directory is the root directory;
* Each of the others is pointed at by an entry in one other directory;
* A path exists from the root to any directory.

In a true hierarchical file system, each directory would be physically separate on the disk. There is actually only one directory on a disk which is accessed by the SHELL: the standard Macintosh directory.

For disks accessed by the SHELL, the true name of a file, as recorded in the real directory on the disk, consists of the complete file name, in the standard SHELL format, less the volume name and the path's leading slash.

For example, the name recorded in the Macintosh directory for the file *hello.c* in directory *work* in the volume depicted in figure 1 would be

work/hello.c

And the name recorded for the file */include/subs/in.c* would be

include/subs/in.c

Folders

The Macintosh Finder program also presents the illusion of a hierarchical file system, using folders. The Finder implements this illusory file system differently than the SHELL, using the real directory on the disk to store information about files and using a section of the 'desktop' file to store information about folders and their interrelationships.

The name of a file, as recorded in the file's entry in the directory, is the same as the name by which the operator or program references it. The entry also contains a field specifying the folder, if any, containing the file.

A disadvantage of the Finder's scheme for simulating a hierarchical file system is that files cannot have the same name, even if they are in different folders. The SHELL's scheme doesn't have this limitation, since the path to a file is part of a file's name, as recorded in the real directory. Files in different directories are reached by different paths, and hence have different filenames recorded in the real directory, even if they appear to have the same name when viewed from the SHELL.

2. Using the SHELL

The previous section presented information on the SHELL's file system, which you need to know before you can use the SHELL. With that information in hand, you can continue on with this section, which shows you how to use the SHELL.

2.1 Simple Commands

You can enter commands to the SHELL in two ways: by typing on the keyboard, and by selecting items from one of the SHELL's menus. In this section we describe keyboard entry of simple commands. Menus are described in another section of this chapter.

Simple commands consist of one or more words separated by blanks. The first word is the name of the command to be executed; the other words are arguments to be passed to the command. The name of the command is always passed to a command as an argument. For example,

```
ls
```

lists the names of the files and directories that are in the current directory. The first word on the command line, *ls*, is the name of the command. No other words are specified, so the only argument passed to the 'ls' command is the name of the command.

The *ls* command can also be passed arguments; the command

```
ls sys:/bin/
```

displays the names of the files and directories in the directory named */bin* on the volume *sys:*. The first word on this command line, *ls* is the name of the command to be executed. Two words are passed to the *ls* command as arguments: *ls* and *sys:/bin/*.

The command

```
rm hello.bak sys:temp /include/head.o
```

removes the files *hello.bak*, *sys:temp*, and */include/head.o*. The name of this command is *rm*. Four words are passed to it as arguments: *rm*, *hello.bak*, *sys:temp*, and *include/head.o*.

The command

```
ls -l /include/
```

displays the names of the files and directories in the directory */include* on the current volume. The '-l' causes the *ls* command to display other information about the files and directories in addition to their names. For this command, three words are passed to the *ls* command: *ls*, *-l*, and */include/*.

The meaning of the arguments following the command name on a command line is particular to each command. Usually, either they are 'switches', indicating a particular command option, as in the *ls -l /include/* command above, or they are file names. By convention, switches usually precede file names in a command line, although there are exceptions to this.

2.2 Pre-opened I/O channels

When a builtin command or command program is started by the SHELL, three I/O channels are automatically pre-opened for it by the SHELL: standard input, standard output, and standard error. By default, these channels are connected to the console, and most programs use these devices when communicating with the operator. For example, the *ls* command displays information about files and devices on the standard output channel and writes error messages to the standard error channel.

2.2.1 Standard output

The operator can request that the standard output channel be pre-opened to another file or device other than the console by including a phrase of the form '> name' on the command line . For example, the following command causes *ls* to write information about the files and directories in the current directory to the file *files.out*, instead of the console:

```
ls > files.out
```

If the specified file doesn't exist, it is created; otherwise, it is truncated to zero length.

The standard output channel can also be redirected so that output to a file via the standard output is appended to the file. This is done by including a phrase of the form '>> file' on the command line. For example, the following command causes *ls* to append information about the files and directories in the current directory to *files.out*:

```
ls >> files.out
```

If the specified file doesn't exist, it is created; otherwise it is opened and positioned at its end.

2.2.2 Standard input

The operator can request that the standard input device be pre-opened to a file or device other than the console by including a phrase of the form '< name' on the command line. For example, if the program *prog* reads from the standard input channel, then the command

```
prog
```

causes *prog* to read from the console, and the command

```
prog <names.in
```

causes it to read from the file *names.in*.

2.2.3 Standard error

A program's standard error channel can also be redirected to another file or device other than the console, by including a phrase of

the form:

2> name

where *name* is the name of the device or file to which standard output is to be connected.

For example, the following causes *ls* to display the names of all files in the directory *sys:/work* having extension *.c*. The names are sent to the file *ls.out* in the current directory and any error messages are sent to the printer, .bout:

ls sys:/work/*.c >ls.out 2>.bout

2.2.4 Other I/O channels

Channels other than standard input, standard output, and standard error can be pre-opened for a program. The channel having file descriptor *i* is pre-opened for output to a device or file named *name* by including the phrase

i> name

on the command line. And it's pre-opened for input by including

i< name

on the command line.

For example, the following command pre-opens the channel having file descriptor 3 for output to the file *info.out*:

prog 3>info.out

2.2.5 Creating empty files

The SHELL allows you to enter a command line containing only I/O redirection components. In this case, the SHELL processes the I/O redirection clauses and then reads another command line.

Such a command line can be used for recording the time at which events occur. For example, the command

> mytime

creates an empty file named *mytime*. The *last-modified* field for this file is set to the time at which it was created.

make is a program in which a command line that simply creates an empty file can be useful. For example, you could create a makefile to backup all files that have been modified since the last time a backup was done. This makefile could create an empty file that records the time of the last backup. Like this:

```
CFILES=main.c in.c out.c sub.c add.c
backtime: $(CFILES)
    > backtime
$(CFILES):
    cp $@ backup:
```

2.3 Expansion of file name templates

When the characters '?' and/or '*' appear in a command line argument, the SHELL interprets the argument as a template to be matched to file names. Each matching name is passed to the program as a separate argument, and the template isn't passed. If the template doesn't match any file names, it is passed to the program, unaltered.

These characters can only be used within the filename component of a file name, and not the volume or path components.

2.3.1 The '?' character

The character '?' in a template matches any single character. For example, the command

```
rm ab?d
```

would remove files in the current directory whose names are four characters long, the first two being 'ab' and the last being 'd'. Thus, it would remove files with names such as

```
abcd  abxd  ab.d
```

from the current directory.

Continuing with this example, if the three files listed above were the only ones in the current directory that matched the template "ab?d", then pointers to those three names are passed to the *rm* command in place of a pointer to the template. So the *rm* command would behave as if the operator had entered

```
rm abcd abxd ab.d
```

If no files matched the template, a pointer to the template itself would have been passed to *rm*.

Notice that the template "ab?d" matches "ab.d". This emphasises the fact that extensions in file names, and their preceding period, are simply conventions and are not afforded special treatment by the SHELL, as they are in some other systems.

2.3.2 The '*' character

The character '*' matches any number of characters, even none. For example,

```
rm /work/ab*d
```

removes all files in the */work* directory whose names begin with the characters 'ab' and end with 'd'. Thus, it would match files in the */work/* directory having names such as

```
abd  abcd  ab123d  ab.exd
```

As with templates containing '?', the names of files which match a template containing '*' are passed to the program, each as a separate

argument, and the template isn't passed. The template is passed only if no files match it. Thus, if the files listed above were the only ones that matched the template, then the following would have been equivalent to 'rm /work/ab*d':

```
rm /work/abd /work/abcd /work/ab123d /work/ab.exd
```

The use of '*' templates can be dangerous. For example, if you wanted to type

```
rm abc*
```

but mistyped it as

```
rm abc *
```

then *rm* will remove "abc", if it exists, and then remove all other files in the current directory.

2.4 Quoting

Characters such as *, <, and > are special, because they cause the SHELL to perform some action and are not normally passed to a program. There are occasions when you want such characters to be passed to a program without having the SHELL interpret them. This can be done by preceding the character with a backslash character, '\'. Any character can be preceded by a backslash; when the SHELL encounters '\' in a command line it removes the backslash from the line and treats the following character as a normal character, without attempting to interpret it.

For example, the command

```
echo *
```

displays the names of all files and directories in the current directory on the console. The command

```
echo \*
```

displays the character '*' on the console.

The backslash character and multi-line commands

The backslash character can also be used to enter long command lines on several physical lines. Normally, a newline character causes the SHELL to terminate the reading of a command line and to begin execution of the command. When the newline character is preceded by a backslash, the SHELL removes both characters from the command line and continues reading characters for the command line. For example,

```
echo abc\
def
```

displays 'abcdef' on the console.

When the SHELL needs additional input from the console before it can execute a command, it will prompt you with its secondary prompt. By default, this is the character '>'. The primary prompt, which is displayed when the SHELL is ready for a new command, is by default '-?'. Prompting is discussed in more detail below.

Quoted strings

A string in the command can be surrounded by single quotes. In this case, the SHELL considers the entire string within the quotes to be a single argument. The SHELL doesn't try to interpret any special characters contained in a string that is surrounded by single quotes.

For example, given a volume named "ralphs disk", the following command will make its root directory the current directory:

```
cd 'ralphs disk:'
```

As another example, consider a program, *args*, which prints the arguments passed to it, each on a separate line. The command

```
args 123 234 345
```

would print

```
args
123
234
345
```

(the command name is passed to the program as an argument), while the command

```
args '123 234 345'
```

would print

```
args
123 234 345
```

The command

```
args *
```

would print the names of each of the files on the current directory, each on a separate line, while

```
args '*'
```

would print the character '*'.

A quoted string can contain newline characters. That is, if the SHELL sees a quote character and then reads a newline character before finding another quote, it will keep prompting for additional input until it finds another quote. The argument corresponding to the quoted string then consists of the string with the newline characters still imbedded in it.

For example, if you enter

```
echo 'ab
```

the SHELL will prompt you for additional input, using its secondary prompt. If you then enter

```
1
2
3'
```

the echo command will be activated with arguments

```
echo
ab\n1\n2\n3
```

(where '\n' stands for the newline character) and will print

```
ab
1
2
3
```

Double-quoted strings

A string on the command line can also be surrounded by double quotes. The only difference in the treatment of singly- and doubly-quoted strings by the SHELL is that variable substitution is done for double-quoted strings but not for single-quoted strings. This is discussed in detail in the section on environment variables.

2.5 Prompts

The SHELL prompts you when it wants you to enter information, by writing a character string, called a 'prompt' to the console. There are two types of prompts: one when the SHELL is waiting for a new command to be entered, and the other when it needs additional input before it can process a partially-entered command.

2.5.1 The primary prompt

The first type of prompt is called the 'primary' prompt. By default, it is the string '-?'. This can be changed by entering the command of the form

```
set PS1=prompt
```

where 'prompt' is the desired prompt string. For example,

```
set PS1='>>'
```

sets the primary prompt to '>>'. Note the single quotes surrounding >>. These are necessary to prevent the SHELL from trying to interpret these special characters.

```
set PS1='hi there, fred. please enter a command: '
```

sets the primary prompt to the specified, space-containing string.

2.5.2 The secondary prompt

The second type of prompt is called the 'secondary' prompt. By default, it is the string '>'. This can be changed by entering a command of the form

```
set PS2=prompt
```

2.5.3 The command logging prefix

When command logging is enabled, the SHELL logs each command to the console, and precedes it with a character string called the 'command logging prefix'. By default, this prefix is the character '+', and can be set by entering a command of the form

```
set PS3=prefix
```

2.5.4 Special substitutions

The prompts and prefix described above can contain codes that cause variable information to be included in a prompt. The codes consist of a lower case letter preceded by the character '%'. For example, to set the primary prompt to the time, followed by ' :' enter

```
set PS1='%t :'
```

The list of letters and their substituted values are:

letter	*substituted value*
d	Date
t	Time
v	Current volume
c	Current directory
h	Amount of free space available in the system area
m	Amount of available program memory

2.6 Selecting screen fonts

The Macintosh allows you to control the appearance of characters that programs write to the screen. The Aztec C software distinguishes between the appearance of SHELL output to the screen and that of other programs. This section first discusses SHELL output, and then that of other programs.

2.6.1 SHELL output to the screen

The SHELL has commands that allow you to select the font, size, and face of characters that the SHELL writes to the screen. By default, characters that the SHELL writes to the screen are displayed in the System Font.

2.6.1.1 Selecting fonts

The font used for SHELL output to the screen is selected with the command

set FONT=code

where *code* is the code of the font. The available fonts and their associated codes are:

code	*name*
0	System Font (Chicago)
1	Application font (Geneva)
2	New York
3	Geneva
4	Monaco (fixed pitch)
5	Venice
6	London
7	Athens
8	San Francisco
9	Toronto

2.6.1.2 Selecting character size

The size of characters that the SHELL writes to the screen is selected with the command

set SIZE=val

where *val* is the size of the characters, in points.

2.6.1.3 Selecting faces

The face of characters that the SHELL writes to the screen is selected with the command

set FACE=code

where *face* can have the following values:

code	*face*
1	Bold
2	Italics
4	Underline
8	Outline
16	Shadow
32	condensed
64	Extended

Several faces can be selected at one time by adding their codes together. For example, to select Bold and Underline faces, the following command would be used:

```
set FACE=5
```

2.6.2 Program output to the screen

The font, size, and face of characters that programs write to the screen is independent of that used for the SHELL. By default, a program's output to the screen appears in the System Font.

2.6.2.1 Selecting the style of a program's screen output

If a program wants its screen output to appear in a style other than the default, it must issue the appropriate Macintosh function calls.

The font, size, and face used for SHELL output to the screen are stored in the environment variables FONT, SIZE, and FACE, respectively. Thus, if a program wants its screen output to use the same style as the SHELL's, it can fetch these environment variables, using the *getenv* function, and then issue the appropriate Macintosh function calls. If these variables don't exist, then the SHELL is using the System Font.

2.6.2.2 Screen output style for standard Manx programs

The *ls* and *mount* commands always display in Monaco font so that things line up. They will use the size and face specified by the SIZE and FACE environment variables.

The *cc, as,* and *ln* programs display in 10 point Monaco so that the tables needed for other styles do not take up memory space.

Z only supports the Monaco font. It does support different sizes using the ZSIZE environment variable. This can be either 9 or 12. With a setting of 9, the screen will have 30 lines, each containing 85 characters. With a setting of 12, it will have 20 lines, each containing 63 characters.

2.7 The program's view of command line arguments

In this section we want to describe the passing of arguments to command programs, first for programs that can be activated by the SHELL but not by the Finder and then for programs that can be started by either the Finder or the SHELL.

Programs of the first type have been linked with the startup routine *shcroot*, which is in the standard library *c.lib*, and not with a special startup routine such as *sacroot* or *mixcroot*. Programs of the second type have been linked with one of the special startup routines.

For more information on the different types of Aztec-generated programs, see the *Command Programs* section of the *Technical Information* chapter.

2.7.1 Passing arguments to programs that can't be activated by the Finder

The *main* function of a program is the first user-written function to be executed when the program is started. The *main* function of a program that has been linked with *shcroot* is passed two arguments, as follows:

```
main(argc, argv)
int argc; char *argv[];
```

argc contains the number of command line arguments passed to the program. The command itself is included in the count.

argv is an array of character pointers, each of which points to a command line argument.

For example, if the operator enters the command

```
prog abc def ghi
```

then the *argc* parameter to *main* will be set to 4, and the *argv* array is set as follows:

argv element	*points to*
0	"prog"
1	"abc"
2	"def"
3	"ghi"

As another example, for the command

```
prog "abc def ghi"
```

argc is set to 2, and the *argv* array as follows:

argv element	*points to*
0	"prog"
1	"abc def ghi"

With the command

prog *.c

and the current directory containing the files

a.c a.o a b.c

argc will be set to 5, and the *argv* array as follows:

argv element	*points to*
0	"prog"
1	"a.c"
2	"a.o"
3	"a"
4	"b.c"

2.7.2 Passing arguments to programs that can be activated by the Finder

A program that can be activated by the Finder can also be activated by the SHELL. When the SHELL starts such a program, the program is started as though a document has been double-clicked, thus allowing arguments to be passed to it.

Arguments that are specified in the command line are passed to the program using the standard Macintosh convention, rather than using C conventions; that is, the names of the file or files that the program is to use are placed after the name of the program. See your Macintosh documentation for more details.

For example, typing

MacWrite include/quickdraw.h myprog.c

invokes the *MacWrite* program with the two files *quickdraw.h* and *myprog.c*. If the files do not exist, an error is generated.

2.8 Devices

Programs can access the following devices:

* The console, named *.con*
* The input channel of the *A* serial port,*.ain*
* The output channel of the *A* serial port, *.aout*
* The input channel of the *B* serial port,*.bin*
* The output channel of the *B* serial port, *.bout*

For example, the following command copies the output of the *ls* command to the printer, which is attached to the B serial port:

```
ls > .bout
```

2.8.1 The keyboard

When a program is reading from the keyboard, some translations are performed:

* ‘ is translated to the escape character, ESC;
* The key next to the 'Option' key is interpreted as the control key; thus, holding down this key and typing another key generates the appropriate control character.
* The only exception to the interpretation of the control key concerns the ‘ key: when the control key and the ‘ key are depressed, ‘ is returned.

When I/O is being performed to the console, the following checks are made:

* If the control key and a number key are depressed, the disk is ejected from the corresponding drive (1 is the internal drive, 2 the external);
* If the control key and the 'S' key are depressed, the console driver waits until control and 'Q' are depressed before continuing;
* If the control key and the 'D' key are depressed, the program is returned EOF;
* If the control key and the 'X' key are depressed, the SHELL deletes the current line and waits for another line to be entered. The character that causes this action can be changed by a program, using the *ioctl* function;
* If the control and '.' keys are depressed together, the program is halted, and the SHELL reloaded.
* If the backspace key has been depressed, the previously-typed character is erased from the screen. The character that causes this action can be changed by a program, using the *ioctl* function.

2.8.2 The printer

Before a program can write to the printer, the printer must be initialized to generate a line feed automatically, following a carriage return, and to correctly respond to tab characters.

The program *prsetup* will perform this function. The program is started with:

```
prsetup [tabwidth]
```

where [tabwidth] is an optional number specifying the spacing between tab stops. If tabwidth isn't entered, it defaults to 4.

2.9 Error trapping

The SHELL can trap the following Macintosh system errors:

bus error
address error
illegal instruction
divide by zero
line 1111 (unimplemented op code)

By default, the SHELL won't trap these errors. In this case, one of these errors will cause the Macintosh to bomb, and the system will have to be reloaded (thus requiring the SHELL to search for the key disk again). In addition, the SHELL will also trap the programmer's switch if installed on the Macintosh.

To have the SHELL trap these errors, enter:

```
set -a
```

Once the SHELL is trapping these errors, it won't stop trapping them. You can enter

```
set +a
```

but it won't have any effect.

When an error is trapped, the SHELL displays on the screen the contents of the registers and the error type.

Don't use *set -a* if you are using any of the Apple Debuggers.

2.10 Exec files

An "exec file" is a file containing a sequence of commands. The operator causes the SHELL to execute the commands in an exec file by simply typing its name.

For example, if the file named *dir* in the current directory contains the commands

```
pwd
ls -l
```

then when the operator types

```
dir
```

the SHELL will execute the commands *pwd* and *ls -l*.

An exec file can contain any command that can be entered from the console. In particular, an exec file can execute another exec file; that is, exec files can be nested.

2.10.1 Exec file arguments

The command line that activates an exec file looks just like a command line that activates a builtin or program command. Exec files can be passed arguments in the same way that builtin and program commands are passed arguments:

* a space-delimited string is normally passed to the exec file as a single argument;
* A quoted string is passed as a single argument;
* Filename-matching templates, containing '?' and '*', are replaced, when a match is made, by the matching file names;
* '\' causes the next character to be passed to the exec file without interpretation, and the '\' isn't passed. '\\' is replaced by a single backslash character.

The method by which an exec file accesses command line arguments is necessarily different from that used by builtin and program commands, since the exec file is not a program. The exec file can be passed any number of arguments, and it refers to them as $1, $2, ..., where $1 represents the first argument, $2 the second, and so on. $0 refers to the name of the exec file.

Before executing a command in an exec file, the SHELL replaces the $x variables with the corresponding command line arguments. $x variables which don't have a corresponding argument are replaced by the null string.

For example, the following exec file displays the value of the first, fourth, and ninth arguments, and the name of the command itself, each on a separate line:

```
echo the first argument is $1
echo the fourth argument is $4
echo the ninth argument is $9
echo and me, I'm $0
```

If the exec file is named *names* then

```
names a b c d e f g h i j
```

would print

```
the first argument is a
the fourth argument is d
the ninth argument is i
and me, I'm names
```

and the command

```
names *
```

would display the names of the first, fourth, and ninth files in the current directory, and the name of the command.

The command

```
names "this is one argument"
```

would print

```
the first argument is this is one argument
```

The $# variable

Several other variables are set when an exec file is activated. **$#** is set to the number of arguments that were passed to the exec file. For example, an exec file named *hello* might contain

```
echo My name is $0
echo I was run with $# arguments
```

Typing

```
hello one two three
```

would print

```
My name is hello
I was run with 3 arguments
```

The $* and $@ variables

$* and $@ are two other variables that are set when an exec file is activated. Both of these are set to a character string consisting of all the exec file's arguments, less $0. For example, consider an exec file *allargs*, which contains

```
args $*
```

where *args* is a command program that prints its arguments, each on a

separate line. Typing

```
allargs one two three
```

would give

```
args
one
two
three
```

Exec file variables and quoted strings

When an exec file variable is contained within a character string surrounded by single quotes, the SHELL does not replace the variables with their values. Thus, given the exec file *info*, which contains

```
echo 'number of args = $0'
echo 'args = $0 $1 $2'
echo 'all args = $* and $@'
```

then typing

```
info one two three
```

gives

```
number of args = $0
args = $0 $1 $2
all args = $* and $@
```

As mentioned in section 2, the SHELL does substitute variables that are contained within character strings that are surrounded by double quotes. Thus, the exec file

```
args "$*"
```

will pass the exec file arguments to echo as a single argument and is equivalent to

```
args "$1 $2 $3 ..."
```

$* and $@ are the same, except when surrounded by double quotes.

The exec file

```
args "$@"
```

is equivalent to

```
args "$1" "$2" ...
```

2.10.2 Exec file options

There are three options related to exec files: logging of exec file commands to the screen, continuation of an exec file following execution of a command which terminates with a non-zero exit code, and execution of commands.

Each option has an identifying character. An option's value is set by issuing a *set* command, giving the option's character preceded by a minus or plus sign. Minus enables an option and plus disables it.

The options, their identifying characters, and their default values are listed below:

character	*option*	*default*
x	log commands	disabled
e	abort on non-zero	enabled
n	don't execute cmds	disabled

Several options can be enabled or disabled in a single *set* command, and an exec file can contain several option-setting commands.

The same *set* command is used to set exec file options and to set environment variable values. *set* commands which set environment variables can also be contained in an exec file. However, a single *set* command cannot set both environment variables and exec file options.

When the SHELL logs exec file commands to the console, it precedes each command line with the character '+'. This prefix can be changed by entering a command of the form

```
set PS3='string'
```

where 'string' is the desired prefix.

The following are valid *set* commands for manipulating exec file options:

```
set -x          enable logging
set +x          disable logging
set -x -n       enable logging and non-execution of cmds
set -x +e       enable logging, disable return code chk
```

Exec file options are inherited by a called exec file. That is, if you type

```
set -x
docmds
```

where *docmds* is an exec file, the 'x' option is enabled in *docmds*.

An exec file can change the setting of the exec file options, but these changes don't affect the settings of the options in the caller. Thus, if *docmds* includes the command

```
set +x
```

then the 'x' option will be disabled during the execution of *docmds*, but when control returns to the operator, the 'x' option is reenabled.

2.10.3 Comments

In an exec file, any line beginning with the character '#' is considered to be a comment, and is not executed. Argument substitution is performed on it, though, allowing exec files like:

```
set -x
# the first arg is $1
# the second is $2
```

2.10.4 Loops

Exec files can contain 'loops'; that is, sequences of commands that are executed repeatedly, each time with an environment variable assigned a different value.

A loop has the format

```
loop var in varlist
cmdlist
eloop
```

where

var	is the name of the environment variable;
varlist	is the list of values for **var**;
cmdlist	is the sequence of commands.

Within the sequence of commands, the term *$var* is replaced by the current value of *var.*

For example, the following exec file compiles the C source files whose names are passed to it (without the '.c' extension):

```
loop prog in $*
echo compiling $prog
cc $prog
eloop
```

The list of variables is not restricted to exec file variables. For example, the following exec file, named *getname*, executes the program *prog* for each name passed to the exec file, and to the names 'Fred', 'Joseph', and 'R. W. Jones':

```
loop name in Fred Joseph 'R. W. Jones' $*
prog $name
eloop
```

This example also demonstrates that quoted strings can be assigned to the SHELL variable.

2.10.5 The *shift* command

The command

```
shift
```

causes the exec file variable $1 to be assigned the value of $2, $2 to be assigned the value of $3, and so on. The original value assigned to $1 is lost. When all arguments to the exec file have been shifted out, $1 is assigned the null string.

For example, the following exec file, *del*, is passed a directory as its first argument and the names of files within the directory that are to be removed:

```
set j = $1
shift
loop i in $*
rm $j/$i
eloop
```

In this example, 'j' is an environment variable. Environment variables are described in the section on environment variables, so you may want to reread this section after reading that section.

The first two statements in the exec file save the name of the directory and then shift the directory name out of the exec file variables.

The loop then repeatedly calls *rm* to remove one of the specified files from the directory.

Entering

```
del sys:/work *.bak
```

will remove all files having extension *.bak* from the directory *sys:/work*.

2.11 Environment variables

An environment variable is a variable having a name and having a character string as its value. Environment variables have two functions:

* They can be used to pass information to a program;
* They can be used to represent character strings within command lines.

Information can also be passed to programs as command line arguments, as described in a previous section.

2.11.1 Defining environment variables

Environment variables can be created by the operator, using the *set* command, and retain their value until changed by another *set* command. In particular, environment variables retain their existence and values even when programs are executed.

Environment variables are case-sensitive, so the variable named *VAR* is different from one named *Var*.

The format of the *set* command which sets the value of an environment variable is:

```
set VAR=string
```

where *VAR* is the name of the variable, and *string* is the character string to be assigned to it. *string* can be null, in which case the specified variable is deleted. The variable will be created, if it didn't previously exist.

For example, to set the environment named *PATH* to the string ";sys:/bin;data:/progs" the following command would be used:

```
set PATH=;sys:/bin;data:/progs
```

To delete the PATH variable, the following command would be used:

```
set PATH=
```

Environment variables can be assigned quoted strings:

```
set NAMES='Penelope Matilda Esmarelda'
```

The *set* command, when issued without any arguments, will display the names and values of the environment variables.

The *set* command can also be used within exec files to set exec file options. This use of the *set* command is discussed in the exec file section of this chapter.

2.11.2 Passing environment variables to programs

A program can fetch the value of an environment variable using the *getenv* function, passing to it the name of the variable. Programs

cannot change the value of an environment variable.

2.11.3 Use of environment variables in command lines

When the SHELL finds an environment variable name in a command line, preceded by the character '$', it replaces the name and the '$' with the value of the variable.

For example, if the environment variable *color* has the value *violet*, then entering

```
echo $color
```

is equivalent to entering

```
echo violet
```

and results in the displaying of

```
violet
```

on the screen.

As another example, given the environment variable *b*, having value '*freds disk:/usr/bin/*', the following command will move the file *pgm* from the current directory to the directory */usr/bin* on the volume '*freds disk:*':

```
mv pgm $b
```

The use of environment variables isn't restricted to command line arguments. For example, given the environment variable *cmd*, having value '*ls -l ralphs disk:/usr/math/lib/*', the following command will list the contents of */usr/math/lib* on *ralphs disk*:

```
$cmd
```

Environment variables names that are used in command lines can be surrounded by { and } to prevent ambiguity in cases where the variable is immediately followed by a character string. For example, if the following environment variables are defined

```
user=fred
userdy=john
```

then

```
echo ${user}
```

is equivalent to

```
echo $user
```

and displays

```
fred
```

Entering

```
echo $userdy
```

will display

```
john
```

since the SHELL interprets the entire string following $ to be the name of the variable. And entering

```
${user}dy
```

will display

```
freddy
```

since the SHELL assumes that the environment variable name is contained in the braces.

2.11.4 Standard environment variables

A few environment variables are created and assigned initial values by the SHELL when it is first activated. These are described in the section on starting the SHELL.

2.12 Searching for commands

When the operator enters a command, the SHELL first checks to see whether it is a builtin command. If so, the SHELL executes it. Otherwise, the command must be the name of a file to be executed, so the SHELL attempts to find the file.

2.12.1 Searching for command files

The SHELL looks for a command file in a sequence of directories. By default, it looks in the current directory and then in the directory */bin* on the volume containing the SHELL.

The directories to be searched for a command file can be specified using the command

```
set PATH=dir1;dir2; ... ;dirn
```

where dir1, dir2, ..., dirn are the directories to be searched. These directories are searched in the order specified.

That is, the directories to be searched are specified on the command line, separated by semicolons. If an entry doesn't specify a volume, but does specify a directory, the directory is assumed to be on the current volume; that is, the volume that contains the current directory. If it specifies a volume and not a directory, it's assumed to be the root directory of the volume. And if neither volume nor directory is specified (that is, the entry is null), the directory is assumed to be the current directory.

For example, the following command will cause the SHELL to search the current directory, then the directory */bin* on the current volume, and finally the directory */progs* on the volume *sys:*.

```
set PATH=;/bin;sys:/progs
```

In the next example, the set command causes the SHELL to search the directory */bin* on the *sys:* volume, then the */bin* directory on the current volume, and finally the current directory:

```
set PATH=sys:/bin;/bin;;
```

The *set PATH* command causes the environment variable named *PATH* to be set to the indicated character string. To display the value of all the environment variables, including *PATH*, enter the *set* command by itself; eg,

```
set
```

2.12.2 Program or exec file?

When the SHELL finds a file that matches the name that the operator entered, it has to decide whether it contains a program or is an exec file. It bases its decision on the type of the file's "fork": if it has a resource fork, as are linker-created files and standard Macintosh

applications, then it's assumed to contain a program. If it only has a data fork, and its type is *TEXT*, then it's assumed to be an exec file.

2.13 Starting and stopping the SHELL

2.13.1 Starting the SHELL

The SHELL can be started when the Finder is active. It can also be made to start automatically when the Macintosh is turned on or reset

2.13.1.1 Starting the SHELL from the Finder

To start the SHELL when the Macintosh Finder program is active, simply open the SHELL as you would any application. That is, open the volume containing the SHELL and then open the SHELL by either double-clicking its icon or clicking the 'open' item in the 'file' menu.

2.13.1.2 Automatic activation of the SHELL

When the Macintosh is turned on or when the reset button is depressed, the Macintosh automatically scans the drives, beginning with the internal drive, and activates the 'startup program' from the first disk it finds. Thus, if the SHELL is the startup program on the disk containing it, the SHELL can be automatically started without having to first start the Finder.

Each disk can have one file designated as the disk's startup program. This is done while the Finder is active by clicking the file, and then clicking the 'startup program' item in the 'special' menu.

You can see that there are several ways to automatically activate the SHELL:

* Put the SHELL's disk in the Mac's internal drive and turn the power on or hit reset;
* With the internal drive empty, put the SHELL's disk in the external drive and turn on the power or hit reset.

Given the Macintosh's startup algorithm, there is one situation to avoid when attempting to automatically activate the SHELL: don't try to boot the SHELL with the external drive containing the SHELL disk and the internal drive containing a non-bootable disk or disk which contains a startup program other than the SHELL. In the first case, the Macintosh won't load anything; it'll just stop. In the second case, the Macintosh will activate the other startup program, and not the SHELL. The SHELL can be set so that it is automatically started when the Macintosh is turned on or when the reset button is depressed.

2.13.1.3 Executing the .profile

If you have a sequence of commands that you want to always execute as soon as the SHELL starts, you can put them in a file named *.profile* on the root directory of the volume containing the SHELL. When the SHELL starts, it will automatically execute the commands in this file. For example, the *.profile* could create environment variables or change the default values assigned to the SHELL-created variables.

2.13.2 Initial environment variables

A few environment variables are created and assigned initial values by the SHELL when it is first activated. These are:

PATH	-	Defines the directories to be searched for a command line. Initially set to *;sys:/bin*, where *sys:* is the name of the volume containing the SHELL.
PS1	-	Primary prompt. Initially set to '-? '.
PS2	-	Secondary prompt. Initially set to '>'.
PS3	-	Cmd logging string. Initially set to '+'.
INCLUDE	-	Defines the directory to be searched by the compiler for files specified in 'include' statements. Initially set to *sys:/include*, where *sys:* is the name of the volume containing the SHELL.
CLIB	-	Defines the directory to be searched by the linker for a libraries. Initially set to *sys:/lib/*, where *sys:* is the name of the volume containing the SHELL.

The values of these variables can be modified by the operator, when desired.

2.13.3 Stopping the SHELL

The SHELL is stopped by starting the Finder. There are two ways to do this: reboot the system, or start the Finder from the SHELL.

Rebooting

The Finder can be started by rebooting the system, letting the Macintosh automatically activate the Finder.

SHELL activation of the Finder

With the SHELL active, the Finder can also be started as is any program. That is, with the Finder in a directory specified in the PATH command, simply enter the name of the file containing the Finder.

You probably won't want to start the Finder in this way, however; if you do, the first time that a Finder-activated program exits, the SHELL will be reactivated, and not the Finder.

The reason for this is that in low memory is a field containing the name of the command processor program. When the SHELL starts, it puts its name in this field so that when any program finishes the SHELL will be reloaded. If you simply start the Finder from the SHELL, this field isn't changed, so when the first Finder-activated program finishes, the operating system will reload the SHELL.

A better way to start the Finder from the SHELL is to run the following program:

```
/* bye - exit to Finder */
main()
{
    *(short *)0x210 = 1;
    strcpy(0x2e0L, "\PFinder");
}
```

This program's first assignment statement sets the field that defines the 'boot drive'; that is, the drive that contains the command processor program. This field is set to 1 or 2, depending on whether the internal or external drive is the boot drive. Here, we have assumed that the internal drive is the boot drive.

The second assignment statement sets the command processor definition field, making the Finder the command processor.

When this program exits, the Finder will be loaded, and will remain the command processor until you explicitly restart the SHELL.

2.14 Menus

The SHELL can optionally display several menus from which you can select SHELL commands to be executed and display information. By default, menus are enabled. Menus are disabled with the command

```
set +m
```

and enabled with the command

```
set -m
```

The following paragraphs discuss menus.

2.14.1 The *Apple* Menu

The *Apple* menu is the standard Apple menu, which displays the desk accessories that are in the System file.

The *Apple* menu has one special item, *About the SHELL*; clicking this item causes the SHELL to display information about itself, such as its version number. When this item has been selected, clicking the mouse will return to the SHELL.

When you select a desk accessory, its window moves to the foreground, the SHELL's moves to the background, and the SHELL won't accept commands. In this situation, clicking the mouse outside accessory's window will cause the SHELL to begin accepting commands again, moving its window to the foreground, and the desk accessory's window to the background; the desk accessory program is still running but you can't talk to it or see its window.

2.14.2 The *File* Menu

The *File* menu allows you to perform operations related to desk accessories: clicking its *See Windows* item, which can only be done when the SHELL is active and its window is in the foreground, moves background windows to the foreground and the SHELL window to the background. Clicking the *File* menu's *Close* item, which can only be done when desk accessory windows are in front of the SHELL's window, deactivates the frontmost window and its related desk accessory.

2.14.3 The *Edit* Menu

The *Edit* menu allows you to perform editing operations, such as cutting and pasting, on appropriate desk accessory windows. Items in the *Edit* menu can be clicked only when the window of a desk accessory is in the foreground.

2.14.4 The *Commands* Menu

The *Commands* menu lists all of the SHELL's builtin commands. The commands are separated into two groups: those that don't require any arguments, and those that do. The groups are separated by a

horizontal line, with the former group above the line and the latter below it. You can activate a command that doesn't require arguments by clicking its item. If you click the item of a command that requires arguments, a message defining its arguments will be displayed.

2.14.5 The *Programs* Menu

The *Programs* menu contains items, which you define, each of which is a SHELL command. When one of these items is clicked, the corresponding command is executed.

For example, you might have the following items in the *Programs* menu:

```
cd /
ls -lt
myprog arg1 arg2 arg3
```

Clicking one of these items causes that command to be executed.

The environment variable *PRG_LIST* defines the *Programs* items: this variable consists of the items, with the items separated by semicolons. For example, for the above *Programs* menu, the following command would be used to initialize *PRG_LIST*:

```
set PRG_LIST='cd /;ls -lt;myprog arg1 arg2 arg3'
```

2.14.6 The *Source* Menu

The *Source* menu lists the names of *.c* and *.asm* files, if any, that are in the current directory. The names are listed in alphabetical order, beginning with the *.c* files. A maximum of 20 names can be displayed. If the current directory doesn't contain any *.c* or *.asm* files, the *Source* menu isn't displayed in the menu bar.

One source file in the current directory can be distinguished for use by the *Compile* menu, as discussed below. This file, called the 'current file', is marked by clicking its name while the *Source* menu is selected. Once a file is made current, it stays current until you explicitly mark another. The SHELL remembers the name of the current file by creating an environment variable, *SRC_FILE*, and associating the name of the current file with this variable.

If you have more than 20 source files in a directory and want to make current one that's not displayed in the *Source* menu, you can set the *SRC_FILE* environment variable to the file name by typing a *set* command. For example, to make the file *x.c* current when it's not displayed in the *Source* menu, you could enter the command

```
set SRC_FILE=x.c
```

2.14.7 The *Compile* Menu

The *Compile* menu has items that allow you to conveniently perform program development operations (editing, compiling,

assembling, linking, executing) on the current file.

In addition to items which, when clicked, initiate an operation on the current file or on one of the files generated from it, there are other items that, when clicked, enable and disable options:

* If the *Auto Assemble* item is enabled when a compilation initiated by clicking the *Compile* item is done, the assembler will automatically assemble the assembly language source that was generated by the compiler.
* If the *Auto Link* item is enabled when an assembly that was initiated by clicking the *Compile* or *Assemble* item is done, the linker will automatically link the object module that was generated by the assembler.
* If the *Math Lib* item is enabled when the linker is started in response to a clicking of the *Compile*, *Assemble*, or *Link* item, the linker will search the math library, *m.lib* for needed modules in addition to the standard libreary, *c.lib*.
* If the *Auto Run* item is enabled when a linkage that was initiated by clicking the *Compile*, *Assemble*, or *Link* items is done, the resulting program will be run.

If an options item is enabled, a check mark appears beside it. To change the state of an option, click it.

The *Compile* menu, like the *Source* menu, appears in the menu bar, and can hence only be selected, when the current directory contains *.c* or *.asm* files.

The following paragraphs discuss in detail the operations initiated by clicking one of the operational items.

The *Edit* Item

Clicking the *Compile* menu's *Edit* item causes an editor to prepare the current source file for editing. The environment variable *EDIT* contains the name of the editor that is activated by the *Edit* item. If this environment variable doesn't exist, the *edit* editor is used. The *EDIT* variable is initialized using the *set* command. For example, the command to make the *Z* editor the *Edit* items' editor is

```
set EDIT=z
```

A convenient place to put the command that initializes the *EDIT* variable is the *.profile* file, since the SHELL automatically looks for a file having this name when it starts and, if found, executes the file's commands. In fact, the *.profile* that is on your distribution disks contains this *set* command.

The *Compile* item

The *Compile* item compiles the current source file, generating an assembly language source file. For example, if the current file is

hello.c, clicking *Compile* will compile *hello.c*, generating the file *hello.asm*.

By default, when the *Compile* item is clicked a command line of the following form is generated and executed:

```
cc hello.c
```

where *hello.c* is the name of the current file.

The environment variable *CFLAGS* allows you to define options that will be included in the command line that is generated in response to clicking the *Compile* item, and hence that will be passed to the compiler. The string associated with *CFLAGS* is simply included in the generated command line. For example, if you want the variables *FLOAT* and *M68K* to be defined when the current file is compiled, initialize *CFLAGS* using the command

```
set CFLAGS='-DFLOAT -DM68K'
```

Then, supposing that the current file is named *xxx.c*, clicking the *Compile* item in the *Compile* menu will cause the compiler to be started with the line

```
cc -DFLOAT -DM68K xxx.c
```

As mentioned above, when a compilation initiated by clicking *Compile* is completed with no errors having been detected, the assembler will automatically assemble the compiler output as if the *Assemble* item was clicked, if the *Auto Assemble* item in the *Compile* menu is enabled.

The *Assemble* Item

Clicking the *assemble* item in the *Compile* menu (or successful completion of a compilation that was initiated by clicking the *Compile* item with the *Auto Assemble* option enabled) causes the assembly language source file that's associated with the current file to be assembled.

If the current file is a *.asm* file, then the current file is the file that's assembled. If the current file is a *.c* file, then the assembly language file generated by the compiler is the file that's assembled. For example, if the current file is *hello.c*, then clicking *Assemble* causes *hello.asm* to be assembled.

The assembler generates an object module, placing it in a file whose name is derived from that of the assembly language source file by changing its extension to *.o*. For example, when *hello.asm* is assembled, the file *hello.o* is generated.

If, when an assembly started by clicking *Assemble* or *Compile* is completed, the *Auto Link* item in the *Compile* menu is enabled, (as shown by a mark beside it) the linker will automatically link the generated object module into an executable program, just as if the *Link*

item was clicked.

The *Link* Item

Clicking the *Link* item in the *Compile* menu (or successful completion of an assembly that was initiated by clicking the *Compile* or *Assemble* item when the *Auto Link* item is enabled) causes the object module that's associated with the current source file to be linked.

The name of the file containing the object module is derived from that of the current source file, by changing its extension to *.o*. For example, if the current file is *hello.c*, then clicking *Link* causes the object module *hello.o* to be linked.

To start the linker, a command line is generated and then executed. The default command line has the following form:

```
ln file.o -lc
```

where *file.o* is the current source file's object module. *-lc*, of course, causes the linker to search the standard library *c.lib* for needed modules; as usual, the environment variable *CLIB* defines the directory that contains the library.

You have some control over the command line that's generated in response to clicking the *Compile*, *Assemble*, or *Link* item. If the *Math Lib* item on the *Compile* menu is enabled (which it is when a check mark is beside it), the generated command line tells the linker to search the math library, *m.lib*, for object modules. That is, the command line has the form:

```
ln file.o -lm -lc
```

The environment variable *LFLAGS* gives you additional control over the command line generated by the *Link* item: the string associated with this variable is included in the command line. For example, if *LFLAGS* is initialized with the command

```
set LFLAGS='-M -T a.o b.o c.o'
```

then when *Link* is clicked the command line looks like this:

```
ln file.o -M -T a.o b.o c.o -lc
```

And if *Math Lib* is also enabled, the command line looks like this:

```
ln file.o -M -T a.o b.o c.o -lm -lc
```

When a linkage that was initiated by the clicking of a *Compile* menu item is completed, the generated program will automatically started if the menu's *Auto Run* item is enabled (which it is when a check mark is beside it). This item is enabled and disabled by clicking it.

THE COMPILER

Chapter Contents

The Compiler

This chapter describes how to use the Aztec C68K C compiler. The Aztec C68K compiler is implemented according to the language definition found in the text *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie. For information on the C language and its use the above reference is recommended. For the C student, there are several tutorial texts listed in the Bibliography.

This chapter has four major sections: the first describes how to use the compiler, the second describes the compiler options, the third describes error handling, and the fourth discusses topics of interest to programmers, such as the use of register variables and the writing of machine-independent code.

1. Operating Instructions

The Aztec C68K compiler is invoked by a command of the format:

cc [-options] filename.c

where [-options] specify optional parameters, and *filename.c* is the name of the file containing the C source program. The option specification should appear before the filename.

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

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

A compilation can be aborted by holding down the key that has the cloverleaf symbol and then typing the period key.

1.1 Compilation Environment

The *cc* compiler executes in the Aztec SHELL environment. For information on using the SHELL facilities, refer to the SHELL reference section of this document.

If the *cc* file is in the current directory, the compiler can be invoked as described above. Often this will not be the case. When the Aztec C68K compiler, *cc*, is in a directory other than the current one, the path to that directory can be defined through the *set PATH*

command. If the concept of current directory or path is unfamiliar, read the SHELL reference section. By way of example, if the *cc* compiler is in the directory *sys:bin* and the current directory is some other directory, the path could be set as,

```
set PATH=sys:bin
```

Now when the *cc* command is referenced, the system will search for *sys:bin/cc.*

The compiler can also be invoked by prefixing the path name to *cc* in the command line. For instance,

```
/bin/cc myprog
```

Or,

```
dsk2:sys/bin/cc myprog
```

If the volume specification is omitted, then the current directory path is prefixed to the *cc* command. Thus if the current directory was *dsk2:sys/space* and the compiler, *cc*, was in the directory *dsk2:sys/space/csys*, then the following would run the compiler,

```
csys/cc myprog
```

It is generally easiest to use *set PATH* to specify the pathname qualification for the *cc* command.

1.2 The Input File

When the compiler is started, the name of the file containing the C source can optionally specify the volume and directory that contains the file. By default, the input file is assumed to be in the current directory. For example, if the file *prog1.c* contains C source, and is in the directory *db/source* on the volume *dsk2:*, it could be compiled with the command

```
cc dsk2:db/source/prog1.c
```

If the directory containing this file is also the current directory, then the file could also be compiled with the command

```
cc prog1.c
```

And if the current directory is *db*, on the *dsk2:* volume, the file could also be compiled with the command

```
cc source/prog1.c
```

Source filename extensions

If the command that starts the compiler doesn't specify the extension of the file containing the C source, the compiler assumes that the extension is *.c*. For example, the command

cc prog

will compile a file named *prog.c* in the current directory.

Although *.c* is the recommended file extension name, it is not mandatory. The specification

cc prog.see

will read the file *prog.see* from the current directory as the input to the compiler.

1.3 The output files

1.3.1 Creating an object code file

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

By default, the object code generated by a compiler-started assembler is sent to a file whose name is derived from that of the file containing the C source by changing its extension to *.o*. This file is placed in the directory that contains the C source file. For example, if the compiler is started with the command

cc prog.c

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

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

cc -O myobj.rel prog.c

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

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

cc -T prog.c

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

1.3.2 Creating just an assembly language file

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

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

When this option is specified, the compiler by default sends the assembly language source to a file whose name is derived from that of the C source file, by changing the extension to *.asm*. This file is placed in the same directory as the one that contains the C source file. For example, the command

cc -A prog.c

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

When using the -A option, you can specify the name of the file to which the assembly language source is sent, using the -O option. For example, the command

cc -A -O ram:temp.asm prog.c

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

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

1.4 Searching for #*include* Files

By default the Aztec C68K compiler searches the current directory to locate files specified in #*include* statements. It can also search a user-specified sequence of directories for such files, thus allowing program source files and header files to be contained in different directories.

The compiler option *-I* and the environment variable *INCLUDE* define the directories in which the compiler will search for #*include* files.

The compiler will automatically search just the current directory for a #*include* file if the following conditions are met: (1) the compiler

was started without a -I option having been specified, (2) *INCLUDE* is not an environment variable, and (3) the *#include* statement doesn't specify the drive and/or directory containing the file.

If a *#include* statement specifies either the drive or directory, just that location is searched for the file.

1.4.1 The -I option

The compiler *-I* option defines a single directory to be searched for a file specified in a *#include* statement. The path descriptor follows the *-I*, with no intervening blanks. For example, the specification

```
cc -isys:db/include prog1
```

directs the compiler to search the *sys:db/include* area when looking for a *#include* file.

Multiple *-I* options can be specified when the compiler is started, if desired, thus defining multiple directories to be searched.

1.4.2 The *INCLUDE* environment variable

The *INCLUDE* environment variable, if it exists, also defines directories to be searched for *#include* files. This variable has the same format as the *PATH* environment variable; that is, it consists of the names of the directories to be searched, separated by semicolons. For example, the following command creates the *INCLUDE* environment variable, defining three directories to be searched:

```
set INCLUDE=work:/include;work:;sys:include
```

These directories are (1) the *include* directory on the *work:* volume; (2) the root directory on the *work:* volume; (3) the *include* directory on the *sys:* volume.

1.4.3 #include Search Order

When the compiler encounters a *#include* statement, it searches directories for the file specified in the statement in the following order:

1. If the file name was delimited by double quotes, "filename" the current directory is searched. If the name was delimited by angle brackets, <filename>, the current directory is searched only if no -I options were specified and if the *INCLUDE* environment variable doesn't exist.
2. The directories specified in the *-I* options are searched, in the order listed on the line that started the compiler;
3. The directories specified in the *INCLUDE* environment variable are searched, in the order listed.

2. Compiler Options

Utility Options

-D Defines a symbol for the preprocessor.

-I Defines an area to be searched for files specified in a #include statement.

-O Used to specify an alternate name for the output file.

-S Causes search for undefined structure members as described below.

-T This option will insert the C source statements as comments in the assembly code output. Each source statement appears before the assembly code it generates.

-A Causes the compiler to not start the assembler after it has compiled a program.

-B Causes the compiler to not generate the statement *public .begin* when it compiles a program.

-U Causes the compiler to generate code that uses register A4 instead of A5 to reference data and to not generate code that uses A4 for holding register variables and temporary values.

-Q Causes the compiler to put character string constants in a program's data segment rather than in its code segment.

Table Manipulation Options

-E Specifies the size of the expression table.

-L Specifies the size of the local symbol table.

-Y Specifies the maximum number of outstanding cases allowed in a switch.

-Z Specifies the size of the table for literal strings.

2.1 Utility Options

-D Option

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

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

For example,

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

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

```
#define  MAXLEN  1000
```

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

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

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

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

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

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

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

-I Option

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

The name of the area immediately follows the -I, with no intervening spaces. For example, the following defines directory /source/inc on volume sys: search area:

```
-Isys:/source/inc
```

For more details, see the Compiler Operating Instructions, above.

-S Option

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

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

struct btype {
   char b1, b2;
} b;

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

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

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

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

The -S option refers only to previously defined structures.

-B option

The -B option prevents the compiler from generating the statement

```
public .begin
```

which it otherwise does generate.

.begin is the startup routine used by most command programs, and is contained in the module *crt0* in the library *c.lib.*

The presence of the *public .begin* statement in a compiled program causes the linker to include *crt0* in the executable version of the program.

Drivers and desktop accessories generally perform their own startup procedures, and don't need *.begin.*

Thus, command programs are usually compiled without the -B option while drivers and desktop accessories are compiled with it.

-U option

If a program is compiled without this option, the code generated for it uses register A5 to access global and static data, and uses register A4 for holding register variables and temporary values.

Register A5 can't be used by drivers and desktop accessories, since it's already being used by command programs, and by QuickDraw. The -U option causes the compiler to generate code that uses register A4 as a base register, and that doesn't use A4 for holding a register variable or a temporary value.

Thus, command programs should normally be compiled without the -U option, while drivers and desktop accessories should be compiled with it.

2.2 Table Manipulation Options

The compiler has several memory-resident tables in which to store information about a program it is compiling. Some of these tables are used to keep track of the symbols defined within the program, and some as a "scratch pad" for temporarily storing information.

The compiler uses the following tables: macro/global symbol table, local symbol table, label table, string table, expression work table, and case statement work table.

The sizes of these tables are determined when the compiler starts. For all tables except the macro/global symbol table and the label table, the size can be specified by the user with a command line option; if the user doesn't specify the size of one of these tables, the compiler sets it to a default value.

The macro/global symbol table is located in the application heap above all the other tables. Its size is set after all the other table sizes have been set, so that it uses all the rest of available memory. Hence, the user can't set the size of this table.

If, during a compilation, the macro/global symbol table that is in the application heap is filled and if no errors have been detected, the compiler will automatically use approximately 10K of the Macintosh screen memory to hold more macro and global symbols. When it does this, the compiler clears the screen and changes the size of the screen window to be about half of its previous size. It then displays any error messages in the contracted window and, when done, resets the screen window to its original size and clears to the end of the screen.

The size of the label table is built into the compiler; if this table overflows, you must reduce the number of labels in your program.

If a table overflows, the compiler will print an error message and stop. If any table except the macro/global symbol table overflowed, the compilation can be restarted, using a different size for the table which overflowed. If the macro/global symbol table overflowed, the compilation can be restarted, using smaller sizes for one or more of the other tables.

The Macro/Global Symbol table

This table is where macros defined with the #define statement are remembered. It also contains information about all global symbols.

If this table overflows, the message *Out of Memory!* will be printed.

The Local Symbol Table:

New symbols can be declared after any open brace. Most commonly, a declaration list appears at the beginning of a function body. The symbols declared here are added to the local symbol table. If

a variable is declared in the body of, say, a *for* loop, it is added to the table. When the compiler has finished compiling the loop, that entry in the table is freed up. And when it has finished the function, the table will be empty.

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

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

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

The Label Table

This table contains information on all the labels in a program, where a label is the destination of a *goto* statement.

If it overflows, error 54 will be displayed. Since the size of this table is fixed, if it overflows you must decrease the number of labels in your program.

The Expression Table:

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

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

The following expression uses 15 entries in the table:

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

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

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

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

```
cc -E100 filename
```

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

The Case Table

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

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

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

```
cc -Y100 file
```

The String Table

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

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

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

```
cc -Z2000 file
```

3. Error checking

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

The compiler will report any errors it finds in the source file, displaying first the source line in which the error was detected. If the error messages are sent to the screen (as discussed below) the line is underlined up to the approximate point at which the error was detected. If the messages are sent elsewhere, the source line, instead of being underlined, is followed by a line containing the "^" character at the approximate point of error.

The compiler will then display a line containing the following information:

1. the name of the source file containing the line,
2. the number of the line within the file.
3. an error code,
4. optionally, a message describing the error,
5. the symbol which caused the error, when appropriate .

The error codes are defined and described in the *errors* chapter.

The message describing an error will only be displayed if the compiler can find the file *cc.msg*, which contains the message, when it starts. The compiler searches for this file in the directories specified in the *INCLUDE* environment variable.

The compiler sends error messages to its standard output device. This can be redirected to a file in the normal way. Without the redirection of its standard output, the compiler sends error messages to the console. For example, to compile *prog.c* and send error messages to the file *prog.err*, the following command could be used:

```
cc prog >prog.err
```

When the compiler sends error messages to the screen, it will pause after several error messages have been displayed, and ask if you want it to continue. If you type Y, followed by a return, the compiler will continue. If you type anything else, followed by a return, the compiler will halt.

When the compiler sends error messages to a device or file other than the screen, it will process the entire file without giving the operator the opportunity to abort the compilation, even if errors are detected.

The compiler is not always able to give a precise description of an error. Usually, it must proceed to the next item in the file to ascertain

that an error was encountered. Once an error is found, it is not obvious how to interpret the subsequent code, since the compiler cannot second-guess the programmer's intentions. This may cause it to flag perfectly good syntax as an error.

If errors arise at compile time, you should first correct the first error, since this may clear up some of the errors which follow.

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

4. Programmer Information

4.1 Register Variables

The Aztec C68K C compiler supports up to six register variables. There are 4 data registers and 2 address registers reserved for user variables. *char* register variables will only be placed in data registers. If 6 data type variables are defined before a pointer register variable is encountered, the data type variables will be assigned to the 4 data and 2 address registers and the pointer register variables will not be assigned to registers.

Register variables may be of type *char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long* and *pointer.*

4.2 Writing machine-independent code

The Aztec family of C compilers are almost entirely compatible. The degree of compatibility of the Aztec C compilers with v7 C, system 3 C, system 5 C, and XENIX C is also extremely high. There are, however, some differences. The following discussion should assist developers looking to import C code to the Macintosh or to export Macintosh C programs to other environments.

4.2.1 Bit Fields

The major incompatibility of Aztec C with the various versions of UNIX C is the absence of bit fields. Bit field support tends in general to be incompatible from one C compiler to another and developers concerned with portability should avoid using them. Existing code using bit fields is fairly easily converted to use character or integer constructs. Bit field support will be implemented in the Aztec C compilers some time in the near future.

4.2.2 Enumerated Data Types And Structures

C programs using enumerated data types or structure passing must be modified to work with the Aztec C68K compiler. Structure passing can be implemented by passing pointers to structures instead of the structures themselves. This approach is generally more efficient in that it eliminates copying the structure onto the run time stack.

4.2.3 Compatibility Between Aztec Products

Within releases, code can be easily moved from one implementation of Aztec C to another. Where release numbers differ (i.e. 1.06 and 2.0) code is upward compatible, but some changes may be needed to move code down to a lower numbered release. The downward compatibility problems can be eliminated by not using new features of the higher numbered releases.

4.2.4 Sign Extension For Character Variables

None of the 8 bit implementations of Aztec C sign extend characters used in arithmetic computations, whereas all 16 bit implementations do sign extend characters. This incompatibility can be corrected by declaring characters used in arithmetic computations as unsigned, or by AND'ing characters used in arithmetic expressions with 255 (0xff). For instance:

```
char a=129;
int b;
b = (a & 0xff) * 21;
```

4.2.5 The MPU... symbols

To simplify the task of writing programs that must have some system dependent code, each of the Aztec C compilers defines a symbol which identifies the machine on which the compiler-generated code will run. These symbols, and their corresponding processors, are:

symbol	*processor*
MPU68000	68000
MPU8086	8086/8088
MPU80186	80186/80286
MPU6502	6502
MPU8080	8080
MPUZ80	Z80

Only one of these symbols will be defined for a particular compiler.

For example, the following program fragment contains several machine-dependent blocks of code. When the program is compiled for execution on a particular processor, just one of these blocks will be compiled: the one containing code for that processor.

```
#ifdef MPU6800
   /* 68000 code */
#else
#ifdef MPU8086
   /* 8086 code */
#else
#ifdef MPU8080
   /* 8080 code */
#endif
#endif
#endif
```

4.3 Writing programs for the Macintosh

4.3.1 Pointer Considerations.

Pointers are 32 bits wide in Aztec C68-compiled programs, whereas *int*s are 16 bits wide. Because of this difference, a program will not

work if it assumes that pointers and *int*s are the same size, and that they are treated the same.

It's easy for a C program to accidentally or purposely make these assumptions, since in C the type of an undeclared function or function argument is assumed to be *int*. A program making these assumptions will run on machines for which the assumptions are true, of course. But when the program is recompiled with Aztec C68, it will malfunction.

To avoid problems, a program that uses pointers should obey the following rules:

* If a function returns a pointer, explicitly declare it, both in the function itself and in any module that calls the function.
* If a function argument is a pointer, explicitly declare it in the function itself, and be careful not to accidentally pass an *int* as the argument.
* Beware when subtracting pointers, when the difference may be greater than 64K.

The following paragraphs discuss these rules.

Declare Functions That Return A Pointer

The following code demonstrates the importance of declaring a function that returns a pointer, both in the function itself and in a function that calls it. For this example, assume that the function *g* must be passed a pointer.

```
char *f();
...
g(f());
```

The compiled code will pass the 32-bit pointer returned by the function *f* to the function *g*. If *f* hadn't been declared as returning a pointer, the compiled code would assume that a 16-bit *int* was returned by *f*, and hence pass just part of the pointer returned by *f* to *g*.

Declare Function Arguments That Are Pointers

The following code demonstrates the importance of declaring that a function argument is a pointer, in the function itself. For this example, assume again that the function *g* must be passed a pointer.

```
f(y)
char *y;
{
...
    g(y);
...
}
```

The compiled code will take the 32-bit pointer *y*, which is passed to the function *f*, and pass it to the function *g*. If the declaration *char *y* was omitted, the compiler would assume that *y* was a 16-bit *int*, and hence generate code that would pass just part of the pointer to *g*.

Pointer Variables And Constants As Arguments On Calls

Code that passes integer constants or variables as function arguments where pointers are expected, will not work with Aztec C68K.

One common problem is attempting to pass the constant *int* 0 as a null pointer. For example, if *g* is again a function that is passed a pointer, the following code demonstrates one way to correctly pass a null pointer to *g*:

```
g((void *) 0);
```

(The code didn't have to cast 0 to be a pointer to a *void*; casting it to be a pointer to any other type of object would also have worked.) If instead the code had said

```
g(0);
```

Then a 16-bit null value would have been passed, which would be wrong.

Subtracting Pointers

The difference between two pointers is an *int*. This fact allows a program to easily pass to a function a pointer to a buffer and an *int* defining the size of the buffer. For example, the function *write* is passed three arguments: an *int* defining the file to be written to, a pointer to the beginning of the data that's to be written, and an *int* defining the number of bytes to be written. If *buf* is the name of an i/o buffer and *cp* is a pointer to the last byte of valid data in *buf*, then the following statement tells the function *write* to write all valid bytes in *buf* to the device having file descriptor *fd*:

```
write(fd, buf, cp-buf);
```

However, you can create buffers that contain more than 64K bytes, for example, by calling the *lmalloc* function. To determine the number of bytes between two arbitrary locations in such a buffer, simply subtract pointers that reference the locations and either assign the result to a long or cast it to a long. The reason that this works is that the difference between two pointers is always computed using a full 32-bit subtraction. When only 16 bits of the difference is needed (the default case) the high-order component of the difference is discarded; when all 32 bits is needed, they are available for use.

4.3.2 Internal Storage Of Numeric Data

Programs written for processors that store data items in least significant to most significant order may need to be changed. The MC68000 processor stores data in most significant to least significant order. This is true of both non-floating point and floating point data. The following short program illustrates a program that will run differently depending on the manner in which *int* data items are stored.

```
cput(c)
int c;
{
    write (1, &c, 1);
}
```

Problems can also occur reading data created in another environment where the data was stored in the reverse order.

The MC68000 requires that any memory access must be aligned on an even address unless it is a single byte access. The Aztec C68K compiler aligns non-byte data items on even boundaries to avoid memory faults. Code that accesses non-byte data through pointers that specify an odd memory address will cause a system crash.

Converting Data

Programs reading data written in another C environment that does not force alignment will probably not produce correct results. Programs writing data that will be accessed in an environment that does not force alignment will likewise probably fail. Some conversion will probably be needed to insert slack bytes to assure even alignment when importing data created for an unaligned environment, and to remove slack bytes when exporting data for an unaligned environment.

Most of the common 8-bit microprocessors store numeric data in an order that is the reverse of the MC68000. The 808x 16-bit processors and the PDP-11 also store numeric items this way.

4.3.3 Long Character Items

Aztec C68K recognizes long character constants. The following code will work:

```
long l;
...
l='abcd'
```

Not all compilers will recognize this construct. Many are limited to two character character constants.

4.3.4 Extensions to the C language for the Macintosh

Aztec C68K has a few extensions to the C language to support the special features of the Macintosh.

Calling Pascal functions from C

First, a C program can call a Pascal function that is in ROM, by declaring the Pascal function with a statement of the form

```
pascal type func ()=0x1234;
```

where

type	is the type of value returned by the function,
func	is the name of the function,
0x1234	(or whatever) is its trap value;

A program can also define a global pointer to a Pascal function, with a statement of the form

```
pascal type (*fp)();
```

where *fp* is a variable that points to a pascal function and *type* is the type of value returned by the function.

Calling C functions from Pascal

There are several toolbox routines that call a function whose address is passed to them. Such functions can be written in C by preceding the C function with the keyword *pascal.* The function can return a value, if required; if not, the function should be declared to be of type *pascal void.*

The code generated for such a function differs from that generated for a normal C function in the following ways:

* It will preserve registers D3 and A2, which are sacred to Pascal.
* It will access arguments on the stack using the Pascal conventions, rather than the C conventions.
* At the return of the function, all arguments will be popped off the stack and, the return value of the function, if any, will be pushed onto the stack.

Character strings

The format of a character string differs in C and Pascal: in C, the string consists of the characters, with a terminating null character. In Pascal, the first byte of the string contains the number of characters in the string.

To have the compiler generate a Pascal format character string, begin the string with the sequence "\P". For example,

```
"\PThis is a Pascal string"
```

The string will still be null-terminated, so it can be passed to functions like *strcpy* and *strcmp*.

There are two functions which can be used to convert strings from one format to the other: *ctop* converts a string from C to Pascal format, and *ptoc* converts a string from Pascal to C format. For more details on these functions, see their description in the section of the Library Functions chapter that describes Macintosh functions.

4.4 Additional features

4.4.1 Line continuation

If the compiler finds a source line whose last character is the backslash character \, it will consider the following line to be part of the current line, without the backslash. For example, the following statements define a character array containing the string "abcdef":

```
char arr[]="ab\
cd\
ef";
```

4.4.2 Special symbols

The following symbols are defined by the compiler:

___FILE___ Name of the file being compiled. This is a character string.

___LINE___ Number of the line currently being compiled. This is an integer.

___FUNC___ Name of the function currently being compiled. This is a character string.

For example,

```
printf("file= %s", ___FILE___);
printf("line= %d, ___LINE___);
printf("func= %s", ___FUNC___);
```

4.4.3 The #*line* statement

The statement

```
#line number "file"
```

causes the compiler to think that *file* is the name of the file being compiled, and that *number* is the number of the line currently being compiled. The file name is optional; if not specified, the compiler assumes that it's still compiling the same file.

4.4.4 In-line assembly code

Assembly language code can be interspersed within C source code by surrounding the assembly code with the statements #*asm* and

#endasm.

For example,

```
main()
{
   /* C code */
#asm
; assembly language code
#endasm
   /* C code */
}
```

For a discussion of register usage by assembly language programs, see the Programmer's Information section of the Assembler chapter.

New Options for Compiler (CC)

These pages describe compiler changes for this release and should be placed at the end of the Compiler section in your manual.

The following options are new to the compiler, *cc.*

+FI Generates code used for the IEEE Double Precision Floating Point Emulation. Users may choose between linking with the Manx Aztec C library (*m.lib*) or the SANE library (*ms.lib*). Compiling defaults to the SANE format.

+F8 Generates code used for the 68881 Floating Point format. Users link in with the math library *m8.lib.*

-N Suppresses the insertion of source debug information in the output module.

The following options are added in the compiler:

+L Defines an integer (*int*) to be 32 bits instead of 16 bits. When linking, you must use C32.lib and M32.lib if this flag is set.

Note: This option cannot be used in conjunction with Macintosh Toolbox calls, where integers are expected to be 16-bits.

+N Imbeds function names into the executable code that MacsBug or TMON debuggers can read as symbols.

-B Switches off the compiler pause after encountering five errors.

-V Generates verbose compiler messages.

+C Generates code that uses the large code memory model.

+D Generates code that uses the large data memory model.

Note: Flags *C* and *D* require the use of large data libraries.

The following options are changes to the compiler,

+B Causes the compiler not to generate the *public.begin* statement.

+U Causes the compiler to generate code that uses the register A4 instead of A5 to reference data and to not generate code that uses A4 for holding register variables and temporary values.

Other New Features

* Compiler supports enumerated data types as 8, 16, or 32 bits, depending on the range of the enumerated literal.

* Compiler supports bit fields, structure assignment, and structure passing. Prior to this release, only an address of a structure could be passed.

* Compiler expands variable name length from 8 to a maximum of 31 characters. External symbols are also significant up to 31 characters throughout assembly and linkage.

* Compiler reserves words "const," "signed," and "volatile"; these must not be used as symbol names.

* Compiler attaches a leading underscore to the filename to ensure compatibility with MPW. (In previous releases the compiler appended a *trailing* underscore.) This change, together with the object module format change to the linker, prohibits linking old libraries or object files. This release includes a utility *su* that you may use to switch trailing underscores in assembly source to leading underscores. Type the command line as follows:

  su file1.asm file1.asm file3.asm...

  su switches the trailing underscores for all files passed to it whether identifiers are declared public, global, or bss. The identifiers without trailing underscores are not modified, but a warning listing these cases is displayed. *su* depends on the public, global, or bss declarations occurring before the identifiers are actually used.

* The compiler defines the names __LARGE_CODE and __LARGE_DATA when the *+C* and *+D* options are given. These memory modules are used in some of the header files to switch between an external definition of some hardware addresses and a macro definition with the address hard-coded in. See the appendices to the "Technical Information" section of your manual included with this release for a discussion of memory modules and how to generate libraries using the *make* function.

* The *Void* data type is added to provide a safety check on the use of *void* functions--those functions that do not return a value.

  If a *void* function attempts to return a value, or if a function tries to use the value returned by a *void* function, the compiler generates an error message.

  Variables can be declared to point to a *void*, and functions can be declared as returning a pointer to a *void*.

Unlike other pointers, a pointer to a *void* can be assigned to a pointer to any type of object, and vice versa. For other types of pointers, the compiler generates a warning message if an attempt is made to assign one pointer to another, when the types of objects pointed at by the two pointers differ.

For example, the compiler generates a warning message for the assignment statement in the following program:

```
main()
{
   char *cp;
   int *ip:
   ip = cp;      }
```

But the compiler won't complain about the following program:

```
main()
{
   char *cp;
   void *getbuf();
   cp = getbuf();
}
```

* The INCLUDE environment variable accepts the ";" character as a separator between multiple directory names. For example, to specify the C and assembler header files, use:

```
set INCLUDE=sys2:include;sys2:asm
```

* To shorten compilation time, the compiler supports precompiled *#include* files.

  This option only applies to a block of *#include* files that are located in the beginning of a file. For example, if an *#undef* is placed between two *#include* file statements, it will not have the desired effect.

  To use this feature, you first compile frequently-used header files, specifying the *+H* option; this causes the compiler to write its symbol table, which contains information about the contents of the header files, to a disk file. Then, when you compile a module that *#include*s some of these header files, you specify the *+I* option; this causes the compiler to load into its symbol table the pre-compiled symbol table information about the header files. When the

compiler encounters a *#include* statement of a header file for which it has already loaded pre-compiled symbol table information, it ignores the *#include* statement. This ignoring occurs even if the *#include* file was nested within another *#include* file in the C source from which the pre-compiled symbol table was generated.

The *+H* option tells the compiler to write its symbol table to a file. The name of the file immediately follows the *+H*, with no intervening spaces. For example, you might create a file named *x.c* that consists just of *#include* statements for all the header files that you want pre-compiled. You could then generate a file named *x.dmp* that contains the symbol table information for these header files by entering the following command:

```
cc +Hx.dmp x.c
```

The *+I* option tells the compiler to read pre-compiled symbol table information from a file and uses the normal include search path. The name of the file immediately follows the *+I*, with no intervening spaces. For example, to compile the file *prog.c* that accesses the header files that were defined in *x.c*, and to have the compiler preload the symbol table information for these files from *x.dmp*, enter the following command:

```
cc +Ix.dmp prog.c
```

THE ASSEMBLER

Chapter Contents

The Assembler

The *as* assembler translates assembly language source statements into relocatable object code. Assembler source statements are read from an input text file and the object code is written to an output file. A listing file is written if requested. The relocatable object code must be linked by *ln*, the Manx Linker, before it can be executed. At linkage time it may be combined with other object files and run time library routines from system or private libraries. Object modules produced from C source text and Assembler source text can be combined at linkage time into a composite module.

Assembly language routines are generally not required when programming in C. Assembly language routines should only be necessary where critical execution time or critical size requirements exist. Some system interfacing or low level routines may also require assembler code.

Information on the MC68000 architecture and instructions can be found in the *Motorola MC68000 16-bit Microprocessor User's Manual* (Prentice-Hall, Inc., Englewood Cliffs, N. J. 07632)

1. Operating Instructions

The assembler is started by entering the command line:

as [-options] filename

where *[-options]* specify optional parameters and *filename* is the name of the file to be assembled.

The assembler is also invoked by the C68K-C compiler to assemble its output file.

The assembler reads assembly source statements from the input file, writes the translated relocatable object code to an output file, and if requested writes a listing to an output file. The Assembler also will merge assembly code from other files on encountering an *include* directive.

1.1 Execution Environment

The Manx Assembler executes in the Aztec SHELL environment. For information on the using the SHELL refer to the SHELL reference section of this document.

The SHELL will search for the assembler in the directories specified in the PATH environment variable. See the SHELL chapter

for more information about this.

1.2 The Input File

The input file is a text file that will usually be created by a text editor or the Aztec C68K compiler. The input file is assumed to reside in the current directory. If it does not, a fully qualified or partially qualified path name can be prefixed to the file name to designate the source directory. Although *.asm* is the recommended file name extension any extension is acceptable. Do not use filenames without extensions as input to the assembler. The specification:

```
as x
```

will assemble the file *x.asm* if there is no file named *x* in the same directory. If there is a file named x in the same directory as the input file the results are unpredictable.

1.3 The Object Code File

The object code produced by the assembler is written to a file. By default, this file is placed in the directory that contains the source file, and its name is derived from that of the input file by changing the extension to *.o*.

To write the object code to a file in another directory, and/or to a file having another name, use the *-o* option. For example, the following command assembles the source that's in *prog.asm*, sending the object code to the file *new.obj*. This latter file is placed in the current directory, since the *-o* option didn't specify otherwise.

```
as -o new.obj prog.asm
```

1.4 Listing File

If the *-L* option is specified, the assembler will produce a listing file with the same root as the input file and a filename extension of *.lst*. The listing file displays the source statements and their machine language equivalent. The listing also indicates the relative displacement of each machine instruction.

1.5 Optimizations

The assembler by default performs some optimizations on an assembly language source file, making just two passes through the assembly source file. Optimization can be disabled using the *-N* option; this causes the asssembler to run faster, since it makes just a single pass through the source and since it needn't optimize the code, but it makes the resultant code larger and slower.

The instructions affected by these optimizations are:

branches Long branches are converted to short if possible, and branches to the following location will be deleted.

movem If there are no registers, the instruction is deleted. If there is only one register, the shorter *move* instruction is substituted.

jsr *bsr* is substituted if possible.

To make these optimizations, the assembler uses a dynamically-allocated table. If this table is filled, the assembler will continue, will generate correct, but not completely optimized, object code, and will tell you the number of additional entries that it could have used. You can then recompile the module using the *-S* option to define a different table size.

1.6 Searching for *include* Files

By default the assembler searches just the current directory for files specified in *include* statements. It can also search a user-specified sequence of directories for such files, thus allowing program source files and header files to be contained in different directories.

The *-I* option and the *INCLUDE* environment variable define the directories in which the assembler will search for *include* files.

The assembler will automatically search just the current directory for a *include* file if the following conditions are met: (1) the assembler was started without a -I option having been specified, (2) *INCLUDE* is not an environment variable, and (3) the *include* statement doesn't specify the drive and/or directory containing the file.

If a *include* statement specifies either the drive or directory, just that location is searched for the file.

1.6.1 The -I option

The *-I* option defines a single directory to be searched for a file specified in a *include* statement. The path descriptor follows the *-I*, with no intervening blanks. For example, the specification

```
as -isys:db/include prog1
```

directs the assembler to search the *sys:db/include* area when looking for an *include* file.

Multiple *-I* options can be specified when the assembler is started, if desired, thus defining multiple directories to be searched.

1.6.2 The *INCLUDE* environment variable

The *INCLUDE* environment variable, if it exists, also defines directories to be searched for *include* files. This variable has the same format as the *PATH* environment variable; that is, it consists of the names of the directories to be searched, separated by semicolons. For example, the following command creates the *INCLUDE* environment variable, defining three directories to be searched:

```
set INCLUDE=work:/include;work:;sys:include
```

These directories are (1) the *include* directory on the *work:* volume; (2) the root directory on the *work:* volume; (3) the *include* directory on the *sys:* volume.

1.6.3 Include Search Order

When the assembler encounters a *include* statement, it searches directories for the file specified in the statement in the following order:

1. The current directory is searched.
2. The directories specified in the *-I* options are searched, in the order listed on the line that started the assembler;
3. The directories specified in the *INCLUDE* environment variable are searched, in the order listed.

2. Assembler Options

2.1 Summary of options

-O filename	Send object code to *filename*.
-Iarea	Defines an area to be searched for files specified in an *include* statement.
-L	Generate listing.
-N	Don't optimize object code.
-S#	Create squeeze table having # entries.
-P	Generate position-dependent code.
-U#	Use address register # as the data segment/jump table base register.
-V	Verbose option. Generate memory usage statistics.
-ZAP	This option is used primarily by the Aztec C68K C compiler and directs the assembler to delete the input file after processing.

2.2 Description of options

Multiple options to the assembler should be separated. The following will produce the desired results:

```
as -l -s x.asm
```

But this will not:

```
as -ls x.asm
```

If more than one option follows the - it sometimes happens that only the first option takes effect. To avoid the problem, specify the options separately.

The '-O *filename*' option

This option causes *as* to send the object code to *filename*. If this option isn't specified, *as* sends the object code to a file whose name is derived from that of the assembler source file by changing the extension to *.o*; in this case, the file is placed in the directory containing the source file.

-I Option

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

The name of the area immediately follows the -I, with no intervening spaces. For example, the following defines directory /source/inc on volume *sys:* search area:

-Isys:/source/inc

For more details, see the Assembler Operating Instructions, above.

The -L option

Causes *as* to generate a listing. The name of the file to which the listing is sent is derived from that of the source file by changing the extension to *.lst*. The listing file is placed in the directory containing the source file.

The -S option

The *-S* option defines the number of entries in the squeeze table. If this option isn't specified, the table contains 1000 entries.

The number of entries immediately follows the *-S*, with no intervening spaces. For example, the following option tells the assembler to use a squeeze table containing 1050 entries:

-s1050

The -P option

This option causes the assembler and linker to generate position dependent code.

If this option isn't specified, the assembler and linker produce position independent code. They do this by generating code that makes memory references as follows:

* Instructions access data in the initialized and uninitialized data segments via an index register, which is assumed to point to the first byte beyond the end of the two data segments. The index register can be specified with the assembler's -U option or with the USEA pseudo-op, and defaults to register A5.

* Instructions that call or jump to locations that are in the code segment are made PC-relative, if the referenced location is within 32K bytes of the instruction making the reference. Otherwise, the call instruction is made to call an item in the jump table, which in turn jumps to the location in the code segment. The jump table is pointed at by A5, so the call or jump instruction in the code segment uses A5 as an index register.

A program that is assembled without the -P option (ie, that is to be position independent) doesn't have to explicitly specify PC-relative and/or A5-relative addressing in its instructions. The assembler and linker will automatically generate the correct addressing mode for each instruction.

The -U# option

This option causes the assembler and linker to generate code that uses address register # to access the program's data segment and jump table. If this option isn't specified, and if the assembler pseudo-op USEA isn't specified, address register A5 is used.

3. Programmer Information

3.1 Source Program Structure

There are four types of Assembler statements:

1. Comments
2. Instructions
3. Directives
4. Macro Calls

3.1.1 Comments

A comment can appear after a semicolon or after the operand field. For example:

```
; this is a comment
    link a6,#.2   this is also a comment
```

3.1.2 Executable Instructions

Executable instructions have the general format:

```
label operation operand
```

Labels

Assembler labels can be any length. External labels are only significant for the first 8 characters. Any additional characters will be ignored. Valid label characters include letters, numbers, or the special characters . and _. A label cannot begin with a digit.

Labels that do not start in the first column require a colon suffixed.

Operations

The assembler recognizes all of the mnemonics found in Motorola's *16-bit MICROPROCESSOR USER'S MANUAL.*

To specify a length for instructions which support multiple lengths, it is sufficient to suffix the instruction mnemonic with:

.B Specifies a length of one byte
.W Specifies a length of 16-bits
.L specifies a length of 32-bits

Operands

The operand field consists of one expression, or two expressions separated by a comma with no imbedded spaces. An expression is comprised of register mnemonics, symbols, constants, or arithmetic combinations of symbols or constants.

Symbols or labels represent relocatable or absolute values. An absolute value is one whose value is known at assembly time. A relocatable value is one whose value is not known until the program is actually loaded into memory for execution.

Relocatable expressions can only be expressed arithmetically as sums or differences. The difference between two relocatable expressions is absolute. The result of summing two relocatable expressions is undefined.

There are five type of constants: octal, binary, decimal, hexadecimal and string. An octal constant is expressed as an @ followed by a string of digits from the set 0 through 7 such as **@123** or **@777**. A binary constant is expressed as a **%** followed by a string of ones and zeroes such as **%10101** or **%11001100**. A decimal constant is a string of numbers. A hexadecimal constant is a **$** followed by a string of characters made up of numbers or alphabetics from **a** through **f** such as **$ffff** or **1a2e**. A string constant is any string of characters enclosed in single quotes such as **'abdc'**.

Register mnemonics include the data register mnemonics **D0** through **D7**, the address registers **A0 through A7**, **SP** or **A7** the stack pointer, **PC** the program counter (forces PC relative mode), **SR** the status register, the condition code register **CCR**. And the user stack pointer **USP**.

The assembler supports addition (+), subtraction (-), multiplication(*), division (/), shift right (>>), shift left (<<), unary minus, and (&), or (|). The order of precedence is innermost parenthesis, unary minus, shift, and/or, multiplication/division, and addition/subtraction.

3.1.3 Directives

The following paragraphs describe the directives that are supported by the assembler.

EQU

label equ <expression>

This directive assigns the value of the expression on the right to the label on the left.

REG

label reg <register list>

This directive assigns the value of the register list to the label. Forward references are not allowed. A register list consists of a list of register names separated by the / character. The - character may be used to identify an inclusive set of registers. The following are valid register lists:

a0-a3/d0-d2/d4
a1/a2/a4/a6/d0-d2

PUBLIC

[label] public <symbol>[,<symbol>...]

This directive identifies the specified symbols as having external scope. These symbols are visable to the linker and are used to resolve references between modules. The type of the symbol is CODE if it appears within the code segment and DATA if it appears within the data segment.

GLOBAL and BSS

[label] global <symbol>,<size>
[label] bss <symbol>,<size>

These directives restore storage for uninitialized data items. The area is reserved in the uninitialized data area. If *global* is used then the data item is known to other modules that are external to the routine. If *bss* is used then the data item is local to the routine in which it is defined.

If a *global* is defined in more than one module then the linkage editor will reserve the maximum value of those assigned.

A symbol that appears in both a *global* and a *public* directive is located in the initialized data area and the global statements size parameters are ignored.

ENTRY

[label] entry <symbol>

This directive defines the entry point of the program. Only one entry can be declared per program. If no entry point is defined, the first instruction of the first module becomes the default entry point. The entry point must be in the first 32K of the root segment.

END

This directive defines the end of the source statements. All files are closed and the assembler terminates.

CSEG

Assembled output following this directive is output into the code segment of the program output file.

DSEG

Assembled output following this directive is placed in the initialized data segment of the program file.

DC - Define Constant

[label]	*dc.b*	*<value>[,<value>, <value> ...]*
[label]	*dc*	*<value>[,<value>, <value> ...]*
[label]	*dc.w*	*<value>[,<value>, <value> ...]*
[label]	*dc.l*	*<value>[,<value>, <value> ...]*
[label]	*dc.b*	*"string"*

The *dc* directive causes one or more fields of memory to be allocated and initialized.

Each <value> operand causes one field to be allocated and then to be initialized with the specified value. A <value> can be an expression. An expression may contain forward references.

For command programs, a value can contain a reference to a memory location whose address won't be known until the program is loaded into memory. In this case, an item for this value will be added to the program's relocation table; when the program is loaded, the field containing this value will be set to the correct value.

Each field for a particular *dc* directive is the same length. A period followed by b, w, or l can be appended to a directive, defining the field length to be one, two, or four bytes, respectively. If the field length isn't specified in this way, it defaults to 2 bytes.

Fields that are two or four bytes long are aligned on word boundaries.

The last form listed above for *dc* allocates a field having exactly the number of characters in the string, and places the string in it.

DCB - Define Constant Block

[label]	*dcb.b*	*<size>[,<value>]*
[label]	*dcb*	*<size>[,<value>]*
[label]	*dcb.w*	*<size>[,<value>]*
[label]	*dcb.l*	*<size>[,<value>]*

The *dcb* directive allocates a block of storage containing <size> fields, and initializes each field with <value>. If <value> isn't specified, it's assumed to be 0.

Each field for a particular *dcb* directive is the same length. A period followed by b, w, or l can be appended to a directive, defining the field length to be one, two, or four bytes, respectively. If the field length isn't specified in this way, it defaults to 2 bytes.

Fields that are two or four bytes long are aligned on word boundaries.

DS - Define Storage

[label]	*ds.b*	*<size>*
[label]	*ds*	*<size>*
[label]	*ds.w*	*<size>*
[label]	*ds.l*	*<size>*

This directive allocates a block of storage containing <size> fields, and sets each field to 0.

Each field for a particular *ds* directive is the same length. A period followed by b, w, or l can be appended to a directive, defining the field length to be one, two, or four bytes, respectively. If the field length isn't specified in this way, it defaults to 2 bytes.

Fields that are two or four bytes long are aligned on word boundaries.

PIND and NOPIND

The pseudo-ops *pind* and *nopind* cause the assembler to generate position-independent and position-dependent code, respectively.

LIST and NOLIST

The directives *list* and *nolist* turn on and off, respectively, the listing of assembly language statements to the listing file.

MLIST and NOMLIST

The directives *mlist* and *nomlist* specify whether or not the assembly language statements generated by a macro expansion should be written to the listing file.

CLIST and NOCLIST

The directives *clist* and *noclist* specify whether or not statements should be included in the listing file, when the statements were not assembled as a result of assembler conditional statements. By default, such statements are not listed.

INCLUDE

include <filename>

This directive causes the contents of the file specified to be processed by the assembler as if they had appeared at the same relative location as the *include* statement.

MACRO and ENDM

[label] macro <symbol>
...
text
...
endm

The specified symbol is entered in the assembler opcodes table. The text between the *macro* and *endm* is saved in memory. When the macro symbol is encountered as an opcode the text is placed in line. Up to nine arguments can be specified. They are referenced in the macro text as %1 through %9. In expanding a macro symbolic argument references are replaced by their actual value.

MEXIT

Upon encountering this directive expansion of the current macro stops and the assembler scans for the statement following the ENDM directive.

IF, ELSE, and ENDC

if <test>
...
[else]
...
endc

These directives are used to allow conditional assembly of parts of the input file. The general form of the IF test is:

<exp>		
<exp> == <exp>	||	*<exp> = <exp>*
<exp> != <exp>	||	*<exp> <> <exp>*
'str1' == 'str2'	||	*'str1' = 'str2'*
'str1' != 'str2'	||	*'str1' <> 'str2'*

If the test result is true, then the lines up to an *ELSE* or *ENDC* are assembled. If there is an *ELSE*, then lines up to the *ENDC* are skipped. The skipped lines are not displayed in the listing file unless the *CLIST* directive has been used. If the test is false, then lines are skipped until an *ELSE* or *ENDC* is encountered. If it is an *ELSE*, then the following lines up to an *ENDC* are assembled.

USEA

The directive *usea n* causes the assembler to generate code that uses address register n to access the program's jump table or data segments. By default, A5 is used.

3.2 Interfacing With C

Interfacing 68000 assembly language routines with C is relatively easy. The linkage conventions are straight forward and simple.

Register Conventions

It is the responsibility of an assembly language subroutine to preserve the values in registers A3 through A7 and D4 through D7. Register D0 through D3 and A0 through A2 are available as work registers. There is no need to preserve the values of work registers for other routines.

Register D0 contains the return value of the subroutine if the return type is non floating point. If the return value is floating point, then the return value is in a pseudo register with a global label of **f0**. The floating point pseudo register is 10 bytes long. The first bit is the sign bit. The next 15 bits are the exponent, and the last 8 bytes are the binary value.

Arguments to Subroutines

Upon entry register A7 points to the stack. The first item on the stack is a 32 bit absolute return address. The second item on the stack is the first argument to the subroutine, followed by the second, and so on. Arguments to C subroutines are passed by value. Therefore character, integer, long, and floating point arguments are copied onto the stack by value. Character items are promoted to type *int* before being pushed on the stack.

The C Run Time Environment

For an overview of the memory structure of the C run time environment, refer to the Technical Information chapter. The following discusses some points of interest to assembly language programmers in regards to the run time environment.

Programs are loaded into memory starting at the lowest available memory address. Above the program area is the system heap. Above the heap, is a free storage area. The free storage area is shared by the heap and the stack. The stack builds from high memory down while the heap builds from low memory up. Register A7 always points to the current top of stack. Above the stack is the initialized data area. Register A5 points to the top of this data area. Register A5 serves a dual purpose, it also points to the bottom of the run time jump transfer table. The use of the jump transfer table is described in the technical notes section.

Returning From a C Function

To return from a C function it is necessary to restore the values of registers D4 through D7 and A3 through A7 to the values they had at entry to the routine, to place the return value, if any, in register D0 or pseudo register **f0** if floating point, and to execute an RTS instruction. It is not necessary to restore values to registers that have not been changed.

Upon return to the calling routine, the stack still contains the arguments that were passed to the subroutine. The first argument is on the top of the stack.

3.3 Interfacing With Pascal

Assembly language routines can be written to interface with both C and Pascal.

The Pascal register usage conventions are the same as those for C with the exception of register D0. Pascal does not use D0 to return values from functions. Pascal returns from a function with a pointer to the returned value on the top of the stack or, if the value is four bytes or less in size, with the returned value itself on the top of the stack.

Pascal expects that upon entry to a subroutine that the top item on the stack is the return address, that the next entry is the last argument to the routine, followed by the next to the last argument, and so on. After the subroutine arguments is a 32-bit field for the return value or its address. Pascal passes pointers to arguments that are longer than four bytes. Otherwise the items themselves are placed on the stack. Character variables are promoted to 16-bit integers and are therefore right justified. Booleans have a one byte slack byte appended to insure alignment of other arguments. Booleans are, therefore, left aligned in a two byte field.

Register preservation rules for Pascal are the same as for C.

Pascal routines when they return do not leave the arguments to the subroutine on the stack. The stack instead is positioned to the 32-bit return value field that followed the arguments. The basic return sequence for a Pascal routine is, therefore:

* restore registers to initial values if necessary
* place the absolute return address into a register
* position the stack to the return value field
* place the return value or its pointer into the field
* jump to the return address.

New Options for Assembler (AS)

This appendix describes assembler changes for this release and should be placed at the end of the Assembler section in your manual.

New Processor Support

The assembler is partly redesigned and supports the MC68010, MC68020, and the MC68881 instruction sets and addressing modes in addition to those of the MC68000. The assembler defaults to assuming that only the MC68000 instructions are valid. The *MACHINE* and *MC68881* directives enable and/or disable the additional instructions and addressing modes.

There are a number of new options, replacements, and directives in this version of the assembler (*as*). Each will be described in detail.

The following options are new to the assembler

-C	Makes *large code* the default code memory model. May be overridden by the near code and/or far code directives.
-D	Makes *large data* the default data memory model. May be overridden by the near code and/or far code directives.
-*e*name[=val]	Creates an entry in the symbol table for *name* and assigns it the constant value *val.* If *val* is not specified, *name* is assigned the value 1.

The following options are replaced:

-U#	Linker automatically specifies A4 in place of A5 when linking drivers and desk accessories.
-P	User specifies compiler *+C* or assembler *-C* to generate large code memory model.
-Snum	Newly-designed algorithm generates squeeze tables. The algorithm is nonrecursive and therefore no longer requires more than a 4K stack. Space for the table is now dynamically allocated, so all instructions should be considered for squeezing. The new algorithm is orders of magnitude faster on large files.

The following new operators are supported:

!	- inclusive or
^	- exclusive or
~	- bitwise not
//	- modulo

The following new directives are supported:

BLANKS

blanks on/off
blanks yes/no
blanks y/n

This directive controls where the assembler will accept blanks or tabs in the operand field of the instruction.

The default setting of *on* allows blanks to be placed between any two complete items. With this setting all comments must be preceded by a ';'.

The blanks *off* setting treats a blank as the end of the operand field.

CNOP

label cnop n1,n2

This directive is used to force alignment on any boundary at a particular offset. The first value, *n1*, is an offset while the second value, *n2*, specifies the alignment to be used as the base of the offset. For example, to align to an even word boundary:

```
cnop  0,2
```

while to align to a long word boundary:

```
cnop  0,4
```

and finally to align to a word beyond a long word boundary:

```
cnop  2,4
```

Note that this will only take effect relative to the beginning of the current module's code or data. Normally, the linker will not align individual modules to long word boundaries. So, for this directive to work, it must either be the first module linked into the program, or else the *+A* option of the linker must be used to force long word alignment of modules.

EQUR

label equr register

This directive allows a register to be referenced by an alternate name. Reference to the new name is made without regard to case.

EVEN

label even

This directive forces alignment to a word (16 bit) boundary.

FAIL

fail

This directive causes the assembler to generate an error for this line. This can be used in macros which detect the incorrect number of arguments and wish to prevent assembly.

FREG

label freg <register list>

This directive is like the REG directive, except that it is used to specify the floating point registers of the MC68881. The list is either composed of the floating point registers *FP0* through *FP7* or of the floating point control registers *FPIAR/FPCR/FPSR*, but not both.

IFC and IFNC

ifc 'string1','string2'
ifnc 'string1','string2'

These conditionals check to see if the two strings are equal. If they are, the *ifc* will assemble the following code, while *ifnc* will skip it.

IFD and IFND

ifd symbol
ifnd symbol

These conditionals check to see if the specified symbol has been defined or not. If the symbol has been defined, then *ifd* will assemble the following code, while *ifnd* will not.

OTHER IFS

ifeq absolute_expression
ifgt absolute_expression
ifge absolute_expression
ifle absolute_expression
iflt absolute_expression
ifne absolute_expression

These conditionals perform a comparison of the value of the absolute expression to zero. If the specified condition is true, then the following assembly language is processed, otherwise it is skipped.

MACHINE

machine MC68000
machine MC68010
machine MC68020

This directive enables or disables the additional instructions and addressing modes associated with different processors in the MC68000 family.

MC68881

mc68881

This directive enables the MC68881 floating point instructions to be recognized and assembled by the assembler.

SECTION

label section name,CODE
label section name,DATA
label section name,BSS

This directive performs the same functions as the *cseg* and *dseg* directives. The name parameter, if present, is ignored at the current time. The type parameter is used to switch from CODE and back again. If only a name parameter is specified, the type defaults to CODE.

SET

label set expression

This directive assigns the value of the absolute expression to the symbol specified by *label*. This definition is similar to the *EQU* directive, with the exception that this symbol's value can be changed with another *SET* directive.

TTL

ttl title_string

This directive sets the title of the current module being assembled. This directive is implemented for compatibility with other assemblers and has no effect at the current time.

XDEF and XREF

xdef symbol
xref symbol

These directives are used to specify the definition and reference of global symbols. Currently these are both mapped onto the *PUBLIC* directive.

THE LINKER

Chapter Contents

The Linker

The Manx linker creates executable programs by linking together the pieces of the program, which, having been compiled and assembled, are in relocatable object format.

It can create three types of executable programs:

* Command programs, which can be activated by the operator from the SHELL or the Finder, or by another command program;
* Drivers, which other programs can call to access devices;
* Desktop accessories, which the operator can activate just like the standard Macintosh desktop accessories.

A command program can be made larger than available memory by dividing its code into several segments. Only a program's segments containing currently-executing functions need be in memory; when a memory-resident segment is no longer needed it can be 'unloaded' and its memory reused.

1. Introduction to linking

This section is a brief introduction to linking in general and the Manx linker in particular. It's intended for those with limited exposure to linkers, so if you have such exposure, you may want to skip this section and continue with the next.

Linking hello.o

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

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

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

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

The Linking Process

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

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

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

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

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

file 1

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

file 2

```
func2()
{
   return;
}
```

Libraries

A library is a collection of object files put together by a librarian. Libraries intended for use with *ln* must be built with the Manx librarian, *libutil.* This utility is described in the Utility Program chapter.

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

For Example

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

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

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

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

```
ln hello.o

ln c.lib hello.o
```

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

last of all.

The Order of Library Modules

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

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

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

ln main.o proglib.lib c.lib

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

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

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

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

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

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

Undefined symbol: put_value_

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

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

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

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

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

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

2. Using the Linker

As mentioned in the introduction to this chapter, the linker can create several types of executable programs. Much of the actual use of the linker is the same, regardless of the type of program generated.

This section describes how the linker is used to create a program, without getting into information which applies to a particular type of executable program.

For information specifically related to the creation of commands, programs, drivers, or desktop accessories, see the appropriate section of the Technical Information chapter. This section is divided into the following paragraphs:

2.1 Starting the linker

The linker is started with a command of the form:

```
ln [-options] file1 file2 ...
```

where

```
[-options]
```

are linker options and

```
file1, file2, ...
```

are the names of files containing object modules and libraries of object modules.

For example, the very simplest linker command, which will create an executable "hello, world" program, linking object code in *hello.o*, with modules in *c.lib*, is

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

This creates a command program in the file *hello*, which can be started by the SHELL.

2.2 Input files

The linker scans the input files in the order in which they are specified on the command line. If a file contains a single object module, the module is automatically included in the program being built.

If an input file contains a library of object modules, the linker will make one pass through the library, looking for modules containing a function which has been called by an already-included module and which is not in any already-included module. When such a module is found, it is included in the program being built.

In other words, when scanning a library of object modules, the linker only includes needed modules in the program it's building.

A file name passed to the linker has the standard SHELL format. That is, it consists of an optional volume name, optional path to the directory containing the file, and the filename itself. The volume defaults to the current volume and the directory to the current directory.

2.3 The executable file

The name of the file to which the linker writes the executable program can be specified using the linker's '-O' option. If this option isn't used, the linker derives the name of the executable file from that of the first object module listed on the command line, by deleting the extension. In the default case, the executable file is placed in the same directory in which the first object module is located.

For example, the following will link *main.o*, *menu.o* and *add.o* with the Manx library *c.lib*, all of which are in the current directory, and send the executable program to the file named *main* in the current directory:

```
ln main.o menu.o add.o c.lib
```

and the following will link the same modules, placing the output in *sys:/bin/myprog*

```
ln -o sys:/bin/myprog main.o menu.o add.o c.lib
```

The maximum size of an executable file is 1024K bytes.

2.4 Libraries

Two libraries are provided with Aztec C: *c.lib* and *m.lib*. All programs must be linked with *c.lib*; in addition to all the non-floating point functions described in the library chapter, it contains internal functions which are called by compiled-generated code, such as functions to process switch statements.

m.lib contains the transcendental floating point functions described in the library chapter, such as *sin*, and versions of the functions *printf*, *fprintf*, *sprintf*, *scanf*, *fscanf*, and *sscanf*. A program which calls any of these functions must be linked with *m.lib* in addition to *c.lib*. Otherwise, a program needn't be linked with *m.lib*. In particular, if it performs floating point operations without calling any of these functions, it doesn't need to be linked with *m.lib*.

When a program calls a printf- or scanf-type function to perform a floating point conversion, *m.lib* must be searched by the linker before *c.lib*. The reason for this is that there are two versions of these functions: the ones that support floating point are in *m.lib*; the others, which don't, are in *c.lib*. If *c.lib* is searched before *m.lib* when a program which requires the floating point versions is linked, the non-floating point version of a function will be used, and the program will misbehave.

Libraries of user-written object modules can also be searched by the linker. These are created by the Manx program *libutil*, and must be searched by the linker before the Manx libraries.

For example,

```
ln prog.o mylib.lib m.lib c.lib
```

creates an executable program, *prog* from the object module *prog.o*, pulling needed object modules from the libraries *mylib.lib*, *m.lib*, and *c.lib*.

2.5 The -L option

The -L option provides a convenient way to link programs located in one directory with libraries located in another.

The -L option is immediately followed by the partial name of a library file. The linker builds the complete name by prefixing it with the string associated with the environment variable CLIB and appending to it the string ".lib".

For example, if the libraries are located in the directory *sys:/lib*, then CLIB would be set to

```
sys:/lib/
```

and the object modules prog.o could be linked with the libraries *mylib.lib*, *m.lib*, and *c.lib* using the command

```
ln prog.o -lmylib -lm -lc
```

2.6 The -F option

This option causes the linker to continue reading options file name from a file; when done, it then continues reading arguments from the command line. The name of the file follows the -F option.

For example, the following command links *prog.o* with *sub1.o*, *sub2.o*, ..., *m.lib* and *c.lib*; it reads some file names from the file *prog.lnk*:

```
ln -f prog.lnk -lm -lc
```

where *prog.lnk* contains

```
-o prog.out
sub1.o sub2.o
sub3.o
sub4.o
```

2.7 Where to go from here

We have now presented all the information that is independent of the type of program being generated. For program-specific information, see the appropriate section of the Technical Information chapter.

3. Summary of Linker options

This section summarizes the linker options. The options are:

-O <file> Specifies the name of the file to which the executable program will be sent. If not given, the name of the file is the same as that of the first input file, with the extension deleted.

-F <file> Read linker arguments from the file *file*.

-T Produce a symbol table. The table is written to a file whose name is the same as that of the file containing the executable code, with extension *.sym*. Each symbol has an entry in the table containing the name of the symbol, the number of the segment containing it, and its offset from the beginning of the segment. See the -B option for more information about offsets.

-B val Add the hexadecimal value *val* to the offset of segment 0 symbols when producing a symbol table. This option is used to display the absolute load addresses of segment 0 symbols.

-M Produce a Finder-executable command program.

-D Create a driver.

-A Create a desktop accessory.

-S val Use the hexadecimal value *val* as the size of the stack area for the program. If this option isn't specified, a program is given an 8K byte stack area.

+O[i] Place the following object modules in code segment i. If no number is specified, use the first empty segment. If the segment already exists, the modules are added to its end.

-N name When creating a driver or desktop accessory, *name* is used as the name of the created resource. The name is prepended with a '.' for drivers and a '\0' for desktop accessories.

-I id When creating a driver or desktop accessory, its ID number is set to the decimal value *id*. If this option isn't used, the ID number is set to 31

-R attr When creating a driver or desktop accessory, its attributes are set to the hexadecimal value *attr*. If this option isn't used, the attributes are set to 0x30.

4. Linker Error Messages

This section discusses the error messages that the linker *ln* may display as it creates an executable program. It's divided into two subsections; the first summarizes the messages that *ln* can display, and the second explains the messages.

4.1 Summary of linker error messages

Command line errors:

1. unknown option '<bad option letter>'
2. too few arguments in command line.
3. No input given!
4. Cannot have nested -f options.
5. too few arguments in -f file: <filename>
6. Multiple entry points defined

I/O errors:

7. file <filename>: can't open
8. Cannot open -f file: <filename>
9. I/O error (<error number>) reading/writing output file
10. Cannot write output file
11. Cannot create output file: <filename>
12. Cannot create symbol table output

Corrupted object files:

13. object file is bad!
14. invalid operator in evaluate <hex value>
15. library format is invalid!
16. Cannot read module from <input> on pass2
    can't find symbol, <symbol name>, on pass two
17. Not an object file

Errors in use of Memory:

18. Insufficient memory!
19. Too many symbols!

Errors arising from source code:

20. Undefined symbol: <symbol name>
21. <symbol name> multiply defined
22. pass1(<hex value>) and pass2(<hex value>) values differ:
    symbol type differs on pass two: <symbol name>
23. Branch out of range @pc=<addr>
24. Short branch to next location @pc=<addr>
25. Entry point must be in root segment
26. Entry point must be in first 64K of program
27. Attempt to store out of bounds
28. Program is too large to link
29. Attempt to perform relocation in overlay code

30. data ref to overlay code not in jump table

4.2 Explanation of linker error messages

When started, *ln* first displays a message on the screen which indicates that the linker has been loaded and is running. If everything goes well, the linker will print on the screen several messages listing the sizes of the programs segments; then the linker will finish. The linker may encounter an error while it is running, in which case it will send a message to the screen.

Errors may be reported at a variety of points during the linking process. *ln* generates an executable program in two stages, known as pass 1 and pass 2. The size messages are printed at the end of pass 1, so any errors occurring after that have been detected during pass 2 of the linker.

Following is a list of the messages which the linker will generate in response to an error. The messages are grouped according to the source of the errors which cause them. Elements which are variable are enclosed by angled brackets: <>.

Command line errors:

1. **unknown option '<bad option letter>'**

   An option letter has been specified which the linker does not recognize. Only the letter will be ignored; everything else on the command line has been preserved, and the linker will try to execute what it has interpreted. See the linker chapter for a list of options which are supported.

2. **too few arguments in command line.**

   Several of the linker options have an associated value or name, such as -B 2000. If a needed value is missing, the linker will give this message and die.

3. **No input given!**

   The linker will quit immediately if not given any input to process.

4. **Cannot have nested -f options.**

   A file which is given as a -f argument can contain any option letter except -f itself. However, more than one -f is allowed on a command line.

5. **too few arguments in -f file: <filename>**

   An option letter specified in the file, "filename," requires a value or name to follow it. If an option appears at the end of the file, its associated value may not appear back on the command line.

6. **Multiple entry points defined**

   Multiple global symbols have been found that have the same name.

I/O errors:

7. can't open <filename>, err=<errno>

If any file in the command line cannot be opened, this message will be sent to the screen, specifying the filename and the current value of errno.

8. Cannot open -f file: <filename>

A file given with the -f option cannot be opened.

9. I/O error (<errno>) reading/writing output file

An error reading or writing the output file probably means there is no more disk space available. In particular, a block of the output file was written to disk and then could not be read back. The current value of errno is given in these messages.

10. Cannot write output file

See error 9.

11. Cannot create output file: <filename>

This message usually indicates that all available directory space on the disk has been exhausted.

12. Cannot create symbol table output

The -T option was given in the command line, but the file containing the linkage symbol table cannot be written to disk. It is possible that there is no more space on the disk.

Corrupted object files

13. object file is bad!

This is the most explicit indication that an object file in the linkage has been corrupted. The solution is simply to recompile and assemble the source file. A bad object file will not be discovered until the second pass of the linker.

14. invalid operator in evaluate <hex value>

This is really the same as error 14. Unless you have changed the object code by hand, the file has been corrupted.

15. library format is invalid!

A library in the linkage has been corrupted.

16. Cannot read module from <input> on pass2

or

can't find symbol, <symbol name>, on pass two

Either message indicates that a module has been corrupted between

pass 1 and pass 2. On a multiuser system, it is possible that another user changed the file while the linker was running. Otherwise, the error was probably due to a hardware failure.

17. Not an object file

A file given to the linker does not contain relocatable object code which LN can process. For instance, a source file may have been included in the link.

Errors in use of memory:

18. Insufficient memory!

The linkage process needs memory space for LN, global and local symbol tables, and approximately 5K for buffers. Just as with compilation, most memory use is devoted to the program software and symbol tables. Since LN is not especially large, only an extremely complicated linkage might run out of memory.

19. Too many symbols!

This is another way of saying that not enough memory was available for the symbol tables needed for the linkage.

Errors arising from source code:

20. Undefined symbol: <symbol name>

A global symbol name has remained undefined. This is commonly a function which has been referenced in the source code but not included anywhere in the link.

21. <symbol name> multiply defined

A global symbol has been defined more than once. For instance, if two functions are accidentally given the same name, this message will be generated.

22. pass1(<hex value>) and pass2(<hex value>) values differ:

or

symbol type differs on pass two: <symbol name>

Either of these errors may be generated during pass 2 when error 24 appeared in pass 1. They may be considered a confirmation of what was discovered in pass 1 of the linker.

23. Branch out of range @pc=<addr>

A branch or jump instruction was encountered that attempted to branch farther than it should. This error shouldn't be generated from C programs.

24. Short branch to next location @pc=<addr>

The 68K processor doesn't accept instructions of this type. This error shouldn't occur, since the Manx assembler will detect such instructions and remove them from the object code.

25. Entry point must be in root segment

The entry point for a program must be in the program's root segment. For example, this error would be generated if you tried to link the "hello, world" program with the command:

```
ln hello.o +O -lc
```

26. Entry point must be in first 64K of program

Not only must a program's entry point be in the root segment, it must also be in the first 64K bytes of this segment. The reason for this is that a 16-bit field in the jump table points to the entry point.

27. Attempt to store out of bounds

This error shouldn't occur. It indicates a linker bug.

28. Program is too large to link

This error shouldn't occur. It indicates a linker bug.

29. Attempt to perform relocation in overlay code

The only segments of a program which can contain addresses which must be relocated when the program is loaded are the program's root code segment and it's initialized data segment. The relocation of these two segments is performed when the program is first loaded, by the Manx-supplied startup code.

30. data ref to overlay code not in jump table

This error is caused by C programs that attempt to initialize a global pointer to a static function, where the function is contained in an overlay. Such initialization is permitted when the function is located in the root segment, but not when it is in another segment.

New Options for Linker (LN)

These pages describe linker changes for this release and should be placed at the end of the Linker section of your manual.

The following options are changed in the linker:

System Dependent options are prefixed with a "+", instead of a "-" as follows:

+A	Creates a desk accessory.
+D	Creates a driver.
+I id	Specifies a decimal ID number for driver or desk accessory.
+M	Produces a finder-executable application.
+N name	Gives name to created driver or desk accessory.
+R attr	Defines hex resource attribute value for driver or desk accessory.
+S val	Defines hex value for size of stack area.
+T	Generates a .map file that may be read and used by *TMON* to view symbols as code resource relative.

System Independent options are prefixed with a "-" as follows:

-F <file>	Reads linker arguments from file.
-G	Collects source level debug information and places it into a *.dbg* extension file. (To be used by *sdb* when it is available.)
-M	Turns off warnings.
-Q	Turns off source level debugging file generation.
-V	Specifies verbose link.
-W	Creates code resource SYMS used with db or the profiler that contains the required symbol table information.

The following options are removed:

-B Values for all symbols listed in the *.sym* symbol table file are now given with a zero offset rather than the 0xcc48 offset previously used.

+R The linker automatically detects MDS rel files and therefore no longer uses this option. The linker supports both MDS 1.0 and 2.0 object files, including 2.0 libraries.

Other Features

* The object module format is changed.

  Note: This change, and flexname changes in the compiler, prohibit linking of old libraries or object files. Use the *su* utility included with this release to switch trailing underscores in assembly source to leading underscores

* A new process substantially decreases link time.

* The linker automatically adds a ".o" extension to files that have no extension. It checks the current directory *and* all directories defined in the CLIB environment variable. This means that if you want to link with "segload.o", give the name and the linker will check the current directory and all the CLIB directories.

* The CLIB environment variable, used to specify where libraries may be found, supports multiple entries when they are separated by a ';'. For example, if libraries are in both the "sys2:lib" and "ram:" directories, then CLIB would be defined like this:

  set CLIB=sys2:lib;ram:lib/

  The null entry at the end means to check the current directory as well.

Z - The Text Editor

Chapter Contents

Z - the text editor

Z is a text editor which is especially useful for creating source programs in the C programming language. It has the following features:

* It's very similar to the Unix editor *vi*: if you know *vi*, you know *Z*.
* It's a full-screen editor: the screen acts as a window into the file being edited.
* Z has a wealth of commands, and commands are specified with just a few keystrokes, allowing editing to be performed quickly and efficiently. The simple and natural way of entering commands and the mnemonic assignment of commands to keys makes the commands easy to remember and use.
* Z has commands for the following:
  + Bringing different sections of a file into view;
  + Inserting text;
  + Making changes to text;
  + Rearranging text by moving blocks of text around and by inserting text from other files;
  + Accessing files;
  + Searching for character strings and "regular expressions".
* Z has several commands which are useful for editing C programs: there are commands for finding matching parentheses, square brackets, and curly braces; for finding the beginning of the next or preceding function; and for finding the next or preceding blank line.
* Most commands can be easily executed repeatedly.
* Sequences of commands, called macros, can be defined and executed one or more times.
* Changes are made to an in-memory copy of a file; the file itself isn't changed until a command is explicitly given;
* Z has a feature which is useful when editing a large number of related files: the operator can request that a file containing a certain function be edited; Z will automatically find the file and prepare it for editing.

Requirements

Z runs on several systems, including

* IBM PC, running PCDOS version 2.0 or later
* 8086-based systems running CP/M-86 and using an ADM-3A or LSI terminal;
* The Macintosh
* The Amiga
* TRS-80, model 4, using TRSDOS

For 8086-based systems, *Z* requires at least 128KB of memory and allows you to edit programs containing up to 58 K bytes of text.

For 8080- and Z80-based systems, *Z* requires 64 KB of memory and allows you edit programs containing up to 11 K bytes of text.

Components

The *Z* package contains two programs:

Z, the text editor;

ctags, a utility for creating a file which relates tags to C source files.

Preview

The remainder of this description of Z is divided into the following sections:

getting started , which describes how to quickly start using *Z*;

commands and features , which presents an overview of the features and commands of *Z*;

summary , which summarizes the *Z* commands.

1. GETTING STARTED

Z is a very powerful tool for creating and editing C source programs, but its wealth of commands and options can be overwhelming to someone not familiar with it. The purpose of this chapter is to get you using *Z* as quickly as possible, by presenting a small subset of the *Z* commands, with which programs can be created and edited. Then, with the ability to create and edit programs, you can continue reading the rest of this manual at your leisure to learn about the other features and commands of *Z*.

This section is divided into two subsections: the first describes how to create a new C program, and the second how to edit an existing program.

1.1 Creating a new program

Z is activated by entering a command of the form:

```
z hello.c
```

where *hello.c* is the name of the file to be edited. Since we're creating a new program, the file doesn't exist yet, so *Z* says so by displaying a message on its status line (which may be either the first or last line of the display, depending on the system on which Z is running). On systems that use the first display line for status information, the screen then looks like this:

```
"hello.c" no such file or directory
~
~
...
~
~
```

with the cursor on the left-hand column of the second line. On systems that use the last display line for status information, the screen looks like this:

```
~
~
...
~
hello.c doesn't exist
```

with the cursor on the left-hand column of the first line.

Z is now waiting for you to enter a command.

The screen

As mentioned above, *Z* uses the one line of the display for displaying information and for echoing the characters of some commands which are entered. On the Macintosh, the last line is the status line; on other systems, the first line is the status line.

The rest of the lines on the screen are used to display text of the file being edited.

The tilde characters on the screen lines are *Z*'s way of saying that the end of the file has been reached: these characters are not actually in the file.

Modes of Z

Z has two modes: command and insert, which allow you to enter commands and to insert text, respectively.

In this section, we'll spend most of our time in insert mode, using commands only to enter insert mode and to exit *Z*. When we get to the next section, in which we edit a file, we'll discuss more commands.

Insert mode

With *Z* in insert mode, characters that you type are entered into a memory-resident buffer; the characters don't appear in the file until you exit insert mode and explicitly issue a command which causes *Z* to write the buffer to the file.

Z has several commands for entering insert mode; the one we want to use is *i*, which allows text to be entered before the cursor. So type *i*. Notice that *Z* doesn't echo this command on the screen; it only does that for a few commands. Notice also that we are in command mode, as evidenced by the message

<insert mode>

on the right-hand side of the status line.

Now you can enter a program, just as you would on a typewriter. Notice that the cursor is positioned where the next character will be entered. Try entering the "hello world" program:

```
main()
{
   printf("hello, world\n");
}
```

When you hit the <return> key after entering the printf line, the cursor was left positioned on the next line of the screen underneath the first non-white space character of the preceding line. This feature, which is known as "autoindent", is useful when creating C programs, encouraging statements within a compound statement to be indented and lined up. Autoindent can be disabled and enabled, and we'll show you how later.

We want the closing curly brace of the main function to be on the first column of the line, not indented. So type the backspace key to get back to the first column, and then type the '}' key.

The backspace key can also be used to backspace over characters that you incorrectly type.

When you're done inserting the program, hit the escape key to exit insert mode and return to command mode. The key used as the escape key varies from system to system. On the IBM PC, the key labeled ESC is the escape key. On the TRS-80, models III and 4, the key labeled BREAK is the escape key. And on the Macintosh, the backquote key, `, is the escape key.

Exiting Z

To write the program you've just entered from the text buffer to the disk file *hello.c* and then exit Z, type *ZZ*.

Occasionally you may want to exit Z without writing the text you've entered to a file; in this case, type

```
:q!
```

followed by a carriage return, CR.

1.2 Editing an Existing File

In this section we're going to present a few commands which will allow you to make changes to an existing file.

Starting and stopping Z

You get in and out of *Z* when editing an existing file just as you do when creating a new file. To start *Z*, enter

```
z hello.c
```

where hello.c is the name of the file to be edited. And to stop *Z* and save the changes you've made, put *Z* in command mode and enter:

```
ZZ
```

Z knows if you made changes to the original text or not; if you did, it saves the original file by changing the extension of its name to *.bak* and then writes the modified text to a new file having the specified name. If a *.bak* file with that name already exists, it will be deleted before the rename occurs.

If you didn't make any changes, the *ZZ* command causes *Z* to halt without changing any disk files.

The command *:q!* will cause *Z* to halt without writing anything to the file being edited.

Going back to the startup of *Z*, *Z* reads the specified file into the text buffer, displays the first screenful of the file's text, displays the file's statistics (name, number of lines, number of characters) on the status line, positions the cursor at the first character of the first line, and enters command mode, waiting for you to enter a command.

The cursor

Before describing the commands for viewing and changing the text in *Z*'s memory-resident buffer, we need to discuss the cursor.

In *Z*, the character position in the text which is pointed at by the cursor acts as a reference point: most commands perform an action relative to that position. For example, the *i* command, described in the last section, allows you to enter text before the cursor. And the *x* command, to be discussed, deletes the character at which the cursor is located.

So we will be describing two types of commands in this section: those that move the cursor around in the text, thus bringing different sections of text into view, and those that modify text in the vicinity of the cursor.

Moving around in the text: scrolling

The text you created in section 1, for the "hello, world" program, easily fit on a single screen. But most text files are too large to be

viewed all at once, so we need commands to bring different sections into view.

Two such commands are the "scroll" commands: "scroll down", represented by the character control-D, and "scroll up", represented by control-U. That is, to execute the "scroll down" command, you hold down the control key and then depress the 'D' key.

The key used as the control key differs from system to system. On the IBM PC, it's the key labeled 'Ctrl'. On the Macintosh, it's the cloverleaf key (the key next to the 'Option' key that has the unusual symbol).

In the rest of this manual, we will refer to control characters using notation of the form ^D rather than control-D, for brevity. Thus, the "scroll up" and "scroll down" commands are represented as ^U and ^D, respectively.

A scroll command moves the screen up or down in the file, bringing another half-screen's worth of text into view. It's as if the text was on a reel of tape and the screen is a viewer: scrolling down moves the viewer down the reel, and scrolling up moves the viewer up the reel.

When scrolling, the cursor will be left on the same position within the text after the scroll as before, if that position is still within view. Otherwise, the cursor is moved to a line in the text which was newly brought into view.

Moving around in the text: the 'Go' command

Scrolling is one way to move around in the text, but it's slow. If we have a large text file, to which we want to append text, it would take a long time and many scroll commands to reach the end.

The *go* command, *g*, is one way to move rapidly to the point of interest in the text: entering *g* by itself will move the cursor to the end of the text and, if necessary, redraw the screen with the text which precedes it.

The *g* command can also be preceded by the number of the line of interest; in this case, the cursor is moved to the beginning of that line. So to move back to the first line of text, enter:

```
1g
```

The *g* command can be used to move to any line within the text, but since you usually don't know the numbers of the lines, the *g* command is mainly used to move to the beginning and end of the text.

Moving around in the text: string searching

So, scrolling allows us to take a casual stroll through text, and the *g* command to move rapidly to the beginning and end of the file. What

we need is a command to rapidly move to a specific point in the middle of the text.

The "string search" command, /, is such a command. When you enter /, followed by the string of interest, followed by a carriage return, *Z* searches forward in the text from the cursor position, looking for the string If *Z* reaches the end of the text without finding the string, it will "wrap around", and continue searching from the beginning of the text.

If the string is found, the cursor is positioned at its first character and, if necessary, the screen is redrawn with its surrounding text.

If the string isn't found, a message saying so is displayed on the status line of the screen and the cursor isn't moved.

While the "string search" command and its string are being entered, the characters are displayed on the status line, and normal editing operations can be used, such as backspacing over mistyped characters.

Z remembers the last string searched for. To repeat the search, enter the "find next string" command, *n*.

Finely tuned moves

With the commands presented up to now, you can move to the area of interest in the text. The next few paragraphs present commands which move the cursor from somewhere within the area of interest to a specific character position, from which changes will be made.

Some commands for this, from the many available in *Z*, are:

- and *CR* (carriage return)
  Move the cursor up and down one line, respectively, to the first non-whitespace character on the line;

space and backspace
  Move the cursor right and left, respectively, on the line on which the cursor is located.

These commands can be preceded by a number, which cause the command to be performed the specified number of times. For example,

```
3-
```

moves the cursor up three lines, and

```
5<space>
```

moves the cursor right five characters. Note that <space> represents the space bar.

Deleting text

You now have a repetoire of commands which allow you to move the cursor fairly quickly to any location in a text file. We're ready to

move on to a few commands for modifying the text.

Two such commands, for deleting text, are "delete character", *x*, and "delete line", *dd*:

x	Deletes the character under the cursor;
dd	Deletes the entire line on which the cursor is located.

Each of these commands can be preceded by a number, causing the command to be repeated the specified number of times. For example,

```
2x
```

deletes two characters, and

```
3dd
```

deletes three lines.

More insert commands

You already know one command for inserting text: *i*, which allows text to be inserted before the cursor. We need a few more insert commands:

a	Enters insert mode such that text is inserted following the cursor;
o	Creates a blank line below the current line (ie, the line on which the cursor is located), moves the cursor to the new line, and enters insert mode;
O	Same as *o*, but the new line is above the current line.

Summary

With the set of commands presented in this chapter, you can edit any text file. You should continue reading this manual, to learn more about *Z*, while you use the basic command set for performing your editing chores.

You'll find that *Z* has many more capabilities, which allow you to perform functions more quickly, with fewer keystrokes, than with the basic command set, and which allow you to perform functions which you can't perform with the basic command set.

The commands in the basic set are listed on the next page.

Starting and stopping Z

Z filename	Start Z, and prepare 'filename' for editing
ZZ	Stop Z, and write modified text to the edit file
:q!	Stop Z, without writing anything to the edit file

scrolling

^D	Move down half a screenful
^U	Move up half a screenful

Moving the cursor

g	Move the cursor to the end of the text, or to a specific line
/str	Search for the character string "str" and move the cursor to it
n	Search again, using the same string
-	Move cursor up a line
CR	Move cursor down a line
space	Move cursor right one character
backspace	Move cursor left one character

Inserting text

i	Insert before cursor
a	Insert after cursor
o	Insert new line below current line
O	Insert new line above current line

Deleting text

x	Delete character under cursor
dd	Delete line on which cursor is located

2. More commands

In this section we're going to describe the rest of the features and commands of Z, building and expanding on the information presented in the previous chapter. The section is organized into subsections; some describe a group of related commands, some a particular feature, and some how to perform a specific function with Z.

2.1 Introduction

Before getting into the *Z* commands, we want to discuss in more detail the way that *Z* displays information on the screen and the way that commands are entered.

2.1.1 The screen

We've already discussed the basic details on *Z*'s use of the screen. There's just a few more things to discuss: the display of unprintable characters and the display of lines which don't fit on the screen.

2.1.1.1 Displaying unprintable characters

A file edited by *Z* can contain any character whose ASCII value in decimal is less than 128, including unprintable characters, such as SOH, LF, and ESC. *Z* displays unprintable characters as two characters; the first is ^, and the second is the character whose ASCII value equals that of the character itself plus 0x40. For example, the unprintable character SOH is displayed as the pair of characters ^*A*, since the ASCII value of SOH is 1, and 1 plus 0x40 is 0x41, which is the ASCII value for the character 'A'.

2.1.1.2 Displaying lines that don't fit on the screen

In the previous chapter we said that lines beyond the end of the file are displayed with the character ~ in the first column of the line on the screen. When you see the ~ character in the leftmost column of a line on the screen, this usually signifies that this line of the display doesn't contain a line of text. Lines which don't fit on the screen are displayed by *Z* in a similar manner, as you'll soon see.

Z allows lines to be entered which are longer than a screen line. Normally, *Z* simply displays such lines on several screen lines. In some cases, however, the entire line won't fit on the screen. For example, if the cursor is positioned at the beginning of the file, it may not be possible to display the text of an entire big line at the bottom of the screen. In this case, *Z* displays an @ character in the first column of the screen lines on which the text would be displayed.

Thus, when you see the @ character in the leftmost column of a line on the screen, this usually signifies that the text which would have appeared on this line of the screen was too big, and not that the @ character is in the text.

2.1.2 Commands

When most commands are entered, *Z* doesn't echo the characters on the screen. For some commands, however, it does. In this latter category are the commands whose first character begins with *:* and with the string search commands.

For these commands, the characters are displayed on the screen's status line, and can be backspaced over and reentered, if necessary.

Also, *Z* doesn't act on such commands until you type the carriage return key, CR.

2.1.3 Special Keys

There are two keys that have special meaning for *Z*: the escape key, which is used to exit insert mode, and the control key, which is used in conjunction with another key to generate control characters. The actual keys used for these functions varies from system to system, as mentioned in the previous chapter.

The escape key is ESC on the IBM PC. On the TRS-80, models III and 4, it's the BREAK key. And on the Macintosh, it's the backquote key, `.

The control key is 'Ctrl' on the IBM PC. On theMacintosh it's the key next to the 'Option' key that has the cloverleaf symbol.

On the Macintosh, there are times when you want to generate a backquote, and not escape. For example, backquote is a cursor motion command to *Z*. To generate backquote, hold down the control key (the key next to the option key), and then type backquote.

2.2 Paging and Scrolling

In the last chapter we described commands for scrolling through text, ^*U* and ^*D*. Another pair of commands allow you to page, instead of scroll, through text. They are ^*B* and ^*F*, which page backwards and forwards, respectively.

A page command brings the previous or next screenful of text into view by redrawing the screen with the new text. Whereas scrolling was described as a viewer moving over a reel of tape, paging can be described as the turning of pages of a book.

Paging moves you through text more quickly than scrolling does. However, since paging redraws the screen all at once, while scrolling changes it gradually, it's often more difficult to keep a sense of continuity when paging than when scrolling. As an aid to continuity when paging, two lines of text which were previously in view are still in view after paging.

In the discussion of scrolling in the last chapter, we neglected to mention that the scroll commands can be preceded by a value specifying the number of lines to be scrolled up or down. If a number isn't specified, the last scroll value entered is used; if a scroll value was never entered, it defaults to half a screen's worth of lines. Separate values are maintained for scrolling up and for scrolling down.

The scrolling and paging commands necessarily move the cursor within the text, but they can't be used to home the cursor to an exact position at which changes are to be made. For this, you'll have to use commands described in subsequent sections.

2.3 Searching for strings

In the previous chapter, we described the string search command, /, which causes Z to scan forward, looking for the string. In this section, we describe the rest of the searching capabilities of Z. First, the rest of the string searching commands are described; then, the capability of Z to match patterns called "regular expressions", of which specific character strings are a special case, is described.

2.3.1 The other string-searching commands

The other string-searching commands are:

? Behaves like /, but Z finds the previous occurence of the string rather than the next;

n Repeats the last string-search command;

N Repeats the last string-search command, but in the opposite direction;

:se ws=0 and :se ws=1
Turns the wrap scan option off or on, respectively.

When Z reaches the end or beginning of text without finding the string of interest, it normally "wraps around" to the opposite end of the text and continues the search. It does this because by default the "wrap scan" option is on. This option can be disabled by entering the "set option" command:

:se ws=0

thus causing the search to end when it reaches the end of text. The option can be reenabled by entering:

:se ws=1

Note that for this colon command, as for all colon commands, carriage return must be typed before the command is executed.

2.3.2 Regular expressions

The string you tell Z to search for is actually a "regular expression". A regular expression is a pattern which is matched to character strings. The pattern can define a specific sequence of characters which comprise the string; in this case, only that specific string matches the pattern. The pattern can also contain special characters which match a class of characters; in this case, the pattern can match any of a number of character strings.

For example, one such special construct is square brackets surrounding a character string; this matches any character in the enclosed string. So the regular expression

ab[xyz]cd

matches the strings

abxcd
abycd
abzcd

Another special character is *, which matches any number of occurences of the preceding pattern. For example, the regular expression

ab*c

matches many strings, including

ac
abc
abbc

and so on. And the pattern

ab[xyz]*cd

matches many strings, including:

abcd
abxcd
abxycd
abzzxcd

and so on.

The complete list of special characters and constructs that can be included in regular expressions is:

^	When the first character of a pattern, it matches the beginning of the line
$	When the last character of a pattern, it matches the end of the line;
.	Matches any single character;
<	Matches the beginning of a word;
>	Matches the end of a word;
[str]	Matches any single character in the enclosed string;
[^str]	Matches any single character *not* in the enclosed string;
[x-y]	Matches any character between x and y;
*	Matches any number of occurences of the preceding pattern.

2.3.3 Disabling extended pattern matching

The "magic" option enables and disables the extended pattern matching capability. To turn off this option, enter:

:se ma=0

And to turn it on, enter:

:se ma=1

By default, extended pattern matching is disabled.

With the magic option off, only the characters ^ and $ are special in patterns.

2.4 Local Moves

In this section we're going to present more commands for moving the cursor fairly short distances; up or down a few lines, along the line on which it's located, and so on. We've already presented several, namely CR (carriage return), space, and backspace; but there are many more, reflecting the importance of finely-tuned, quickly-executed movements to the user.

2.4.1 Moving around on the screen:

Here are some commands for moving the cursor short distances:

h	Moves to the left one character;
j	Moves down one line, leaving the cursor in the same column;
k	Moves up one line, leaving the cursor in the same column;
l	Moves right one cursor;

The keys ^H, LF, ^K, and ^L are synonyms for h,j,k, and l, respectively.

These commands can be preceded by a number, which specifies the number of times the command is to be repeated.

Z has commands for moving the cursor to the top, middle, and bottom of the screen; they are H, M, and L, respectively. The cursor is positioned at the beginning of the line to which it's moved.

Remember the - command, which moved the cursor up a line, to the first non-whitespace character? As you might expect, + will move the cursor down a line, to the first non-whitespace character. + is thus equivalent to CR, the command presented in the last chapter.

2.4.2 Moving within a line

We've already presented several commands for moving the cursor around within the line on which it's located:

h, ^H, backspace	Left one character;
l, ^L, space	Right one character;

Here are a few more:

^	Moves the cursor to the first non-whitespace character on the line;
0	Moves to the first character on the line;
$	Moves to the last character on the line;

A few commands fetch another character from the keyboard, search for that character, beginning at the current cursor location, and leave the cursor near the character:

f	Scan forward, looking for the character, and leave the

cursor on it;

t Same as *f*, but leave the cursor on the character preceding the found character;

F Same as f, but scan backwards;

T Same as t, but scan backwards.

; Repeat the last f, t, F, or T command;

, Repeat the last f, t, F, or T command in the opposite direction.

Finally, the command | moves the cursor to the column whose number precedes the command. For example, the following command moves the cursor to column 56 on the current line:

```
56|
```

2.4.3 Word movements

Z has several commands for moving the cursor to the beginning or end of a word which is near the cursor:

w Moves to the beginning of the next word;

b Moves to the beginning of the previous word;

e Moves to the end of the current word.

For the preceding commands, a "word" is defined in the normal way: a string of alphabetical and numerical characters surrounded by whitespace or punctuation. There is a variant of each of these commands, differing only in the definition of a "word": they think that a word is any string of non-whitespace characters surrounded by whitespace. The variant of each of these commands is identified by the same letter, but in upper case instead of lower:

W Moves to the beginning of the next big word;

B Moves to the beginning of the previous big word;

E To the end of the current big word.

Each of these commands can be preceded by a number, specifying the number of times the command is to be repeated. For example,

```
5w
```

moves forward five words.

The word movement commands will cross line boundaries, if necessary, to find the word they're looking for.

2.4.4 Moves within C programs

Z has several commands for moving the cursor within C programs:

]] and [[Move to the opening curly brace, {, of the next or previous function, respectively;

% Move to the parenthesis, square bracket, or curly bracket which matches the one on which the cursor is currently located;

{ and } Move to the preceding or next blank line.

The [[and]] commands assume that the opening and closing curly braces for a function are in the first column of a line, and that all other curly braces are indented.

As an example of the '%' command, given the statement:

```
while (((a = getchar()) != EOF) && (c != 'a'))
```

with the cursor on the parenthesis immediately following the 'while', the % command will move the cursor to the last closing parenthesis on the line.

2.4.5 Marking and returning

Z has commands which allow you to set markers in the text and later return to a marker. Twenty six markers are available, identified by the alphabetical letters.

Unlike the other commands described in this section, these commands are not limited to moves within the current area of the cursor - they can move the cursor anywhere within the text.

A marker is set at the current cursor location using the command

*m*x

where x is the letter with which you want to mark the location.

There are two commands for returning to a marked position:

`x Moves the cursor to the location marked with the letter 'x';

'x Moves the cursor to the first non-whitespace character on the line containing the 'x' marker.

Remember, to generate backquote on the Macintosh, hold down the control key and then type backquote.

Occasionally, you may accidently move the cursor far from the desired position. There are two single quote commands for returning you to the area from which you moved:

" Returns the cursor to its exact starting point;

" Returns the cursor to the first non-whitespace character on the line from which the cursor was moved.

For example, if the cursor is on the line:

```
if (a >= 'm' && a <= 'z')
```

at the character '<', then following a command which moves the cursor far away, the command " will return the cursor to the '<' character, and the command " will return it to the beginning of the word 'if'.

2.4.6 Adjusting the screen

The *z* command is used to redraw the screen, with a certain line at the top, middle, or bottom of the screen.

To use it, place the cursor on the desired line, then enter the *Z* command, followed by one of these characters:

CR	To place the line at the top of the screen;
.	To place it in the middle of the screen;
-	To place it at the bottom.

The *z* command isn't a true cursor motion command, because the cursor is in the same position in the text after the command as before.

On the Macintosh, control L repaints the screen.

2.5 Making changes

That concludes the presentation of cursor movement commands. The next several sections describe commands for making changes to the text.

2.5.1 Small changes

In this section we present commands for making small changes. We've already presented two such commands in the previous chapter:

x	Which deletes the character at which the cursor is located;
dd	Which deletes the line at which the cursor is located.

The other commands are:

X	Delete the character which precedes the cursor; can be preceded by a count of the number of characters to be deleted;
D	Delete the rest of the line, starting at the cursor position;
rx	Replace the character at the cursor with 'x';
R	Start overlaying characters, beginning at the cursor. Type the escape key to terminate the command. (Remember, this key differs from system to system);
s	Delete the character at the cursor and enter insert mode; when preceded by a number, that number of characters are deleted before entering insert mode;
S	Delete the line at the cursor and enter insert mode; when preceded by a count, that number of lines are deleted before entering insert mode;
C	Delete the rest of the line, beginning at the cursor, and enter insert mode;
J	Join the line on which the cursor is positioned with the following line; when preceded by a count, that many lines are joined.

2.5.2 Operators for deleting and changing text

Z has a small number of commands, called 'operators', for modifying text. They all have the same form, consisting of a single letter command, optionally preceded by a count and always followed by a cursor motion command. The count specifies the number of times the command is to be executed. The command affects the text from the current cursor position to the destination of the cursor motion command, if the starting and ending position of the cursor are on the same line. If these positions are on different lines, the command affects all lines between and including the lines which contain the starting position and ending positions.

In this section, we're going to describe the operators for deleting and changing text, *d* and *c*:

d	Deletes text as defined by the cursor motion command;
c	Same as *d*, but *Z* enters insert mode following the deletion.

For example,

dw	Deletes text from the current cursor location to the beginning of the next word;
3dw	Deletes text from the cursor to the beginning of the third word;
d3w	Same as '3dw';
db	Deletes text from the current to the beginning of the previous word;
d'a	Deletes text from the cursor to the marker 'a', if the marker and the starting cursor position are on the same line. Otherwise, deletes lines from that on which the cursor is located through that on which the marker is located; On the Macintosh, generate backquote by holding down the control key and then typing the backquote key.
d/var	Deletes text either from the cursor to the string "var" or between the lines at which the cursor is currently located and that on which the string is located.
d$	Deletes the rest of the characters on the line, and hence is equivalent to *D*.

2.5.3 Deleting and changing lines

In the last chapter, we presented a command for deleting lines: *dd*. As you can see now, this is a special form of the *d* command, because the character following the first *d* is not a cursor motion command.

For all the operator commands, typing the command character twice will affect whole lines. Thus, typing *cc* will clear the line on which the cursor is located and enter insert mode. Preceding *cc* with a number will compress the specified number of lines to a single blank line and enter insert mode on that line.

2.5.4 Moving blocks of text

When text is deleted using the *d* or *c* command, it's moved to a buffer called the "unnamed buffer". (There are other buffers available, which have names. More about them later).

Data in the unnamed buffer can be copied into the main text buffer using one of the "put" commands:

p	Copies the unnamed buffer into the main text buffer, after the cursor;

P Same as *p*, but the text is placed before the cursor.

Thus, the delete and put commands together provide a convenient way to move blocks of text within a file.

The contents of the unnamed buffer is very volatile: when any command is issued that modifies the text, the text which was modified is placed in the unnamed buffer. This is done so that the modification can be 'undone', if necessary, using one of the 'undo' commands. For example, if you delete a character using the *x* command, the deleted character is placed in the unnamed buffer, replacing whatever was in there. So you have to be careful when moving text via the unnamed buffer: if you delete text into the unnamed buffer, expecting to put it back somewhere, then issue another command which modifies the text before issuing the put command, the deleted text is no longer in the unnamed buffer.

As you'll see, the named buffers can also be used to move blocks of text, and their contents are not volatile.

2.5.5 Duplicating blocks of text: the 'yank' operator

The 'yank' operator, *y*, copies text into the unnamed buffer without first deleting it from the main text buffer. When used with the 'put' command, it thus provides a convenient way for duplicating a block of text.

Since *y* is an operator, it has the same form as the other operators: an optional count, followed by the *y* command, followed by a cursor motion command. The command yanks the text from the cursor position to the destination of the cursor motion command, if the starting and ending positions are on the same line. If they are on separate lines, a whole number of lines are yanked, from that on which the cursor is currently located through that to which the cursor would be moved by the cursor motion command. The text is yanked into the unnamed buffer.

For example,

yw Copies text from the cursor to the next word into the unnamed buffer;

y3w Copies text from the cursor to the beginning of the third word;

3yw Same as 'y3w';

y`a Copies text from the cursor location to the marker 'a' into the unnamed buffer, if the two positions are on the same line. Otherwise, copies entire lines between and including those containing the two positions.

As a special case, the command *yy* will yank a specified number of whole lines. The command *Y* is a synonym for *yy*. For example,

yy Yanks the line at which the cursor is located;

3Y Yanks three lines, beginning with the one on which the cursor is located.

2.5.6 Named buffers

In addition to the unnamed buffers, *Z* has twenty six named buffers, each identified by a letter of the alphabet, which can used for rearranging text. Text can be deleted or yanked into a named buffer and put from it back into the main text buffer.

The advantage of these buffers over the unnamed buffer in rearranging text is that their contents are not volatile: when you put something in a named buffer, it stays there, and won't be overwritten unexpectedly. Also, as you'll see, the named buffers can be used to move text from one file to another.

To yank text into a named buffer, use the yank operator, preceded by a double quote and the buffer name, and followed by a cursor motion command. For example, the following will yank three words into the 'a' buffer:

```
"ay3w
```

and the following yanks four lines into the 'b' buffer, beginning with the line on which the cursor is located:

```
"b4yy
```

Text is deleted into a named buffer in the same way: the delete command is used, preceded by a double quote and the buffer name. For example, to delete characters from the cursor to the 'a' marker into the 'h' buffer:

```
"hd`a
```

The preceding command, when the source and destination cursor positions are on separate lines, will delete a number of whole lines into the 'h' buffer, from that on which the cursor is initially located through that containing the destination position.

As you remember, on the Macintosh, the backquote key is interpreted as the escape key. To generate a real backquote for use in the preceding example, you must hold down the control key (the key with the strange symbol next to the option key) and then type backquote.

To delete ten lines into the 'c' buffer:

```
"c10dd
```

Text in a named buffer is put back into the main text using the 'put' commands *p* and *P*, preceded by a double quote and the buffer name. For example:

ap	Puts text from the 'a' buffer, after the cursor;
zP	Puts text from the 'z' buffer, before the cursor.

2.5.7 Moving text between files

The named buffers are conveniently used to move text from one file to another. First yank or delete text from one file into a named buffer; then switch and begin editing the target file, using the *:e* command:

```
:e filename
```

(More on this later). Then move the cursor to the desired position; then put text from the named buffer.

This technique only works when using named buffers, not with the unnamed buffer. When switching to a new file, the unnamed buffer is cleared, but the named buffers are not.

2.5.8 Shifting text

The last two operator commands to introduce are the 'shift' operators, < and >, which are used to shift text left and right a tabwidth, respectively.

For example,

```
>/str
```

shifts right one tab width the lines from that on which the cursor is located through that containing the string "str".

Following the standard operator syntax, repeating the shift operator twice affects a number of whole lines:

5<<	Shifts five lines left;
>>	Shifts one line right.

2.5.9 Undoing and redoing changes

Z remembers the last change you made, and has a command, *u*, which undoes it, restoring the text to its original state.

Z also remembers all the changes which were made to the last line which was modified. Another 'undo' command, *U*, undoes all changes made to that line.

Finally, the period command, '.', reexecutes the last command that modified text.

2.6 Inserting text

We've already presented most of the commands for entering insert mode:

a	Append after cursor;
i	Insert before cursor;
o	Open new line below cursor;
O	Open new line above cursor;
C	Delete to end of line, then enter insert mode;
s	Delete characters, then enter insert mode;
S	Delete lines, then enter insert mode;

In this section we want to present the remaining few commands for entering insert mode, and present some other features of insert mode.

2.6.1 Additional insert commands

The other commands for entering insert mode are:

A	Append characters at the end of the line on which the cursor is located. This is equivalent to *$a*;
I	Insert before the first non-whitespace character on the current line. This is equivalent to *^i*.

2.6.2 Insert mode commands

Some editing can be done on text entered during insert mode, using the following control characters:

backspace	Delete the last character entered;
^H	Same as 'backspace' character;
^D	Same as "backspace";
^X	Erase to beginning of insert on current line;
^V	Enter next character into text without attempting to interpret it.

^V is used to enter non-printing characters into the text. For example, to enter the character 'control-A' into the text, type

```
^V^A
```

That is, hold down the control key, then type the 'V' key, then the 'A' key, then release the control key. As mentioned earlier, non-printing characters are displayed as two characters: '^' followed by a character whose ASCII code equals that of the non- printing character plus 0x40.

2.6.3 Autoindent

The *Z* 'autoindent' option is useful when entering C programs. When you are in insert mode and type the 'carriage return' key, with the autoindent option enabled, the cursor will be automatically indented on the new line to the same column on which the first non-whitespace character appeared on the previous line. This feature is useful for editing C programs because it encourages statements which

are part of the same compound statement to be indented the same amount, thus making the program more readable.

Z autoindents a line by inserting tab and space characters at the beginning of a new line. If you don't want to be indented that much, you can backspace over these automatically inserted tabs and spaces until you reach the desired degree of indentation.

The autoindent option can be selectively enabled and disabled using the 'set options' command:

```
:se ai=0    to disable autoindent
:se ai=1    to enable autoindent
```

When *Z* is activated, autoindent is enabled.

2.7 Macros

Z allows you to define a sequence of commands, called a 'macro', and then execute the macro one or more times.

When a macro is defined to *Z*, it's placed in a special buffer, called the macro buffer, and then executed once. There are two ways to define a macro to *Z*: immediately and indirectly.

2.7.1 Immediate macro definition

An 'immediate' macro definition is initiated by typing the characters

```
:>
```

Z responds by clearing the status line, displaying these characters on the line, and waiting for you to enter the sequence of commands.

As you enter the commands, *Z* displays them on the status line and enters them immediately into the macro buffer; that's why it's called 'immediate macro definition'.

If you make a mistake while entering commands, you can simply backspace and enter the correct characters.

To terminate the definition, type the carriage return key. *Z* will then execute the sequence of commands in the macro buffer. The contents of this buffer are not altered by executing the macro, so you can reexecute the macro without reentering it, as described below.

2.7.2 Some examples

The following macro advances the cursor one line, and deletes the first word on the new line:

```
+dw
```

contains two commands: +, which advances the cursor, and *dw*, which deletes the word beneath the cursor.

The next macro moves the cursor to the previous line and deletes the last character on the line:

```
-$x
```

It contains three commands: -, which moves the cursor to the previous line; *$*, which moves the cursor to the last character on that line; and *x*, which deletes the character beneath the cursor.

You can also insert text using a macro. You enter insert mode using one of the normal insert commands. The characters which follow the insert command on the macro line, up to a terminating escape character, are then inserted into the text. The escape character causes *Z* to return to command mode and continue executing commands in the macro which follow the insert command.

Remember, the key used as the escape character differs from system to system. See section 1 of this chapter for details.

For example, the following macro advances the cursor to the next line, deletes the second word on the line, inserts the character string "and furthermore", and deletes the last word on the line:

```
+wdwiand furthermore<ESC>$bdw
```

The last macro contains the following commands:

+	Advances the cursor to the next line;
w	Moves the cursor to the second word on the line;
dw	Deletes the word beneath the cursor;
iand furthermore<ESC>	Inserts the text "and furthermore". <ESC> stands for the escape key;
$	Moves the cursor to the last character on the line;
b	Moves the cursor to the beginning of the last word on the line;
dw	Deletes that word.

Z also allows you to search for a string from within a macro. Enter in the macro the 'string search' command (for example, /), followed by the string, followed by the ESC character. For example, the following macro moves the cursor to the word "Ralph" and deletes it:

```
/Ralph<ESC>dw
```

It contains the commands

/Ralph<ESC>	Moves the cursor to "Ralph". <ESC> stands for the escape key;
dw	Deletes "Ralph".

The following macro finds "Ralph" and replaces it with "Sarah":

```
/Ralph<ESC>cwSarah<ESC>
```

It contains the commands:

/Ralph<ESC>	Moves the cursor to "Ralph";
cwSarah<ESC>	Changes "Ralph" to "Sarah".

2.7.3 Indirect macro definition

The other way of defining a macro is to yank a line containing a sequence of commands from the main text buffer into a named buffer and then have *Z* move the contents of the named buffer to the macro buffer.

Commands for indirect macro definition are:

@x	Causes *Z* to move the contents of the 'x' buffer to the macro buffer and then execute it once;
xv	A synonym for '@x'.

Indirect macro definition of macros has several advantages over immediate definition: for one, if a macro defined immediately is incorrect, you have to reenter the entire macro. With an indirectly defined macro, you can edit the macro definition in the main text buffer and then move it back to the macro buffer.

Another advantage is that you can store several macros in the named buffers and easily reexecute a macro, without having to reenter it. With immediate definition, when a new macro is defined, the previously defined macro is lost, and must be reentered to be reexecuted.

One difference between entering macros immediately and via the text buffer and named buffer concerns the method for specifying the end of a search string and for exiting insert mode. With immediate definition, you do this by typing the ESC key directly. For indirect definition, in which the macro is first entered into the main text buffer, typing the ESC key would cause *Z* to exit insert mode, not to enter the ESC key into to text of the macro. In this case, you enter the ESC key by first typing control-V, then ESC. This causes *Z* to enter the ESC character into the text of the macro and remain in insert mode.

2.7.4 Re-executing macros

Once a macro is defined and is in the macro buffer, it can be re-executed by typing one of the commands:

@@
v

Preceding the command with a count will cause the macro to be executed the specified number of times.

2.7.5 Wrapping around during macro execution

While executing a macro, *Z* may reach the beginning or end of the text, and want to continue beyond that point. This is especially true when reexecuting macros. The 'macro wrap' option, *wm*, specifies whether *Z* should terminate the macro execution at that point, or continue at the opposite end of the text.

This option is enabled and disabled using the 'set options' command:

:se wm=0	To disable macro wrapping;
:se wm=1	To enable it.

When *Z* starts, this option is enabled.

2.8 The Ex-like commands

The 'substitute' and 'repeat last substitution' commands are part of a set of commands that are being added to the Z editor and that are similar to commands in the UNIX Ex editor. In this section we will first generally describe the syntax of these commands, then the 'substitute' command, and finally the 'repeat last substitution' command.

The Ex-like commands consist of a leading colon, followed by zero, one, or two addresses identifying the lines to be affected by the command, followed by a single-letter command, followed by command parameters, and terminated by a carriage return. Most commands have a default set of lines that they affect, thus frequently allowing you to enter commands without explicitly specifying a range.

These commands support regular expressions, as defined in the Z documentation, for identifying addresses and strings to be searched for.

2.8.1 Addresses in Ex commands

An address can be one of the following:

* A period, ., addresses the current line; that is, the line on which the cursor is located.
* The character $ addresses the last line in the edit buffer.
* A decimal number *n* addresses the *n*-th line in the edit buffer.
* 'x addreses the line marked with the mark name *x*. Lines are marked with the *m* command.
* A regular expression surrounded by slashes (/) addresses the first line containing a string that matches the regular expression. The search begins with the line following the current line and continues towards the end of the edit buffer. If a line isn't found when the end of the buffer is reached, and if Z's *ws* option is set to 1 (ie, by the *:se ws=1* command) the search continues at the beginning of the buffer, stopping when the current line is reached.
* A regular expression surrounded by question marks (?) also addresses the first line containing a string that matches the regular expression. But in this case, the search begins with the line preceding the current line in the edit buffer and continues towards the beginning of the buffer. If a line isn't found when the beginning of the buffer is reached, and if Z's *ws* option is set to 1 (ie, by the *:se ws=1* command) the search continues at the end of the buffer, stopping when the current line is reached.
* An address followed by a plus or a minus sign, which in turn is followed by a decimal number *n* addresses the *n*-th line

following or preceding the line identified by the address.

When two addresses are entered to define the range of lines affected by a command, the addresses are usually separated by a comma. They can also be separated by a semicolon; in this latter case, the current line is set to the line defined by the first address, and then the line corresponding to the second address is located.

When no value is specified for the first address in an address range, it's assumed to be the current line or the first line in the buffer, depending on whether the second address was preceded with a comma or a semicolon. When no value is specified for the second address in an address range, it's assumed to be the last line in the buffer. Thus, if neither the beginning nor the ending address of a range is specified, the range consists of either all the lines in the buffer or the lines from the current through the last line in the buffer, depending on whether comma or semicolon is used to separate the unspecified addresses.

2.8.2 The 'substitute' command

The 'substitute' command has the following form:

:[range]s/pat/rep/[options]

where square brackets surround a parameter to indicate that the parameter is optional.

Z searches the lines specified by *range* for strings that match the regular expression *pat*, replacing them with the *rep* string. If *range* isn't specified, just the current line is searched. When the command is completed, the cursor is left on the character following the last replaced string.

Normally, Z automatically replaces a string that matches *pat*. Specifying *c* as an *option* causes Z instead to pause when it finds a matching string, ask if you want the string to be replaced, and make the replacement only if you give your permission.

Normally, Z will replace only the first *pat*-matching string on a line. Specifying *g* as an option causes Z instead to replace all matching strings on a line; in this case, after Z replaces a string on a line, it continues searching for more strings on the line at the character following the replaced string.

An ampersand (&) in the replacement string *rep* is replaced by the string that matched *pat*. The special meaning of & can be suppressed by preceding it with a backslash, \.

A replacement string consisting of just the percent character (%) is replaced in the current substitution by the replacement string that was used in the last substitution. The special meaning of % can be suppressed by preceding it with a backslash, \.

2.8.2.1 Examples

:s/abc/def/

Search the line on which the cursor is located for the string *abc*; if found, replace it with the string *def*.

:1,$s/ab*c/xyz/

Search all lines in the edit buffer for strings that begin with *a*, end in *c*, and have zero or more *b*'s in between; replace such strings with *xyz*. On any given line, only the first occurrence of a string that matches the pattern is replaced.

:/{/;/}/s/for/while/c

Find the first line following the current line that contains a {; then find the first line following this line that contains a }. In the lines between and including these lines, search for the string *for*; for each such string, ask if it should be replaced; if yes, replace it with *while*.

2.8.3 The '&' (repeat last substitution) command

The & command has the form

:[range]&

where brackets indicate that the parameters are optional.

The & command causes the last 'substitute' command to be executed again, using the same search pattern, replacement string, and options as were used in the previous command. The command searches the lines that are specified in the & command's *range*; if *range* isn't specified, the substitution is performed on just the current line.

2.9 Starting and stopping Z

You already know how to start and stop *Z*, from the previous chapter. In this section we present more information related to the starting and stopping of *Z*.

2.9.1 Starting Z

In the previous chapter, we said that *Z* was started by specifying the name of the file to be edited on the command line:

```
Z filename
```

Z can also be started without specifying a file name or by specifying a list of files to be edited.

2.9.1.1 Starting Z without a filename

When *Z* is started without a filename being specified, you will normally tell *Z* the name of the file to be edited, once it's active, using the *:e* command:

```
:e filename
```

It isn't absolutely necessary for *Z* to know the name of the file you're editing: *Z* will allow you to create and modify text in the text buffer without knowing the name of the file to which you intend to write the text. But then you'll have to explicitly tell *Z* to write the text, using the command

```
:w filename
```

Z can't automatically write the text, since it doesn't know which file you're editing.

2.9.1.2 Starting Z with a list of files

Z can be started and passed a list of names of files to be edited, as follows:

```
Z file1 file2 ...
```

Z will remember the list, and make the first file in the list the 'edit file'; that is, read the file into the main text buffer and allow it to be edited.

Z has a command, *:n*, which will make the next file in the list the edit file, after writing the contents of the text buffer back to the current edit file.

File lists are discussed in more detail below.

2.9.1.3 The options file

Z has several options for controlling its operation in different situations. You've already met most of them, including the 'autoindent', 'macro wrap', and other options. The complete list of

options will be presented later. In this section, we want to present another feature of *Z* related to options; the ability to set options automatically, when *Z* is started.

When *Z* starts, it will read options from the file named 'z.opt', if it exists. *Z* looks for the file in different places on different systems.

On PCDOS and on the Macintosh, the environment variable ZOPT defines the name of the options file. If this variable doesn't exist, or if the file isn't found there, *Z* then looks for the file *z.opt* on the current directory on the default drive.

Each line in the options file defines the value of one option, with a statement of the form

```
opt=val
```

where 'opt' is the name of the option, and 'val' is its value. For example, the following sets the 'tab width' option to 8 characters:

```
ts=8
```

2.9.1.4 Setting options for a file

When *Z* makes a file the 'edit file' by reading it into the edit buffer, the file itself can specify the options to be in effect during its edit session. This feature is most useful in editing files which have different tab settings.

A file specifies option values by including strings of the form

```
:opt=val
```

in the first ten lines of the file. For example, the following line could be used near the front of a C program, causing a tab width of 8 characters to be used:

```
/*  :ts=8    */
```

When *Z* starts editing a file, the tab width is set back to the default

value, 4 characters, before the file is scanned for option settings.

2.9.2 Stopping Z

In the preceding chapter we presented the following commands for stopping *Z*:

ZZ	If the file's text in the edit buffer has been modified, the text is written to the file, after changing the extension of the original file to ".bak".
:q!	Stops *Z* without writing the text to the file.

Two other commands for exiting *Z* are:

:wq	Which is the name as ZZ, except that the text in the main text buffer is always written to the file, even if no changes have been made;
:q	Which conditionally stops *Z*. If no changes were made to the file's text, *Z* stops; otherwise, it displays a message and remains active.

2.10 Accessing files

Z has other commands for accessing files besides ZZ and :wq, and we're going to discuss them in this chapter.

Z usually knows the name of the file you are editing, and in the sections that follow we will call this the 'edit file'. *Z* makes use of this knowledge, allowing you to write to the edit file without specifying it by name. For example, the ZZ command writes text to the edit file without requiring you to enter the name of the file.

Some commands allow you to access files without redefining *Z*'s idea of the edit file. The commands described in the next two subsections fall into this category.

Other commands cause *Z* to terminate editing of one file and begin editing another; this new file becomes the edit file. The commands described in the other sections of this section are of this type.

2.10.1 File names

In the *Z* commands that require a file name, the name is usually entered using the standard system conventions. However, some characters are special to *Z*:

#	Refers to the last edit file;
%	Refers to the current edit file;
\	Causes the next character to be used in the filename and not be interpreted.

To enter a file name which contains these characters, precede the special character with the character '\'. For example, on PCDOS, to edit the file

```
a:\subs\hello.c
```

use the command

```
:e \\subs\\hello.c
```

On PCDOS, the '/' character can also be used as a separator between directories and between a directory and file name. Thus, the above command could also be entered as:

```
:e /subs/hello.c
```

2.10.2 Writing files

The command *:w* writes the contents of the main text buffer to a file, without redefining the identity of the current edit file. It has the following forms:

:w	Write to the current edit file;
:w	Write to the specified file;
:w!	Same as ':w filename', but the file is overwritten if it exists.

As with all colon commands, carriage return must be typed to cause *Z* to execute the command.

When entered without a filename, *:w* creates a new file having the name of the current edit file and writes the contents of the edit buffer to it. This form of the :w command is commonly used to periodically save text during a long edit session, to guard against system failures.

The option *bk* tells *Z* whether it should save the original edit file before creating a new one. If *bk* is 1 the original will be saved, and if 0 it won't. *Z* saves the original file by changing its name to *.bak*. An existing *.bak* file will be erased before the rename occurs. For details on setting options, see the Options section.

When a filename is entered with the :w command, the text is written to that file, if it doesn't already exist. If it does, nothing is written, and *Z* displays a message on the status line; in this case you must use the *:w!* form of the command to overwrite the file.

The *:w!* command unconditionally writes the text to the specified file, after truncating the file, if it exists, so that nothing is in it. Unlike the *:w* command which doesn't specify a file name, the :w! command doesn't save the original file as a ".bak" file.

2.10.3 Reading files

The command

:r filename

merges one file with a file being edited, without redefining the identity of the edit file.

It reads the contents of the specified file into the main text buffer, inserting the new text following the line on which the cursor is located. It doesn't alter text which is already in the edit buffer.

2.10.4 Editing another file

The following commands cause *Z* to stop editing one file and begin editing another, which thus becomes the 'edit file':

:e	Edit the specified file;
:e!	Edit the file, discarding changes to the current edit file;
:e	Reload the current edit file;
:e!	Reload the current edit file, discarding changes;
:e	Re-edit the previous edit file;
^^	Synonym for ':e #'. (the command is 'control-^').

Z begins editing another file by erasing the contents of the main text buffer and the unnamed buffer, resetting the tab width to four characters, redrawing the display with the first screenful of lines from the file, and setting the cursor at the first character in the text.

When switching to a new edit file, *Z* doesn't change the contents of the named buffers. Thus, these buffers can be used to hold text which is to be moved from one file to another and to contain commonly used macros.

The command

:e filename

causes the specified file to conditionally become the edit file. The condition is that changes must not have been made to the text of the current edit file since it was last written to disk. If this condition is met, then the switch is made; otherwise, *Z* displays a message on the status line and nothing is changed: the identity of the edit file is the same, the contents of the edit buffer are not modified, and the options are not changed.

If *Z* doesn't let you switch edit files when you enter

:e filename

and you want to save the changes to the current edit file, enter the sequence:

:w
:e filename

You can unconditionally cause *Z* to begin editing a new file by entering:

:e! filename

In this case, *Z* doesn't care whether or not you made changes to the current edit file since it was last written to disk; it begins editing the new file without changing the previous edit file.

Sometimes the text in the edit file may get hopelessly scrambled, and you want to get a fresh copy of the edit file contents. The command

:e!

specified without a file name will do just that.

Z not only remembers the name of the current edit file you're editing; it remembers the name of the last file you edited as well. *Z* allows you to refer to this name using the character '#' in :e commands, thus providing a quick means to re-edit the previous edit file:

:e #

causes the previous edit file to conditionally become the current edit file, and

:e! #

causes it to unconditionally become the edit file.

The command ^^ (that is, control-^) is a synonym for ':e #'.

Z also remembers the position at which the cursor was located in the previous edit file, and when you begin re-editing this file it sets the cursor back to this position.

2.10.5 File lists

Z's 'file list' feature is convenient to use when you have several files to edit: you pass *Z* a list of the files and begin editing the first one. When you're finished with one file, a command switches to the next file in the list, after automatically saving the changes to the current edit file. An option to the command prevents *Z* from saving changes, and another command "rewinds" the file list so that you're back editing the first file in the list again.

There's two ways to pass the list of files to be edited to *Z*: as parameters to the command that starts *Z*, and as parameters to the ':n' command. In each case, *Z* remembers the list and makes the first file in the list the 'edit file'. For example,

Z file1 file2 file3

starts *Z* and defines the list of files file1, file2, and file3. *Z* makes file1 the edit file; that is, prepares it for editing by reading it into the edit buffer and displaying its first lines.

When *Z* is active, the command

:n file4 file5 file 6

defines a new list of files: file4 file5 and file6. *Z* makes file4 the edit file.

When used without a files list, the ':n' command switches from one file in the list to the next:

:n	Writes the text in the edit buffer to the current edit file before switching;
:n!	Switches without writing anything to the current edit file.

The ':rew' command "rewinds" the file list; that is, makes the first file in the list the edit file. This command behaves like the ':n' command, in that it by default writes changes to the current edit file before rewinding; and when an exclamation mark is appended to the comand, the rewind occurs without writing to the current edit file.

2.10.6 Tags

Z has a feature useful for editing large C programs which contain many functions distributed over several files. With the aid of a cross-

reference file relating 'tags', that is, function names, to the files containing them, you simply tell *Z* the name of the function that you want to edit and *Z* makes the file containing it the edit file by reading it into the edit buffer and positioning the cursor to the function.

The following commands specify the tag of the function to be edited:

:ta tag	Position to the function named 'tag' in the appropriate file, if the current edit file is up to date;
:ta! tag	Same as ':ta tag', but the switch to the new file occurs even if the current edit file isn't up to date.

When using the ':ta' command, the current edit file is considered 'up to date' if the text in the edit buffer hasn't been modified since it was last written to the file. When used without the trailing '!', the ':ta' command won't switch edit files if the current edit file isn't up to date; it'll just display a message on the status line. You can then either write the text in the edit buffer to the file and re-enter the ':ta' command, or immediately enter the ':ta!' command, to switch edit files anyway.

The command

^]

that is, control-], is convenient when, while editing or viewing one function, you want to edit or examine a function which it calls. You just set the cursor to the name of the called function and enter '^]'; *Z* will make the file containing the called function the edit file, and position the cursor to this function.

For example, while examining the file *crtdvr.c*, you may come across a call to the function *pcdvr*, and want to take a look at it. By positioning the cursor at the beginning of the word 'pcdvr' and typing '^]', *Z* will make the file containing *pcdvr* the edit file and leave the cursor positioned at this function.

2.10.7 The CTAGS utility

The utility program *ctags* creates the cross reference file, *tags*, which relates function names to the file containing them.

ctags is activated by a command of the form

ctags file1 file2 ...

where file1, ..., are names of files whose functions are to be placed in the cross reference file. A file name can specify a group of files using the character '*'. For example:

*.c

specifies all files whose extension is ".c", and

```
f*.c
```

specifies all files whose first character is 'f' and whose extension is ".c".

ctags considers a character string in a file it is scanning to be a function name, for inclusion in the cross reference file if it's a valid C name which begins on the first column of a line and which is terminated by an open parenthesis character. Thus, the function which begins

```
FILE *
fopen(...
```

would be included in the cross reference, but the function which begins

```
FILE * fopen(...
```

wouldn't.

ctags creates the cross reference file, *tags*, in the current directory on the default drive.

When a tags command is given, Z searches for this file in locations which differ from system to system. On PCDOS, it searches for the file in the current directory on the default drive.

2.11 Executing system commands

On PCDOS, *Z* has two commands which allow you to execute system commands while *Z* is active and then return to *Z*:

:!cmd	Executes the system command 'cmd';
:!!	Re-executes the last command.

For example,

```
:!dir *.c
```

executes the system command 'dir *.c' and returns to *Z*.

2.12 Options

Z has has several options under user control which define how *Z* behaves in certain situations. Most of these options have been discussed peripherally in previous sections, when appropriate. In this section we want to focus on the options.

Each option is identified by a code. The options and their codes are:

ai — The 'auto-indent' option. When this option is enabled and you begin inserting text on a new line, *Z* automatically indents the line by inserting tabs and spaces so that the first character you type will be located in the same column as the first non-whitespace character on the previous line. By default, this option is enabled.

eb — The 'error bells' option. When this option is enabled, *Z* will beep when you make a mistake. By default, this option is enabled.

ma — The 'magic' option. When this option is enabled, regular expessions used in string searches can include extended pattern matching characters. Otherwise, only the characters '^' and '$' are special and the extended pattern matching constructs are gotten by preceding them with ". By default, this option is disabled.

ts — The 'tab set' option. Specifies the number of characters between tab settings. By default, the tab width is four characters.

wm — The 'wrap on macro' option. When this option is enabled, and a macro being executed reaches the end of the buffer, the macro will wrap around to the beginning of the buffer and continue. By default, this option is enabled.

ws — The 'wrap on search' option. When this option is enabled, and a search for a string reaches the end of the buffer without finding the string, the search continues at the opposite end of the buffer. By default, this option is enabled.

bk — This option defines whether Z, when a *.w* command is entered to write the edit buffer to the current edit file, should save the original edit file before creating a new one.

An option is enabled by setting it to 1, and disabled by setting it to 0.

2.13 Z vs. Vi

Z is very similar to the UNIX editor VI:

* Both are full-screen editors, display text in the same way, and reserve one line of the display for messages;
* They have the same two modes: command and insert;
* Z supports most of the Vi commands. The Z commands are activated by the same keystrokes and perform the same functions as their Vi counterparts.

Z and Vi differ in the following ways:

* In Z, the buffer in which text is edited is entirely within RAM memory; in Vi, the buffer is both in memory and on disk. Because of this, Z is restricted in the size of program that can be edited, but Vi is not;
* A single copy of Vi can be configured to use any type terminal. A single copy of Z is pre-configured to use just one terminal;
* Vi has an underlying editor, *ex*, whose commands can be executed while Vi is active. Z doesn't have an underlying editor. However, Z does support some *ex* commands directly; these are the commands whose first character is ':'. (Vi interprets the ':' as a request to execute the *ex* command which is entered after the ':');
* Vi has commands and options useful for editing documents and for editing LISP programs, but Z doesn't;
* With Vi, you can create a shell and suspend Vi while executing commands from within the new shell. With some Vis, you can also suspend Vi while executing commands from the shell that activated Vi. Z doesn't support either of these features, although it will allow you to suspend Z while executing a single system command;
* Vi saves the last nine deleted blocks of text, and has commands with which it can recover them, if necessary. Z lets you recover the last deleted block;
* With Vi, operator commands can affect exactly the characters between the starting and ending cursor positions, even when the positions are on different lines. It has variations of these commands which allow whole lines to be affected, between and including the lines containing the two positions.

  In Z, operator commands in which the starting and ending cursor positions are on different lines always affect whole lines, between and including the lines containing the two positions.

2.14 System-dependent features

2.14.1 Macintosh features

On the Macintosh, Z supports only the Monaco font. It allows characters to be displayed using either 9- or 12-point size. When 9-point is used, the screen has 30 lines, each containing 85 characters. when 12-point is used, the screen has 20 lines, each containing 63 characters.

The environment variable *ZSIZE* defines the point size that Z is to use. The command

```
set ZSIZE=9
```

causes Z to use 9-point characters, and

```
set ZSIZE=12
```

causes it to use 12-point.

3. Command Summary

Starting Z

z name	edit file name
z name1 name2	edit file name1, rest via :n

The Display

~ lines	lines past end of file
@ lines	lines that don't fit on screen
^x	control characters
tabs	expand to spaces, cursor on last

Options

ai=1/0	auto-indent on/off
eb=1/0	error bells on/off
ma=0/1	magic off/on
ts=val	tab width (4)
wm=1/0	wrap on search when executing macro
ws=1/0	wrap on search scan
bk=1/0	save original file as *.bak*

Adjusting the Screen

^F	forward screenful
^B	backward screenful
^D	scroll down half screen
^U	scroll up half screen
zCR	redraw, current line at top
z-	redraw, current line at bottom
z.	redraw, current line at center

Positioning within File

g	go to line (default is end of file)
G	go to line (default is end of file)
/pat	move cursor to pat searching forwards
?pat	move cursor to pat searching backwards
n	repeat last / or ?
N	repeat last / or ? in reverse direction
]]	next "^{"
[[	previous "^{"
%	find matching (), {}, or [].

Marking and Returning

`` ` ``	previous context
'	first non-white at previous context
mx	mark position with letter 'x'
`` `x ``	to mark 'x'
'x	first non-white at mark 'x'

Line Positioning

H	top of screen
M	middle of screen
L	bottom of screen
+	next line, first non-white
CR	next line, first non-white
-	previous line, first non-white
LF	next line, same column
j	next line, same column
^K	previous line, same column
k	previous line, same column

Character Positioning

0	beginning of line
^	first non-white at beginning of line
$	end of line
space	forward a character
^L	forward a character
l	forward a character
^H	backwards a character
h	backwards a character
fx	find character 'x' forward
Fx	find character 'x' backwards
tx	position before character 'x' forward
Tx	position before character 'x' backwards
;	repeat last f, F, t or T
,	repeat last f, F, t or T in reverse direction
|	move to specified column number

Words and Paragraphs

w	word forward
W	blank delimited word forward
b	back word
B	back blank delimited word
e	end of word
E	end of blank delimited word
}	to next blank line
{	to previous blank line

Insert and Replace

a	append after cursor
A	append at end of line
i	insert before cursor
I	insert before first non-blank in line
o	open line below current line
O	open line above current line
rx	replace single character with 'x'
R	replace characters

Corrections During Insert

^H	erase last character
^D	erase last character
^X	erase to beginning of insert on current line
^V	insert following character directly

Operators

d	delete
c	delete and insert
<	left shift
>	right shift
y	yank

Miscellaneous Operations

D	delete rest of line
C	change rest of line
s	substitute characters
S	substitute lines
J	join lines
x	delete characters starting at cursor
X	delete characters before cursor
Y	yank lines

Yank and Put

p	put after current
P	put before current
"xp	put from buffer 'x'
"xy	yank to buffer 'x'
"xd	delete to buffer 'x'

Undo and Redo

u	undo last change
U	restore current line
.	repeat last change command

Macros

@x	execute macro in buffer 'x'
"xv	execute macro in buffer 'x'
@@	repeat last macro
v	repeat last macro

Colon Commands

:e name	edit file name
:e	reedit last file
:e! name	edit file name, discarding changes
:e!	reedit last file, discarding changes
:e #	edit alternate file
^^	edit alternate file
:e! #	edit alternate file, discarding changes
:r name	read file "name" into current file
:w	write back to file being edited
:wq	write back to file and quit
:w name	write to file "name" if does not exist
:w! name	write to file "name", delete if exists
:q	quit
:q!	quit, discarding changes
:x	quit, saving file if modified
ZZ	quit, saving file if modified
:f	show current file and line
^G	show current file and line
:n	edit next file in list
:n!	edit next file in list, discarding changes
:n arg1 arg2	specify new list
:rew	point back to beginning of list
:rew!	point back to beginning, discarding changes
:ta tag	position to tag in appropriate file
^]	same as :ta using word at cursor
:ta! tag	position to tag, discarding changes
:!cmd	execute cmd, then return (PCDOS only)
:!!	re-execute last cmd (PCDOS only)
:>macro	specify and execute immediate macro
:set opt1=val opt2=val ...	set editor options
:se opt1=val opt2=val ...	set editor options
:set all	display current option settings
:[range]s/pat/rep/[options]	substitute *rep* for *pat* in *range*
:[range]&	repeat last substitute command

New Options for Z - Text Editor

These pages describe new features in the Z program editor and should be placed at the end of the "Z - Text Editor" section in your manual.

Z now accepts options as part of the command line.

-tname Invokes Z and automatically searches the *tags* file for the specified name. If found, the file is opened and the cursor placed on the appropriate line.

-n# Invokes Z on the first file specified and places the cursor on the requested line number.

New Features

* The tags command, *:ta*, searches the file pointed to by the environment variable TAGS if the *tags* file does not exist or the tag is not found in it.

* A new command, *:fn*, is added. This command takes a string as an argument and searches the file *funclist* for the specified string. If the file does not exist, or the string is not found, a check is made to see if the environment variable FUNCLIST exists. If so, the file indicated by the environment variable is searched as well.

  If the string is found in one of the files, the entire line (up to the width of the screen) is displayed that contained the string. This is most useful for displaying the calling sequence of a particular function. For example, if FUNCLIST=xxx/manx.c, where xxx is the path to the file, then typing:

  :fn AllocMem

  would display in the command line area a line like:

  void *AllocMem(size,requirements) long size,requirements;{}

  This command may also be invoked by placing the cursor on the name to be searched for and typing the ^_ character.

* A new flag setting is available, *sm*, which is used to indicate whether or not macros should perform their operation silently. If non-zero, a macro will perform all iterations and redisplay the screen when finished.

* The new flag, *ak*, allows the user to move the cursor via the keyboard arrow keys when *ak* is set to a nonzero value. When *ak* is not set, the user may use the arrow keys as shortcuts for the following:

up arrow - performs :e # to edit the alternate file
down arrow - performs :fn, using the word the cursor is positioned on
right arrow - performs ^] to search the tag file for the word positioned on

* During insert mode, if a ^W is typed, the previous word typed is deleted.
* When activated from the shell, the number of files that can be specified is limited to 30 instead of 10.

UTILITY PROGRAMS

Chapter Contents

Utility Programs

This chapter describes commands whose code is built into the SHELL and utility programs that are provided with this package.

NAME

arcv & mkarcv - source dearchiver & archiver

SYNOPSIS

arcv arcfile

mkarcv arcfile

DESCRIPTION

arcv extracts the source from the archive *arcfile*, which has been previously created by *mkarcv*, placing the results in separate files in the current directory.

mkarcv creates the archive file *arcfile*, placing in it the files whose names it reads from its standard input. Only one file name is read from a standard input line.

EXAMPLES

For example, the file *header.arc* contains the source for all the header files. To create these header files, enter:

```
arcv header.arc
```

The files will be created in the current directory.

The following command creates the archive *myarc.arc* containing the files *in.c*, *out.c*, and *hello.c*:

```
mkarcv myarc.arc <myarc.bld
```

The names of the three files are contained in the file *myarc.bld*:

```
in.c
out.c
hello.c
```

NAME

cat - catenate and print

SYNOPSIS

cat [file] [file] ...

DESCRIPTION

cat reads each *file* in sequence and writes it to its standard output device. If no files are specified, *cat* reads from its standard input device.

cat will only read a file's data fork. Hence, it can be used to copy files such as those containing C source and object code, but not files containing a resource fork, such as the linker- created files which contain executable code and standard Macintosh application files.

Each argument can specify a complete or partial file name, in the normal manner.

By default, *cat*'s standard input and output devices are assigned to the console. Either or both can also be redirected to another device or file, if desired, in the normal fashion.

The code for *cat* is built into the SHELL.

EXAMPLES

cat hello.c

Writes *hello.c* to the screen.

cat data:/hello.c input.c >.bout

Writes *data:/hello.c* and *input.c*, in that order, to the *.bout* device.

cat

Copies typed characters to the screen.

cat >../newfile

Copies typed characters to *../newfile.*

cat <sys:/stdio/printf.c >tmp.c

Equivalent to *cat sys:/stdio/printf.c >tmp.c.*

SEE ALSO

cp

NAME

cd - change current directory

SYNOPSIS

cd directory

DESCRIPTION

cd makes the specified directory the current directory. If the directory doesn't exist, it is created.

If *cd* creates a directory, and then another *cd* command is issued before any files are created in the new current directory, the new directory will disappear.

The *directory* argument has the format:

[vol:][path]

where:

vol: Defines the volume containing the new current directory. *vol* can be the name of the volume. If the volume is in a drive, it can also be the number of the drive. If not specified, it's assumed to be the volume containing the directory which was current before the *cd* command was issued.

path Defines the path of directories which must be passed through to reach the new current directory. The path can define a complete path from the root directory on the specified volume, or it can define a partial path, which is assumed to begin at the current directory. If the path isn't defined, the root directory on the specified volume is made the current directory; in this case, the volume component must be specified.

The code for *cd* is contained in the SHELL.

EXAMPLES

cd data:/work

The */work* directory on the volume *data:* is made the current directory.

cd /subs/io

The directory */subs/io*, on the volume which contained the current directory before the issuance of this command, is made the current directory.

cd subs/io

The directory *io*, which is reached from the current directory by passing through the subdirectory *subs* of the current directory and then into *io*, is made the current directory. For example, if *data:/work* was the current directory, then after this command, *data:/work/subs/io* is the new current directory.

cd ..

The current directory is set to the parent directory of the directory which was current before the issuance of this command.

cd ../include

The current directory is set to the directory which is reached by passing through the parent directory of the directory which was current before the issuance of this command and then to its *include* subdirectory.

cd ../..

The current directory is set to the directory which is reached by passing through the parent directory of the directory which was current before the issuance of this command and then to its parent directory.

NAME

cmp - File comparison utility

SYNOPSIS

cmp [-l] [-r] file1 file2

DESCRIPTION

cmp compares two files on a character-by-character basis. When it finds a difference, it displays a message, giving the offset from the beginning of the file.

The files' resource forks will be compared if the *-r* option is specified; otherwise, their data forks are compared.

If the *-l* option isn't specified, the program will stop after the first difference, displaying a message in the format:

Files differ: character 10

If the *-l* option is specified, *cmp* will list all differences, in the format:

decimal-offset hex-offset file1-valuefile2-value

EXAMPLES

cmp otst ntst

Files differ: character 10

and

cmp -l otst ntst

```
 10   a:  00 45
100   64:  1a 23
```

NAME

cnm - display object file info

SYNOPSIS

cnm [-s] file [file ...]

DESCRIPTION

cnm displays the size and symbols of its object file arguments. The files can be object modules created by the Manx assembler, libraries of object modules created by the Manx linker, and 'rsm' files created by the Manx linker during the linking of an overlay root.

The *-s* option tells *cnm* to just display the size information for the files.

For example, the following displays the size and symbols for the object module *sub1.o*, the library *c.lib*, and the rsm file *root.rsm*:

```
cnm sub1.o c.lib root.rsm
```

By default, the information is sent to the display. It can be redirected to a file or device in the normal way. For example, the following commands send information about *sub1.o* to the display, and the file *dispfile*, respectively:

```
cnm sub1.o
cnm sub1.o > dispfile
cnm >.bout sub1.o
```

A filename can optionally specify multiple files, using the "wildcard" characters *?* and ***. These have their standard meanings: *?* matches a single character; * matches zero or more characters. For example

```
*.o       Specifies all files with extent '.o'
a??.lib   Specifies all files whose filename has three characters
          the first of which is 'a', and whose extent is '.lib'
```

cnm displays information about an program's 'named' symbols; that is, about the symbols whose first two characters are other than a dollar sign followed by a digit. For example, the symbol *quad* is named, information about it would be displayed; the symbol *$0123* is unnamed, so information about it would not be displayed.

For each named symbol in a program, *cnm* displays its name, a code specifying its type, and an associated value. The value displayed depends on the type of the symbol.

A type code is a single character, and can be either upper or lower case, specifying that the symbol is global or local to the program respectively. The types codes are:

bus error
address error
illegal instruction
divide by zero
line 1111 (unimplemented op code)

By default, the SHELL won't trap these errors; in this case, one of these errors will cause the Macintosh to bomb, and the system will have to be rebooted.

When an error is trapped, the SHELL displays on the screen the contents of the registers and the error type.

NAME

shift - shift exec file variables

SYNOPSIS

shift [n]

DESCRIPTION

shift causes the values assigned to an exec file variable to be reassigned to the next lower-numbered exec file variable. *n* is the number of the lowest-numbered variable whose value is to be reassigned, and defaults to 1.

Thus,

```
shift
```

causes the exec file variable $1 to be assigned the value of $2, $2 to be assigned the value of $3, and so on. The original value assigned to $1 is lost. When all arguments to the exec file have been shifted out, $1 is assigned the null string.

EXAMPLES

The following exec file, *del*, is passed a directory as its first argument and the names of files within the directory that are to be removed:

```
set j = $1
shift
loop i in $*
rm $j/$i
eloop
```

In this example, *j* is an environment variable. The first two statements in the exec file save the name of the directory and then shift the directory name out of the exec file variables.

The loop then repeatedly calls *rm* to remove one of the specified files from the directory.

Entering

```
del sys:/work *.bak
```

will remove all files having extension *.bak* from the directory *sys:/work*.

NAME

styp - set file type

SYNOPSIS

styp type creator file1 file2 ...

DESCRIPTION

styp sets the type and the creator fields of the specified files.

NAME

term - terminal emulation program

SYNOPSIS

term

DESCRIPTION

term is a terminal emulation program that allows source files to be transferred from another computer to the Macintosh.

To transfer a file, select "FILE XFER" from the FILE menu displayed by the *term* program. *term* will then prompt for the name of a file. Type the name of the file, followed by carriage return.

Now tell the host computer to start sending the source file: if the Macintosh is acting as the console of the host computer, enter, on the Macintosh keyboard, the command necessary to start displaying on the screen the file to be transferred. Otherwise, enter, on the host computer's console, whatever command is necessary to start the other computer sending the text down the serial communications line.

The *term* program will display a '.' for each line that is transferred. When the file is transferred (and the dots stop being printed), select "FILE XFER" from the FILE menu again to close the file and return to normal terminal mode.

NAME

lb - object file librarian

SYNOPSIS

lb library [options] [mod1 mod2 ...]

DESCRIPTION

lb is a program that creates and manipulates libraries of object modules. The modules must be created by the Manx assembler.

This description of *lb* is divided into three sections: the first describes briefly *lb*'s arguments and options, the second *lb*'s basic features, and the third the rest of *lb*'s features.

1. The arguments to *lb*

1.1 The *library* argument

When started, *lb* acts upon a single library file. The first argument to *lb* (*library*, in the synopsis) is the name of this file. The filename extension for *library* is optional; if not specified, it's assumed to be *.lib*.

1.2 The *options* argument

There are two types of *options* argument: function code options, and qualifier options. These options will be summarized in the following paragraphs, and then described in detail below.

1.2.1 Function code options When *lb* is started, it performs one function on the specified library, as defined by the *options* argument. The functions that *lb* can perform, and their corresponding option codes, are:

function	*code*
create a library	(no code)
add modules to a library	-a, -i, -b
list library modules	-t
move modules within a library	-m
replace modules	-r
delete modules	-d
extract modules	-x
ensure module uniqueness	-u
define module extension	-e
help	-h

In the synopsis, the *options* argument is surrounded by square brackets. This indicates that the argument is optional; if a code isn't specified, *lb* assumes that a library is to be created.

1.2.2 Qualifier options In addition to a function code, the *options* argument can optionally specify a qualifier, that modifies *lb*'s behavior as it is performing the requested function. The qualifiers and their codes are:

verbose	-v
silent	-s

The qualifier can be included in the same argument as the function code, or as a separate argument. For example, to cause *lb* to append modules to a library, and be silent when doing it, any of the following option arguments could be specified:

-as
-sa
-a -s
-s -a

1.3 The *mod* arguments

The arguments *mod1*, *mod2*, etc. are the names of the object modules, or the files containing these modules, that *lb* is to use. For some functions, *lb* requires an object module name, and for others it requires the name of a file containing an object module. In the latter case, the file's extension is optional; if not specified, the *lb* that's supplied with native Aztec C systems assumes that it is *.o*, and the *lb*

that's supplied with cross development versions of Aztec C assumes that the extension is *.r*. You can explicitly define the default module extension using the *-e* option.

1.4 Reading arguments from another file

lb has a special argument, *-f filename*, that causes it to read command line arguments from the specified file. When done, it continues reading arguments from the command line. Arguments can be read from more than one file, but the file specified in a *-f filename* argument can't itself contain a *-f filename* argument.

2. Basic features of *lb*

In this section we want to describe the basic features of *lb*. With this knowledge in hand, you can start using *lb*, and then read about the rest of the features of *lb* at your leisure.

The basic things you need to know about *lb*, and which thus are described in this section, are:

* How to create a library
* How to list the names of modules in a library
* How modules get their names
* Order of modules in a library
* Getting *lb* arguments from a file

Thus, with the information presented in this section you can create libraries and get a list of the modules in libraries. The third section of this description shows you how to modify selected modules within a library.

2.1 Creating a Library

A library is created by starting *lb* with a command line that specifies the name of the library file to be created and the names of the files whose object modules are to be copied into the library. It doesn't contain a function code, and it's this absence of a function code that tells *lb* that it is to create a library.

For example, the following command creates the library *exmpl.lib*, copying into it the object modules that are in the files *obj1.o* and *obj2.o*:

```
lb exmpl.lib obj1.o obj2.o
```

Making use of *lb*'s assumptions about file names for which no extension is specified, the following command is equivalent to the above command:

```
lb exmpl obj1 obj2
```

An object module file from which modules are read into a new library can itself be a library created by *lb*. In this case, all the modules in the input library are copied into the new library.

2.1.1 The temporary library When *lb* creates a library or modifies an existing library, it first creates a new library with a temporary name. If the function was successfully performed, *lb* erases the file having the same name as the specified library, and then renames the new library, giving it the name of the specified library. Thus, *lb* makes sure it can create a library before erasing an existing one.

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

2.2 Getting the table of contents for a library

To list the names of the modules in a library, use *lb*'s *-t* option. For example, the following command lists the modules that are in *exmpl.lib*:

```
lb exmpl -t
```

The list will include some ***DIR*** entries. These identify blocks within the library that contain control information. They are created and deleted automatically as needed, and cannot be changed by you.

2.3 How modules get their names

When a module is copied into a library from a file containing a single object module (that is, from an object module generated by the Manx assembler), the name of the module within the library is derived from the name of the input file by deleting the input file's volume, path, and extension components.

For example, in the example given above, the names of the object modules in *exmpl.lib* are *obj1* and *obj2*.

An input file can itself be a library. In this case, a module's name in the new library is the same as its name in the input library.

2.4 Order in a library

The order of modules in a library is important, since the linker makes only a single pass through a library when it is searching for modules. For a discussion of this, see the tutorial section of the Linker chapter.

When *lb* creates a library, it places modules in the library in the order in which it reads them. Thus, in the example given above, the modules will be in the library in the following order:

```
obj1 obj2
```

As another example, suppose that the library *oldlib.lib* contains the following modules, in the order specified:

```
sub1   sub2   sub3
```

If the library *newlib.lib* is created with the command

```
lb newlib mod1 oldlib.lib mod2 mod3
```

the contents of the newly-created *newlib.lib* will be:

```
mod1   sub1   sub2   sub3   mod2   mod3
```

The *ord* utility program can be used to create a library whose modules are optimally sorted. For information, see its description later in this chapter.

2.5 Getting *lb* arguments from a file

For libraries containing many modules, it is frequently inconvenient, if not impossible, to enter all the arguments to *lb* on a single command line. In this case, *lb*'s *-f filename* feature can be of use: when *lb* finds this option, it opens the specified file and starts reading command arguments from it. After finishing the file, it continues to scan the command line.

For example, suppose the file *build* contains the line

```
exmpl obj1 obj2
```

Then entering the command

```
lb -f build
```

causes *lb* to get its arguments from the file *build*, which causes *lb* to create the library *exmpl.lib* containing *obj1* and *obj2*.

a The symbol was defined using the assembler's EQUATE directive. The value listed is the equated value of its symbol.

The compiler doesn't generate symbols of this type.

t The symbol is in the code segment. The value is the offset of the symbol within the code segment.

The compiler generates this type symbol for function names; static functions are local to the function, and so have type *t*; all other functions are global, that is, callable from other programs, and hence have type *T*.

d The symbol is in the data segment. The value is the offset of the symbol from the start of the data segment.

The compiler generates symbols of this type for initialized variables which are declared outside any function. Static variables are local to the program and so have type *d*; all other variables are global, that is, accessable from other programs, and hence have type *D*.

C The symbol is the name of a common block. The value is the size of the common block, in bytes. *C* is in upper case because common block names are always global.

The compiler doesn't generate this type symbol.

r The symbol is defined within a common block. The value is the offset of the symbol from the beginning of the common block.

The compiler doesn't generate this type symbol.

u The symbol is used but not defined within the program. The value has no meaning.

The compiler generates *U* symbols for functions that are called but not defined within the program, for variables that are declared to be *extern* and which are actually used within the program, and for uninitialized, global dimensionless arrays. Variables which are declared to be *extern* but which are not used within the program do not make it to the object file.

The compiler generates *u* symbols for variables which are used but not defined within the program.

Arguments in a -f file can be separated by any sequence of whitespace characters ('whitespace' being blanks, tabs, and newlines). Thus, arguments in a -f file can be on separate lines, if desired.

The *lb* command line can contain multiple *-f* arguments, allowing *lb* arguments to be read from several files. For example, if some of the object modules that are to be placed in *exmpl.lib* are defined in *arith.inc*, *input.inc*, and *output.inc*, then the following command could be used to create *exmpl.lib*:

```
lb exmpl -f arith.inc -f input.inc -f output.inc
```

A -f file can contain any valid *lb* argument, except for another -f. That is, -f files can't be nested.

3. Advanced *lb* features

In this section we describe the rest of the functions that *lb* can perform. These primarily involve manipulating selected modules within a library.

3.1 Adding modules to a library

lb allows you to add modules to an existing library. The modules can be added before or after a specified module in the library or can be added to the beginning or end of the library.

The options that select *lb*'s add function are:

option	*function*
-b target	add modules before the module *target*
-i target	same as *-b target*
-a target	add modules after the module *target*
-b+	add modules to the beginning of the library
-i+	same as *-b+*
-a+	add modules to the end of the library

In an *lb* command that selects the *add* function, the names of the files containing modules to be added follows the add option code (and the target module name, when appropriate). A file can contain a single module or a library of modules.

Modules are added in the order that they are specified. If a library is to be added, its modules are added in the order they occur in the input library.

3.1.1 Adding modules before an existing module As an example of the addition of modules before a selected module, suppose that the library *exmpl.lib* contains the modules

```
obj1    obj2    obj3
```

The command

```
lb exmpl -i obj2 mod1 mod2
```

adds the modules in the files *mod1.o* and *mod2.o* to *exmpl.lib*, placing them before the module *obj2*. The resultant *exmpl.lib* looking like this:

```
obj1    mod1    mod2    obj2    obj3
```

Note that in the *lb* command we didn't need to specify the extension of either the file containing the library to which modules were to be added or the extension of the files containing the modules to be added. *lb* assumed that the extension of the file containing the target library was *.lib*, and that the extension of the other files was *.o*.

As an example of the addition of one library to another, suppose that the library *mylib.lib* contains the modules

```
mod1    mod2    mod3
```

and that the library *exmpl.lib* contains

```
obj1    obj2    obj3
```

Then the command

```
lb -b obj2 mylib.lib
```

adds the modules in *mylib.lib* to *exmpl.lib*, resulting in *exmpl.lib* containing

```
obj1    mod1    mod2    mod3    obj2    obj3
```

Note that in this example, we had to specify the extension of the input file *mylib.lib*. If we hadn't included it, *lb* would have assumed that the file was named *mylib.o*.

3.1.2 Adding modules after an existing module As an example of adding modules after a specified module, the command

```
lb exmpl -a obj1 mod1 mod2
```

will insert *mod1* and *mod2* after *obj1* in the library *exmpl.lib*. If *exmpl.lib* originally contained

```
obj1    obj2    obj3
```

then after the addition, it contains

```
obj1    mod1    mod2    obj2    obj3
```

3.1.3 Adding modules at the beginning or end of a library The options *-b+* and *-a+* tell *lb* to add the modules whose names follow the option to the beginning or end of a library, respectively. Unlike the *-i* and *-a* options, these options aren't followed by the name of an existing module in the library.

For example, given the library *exmpl.lib* containing

```
obj1    obj2
```

the following command will add the modules *mod1* and *mod2* to the beginning of *exmpl.lib*:

```
lb exmpl -i+ mod1 mod2
```

resulting in *exmpl.lib* containing

```
mod1 mod2 obj1 obj2
```

The following command will add the same modules to the end of the library:

```
lb exmpl -a+ mod1 mod2
```

resulting in *exmpl.lib* containing

```
obj1    obj2    mod1    mod2
```

3.2 Moving modules within a library

Modules which already exist in a library can be easily moved about, using the *move* option, *-m*.

As with the options for adding modules to an existing library, there are several forms of *move* functions:

option	*meaning*
-mb target	move modules before the module *target*
-ma target	move modules after the module *target*
-mb+	move modules to the beginning of the library
-ma+	move modules to the end of the library

In the *lb* command, the names of the modules to be moved follows the 'move' option code.

The modules are moved in the order in which they are found in the original library, not in the order in which they are listed in the *lb* command.

3.2.1 Moving modules before an existing module As an example of the movement of modules to a position before an existing module in a library, suppose that the library *exmpl.lib* contains

obj1 obj2 obj3 obj4 obj5 obj6

The following command moves *obj3* before *obj2*:

lb exmpl -mb obj2 obj3

putting the modules in the order:

obj1 obj3 obj2 obj4 obj5 obj6

And, given the library in the original order again, the following command moves *obj6*, *obj2*, and *obj1* before *obj3*:

lb exmpl -mb obj3 obj6 obj2 obj1

putting the library in the order:

obj1 obj2 obj6 obj3 obj4 obj5

As an example of the movement of modules to a position after an existing module, suppose that the library *exmpl.lib* is back in its original order. Then the command

lb exmpl -ma obj4 obj3 obj2

moves *obj3* and *obj2* after *obj4*, resulting in the library

obj1 obj4 obj2 obj3 obj5 obj6

3.2.2 Moving modules to the beginning or end of a library The options for moving modules to the beginning or end of a library are *-mb+* and *-ma+*, respectively.

For example, given the library *exmpl.lib* with contents

obj1 obj2 obj3 obj4 obj5 obj6

the following command will move *obj3* and *obj5* to the beginning of the library:

lb exmpl -mb+ obj5 obj3

resulting in *exmpl.lib* having the order

```
obj3    obj5    obj1    obj2    obj4    obj6
```

And the following command will move *obj2* to the end of the library:

```
lb exmpl -ma+ obj2
```

3.3 Deleting Modules

Modules can be deleted from a library using *lb*'s *-d* option. The command for deletion has the form

```
lb libname -d mod1 mod2 ...
```

where *mod1*, *mod2*, ... are the names of the modules to be deleted.

For example, suppose that *exmpl.lib* contains

```
obj1    obj2    obj3    obj4    obj5    obj6
```

The following command deletes *obj3* and *obj5* from this library:

```
lb exmpl -d obj3 obj5
```

3.4 Replacing Modules

The *lb* option 'replace' is used to replace one module in a library with one or more other modules.

The 'replace' option has the form *-r target*, where *target* is the name of the module being replaced. In a command that uses the 'replace' option, the names of the files whose modules are to replace the target module follow the 'replace' option and its associated target module. Such a file can contain a single module or a library of modules.

Thus, an *lb* command to replace a module has the form:

```
lb library -r target mod1 mod2 ...
```

For example, suppose that the library *exmpl.lib* looks like this:

```
obj1 obj2 obj3 obj4
```

Then to replace obj3 with the modules in the files *mod1.o* and *mod2.o*, the following command could be used:

```
lb exmpl -r obj3 mod1 mod2
```

resulting in *exmpl.lib* containing

```
obj1    obj2    mod1    mod2    obj4
```

3.5 Uniqueness

lb allows libraries to be created containing duplicate modules, where one module is a duplicate of another if it has the same name.

The option *-u* causes *lb* to delete duplicate modules in a library, resulting in a library in which each module name is unique. In particular, the *-u* option causes *lb* to scan through a library, looking at module names. Any modules found that are duplicates of previous modules are deleted.

For example, suppose that the library *exmpl.lib* contains the following:

```
obj1    obj2    obj3    obj1    obj3
```

The command

```
lb exmpl -u
```

will delete the second copies of the modules *obj1* and *obj2*, leaving the library looking like this:

```
obj1    obj2    obj3
```

3.6 Extracting modules from a Library

The *lb* option *-x* extracts modules from a library and puts them in separate files, without modifying the library.

The names of the modules to be extracted follows the *-x* option. If no modules are specified, all modules in the library are extracted.

When a module is extracted, it's written to a new file; the file has same name as the module and extension *.o*.

For example, given the library *exmpl.lib* containing the modules

```
obj1 obj2 obj3
```

The command

```
lb exmpl -x
```

extracts all modules from the library, writing *obj1* to *obj1.o*, *obj2* to *obj2.o*, and *obj3* to *obj3.o*.

And the command

```
lb exmpl -x obj2
```

extracts just *obj2* from the library.

3.7 The 'verbose' option

The 'verbose' option, *-v*, causes *lb* to be verbose; that is, to tell you what it's doing.

This option can be specified as part of another option, or all by itself. For example, the following command creates a library in a chatty manner:

```
lb exmpl -v mod1 mod2 mod3
```

And the following equivalent commands cause *lb* to remove some modules and to be verbose:

```
lb exmpl -dv mod1 mod2
lb exmpl -d -v mod1 mod2
```

3.8 The 'silence' option

The 'silence' option, *-s*, tells *lb* not to display its signon message.

This option is especially useful when redirecting the output of a list command to a disk file, as described below.

3.9 Rebuilding a library

The following commands provide a convenient way to rebuild a library:

```
lb exmpl -st > tfil
lb exmpl -f tfil
```

The first command writes the names of the modules in *exmpl.lib* to the file *tfil*. The second command then rebuilds the library, using as arguments the listing generated by the first command.

The *-s* option to the first command prevents *lb* from sending information to *tfil* that would foul up the second command. The names sent to *tfil* include entries for the directory blocks, ***DIR***, but these are ignored by *lb*.

3.10 Defining the default module extension.

Specification of the extension of an object module file is optional; the *lb* that comes with native development versions of Aztec C assumes that the extension is *.o*, and the *lb* that comes with cross development versions of Aztec C assumes that it's *.r*. You can explicitly define the default extension using the *-e* option. This option has the form

-e .ext

For example, the following command creates a library; the extension of the input object module files is *.i*.

lb my.lib -e .i mod1 mod2 mod3

3.11 Help

The *-h* option is provided for brief lapses of memory, and will generate a summary of *lb* functions and options.

NAME

obd - list object code

SYNOPSIS

obd <objfile>

DESCRIPTION

obd lists the loader items in an object file. It has a single parameter, which is the name of the object file.

b The symbol is in the uninitalized data segment. The value is the space reserved for the symbol.

The compiler generates *b* symbols for static, uninitialized variables which are declared outside all functions and which aren't dimensionless arrays.

The assembler generates *b* symbols for symbols defined using the *bss* assembler directive. If the symbol also appears in the *public* directive, it's type is *B* instead of *b*.

G The symbol is in the uninitialized data segment. The value is the space reserved for the symbol.

The compiler generates *G* symbols for non-static, uninitialized variables which are declared outside all functions and which aren't dimensionless arrays.

The assembler generates *G* symbols for variables declared using the *global* directive which have a non-zero size.

NAME

prof - execution profiler report program

SYNOPSIS

prof -S symfile [-m monfile] [-[ant]] [-[xo]] [-[zh]]
prof -R progfile [-m monfile] [-[ant]] [-[xo]] [-[zh]]

DESCRIPTION

prof processes a monitor file produced by the *monitor* function, and produces a report on the execution of the monitored program. For each function in the range specified in *monitor*, *prof* counts the number of ticks encountered in that function and determines the percentage of program run time spent in the function.

Options available for *prof*

-S the -s argument is the name of the symbol table file for the program generated by the linker -T option. This argument or -*R* must be present.

-R the -R argument is the name of the program containing the SYMS resource generated by the linker -W option.

-M The -M option allows the user to specify the *monitor* output file to be processed. If this option is not present, *prof* assumes the file is named mon.out (the name always used by *monitor*) and is on the current directory.

The -T, -A, and -N options determine the sorting of lines in the report.

-T Sort by percentage of time spent in function, greatest to least (This option is the default).

-A Sort by address of function.

-N Sort alphabetically by function name.

The -O and -X options cause *prof* to display the addresses of the functions in the report along with their names.

-O Specifies function addresses in octal.

-X Specifies function addresses in hexadecimal.

-Z The -Z option causes all symbols in the range specified in the call to monitor to be displayed, regardless of whether any ticks were encountered in these functions. The default is to supress listing any unencountered functions.

-H The -H option causes *prof* to suppress printing its normal header in the report. This is useful if the information is to undergo further processing.

The profiler currently works with code only in segment 1 and does not report on tick counts occuring in ROM.

SEE ALSO

monitor (C)

NAME

cnm - display object file info

SYNOPSIS

cnm [-sol] file [file ...]

DESCRIPTION

cnm displays the size and symbols of its object file arguments. The files can be object modules created by the Manx assembler, libraries of object modules created by the *lb* librarian, and, when applicable, 'rsm' files created by the Manx linker during the linking of an overlay root.

For example, the following displays the size and symbols for the object module *sub1.o* and the library *c.lib*:

```
cnm sub1.o c.lib
```

By default, the information is sent to the console. It can be redirected to a file or device in the normal way. For example, the following commands send information about *sub1.o* to the display and to the file *dispfile*:

```
cnm sub1.o
cnm sub1.o > dispfile
```

The first line listed by *cnm* for an object module has the following format:

```
file (module): code: cc   data: dd   udata: uu   total: tt (0xhh)
```

where

* *file* is the name of the file containing the module,
* *module* is the name of the module; if the module is unnamed, this field and its surrounding parentheses aren't printed;
* *cc* is the number of bytes in the module's code segment, in decimal;
* *dd* is the number of bytes in the module's initialized data segment, in decimal;
* *uu* is the number of bytes in the module's uninitialized data segment, in decimal;
* *tt* is the total number of bytes in the module's three segments, in decimal;

* *hh* is the total number of bytes in the module's three segments, in hexadecimal.

If *cnm* displays information about more than one module, it displays four totals just before it finishes, listing the sum of the sizes of the modules' code segments, initialized data segments, and uninitialized data segments, and the sum of the sizes of all segments of all modules. Each sum is in decimal; the total of all segments is also given in hexadecimal.

The *-S* option tells *cnm* to display just the sizes of the object modules. If this option isn't specified, *cnm* also displays information about each named symbol in the object modules.

When *cnm* displays information about the modules' named symbols, the *-L* option tells *cnm* to display each symbol's information on a separate line and to display all of the characters in a symbol's name; if this option isn't used, *cnm* displays the information about several symbols on a line and only displays the first eight characters of a symbol's name.

The *-O* option tells *cnm* to prefix each line generated for an object module with the name of the file containing the module and the module name in parentheses (if the module is named). If this option isn't specified, this information is listed just once for each module: prefixed to the first line generated for the module.

The *-O* option is useful when using *cnm* in combination with *grep*. For example, the following commands will display all information about the module *perror* in the library *c.lib*:

```
cnm -o c.lib >tmp
grep perror tmp
```

cnm displays information about an module's 'named' symbols; that is, about the symbols that begin with something other than a period followed by a digit. For example, the symbol *quad* is named, so information about it would be displayed; the symbol *.0123* is unnamed, so information about it would not be displayed.

For each named symbol in a module, *cnm* displays its name, a two-character code specifying its type, and an associated value. The value displayed depends on the type of the symbol.

If the first character of a symbol's type code is lower case, the symbol can only be accessed by the module; that is, it's local to the module. If this character is upper case, the symbol is global to the module: either the module has defined the symbol and is allowing other modules to access it or the module needs to access the symbol, which must be defined as a global or public symbol in another module. The type codes are:

ab The symbol was defined using the assembler's EQU directive. The value listed is the equated value of its symbol.

The compiler doesn't generate symbols of this type.

pg The symbol is in the code segment. The value is the offset of the symbol within the code segment.

The compiler generates this type symbol for function names. Static functions are local to the function, and so have type *pg*; all other functions are global, that is, callable from other programs, and hence have type *Pg*.

dt The symbol is in the initialized data segment. The value is the offset of the symbol from the start of the data segment.

The compiler generates symbols of this type for initialized variables which are declared outside any function. Static variables are local to the program and so have type *dt*; all other variables are global, that is, accessable from other programs, and hence have type *Dt*.

ov When an overlay is being linked and that overlay itself calls another overlay, this type of symbol can appear in the rsm file for the overlay that is being linked. It indicates that the symbol is defined in the program that is going to call the overlay that is being linked.

The value is the offset of the symbol from the beginning of the physical segment that contains it.

un The symbol is used but not defined within the program. The value has no meaning.

In assembly language terms, a type of *Un* (the U is capitalized) indicates that the symbol is the operand of a *public* directive and that it is perhaps referenced in the operand field of some statements, but that the program didn't create the symbol in a statement's label field.

The compiler generates *Un* symbols for functions that are called but not defined within the program, for variables that are declared to be *extern* and that are actually used within the program, and for uninitialized, global dimensionless arrays. Variables which are declared to be *extern* but which are not used within the program aren't mentioned in the assembly language source file generated by the compiler and hence don't appear in the object file.

bs The symbol is in the uninitalized data segment. The value is the space reserved for the symbol.

The compiler generates *bs* symbols for static, uninitialized variables which are declared outside all functions and which aren't dimensionless arrays.

The assembler generates *bs* symbols for symbols defined using the *bss* assembler directive.

Gl The symbol is in the uninitialized data segment. The value is the space reserved for the symbol.

The compiler generates *Gl* symbols for non-static, uninitialized variables which are declared outside all functions and which aren't dimensionless arrays.

The assembler generates *Gl* symbols for variables declared using the *global* directive which have a non-zero size.

Enhanced MAKE

(Duplicate Page - First issued with release 1.06h)

Make has been enhanced to support dependencies on files in directories other than the current directory. This is best illustrated by an example.

Suppose the current directory is /src1 and that the program is being built from source there and in the /src2 directory. The makefile line would look like this:

```
program : /src1/main.o /src2/sub.o
        ln -o program /src1/main.o /src2/sub.o -lc
```

This will build main.o and sub.o from their sources, putting the object files (the .o's) in the SAME directory as the sources. The .o file will always be placed in the same directory as the source file.

Note : Unfortunately there isn't a way to say "all my .o's are in one directory, and all my .c's are in another".

Any futher references to these files should be with the same path or else make will be confused.

For object files to be dependent on files in another directory, the full pathname must be used as in:

```
/src2/sub.o : /header/defs.h
```

Make also supports command line macro definition such as:

```
make MACRO=text
```

the equal sign (=) is the key to having it be a macro definition. If the macro is to be empty, enter:

```
make MACRO=
```

Command line macro definition always overrides makefile macro definition. Be careful about your macro definitions. Some operating systems do not support quotes around text, so any white space in the macro text will cause the macro definition to terminate prematurely. This is not a problem under our SHELL on the Macintosh.

NAME

cp - copy

SYNOPSIS

cp [-f] infile outfile

cp [-f] file1 file2 ... directory

cp indrive outdrive

DESCRIPTION

cp copies things: it can copy one or more selected files, and can copy entire disks.

1. Copying selected files

cp can copy any type file, and will copy all of the file's 'forks'. The 'last modified' time of each copied file is set to that of the original file.

The first two forms of *cp* shown above are used to copy selected files. The first one copies a single file, *infile*, to *outfile.*

The second form copies all the specified files into the specified directory, with their original names. In this form, the name of the target directory must contain a terminating '/', unless the directory name is '.', '..', or x:.

The *-f* option forces *cp* to copy, even if the target file already exists. If this option isn't used, and if a target file already exists, *cp* will ask if you want to overwrite it.

2. Copying disks

The third form shown above for *cp* copies one entire disk onto another. *cp* verifies that the copy was correctly done. For example, the following command will copy the disk in drive 1: onto the disk in drive 2:

cp 1: 2:

cp will respond by asking

Are you sure?

Type *y* followed by return if you really want the copy to be made. Typing anything else will cause *cp* to halt without copying.

When *cp* is done, it will eject the newly created disk and return control to the SHELL.

EXAMPLES

cp hello.c data:/source/hithere.c

Makes a copy of the file *hello.c* which is in the current directory. The copy, which is named *hithere.c*, is in the */source* directory on the *data:* volume.

cp sys:/bin/* sys2:/work/

Copies all files in the directory *sys:/bin* to the directory *sys2:/work*. Note the terminating '/' in the name of the target directory.

cp /bin/*.c .

Copies all files having extension *.c* in the directory */bin* on the current volume to the current directory.

SEE ALSO

cat, mv

NAME

cprsrc - Resource copy utility

SYNOPSIS

cprsrc [-f] type id file1 file2

DESCRIPTION

cprsrc is a utility program which copies a resource from *file1* to *file2*. *type* is the type of the resource, and *id* is its resource number.

If the *-f* option isn't specified and if a resource of the specified TYPE and ID exists in the destination file, *cprsrc* will display the following message before copying the resource:

Resource already exists, replace?

If you then type 'y', followed by return, *cprsrc* will remove the old resource and add the new resource. If you type anything else, *cprsrc* will halt without copying the resource.

If the **-f** option is specified, *cprsrc* will automatically remove a pre-existing resource of the specified TYPE and ID from the destination file, without asking for your permission. This should be used if you know a resource already exists and are sure that it should be overwritten.

EXAMPLES

The following examples demonstrate the *cprsrc* program.

```
cprsrc DRVR 31 con System
cprsrc -f DRVR 13 System clock
cprsrc FONT 512 System fontfile
cprsrc FONT 521 System fontfile
```

The first example copies the console driver from the file, *con* to the *System* file. The second example copies the *clock* desk accessory from the *System* file to a file called *clock*. If the resource already exists in *clock*, it will be deleted automatically. The last two examples copy the Monaco font type and the 9 point version from the *System* file to the *fontfile* file.

NAME

date - display date and time

SYNOPSIS

date

DESCRIPTION

Displays the date and time.

NAME

diff - Source file comparison utility

SYNOPSIS

diff [-b] file1 file2

DESCRIPTION

diff is a program, similar to the UNIX program of the same name, that determines the differences between two files containing text. *file1* and *file2* are the names of the files to be compared.

1. The -b option

The -b option causes *diff* to ignore trailing blanks (spaces and tabs) and to consider strings of blanks to be identical. If this option isn't specified, *diff* considers two lines to be the same only if they match *exactly*.

For example, if file1 contains the the line

^abc$

(^ and $ stand for "the beginning of the line" and "the end of the line", respectively, and aren't actually in the file) and if file2 contains the line

^abc $

then *diff* would consider the two lines to be the same or different, depending on whether or not it was started with the -b option.

And *diff* would consider the lines

^a b c$

and

^a b c$

to be the same or different, depending on whether or not it was started with the -b option.

diff will never consider blanks to match a null string, regardless of whether -b was used or not. So *diff* will never consider the lines

^abc$

and

^a bc$

to be the same.

2. The conversion list

diff writes, to its standard output, a "conversion list" that describes the changes that need to be made to *file1* to convert it into *file2*. The list is organized into a sequence of items, each of which describes one operation that must be performed on *file1*.

2.1 Conversion items

There are three types of operations that can be specified in a conversion list item:

* adding lines to *file1* from *file2*;
* deleting lines from *file1*;
* replacing (changing) *file1* lines with *file2* lines.

A conversion list item consists of a command line, followed by the lines in the two files that are affected by the item's operation.

2.1.1 The command line

An item's command line contains a letter describing the operation to be performed: 'a' for adding lines, 'd' for deleting lines, and 'c' for changing lines.

Preceding and following the letter are the numbers of the lines in *file1* and *file2*, respectively, that are affected by the command. If a range of lines in a file are affected, just the beginning and ending line numbers are listed, separated by a comma.

For example, the following command line says to add line 3 of *file2* after line 5 of *file1*:

```
5a3
```

and the next command line says to add lines 8,9, and 10 of *file2* after line 16 of *file1*:

```
16a8,10
```

The next command line says to delete lines 100 through 150 from *file1*, and that the last line in *file2* that matched a *file1* line was number 75:

```
100,150d75
```

The following command says to replace (change) line 32 in *file1* with line 33 in *file2*:

```
32c33
```

and the next command says to replace lines 453 through 500 in *file1* with lines 490 through 499 in *file2*:

```
453,500c490,499
```

2.1.2 The affected lines

As mentioned above, the lines affected by a conversion item's operation are listed after the item's command line. The affected lines from *file1* are listed first, flagged with a preceding '<'. Then come the affected lines from *file2*, flagged with a preceding '>'. The *file1* and *file2* lines are separated by the line

```
---
```

For example, the following conversion item says to add line 6 of *file2* after line 4 of *file1*. Line 6 of *file2* is "for (i=1; i<10;++i)":

```
4a6
> for (i=1; i<10;++i)
```

Since no lines from *file1* are affected by an 'add' conversion item, only the *file2* lines that will be added to *file1* are listed, and the separator line "---" is omitted.

The following conversion item says to delete lines 100 and 101 from *file1*, and that the last *file2* line that matched a *file1* line was numbered 110. The deleted lines were "int a;" and "double b;". Only the deleted lines are listed, and the separator line "---" is omitted:

```
100,101d110
< int a;
< double b;
```

The following conversion item says to replace lines 53 through 56 in *file1* with lines 60 and 61 in *file2*. Lines 53 through 56 in *file1* are "if (a=b){", " d = a;", " a++;", and "}". Lines 60 and 61 of *file2* are "if (a==b)" and "d = a++;".

```
53,55c60,61
< if (a=b){
<   d = a;
<   a++;
< }
---
> if (a==b)
>   d = a++;
```

3. Differences between the UNIX and Manx versions of *diff*

The Manx and UNIX versions of *diff* are actually most similar when the latter program is invoked with the -h option. As with the UNIX *diff* when used with the -h option, the Manx *diff* works best when changed stretches are short and well separated, and works with files of unlimited length.

Unlike the UNIX *diff*, the Manx *diff* doesn't support the options e, f, or h.

Unlike the UNIX *diff*, the Manx version requires that both operands to *diff* be actual files. Because of this, the Manx version of *diff* doesn't support the features of the UNIX version which allows one operand to be a directory name, (to specify a file in that directory having the same name as the other operand), and which allows one operand to be '-' (to specify *diff*'s standard input instead of a file).

NAME

echo - echo arguments

SYNOPSIS

echo [arg] [arg] ...

DESCRIPTION

echo writes its arguments, separated by blanks and terminated by a newline, to its standard output device.

The output thus goes, by default, to the screen. It can also be redirected to another device or file in the normal manner.

Before *echo* is called, substitutions of the appropriate command line arguments are made:

* File name templates are replaced with matching file names.
* If the echo command is in an exec file, any exec file variables (such as $1 and $@) are replaced by their corresponding values.

The code for *echo* is contained in the SHELL.

EXAMPLES

The command

```
echo *
```

prints the names of the files in the current directory to the screen.

The following illustrates use of *echo* in an exec file:

```
loop i in $*
echo compiling $i
cc $i
eloop
```

NAME

Edit - Mouse-based editor

SYNOPSIS

Edit

DESCRIPTION

This section describes the *Edit* mouse-based editor supplied as part of the Apple supplement to the Aztec C system. *Edit* is a stand-alone program which can be run from the Finder or from the SHELL. When run from the Finder, it is possible to double click the *Edit* icon itself, or a text file created with the *Edit* program. From the SHELL, just typing *edit* followed by a carriage return will invoke the editor.

1. About Edit

Edit is a mouse and window based editor. Unlike Z, it can edit a file bigger than the available memory. It does this by only keeping the displayed part of the file in memory. The rest remains on disk. This also allows the editor to have up to four files open in different windows at one time.

Files are created by selecting *New* from the *File* menu. There are several ways to edit an existing file. The first way is to select the first *Open* from the *File* menu. This will give the standard file selection dialog box which can be used to select the file desired. A file can also be opened by selecting the name of the file in an open file using the mouse and then choosing the second *Open* from the *File* menu.

The last way is to type *Clover-K* followed by the volume name, a colon and the file name of the file to be opened followed by a Return. There will be nothing displayed while this is being typed. The volume name and colon are optional.

When the editor is exited, it will automatically check whether an open document has been modified and will ask whether the modifications will be saved.

2. Editing

Files created and edited by the editor consist of lines of normal ASCII characters separated by Returns. There is no formatting information stored in the file, so only one font is permitted per file. Text editing is performed as one would expect with the standard *Edit* menu. Operations may be performed between file windows.

Search and replace operations are performed using the *Find* and *Change* options of the *Search* menu. The search starts at the current insertion point. Subsequent searches and replaces use the last string typed.

The *Format* menu allows the alignment of text to be adjusted. The *Set Tabs* option allows the width of the tab to be adjusted. When a line is ended, normally the next one is started at the left margin. If the *Auto Indent* option is selected, the new line is started at the same indent level as the previous line.

The *Align*, *Move Left*, and *Move Right* options of the *Edit* menu work with a block of text that has been selected. The *Align* option aligns the left margins of all the lines. The *Move* options shift the whole block left or right a space.

NAME

FixAttr

SYNOPSIS

FixAttr

DESCRIPTION

FixAttr changes the attributes of the resources within the current System file that contain the console driver and the Monaco 9- and 12-point fonts.

The new attributes cause these resources to be loaded into the system heap and made non-purgable when the Macintosh is rebooted.

Since, unlike the application heap, the system heap is not cleared when an application program terminates, making the frequently-used resources resident and non-purgable in the system heap means that they will be in memory when needed by an application program, and won't have to be loaded from disk.

If you don't use *FixAttr*, these resources will automatically be loaded into the application heap when needed by an application program. However, the application heap is cleared when each application program terminates, so if these resources aren't in the system heap they must be loaded into the application heap for each application program that needs them.

Thus, if you use *FixAttr*, your system will run faster, since these resources don't have to be loaded for each application that needs them.

FixAttr should only be used on Macintoshes having 512K bytes of RAM memory, since only it has a system heap big enough to hold these resources.

SEE ALSO

The Technical Information chapter has a section that discusses the use of Aztec C68K on Macintoshes having 512K bytes of RAM.

NAME

fldr - folder utility

SYNOPSIS

fldr -d

fldr -f file1 file2 ...

fldr id file1 file2 ...

DESCRIPTION

The first form of the command displays the names of the folders known to the Finder and their associated ids.

The second form of the *fldr* command displays the number of the folders containing the specified files.

The third form moves the specified files into the folder whose number is *id*, and sets the fdFlags field of each file to 0x100.

This form of the command is useful when a disk is being used with both the Finder and the SHELL: it provides a fast way of moving a set of files to a particular folder. Also, it initializes fields that otherwise will be initialized by the Finder. The Finder's algorithm for doing this is very slow, so it's better to initialize them using *fldr*.

NAME

grep - pattern-matching program

SYNOPSIS

grep [-cflnv] pattern [files]

DESCRIPTION

grep is a program, similar to the UNIX program of the same name, that searches files for lines containing a pattern. By default, such lines are written to *grep*'s standard output.

1. Input files

The **files** parameter is a list of files to be searched. If no files are specified, *grep* searches its standard input. Each file name can specify a single file to be searched. A name can also specify a class of files to be searched, using the special characters '*' and '?'. The character '*' matches any string of characters in a file name, and '?' matches any single character. For example,

grep int main.c sub1.c sub2.c

searches *main.c, sub1.c,* and *sub2.c* for the string *int.* The command

grep int *.c

searches all files whose extension is *.c* for the string *int.* The command

grep int a*.txt b*.doc

searches for the string *int* in each file whose (1) extension is *.txt* and *first* character is *a* and whose (2) extension is *.doc* and first character is *b.* The command

grep int sub?.c

searches for the string *int* in each file whose filename contains four characters, the first three being *sub,* and whose extension is *.c.*

2. Options

The following options are supported:

v	Print all lines that don't match the pattern.
c	Print just the name of each file and the number of matching lines that it contained.
l	Print the names of just the files that contain matching lines.
n	Precede each matching line that's printed by its relative line number within the file that contains it.
f	A character in the pattern will match both its upper and lower case equivalent.

3. Patterns

A pattern consists of a limited form of regular expression. It describes a set of character strings, any of whose members are said to be matched by the regular expression.

Some patterns match just a single character; others, which match strings, can be constructed from those that match single characters. In the following paragraphs, we'll first describe the patterns that match a single character, and then describe patterns that match strings of characters.

3.1 Matching single characters

The patterns that match a single character are these:

* An ordinary character (that is, one other than the special characters described below) matches itself.

* A period (.) is a pattern that matches any character except newline.

* A non-empty string of characters enclosed in square brackets, [], matches any one character in that string. For example, the pattern

  [ad9@]

  matches any one of the characters *a, d, 9,* or @.

  If, however, the string begins with the caret character (^), the regular expression matches any character except the other enclosed characters and newline. The '^' has this special meaning only if it is the first character of the string. For example, the pattern

  [^ad9@]

  matches any single character *except a, d, 9,* or @.

  The minus character ,-, can be used to indicate a range of consecutive ASCII characters. For example, [0-9] is equivalent to [0123456789].

* A backslash (\) followed by a special character matches the special character itself. The special characters are:

  ., *, [, and \, which are always special, except when they appear in square brackets, [].

  ^ (caret), which is special when it is at the beginning of an entire regular expression (as discussed in 3.4) and when it immediately follows the left of a pair of square brackets.

  $, which is special at the end of an entire regular

expression (discussed in 3.4).

3.2 Matching character strings

Patterns can be concatenated. In this case, the resulting pattern matches strings whose substrings match each of the concatenated patterns. For example, the pattern

```
abc
```

matches the string *abc*. This pattern is built from the patterns *a*, *b*, and *c*. The pattern

```
a.c
```

matches strings containing three characters, whose first and last characters are *a* and *c*, respectively, such as

```
abc
a@c
axc
```

3.3 Matching repeating characters

A pattern can be built by appending an asterick (*) to a pattern that matches a single character. The resulting pattern matches zero or more occurrences of the single-character pattern For example, the pattern

```
a*
```

matches any line containing zero or more *a* characters. And the pattern

```
sub[1-4]*end
```

matches lines containing strings such as

```
subend
sub132132end
```

3.4 Matching strings that begin or end lines

An entire pattern may be constrained to match only character strings that occur at the beginning or the end of a line, by beginning or ending the pattern with the character '^' or '$', respectively. For example, the pattern

```
^main
```

matches the line that begins

```
main
```

but not one that begins

```
the main ...
```

The pattern

line$

matches the line ending in

... the end of the line

but not the line ending in

a hard-hit line drive.

4. Examples

4.1 Simple string matching

The following command will search the files *file1.txt* and *file2.txt* and print the lines containing the word *heretofore*:

grep heretofore file1.txt file2.txt

If you aren't interested in the specific lines of these files, but just want to know the names of the files containing the word *heretofore*, you could enter

grep -l heretofore file1.txt file2.txt

The above two examples ignore lines in which *heretofore* contains capital letters, such as when it begins a sentence. The following command will cover this situation:

grep -lf heretofore file1.txt file2.txt

grep processes all options at once, so multiple options must be specified in one dash parameter. For example, the command

grep -l -f heretofore file1.txt file2.txt

won't work.

4.2 The special character '.'

Suppose you want to find all lines in the file prog.c that contain a four-character string whose first and last characters are 'm' and 'n', respectively, and whose other characters you don't care about. The command

grep m..n prog.c

will do the trick, since the special character '.' matches any single character.

4.3 The backslash character

There are occasions when you want to find the character '.' in a file, and don't want grep to consider it to be special. In this case, you can use the backslash character, '\', to turn off the special meaning of the next character.

For example, suppose you want to find all lines containing

.PP

Entering

grep .PP prog.doc

isn't adequate, because it will find lines such as

THE APPLICATION OF

since the '.' matches the letter 'A'. But if you enter

grep \.PP prog.doc

grep will print just what you want.

The backslash character can be used to turn off the special meaning of any special character. For example,

grep \\n prog.c

finds all lines in prog.c containing the string '\n'.

4.4 The dollar sign and the caret ($ and ^)

Suppose you want to find the number of the line on which the definition of the function *add* occurs in the file *arith.c.* Entering

grep -n add arith.c

isn't good, because it will print lines in which *add* is called in addition to the line you're interested in. Assuming that you begin all function definitions at the beginning of a line, you could enter

grep ^add arith.c

to accomplish your purpose.

The character '$' is a companion to '^', and stands for 'the end of the line'. So if you want to find all lines in *file.doc* that end in the string *time,* you could enter

grep time$ file.doc

And the following will find all lines that contain just *.PP:*

grep ^\.PP$

4.5 Using brackets

Suppose that you want to find all lines in the file *file.doc* that begin with a digit. The command

grep ^[0123456789] file.doc

will do just that. This command can be abbreviated as

grep ^[0-9] file.doc

And if you wanted to print all lines that don't begin with a digit, you could enter

```
grep ^[^0-9] file.doc
```

4.6 Repeated characters

Suppose you want to find all lines in the file *prog.c* that contain strings whose first character is 'e' and whose last character is 'z'. The command

```
grep e.*z prog.c
```

will do that. The 'e' matches an 'e', the '.*' matches zero or more arbitrary characters, and the 'z' matches a 'z'.

5. Differences between the Manx and UNIX versions of grep

The Manx and UNIX versions of *grep* differ in the options they accept and the patterns they match.

5.1 Option differences

* The option -f is supported only by the Manx *grep.*
* The options -b and -s are supported only by the UNIX *grep.*

5.2 Pattern differences

Basically, the patterns accepted by the Manx *grep* are a subset of those accepted by the UNIX *grep.*

* The Manx *grep* doesn't allow a regular expression to be surrounded by '\(' and '\)'.
* The Manx *grep* doesn't accept the construct '\{m\}'.
* The Manx *grep* doesn't allow a right bracket, ']', to be specified within brackets.

NAME

hd - hex dump utility

SYNOPSIS

hd [-r] [+n[.]] file1 [+n[.]] file 2 ...

DESCRIPTION

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

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

-r causes the file's resource forks to be displayed. This option can occur between any two files. Before it is encountered, the file's data forks are displayed.

+n specifies the offset into the file where the display is to start, and defaults to the beginning of the file. If *+n* is followed by a period, *n* is assumed to be a decimal number; otherwise, it's assumed to be hexadecimal. Each file will be displayed beginning at the last specified offset.

EXAMPLES

hd +16b oldtest newtest +0 junk

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

hd -r +1000. tstfil

Displays the contents of the resource fork of *tstfil*, beginning at byte 1000.

NAME

InstallConsole - Console Driver Installation Utility

SYNOPSIS

InstallConsole

DESCRIPTION

InstallConsole places a copy of the Aztec console driver in a specified file. *InstallConsole* is a command program, and can be activated by either the SHELL or the Finder.

When invoked, *InstallConsole* displays the standard Mini-Finder display, listing all files that have a resource fork. When you select a file and click "Open", *InstallConsole* will read the console driver from the file that contains *InstallConsole* and copy it into the resource fork of the specified file. If a resource of the same name already exists, *InstallConsole* will ask if it should be replaced.

After the driver has been installed, *InstallConsole* displays a message and restores the Mini-Finder display. You can then Click the "Cancel" button, to terminate *InstallConsole*, or install the console driver in another file.

SEE ALSO

For more information on command programs that call the console driver, and on the console driver itself, see the Command Programs and Console Driver sections of the Technical Information chapter.

NAME

libutil - object module librarian

SYNOPSIS

libutil [-o library] [-atxrv] modules

DESCRIPTION

libutil is a program that is used to create and manipulate libraries of object modules.

This description of *libutil* contains two major sections: the first summarizes the use of libutil; the second describes libutil in more detail.

1. Summary

libutil manipulates object file libraries, in the following ways (the command line option which initiates the action is in parentheses):

a. create a library	
b. append to a library	(-a)
c. list library modules	(-t)
d. extract library modules	(-x)
e. replace library modules	(-r)
f. take file names from stdin	(.)
g. be verbose	(-v)

The following paragraphs give examples of *libutil* usage:

a. Creating a library

The following creates a library, *example.lib*, containing the modules *sub1.o* and *sub2.o:*

```
libutil -o example.lib sub1.o sub2.o
```

b. Appending modules to a library

This example appends *exmpl.o* to the library *example.lib:*

```
libutil -oa example.lib exmpl.o
```

c. Listing library modules

The following lists the modules in *progs.lib:*

```
libutil -ot prog.lib
```

d. Extracting modules from a library

This option allows either selected modules or all modules to be extracted from a library and placed in separate files. The library itself is not modified. The following copies the module *exmpl* from the library *mylib.lib* into the file *exmpl.o:*

```
libutil -ox mylib.lib exmpl
```

e. Replacing library modules

This example replaces the module *sub1* in library *example.lib* with the contents of the object file *sub1.o:*

```
libutil -or example.lib sub1
```

f. Taking commands from stdin

The following creates a library, *subs.lib,* and appends to it *sub1.o, sub2.o, sub3.o,* and *sub4.o.* File names are taken from *command.fil:*

```
libutil -o subs.lib . <command.fil
```

where *command.fil* contains:

```
sub1.o sub2.o
sub3.o sub4.o
```

2. In More Detail...

2.1 Creating a Library

The command for creating a new library has this format:

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

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

2.2 How it Works

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

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

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

2.3 Naming Conventions

An input filename has the standard format. The drive or volume component defaults to the current drive or volume, and the directory to the current directory. The extension for the filename is optional; if not specified, it's assumed to be *.o.*

When an input file contains a single relocatable object module, the name of the module in the library will be the filename, less its other components. For example, if the input file is *b:sub1.o*, then the module name inside the new library will be *sub1*.

An input file can be a library itself. In this case, the module names in the new library are the same as those in the input library. For example, if the input file is a library containing modules *sub1*, *sub2* and *sub3*, then the names of these modules in the created library will also be *sub1*, *sub2* and *sub3*.

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

2.4 Order in a Library

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

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

```
sub1    sub2    sub3
```

The following command might be given:

```
libutil -o newlib.lib oldlib.lib sub4 . sub5 <cmd.fil
```

where *cmd.fil* contains the following:

```
sub6.o sub7.o
sub8.o
```

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

When *libutil* finds a period, it continues reading the filenames from standard input. So the next three files copied into *newlib.lib* are *sub6.o*, *sub7.o* and *sub8.o*. Notice that *.o* after a filename in the command is assumed. The last module read in the example is in *sub5.o*. So the final makeup of *newlib.lib* is:

```
sub1    sub2    sub3    sub4    sub6    sub7
sub8    sub5
```

2.5 Listing the Modules in a Library

A listing such as this can be obtained with the -T option. This option simply produces a listing of the modules in the order in which

they appear in a library. The -O option is used in this case to specify which library is to be listed. For example, the listing above would be produced by entering:

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

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

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

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

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

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

2.6 Adding and Replacing Modules

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

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

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

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

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

```
mod1    mod2    mod3
```

and the following command:

```
libutil -oa obslib.lib mod2 . sub2 <cmd.fil
```

where *cmd.fil* contains:

sub1

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

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

Consider the following command:

libutil -oa obslib.lib obslib.lib

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

mod1 mod2 mod3 mod1 mod2 mod3

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

NAME

lock, unlock, flock, funlock

SYNOPSIS

lock file

unlock file

flock file

funlock file

DESCRIPTION

These commands are used to lock and unlock files. They are grouped into two pairs: lock/unlock and flock/funlock.

The lock/unlock commands set/reset the operating system's 'lock flag' for a file, and the flock/funlock commands set/reset the Finder's 'lock flag' for a file.

With the operating system's lock flag set for a file, the file can't be removed or opened for writing by the SHELL, by programs activated by the SHELL, or by the Finder.

With the Finder's lock flag set for a file, the Finder will not allow the file to be deleted; however, the file can still be deleted or modified by the SHELL or by SHELL-activated programs.

Thus, lock/unlock are more useful than flock/funlock.

The code for these commands is contained in the SHELL.

NAME

ls - list files and directories

SYNOPSIS

ls [-lt] [name] [name] ...

DESCRIPTION

ls lists the names of the files and the contents of the directories whose names are passed to it as arguments. If no names are specified, the contents of the current directory are listed.

To specify a directory other than '.' or '..', the name must contain a terminating '/' character.

The information is written to *ls*'s standard output device, and hence goes, by default, to the screen. The information can be sent to another device or file by redirecting *ls*' standard output device in the normal manner.

By default, the output is sorted alphabetically. The *-t* option causes the output to be sorted according to the 'last modified' times, with the most recently modified file listed first and directories listed last.

When the *-l* option isn't specified and a directory's contents are listed, the names of subdirectories within the directory are terminated by a slash character, allowing subdirectories to be distinguished from files.

The *-l* option causes the listing to be made in 'long format', in which additional information is displayed for each file. In this case, the listing for a file has the following format:

Finder Info			Directory Info				
Type	Creator	Flags	Flags	Data-len	Rsrc-len	Mod	Name

where the items correspond to fields within the directory entry for the file, as follows:

Type - The type of the file, as recorded in the directory.

Creator - The creator of the file, as recorded in the directory.

Flags - The first 'flags' field contains flags used by the Finder.

Flags - The second 'flags' field contains flags used by the operating system. It's also one byte wide, and it's bits have the following meanings:

bit 0 - lock flag.
bit 7 - 'in use' flag. Set when file is open.

Data-len - The length, in bytes, of the file's data fork.

Rsrc-len - The length, in bytes, of the file's resource fork.

Mod - The date and time that the file was last modified.

Name - The file's name.

The code for the *ls* command is contained in the SHELL.

EXAMPLES

ls >catalog

Writes the names of the files in the current directory to the file *catalog* in the current directory.

ls -l data:/source/*.c

Displays information about the files having extension *.c* which are are contained in the */source* directory on the *data:* volume.

ls -lt sys:/bin/

Displays information about the files in the directory *sys:/bin.* The listing is sorted by last modified times, with directories listed last. Note the terminating '/' to the directory name.

ls ..

Displays the names of the files and directories in the parent directory to the current directory.

NAME

MacsBug - Macintosh Assembly Language Debugger

SYNOPSIS

MacsBug

DESCRIPTION

This section describes the *MacsBug* family of debuggers supplied as part of the Apple supplement to the Aztec C system. *MacsBug* is a line-oriented assembly language debugger which runs on a single Macintosh.

1. Setting Up MacsBug

MacsBug is not started like a normal Macintosh or SHELL application. Instead, it must be installed as part of the system during the bootup of the machine. The boot code looks for a file with the name *MacsBug* and if found installs it. Since there are several different versions of *MacsBug* supplied, the one that matches the spelling will be installed.

The five versions are:

version	*size*	*description*
MacsBug	18K	8 lines, Macintosh screen
MaxBug	40K	40 lines, Macintosh screen
TermBugA	12K	serial port A terminal
TermBugB	12K	serial port B terminal
LisaBug	40K	40 lines, Lisa screen, MacWorks

2. Starting MacsBug

The simplest way to start the debugger and get to command mode, is to press the interrupt part of the programmer's switch on the Macintosh. For the Lisa, press the '-' key on the numeric pad.

To get control of programs developed using Aztec C68K, get into the debugger while the SHELL is running, and type the following:

AB 175

which sets a breakpoint to occur when a toolbox call to the *TickCount()* routine is made. Just such a call has been placed at the beginning of the various *Croot()* routines used with the Aztec C68K system. Then type **G** to get the SHELL going again and run the program to be debugged. After the program is loaded, the trap will return control to the debugger which can then be used to set breakpoints at specific routine addresses.

When a break is encountered, *MacsBug* disassembles the instruction at the current program location, and displays the contents of all the

registers. Then it displays the '>' symbol which is its command input prompt. The debuggers which talk to the screen directly, save the old screen image in a buffer. To see the old screen, hit the (~) key. Any other key will return to the debugger screen.

3. Command Syntax

Commands are typed as one or two characters followed by some number of arguments. Arguments are expressions ranging from a simple number to more complex combinations.

3.1 Numbers

The default format for numbers is hexadecimal. Decimal numbers may also be entered if preceded by an ampersand (&). Hexadecimals may optionally be preceded by a dollar sign ($). Signed numbers should have the sign before the & or $. For example:

Number	*Unsigned Hex*	*Signed Hex*	*Decimal*
$FF	$000000FF	$000000FF	&255
-$FF	$FFFFFF01	-$000000FF	-&255
&100	$00000064	$00000064	&100
+10	$00000010	$00000010	&16

3.2 Text Literals

Text literals are similar to character literals as specified in C. The can be from one to four characters long, but if less than four, are stored right justified. For example:

String	*Stored As*
'A'	$00000041
'Fred'	$46726564
'1234'	$31323334

3.3 Symbols and Register Names

The symbol and register names recognized by *MacsBug* are:

RA0 thru RA7	Address registers A0 thru A7
RD0 thru RD7	Data registers D0 thru D7
PC	Program counter
.	Last address referenced
TP	Current QuickDraw port

3.4 Expressions

Expressions are formed by using the preceding numbers, literals and symbols in conjunction with operators. The operators supported by the debugger are:

+	addition (infix), assertion (prefix)
-	subtraction (infix), negation (prefix)
@	indirection (prefix)

The @ operator uses the long value at the location specified by the operand. Some examples of valid expressions are:

```
RA7+4
1A700-@10C
TP+&24
-RA0+RA1-'FRED'+@@4C50
```

4. Memory Commands

4.1 DM Display Memory

DM ADDR NUM

Displays **NUM** bytes of memory in hex and ASCII starting at **ADDR**. **NUM** is rounded up to the nearest multiple of 16 bytes. If omitted, it defaults to 16. If both are omitted, the next 16 bytes are displayed.

Pressing Return displays the next set of **NUM** bytes. After the command, the '.' is set to **ADDR**.

If **NUM** is set to certain four character strings, memory is displayed in a special format depending on the particular string. The strings and the formats are:

'IOPB'	Input/Output Parameter Block
'WIND'	Window Record
'TERC'	TextEdit Record

4.2 SM Set Memory

SM ADDR EXPR1 EXPR2 ... EXPRN

Modifies memory starting at **ADDR** using the values represented by **EXPRn**. Each expression has a width associated with it that determines how many bytes are affected by each expression.

The width of text literals corresponds to the number of characters, up to 4. Values from indirection are always four bytes. Decimal and hexadecimal values are as wide as the smallest number of bytes that will hold the value. An expression is as wide as the widest of its elements.

5. Register Commands

5.1 D Data Register

Dn EXPR

This command displays the appropriate register if no expression is specified. If one is specified, the register is set to the value of the expression.

5.2 A Address Register

An EXPR

This command displays the appropriate register if no expression is specified. If one is specified, the register is set to the value of the expression.

5.3 PC Program Counter

PC EXPR

This command displays the appropriate register if no expression is specified. If one is specified, the register is set to the value of the expression.

5.4 SR Status Register

SR EXPR

This command displays the appropriate register if no expression is specified. If one is specified, the register is set to the value of the expression.

5.5 TD Total Display

TD

This command displays all registers, including the disassembled instruction at the current program counter location.

6. Control Commands

6.1 BR Set Breakpoint

BR ADDR CNT

This command sets a breakpoint at **ADDR** that will cause the debugger to stop after it has been encountered **CNT** times. If the **CNT** is omitted, it defaults to one. If the **ADDR** is omitted, all the breakpoints currently set are displayed. Up to 8 breakpoints can be set.

6.2 CL Clear Breakpoint

CL ADDR

Removes the breakpoint at **ADDR**. If omitted, all breakpoints are removed.

6.3 G Go

G ADDR

Resumes execution after setting the PC to the address specified. If the address is omitted, execution is resumed at the current value of the PC.

6.4 GT Go Till

GT ADDR

Sets a temporary breakpoint at **ADDR** and begins executing instructions at the current value of the program counter. When control is returned to the deugger, the breakpoint is cleared and forgotten.

6.5 T Trace

T

Single steps through one instruction at a time. Traps to the Toolbox and Operating System are treated as a single instruction.

6.6 S Step

S NUM

Single steps through **NUM** instructions or just one if omitted. Traps are not treated as a single instruction.

6.7 SS Step Spy

SS ADDR1 ADDR2

This command is used to determine when a particular range of memory gets modified. When executed, a checksum is performed on the memory range specified. Then, the debugger single steps through the program checking the checksum before each instruction and returns control to the user when the checksum changes.

6.8 ST Step Till

ST ADDR

Single steps through all instructions until an instruction at the address, **ADDR**, is about to be executed. This is primarily used to set breakpoints in the ROM.

6.9 MR Magic Return

MR NUM

This command is used to set a temporary breakpoint at the return address of a function. More specifically, it uses the address which is located **NUM** bytes off of the stack pointer as the breakpoint address. If the offset is omitted, it is assumed to be the top of the stack.

6.10 RB Reboot

RB

Reboots the system.

6.11 ES Exit to Shell

ES

This launches the current startup application, aborting the program being debugged.

7. A-Trap Commands

The A-Trap commands are used to set breakpoints based on access to the Macintosh Toolbox or Operating System routines. Traps may be selected based on the trap number, the memory location where the trap was executed, and the contents of the D0 register.

When selecting the trap number, values from 0 through 511 may be used. If one trap number is specified, then that is the only value trapped. If two trap numbers are specified, then any trap in the range will be trapped.

If an address is specified, a second address marking the end of the memory range must be specified as well. Only traps that are in the specified range and that occur in the memory range will be trapped.

Finally, if a range for the D0 register is specified, trapping will only occur if the trap is in the specified trap range, occurs in the designated memory range and the D0 register contains a value in the proper range. Only one A-Trap command may be active at a time.

7.1 AB A-Trap Break

AB TRAP1 TRAP2 ADDR1 ADDR2 D1 D2

Breaks to the debugger when the appropriate conditions have been met.

7.2 AT A-Trap Trace

AT TRAP1 TRAP2 ADDR1 ADDR2 D1 D2

Displays the A-Trap paramaters for each trap which meets the specified conditions. Unlike the **AB** command it does not stop.

7.3 AH A-Trap Heap Zone Check

AH TRAP1 TRAP2 ADDR1 ADDR2 D1 D2

Performs a Heap Check before each trap specified is executed. If an error is discovered, the address of the block in question is displayed, and control returned to the debugger.

7.4 HS Heap Scramble

HS TRAP1 TRAP2

Scrambles the heap zone when certain traps in the specified range are encountered. It always scrambles the heap zone as a result of *NewPtr(), NewHandle()*, and *ReallocHandle()* calls. It scrambles the heap zone as a result of *SetHandleSize()* and *SetPtrSize()* if the new length is greater than the current length.

This command is fastest if you set TRAP1 to $18 and TRAP2 to $2D. The heap zone is not scrambled as a result of traps other than those named above.

7.5 AS A-Trap Spy

AS TRAP1 TRAP2 ADDR1 ADDR2

This command calculates a checksum for the specified range of memory and then checks it before each trap in the range is executed. If the checksum changes, control is returned to the debugger.

7.6 AX A-Trap Clear

AX

Removes all A-Trap breakpoints.

8. Heap Zone Commands

8.1 HX Heap Exchange

HX

Switches the heap zone being examined between the application and the system heap zones. The default is the application heap zone.

8.2 HC Heap Check

HC

Checks the current heap zone for internal consistency. If an error is found, the address of the faulty block is displayed.

8.3 HD Heap Dump

HD MASK

This command displays each block in the current heap zone in the following format

BlkAddr Type Size [MstrPtr] [*] [RefNum ID Type]

BlkAddr is the block address, which points to the beginning of the memory block. The type is 0 for a free block, 4 for a pointer, and 8 for a relocatable block. The size is the physical size of the block, including the contents, header, and any unused bytes at the end of the block.

MstrPtr is the master pointer; it's given for relocatable blocks. The high order byte contains the lock, purge, and resource bits. The

asterisk marks any immobile objects.

For resource file blocks, three additional fields are displayed. They are the resource's reference number, the ID number, and the type.

If mask is omitted, the dump is followed by a summary of the heap zone's blocks. It begins with the six characters 'HLP PF' which serve as reminders for the six values that follow them. The six values are:

H	Number of relocatable blocks (handles)
L	Number of relocatable blocks that are locked
P	Number of purgable blocks space, in bytes, occupied by purgeable blocks
P	Number of nonrelocatable blocks (pointers)
F	Total amount of free space

If mask is present, the heap summary takes the form:

CNT ### NumBlks NumBytes

where NumBlks is the number of blocks of MASK type, and NumBytes is the number of bytes in those blocks.

8.4 HP Heap Print

HP

If you are using one of the terminal based versions of *MacsBug*, this command will do the dump to the other serial port.

8.5 HT Heap Total

HT MASK

Displays just the summary line from a heap zone dump. MASK is defined to function the same as in the HD command.

9. Disassembler Commands

9.1 ID Instruction Disassemble

ID ADDR

Disassembles one line at the address specified. If the address is omitted, the next logical address is used.

9.2 IL Instruction List

IL ADDR NUM

Disassembles NUM lines starting at ADDR. If the number is omitted, it defaults to a full screen. If the address is omitted, it uses the next logical address.

10. Miscellaneous Commands

10.1 F Find

F ADDR CNT DATA MASK

This command searches CNT bytes from ADDR, looking for DATA after masking the target with MASK. As soon as a match is found, the address and value are displayed. To search through the next CNT bytes, just press Return.

10.2 WH Where

WH EXPR

If EXPR is less than 512, this displays the address corresponding to the trap with that number. If EXPR is greater than or equal to 512, the trap whose code is closest to that address is displayed.

10.3 CS Checksum

CS ADDR1 ADDR2

Calculates the checksum for the bytes in the range specified. If the second address is not specified, sixteen bytes are summed. If neither address is specified, it computes the checksum for the last range summed and compares it to the previous value. If the checksum matches, the message:

CHKSUM T

is displayed. Otherwise, the message displayed is:

CHKSUM F

10.4 CV Convert

CV EXPR

Displays the expression as unsigned hexadecimal, signed hexadecimal, signed decimal, and ASCII.

10.5 RX Register Exchange

RX

Toggles whether or not the registers are dumped during a trace command. The disassembly of the current instruction is still performed.

NAME

make - Program maintenance utility

SYNOPSIS

make [-n] [-f makefile] [-a] [name1 name2 ...]

DESCRIPTION

make is a program, similar to the UNIX program of the same name, whose primary function is to create, and keep up-to-date, files that are created from other files, such as programs, libraries, and archives.

When told to make a file, *make* first ensures that the files from which the target file is created are up-to-date or current, recreating just the ones that aren't. Then, if the target file is not current, *make* creates it.

Inter-file dependencies and the commands which must be executed to create files are specified in a file called the 'makefile', which you must write.

make has a rule-processing capability, which allows it to infer, without being explicitly told, the files on which a file depends and the commands which must be executed to create a file. Some rules are built into *make*; you can define others within the makefile.

A rule tells *make* something like this:

> "a target file having extension '.x' depends on the file having the same basic name and extension '.y'. To create such a target file, apply the commands ...".

Rules simplify the task of writing a makefile: a file's dependency information and command sequences need be explicitly specified in a makefile only if this information can't be inferred by the application of a rule.

make has a macro capability. A character string can be associated with a macro name; when the macro name is invoked in the makefile, it's replaced by its string.

Preview

The rest of this description of *make* is divided into the following sections:

1. The basics
2. Advanced features
3. Examples

1. The basics

In this section we want to present the basic features of *make*, with which you'll be able to start using *make*. Section 2 describes the other

features of *make*.

Before you can begin using *make*, you must know what *make* does, how to create a simple makefile that contains dependency entries, how to take advantage of *make*'s rule-processing capability, and, finally, how to tell *make* to make a file. Each of these topics is discussed in the following paragraphs.

1.1 What make does

The main function of *make* is to make a target file "current", where a file is considered "current" if the files on which it depends are current and if it was modified more recently than its prerequisite files. To make a file current, *make* makes the prerequisite files current; then, if the target file is not current, *make* executes the commands associated with the file, which usually recreates the file.

As you can see, *make* is inherently recursive: making a file current involves making each of its prerequisite files current; making these files current involves making each of their prerequisite files current; and so on.

make is very efficient: it only creates or recreates files that aren't current. If a file on which a target file depends is current, *make* leaves it alone. If the target file itself is current, *make* will announce the fact and halt without modifying the target.

> **It is important to have the time and date set for *make* to behave properly, since *make* uses the 'last modified' times that are recorded in files' directory entries to decide if a target file is not current.**

1.2 The makefile

When *make* starts, one of the first things it does is to read a file, which you must write, called the 'makefile'. This file contains dependency entries defining inter-file dependencies and the commands that must be executed to make a file current. It also contains rule definitions and macro definitions.

In the following paragraphs, we want to just describe dependency entries. In section 2 we discuss the somewhat more advanced topics of rule and macro definition.

A dependency entry in a makefile defines one or more target files, the files on which the targets depend, and the operating system commands that are to be executed when any of the targets is not current. The first line of the entry specifies the target files and the files on which they depend; the line begins with the target file names, followed by a colon, followed by one or more spaces or tabs, followed by the names of the prerequisite files. It's important to place spaces or tabs after the colon that separates target and dependent files; on systems that allow colons in file names, this allows *make* to distinguish

between the two uses of the colon character.

The commands are on the following lines of the dependency information entry. The first character of a command line must be a tab; *make* assumes that the command lines end with the last line not beginning with a tab.

For example,consider the following dependency entry:

```
prog: prog.o sub1.o sub2.o
        ln -o prog prog.o sub1.o sub2.o -lc
```

This entry says that the file *prog* depends on the files *prog.o*, *sub1.o*, and *sub2.o*. It also says that if *prog* is not current, *make* should execute the *ln* command. *make* considers *prog* to be current if it exists and if it has been modified more recently than *prog.o*, *sub1.o*, and *sub2.o*.

The above entry describes only the dependence of *prog* on *prog.o*, *sub1.o*, and *sub2.o*. It doesn't define the files on which the '.o' files depend. For that, we need either additional dependency entries in the makefile or a rule that can be applied to create '.o' files from '.c' files.

For now, we'll add dependency entries in the makefile for *prog.o*, *sub1.o*, and *sub2.o*, which will define the files on which the object modules depend and the commands to be executed when an object module is not current. In section 1.3 we'll then modify the makefile to make use of *make*'s built-in rule for creating a '.o' file from a '.c' file.

Suppose that the '.o' files are created from the C source files *prog.c*, *sub1.c*, and *sub2.c*; that *sub1.c* and *sub2.c* contain a statement to include the file *defs.h* and that *prog.c* doesn't contain any #include statements. Then the following long-winded makefile could be used to explicitly define all the information needed to make *prog*

```
prog: prog.o sub1.o sub2.o
        ln -o prog prog.o sub1.o sub2.o -lc

prog.o: prog.c
        cc prog.c

sub1.o: sub1.c defs.h
        cc sub1.c

sub2.o: sub2.c defs.h
        cc sub2.c
```

This makefile contains four dependency entries: for *prog*, *prog.o*, *sub1.o*, and *sub2.o*. Each entry defines the files on which its target file depends and the commands to be executed when its target isn't current. The order of the dependency entries in the makefile is not important.

We can use this makefile to make any of the four target files defined in it. If none of the target files exists, then entering

```
make prog
```

will cause *make* to compile and assemble all three object modules from their C source files, and then create *prog* by linking the object modules together.

Suppose that you create *prog* and then modify *sub1.c*. Then telling *make* to make *prog* will cause *make* to compile and assemble just *sub1.c*, and then recreate *prog*.

If you then modify *defs.h*, and then tell *make* to make *prog*, *make* will compile and assemble *sub1.c* and *sub2.c*, and then recreate *prog*.

You can tell *make* to make any file defined as a target in a dependency entry. Thus, if you want to make *sub2.o* current, you could enter

```
make sub2.o
```

A makefile can contain dependency entries for unrelated files. For example, the following dependency entries can be added to the above makefile:

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

hello.o: hello.c
        cc hello.c
```

With these dependency entries, you can tell *make* to make *hello* and *hello.o*, in addition to *prog* and its object files.

1.3 Rules

You can see that the makefile describing a program built from many .o files would be huge if it had to explicitly state that each .o file depends on its .c source file and is made current by compiling its source file.

This is where rules are useful. When a rule can be applied to a file that *make* has been told to make or that is a direct or indirect prerequisite of it, the rule allows *make* to infer, without being explicitly told, the name of a file on which the target file depends and/or the commands that must be executed to make it current. This in turn allows makefiles to be very compact, just specifying information that *make* can't infer by the application of a rule.

Some rules are built into *make*; you can define others in a makefile. In the rest of this section, we're going to describe the properties of rules and how you write makefiles that make use of *make*'s built-in rule for creating a .o file from a .c file. For more information on rules, including a complete list of built-in rules and how to define rules in a makefile, see section 2.2.

1.3.1 make's use of rules

A rule specifies a target extension, source extension, and sequence of commands. Given a file that *make* wants to make, it searches the rules known to it for one that meets the following conditions:

* The rule's target extension is the same as the file's extension;
* A file exists that has the same basic name as the file *make* is working on and that has the rule's source extension.

If a rule is found that meets these conditions, *make* applies the first such rule to the file it's working on, as follows:

* The file having the source extension is defined to be a prerequisite of the file with the target extension;
* If the file having the target extension doesn't have a command sequence associated with it, the rule's commands are defined to be the ones that will make the file current.

One rule built into *make*, for converting .c files into .o files, says

> "a file having extension '.o' depends on the file having the same basic name, with extension '.c'. To make current such a .o file, execute the command
>
> cc x.c
>
> where 'x' is the name of the file"

Another built-in rule exists for converting .asm files into .o files, using the Manx assembler.

1.3.2 An example

The .c to .o rule allows us to abbreviate the long-winded makefile given in section 1.2 as follows:

```
prog: prog.o sub1.o sub2.o
        ln -o prog prog.o sub1.o sub2.o -lc

sub1.o sub2.o: defs.h
```

In this abbreviated makefile, a dependency entry for *prog.o* isn't needed; using the built-in '.c to .o' rule, *make* infers that the *prog.o* depends on *prog.c* and that the command *cc prog.c* will make *prog.o* current.

The abbreviated makefile says that both *sub1.o* and *sub2.o* depend on *defs.h*. It doesn't say that they also depend on *sub1.c* and *sub2.c*, respectively, or that the compiler must be run to make them current; *make* infers this information from the .c to .o rule. The only information given in the dependency entry is that which *make* couldn't infer by itself: that the two object files depend on *defs.h*.

1.3.3 Interaction of rules and dependency entries

As we showed in the above example, a rule allows you to leave some dependency information unspecified in a makefile. The *prog.o* entry in the long-winded makefile of section 1.2 was not needed, since its information could be inferred by the .c to .o rule. And the dependence of *sub1.o* and *sub2.o* on their respective C source files, and the commands needed to create the object files was also not needed, since the information could be inferred from the .c to .o rule.

There are occasions when you don't want a rule to be applied; in this case, information specified in a dependency entry will override that which would be inferred from a rule. For example, the following dependency entry in a makefile

```
add.o:
    cc -DFLOAT add.c
```

will cause *add.o* to be compiled using the specified command rather than the command specified by the .c to .o rule. *make* still infers the dependence of *add.o* on *add.c*, using the .c to .o rule, however.

2. Advanced features

In the last section we presented the basic features of *make*, with which you can start using *make*. In this section, we present the rest of *make*'s features.

2.1 Dependent Files

A dependent file can be in a different volume or directory than its target file, with the following provisos.

If the file name contains a colon (for example, because the file name defines the volume on which the file is located), the colon must be followed by characters other than spaces or tabs, so that *make* can distinguish between this use of the colon character and its use as a separator between the target and dependent files in a dependency line. This shouldn't be a problem, since most systems don't allow file names to contain spaces or tabs.

All references to a file must use the same name. For example, if a file is referred to in one place using the name

```
/root/src/foo.c
```

then all references to the file must use this exact same name.

2.2 Macros

make has a simple macro capability that allows character strings to be associated with a macro name and to be represented in the makefile by the name. In the following paragraphs, we're first going to describe how to use macros within a makefile, then how they are defined, and

finally some special features of macros.

2.2.1 Using macros

Within a makefile, a macro is invoked by preceding its name with a dollar sign; macro names longer than one character must be parenthesized. For example, the following are valid macro invocations:

```
$(CFLAGS)
$2
$(X)
$X
```

The last two invocations are identical.

When *make* encounters a macro invocation in a dependency line or command line of a makefile, it replaces it with the character string associated with the macro. For example, suppose that the macro OBJECTS is associated with the string *a.o b.o c.o d.o*. Then the dependency entries:

```
prog: prog.o a.o b.o c.o d.o
        ln prog.o a.o b.o c.o d.o

a.o b.o c.o d.o: defs.h
```

within a makefile could be abbreviated as:

```
prog: prog.o $(OBJECTS)
    ln prog.o $(OBJECTS)

$(OBJECTS): defs.h
```

There are three special macros: $$, $*, and $@. $$ represents the dollar sign. The other two are discussed below.

2.2.2 Defining macros in a makefile

A macro is defined in a makefile by a line consisting of the macro name, followed by the character '=', followed by the character string to be associated with the macro.

For example, the macro OBJECTS, used above, could be defined in the makefile by the line

```
OBJECTS = a.o b.o c.o d.o
```

A makefile can contain any number of macro definition entries. A macro definition must appear in the makefile before the lines in which it is used.

2.2.3 Defining macros in a command line

A macro can be defined in the command line that starts *make*. The syntax for a command line definition has the following form:

mac=str

where *mac* is the name of the macro, and *str* is its value.

If *str* contains spaces or tabs, the entire argument must be surrounded by quotes.

For example, the following command assigns the value *-DFLOAT* to the macro *CFLAGS*:

make CFLAGS=-DNOFLOAT

The assignment of a value to a macro in a command line overrides an assignment in a makefile statement.

2.2.4 Macros used by built-in rules

make has two macros, CFLAGS and AFLAGS, that are used by the built-in rules. These macros by default are assigned the null string. This can be overriden by a macro definition entry in the makefile.

For example, the following would cause CFLAGS to be assigned the string "-T":

CFLAGS = -T

These macros are discussed below in the description of builtin rules.

2.2.5 Special macros

Before issuing any command, two special macros are set: $@ is assigned the full name of the target file to be made, and $* is the name of the target file, without its extension. Unlike other macros, these can only be used in command lines, not in dependency lines.

For example, suppose that the files *x.c*, *y.c*, and *z.c* need to be compiled using the option "-DFLOAT". The following dependency entry could be used:

```
x.o y.o z.o:
    cc -DFLOAT $*.c
```

When *make* decides that *x.o* needs to be recreated from *x.c*, it will assign $* the string "x", and the command

cc -DFLOAT x.c

will be executed. Similarly, when *y.o* or *z.o* is made, the command *cc -DFLOAT y.c* or *cc -DFLOAT z.c* will be executed.

The special macros can also be used in command lines associated with rules. In fact, the $@ macro is primarily used by rules. We'll discuss this more in the description of rules, below.

2.3 Rules

In section 1, we presented the basic features of rules: what they are and how they are used. We also noted that rules could be defined in the makefile and that some rules are built into *make*. In the following paragraphs, we describe how rules are defined in a makefile and list the built-in rules.

2.3.1 Rule definition

A rule consists of a source extension, target extension, and command list. In a makefile, an entry defining a rule consists of a line defining the two extensions, followed by lines containing the commands.

The line defining the extensions consists of the source extension, immediately followed by the target extension, followed by a colon.

All command lines associated with a rule must begin with a tab character. The first line following the extension line that doesn't begin with a tab terminates the commands for the rule.

For example, the following rule defines how to create a file having extension *.rel* from one having extension *.c*:

```
.c.rel:
    cc -o $@ $*.c
```

The first line declares that the rule's source and target extension are *.c* and *.rel*, respectively.

The second line, which must begin with a tab, is the command to be executed when a *.rel* file is to be created using the rule.

Note the existence of the special macros $@ and $* in the command line. Before the command is executed to create a *.rel* target file using the rule, the macro $@ is replaced by the full name of the target file, and the macro $* by the name of the target, less its extension.

Thus, if *make* decides that the file *x.rel* needs to be created using this rule, it will issue the command

```
cc -o x.rel x.c
```

If a rule defined in a makefile has the same source and target extensions as a built-in rule, the commands associated with the makefile version of the rule replace those of the built- in version. For example, the built-in rule for creating a *.o* file from a *.c* file looks like this:

```
.c.o:
    cc $(CFLAGS) $*.c
```

If you want the rule to generate an assembly language listing, include the following rule in your makefile:

```
.c.o:
    cc $(CFLAGS) -a $*.c
    as -ZAP -l $*.asm
```

2.3.2 Built-in rules

The following rules are built into *make*. The order of the rules is important, since *make* searches the list beginning with the first one, and applies the first applicable rule that it finds.

```
.c.o:
    cc $(CFLAGS) -o $@ $*.c

.asm.o:
    as $(AFLAGS) -o $@ $*.asm
```

The two macros CFLAGS and AFLAGS that are used in the built-in rules are built into *make*, having the null character string as their values. To have *make* use other options when applying one of the built-in rules, you can define the macro in the makefile.

For example, if you want the options -T and -DDEBUG to be used when *make* applies the *.c.o* rule, you can include the line

```
CFLAGS = -T -DDEBUG
```

in the makefile. Another way to accomplish the same result is to redefine the *.c.o* rule in the makefile; this, however, would use more lines in the makefile than the macro redefinition.

2.4 Commands

In this section we want to discuss the execution of operating system commands by *make*.

2.4.1 Allowed commands

A command line in a dependency entry or rule within a makefile can specify any command that you can enter at the keyboard. This includes batch commands, commands built into the operating system, and commands that cause a program to be loaded and executed from a disk file.

2.4.2 Logging commands and aborting make

Normally, before *make* executes a command, it writes the command to its standard output device; and when the command terminates, *make* halts if the command's return code was non-zero. Either or both of these actions can be suppressed for a command, by preceding the command in the makefile with a special character:

@ Tells *make* not to log the command;

- Tells *make* to ignore the command's return code.

For example, consider the following dependency entry in a makefile:

```
prog: a.o b.o c.o d.o
    ln -o prog a.o b.o c.o d.o -lc
    @echo all done
```

When the *echo* command is executed, the command itself won't be logged to the console.

2.4.3 Long command lines

Makefile commands that start a Manx program, such as *cc*, *as*, or *ln*, or that start a program created with *cc*, *as*, *ln*, and *c.lib*, can specify a command line containing up to 2048 characters.

For example, if a program depends on fifty modules, you could associate them with the macro OBJECTS in the makefile, and also include the dependency entry

```
prog: $(OBJECTS)
    ln -o prog $(OBJECTS) -lc
```

This will result in a very long command line being passed to *ln*.

In the next section we will describe how OBJECTS could be defined.

2.5 Makefile syntax

We've already presented most of the syntax of a makefile; that is, how to define rules, macros, and dependencies. In this section we want to present two features of the makefile syntax not presented elsewhere: comments and line continuation.

2.5.1 Comments

make assumes that any line in a makefile whose first character is '#' is a comment, and ignores it. For example:

```
#
# the following rule generates an 8080 object module
# from a C source file:
#
.c.o80:
    cc80 -o cc.tmp $*.c
    as80 -ZAP -o $*.o80 cc.tmp
```

2.5.2 Line continuation

Many of the items in a makefile must be on a single line: a macro definition, the file dependency information in a dependency entry, and a command that *make* is to execute each must be on a single line.

You can tell *make* that several makefile lines should be considered to be a single line by terminating each of the lines, except the last, with the backslash character, '\'. When *make* sees this, it replaces the current line's backslash and newline, and the next line's leading blanks and tabs by a single blank, thus effectively joining the lines together.

The maximum length of a makefile line after joining continued lines is 2048 characters.

For example, the following macro definition equates OBJ to a string consisting of all the specified object module names.

```
OBJ = printf.o fprintf.o format.o\
    scanf.o fscanf.o scan.o\
    getchar.o getc.o
```

As another example, the following dependency entry defines the dependence of *driver.lib* on several object modules, and specifies the command for making *driver.lib*:

```
driver.lib: driver.o printer.o \
    in.o \
    out.o
        libutil -o  driver.lib driver.o\
          printer.o \
          in.o out.o
```

This second example could have been more cleanly expressed using a macro:

```
DRIVOBJ= driver.o printer.o\
    in.o out.o

driver.lib: $(DRIVOBJ)
    libutil -o  driver.lib $(DRIVOBJ)
```

We did it as we did to show that dependency lines and command lines can be continued, too.

2.6 Starting make

You've already seen how *make* is told to make a single file. Entering

```
make filename
```

makes the file named *filename*, which must be described by a dependency entry in the makefile. And entering

```
make
```

makes the first file listed as a target file in the first dependency entry in the makefile.

In both of these cases, *make* assumes the makefile is named 'makefile' and that it's in the current directory on the default drive.

In this section we want to describe the other features available when starting *make*.

2.6.1 The command line

The complete syntax of the command line that starts *make* is:

make [-n] [-f makefile] [-a] [-dmacro=str] [file1] [file2] ...

Square brackets indicate that the enclosed parameter is optional.

The parameters *file1*, *file2* ... are the names of the files to be made. Each file must be described in a dependency entry in the makefile. They are made in the order listed on the command line.

The other command line parameters are options, and can be entered in upper or lower case. Their meanings are:

-n Suppresses command execution. *make* logs the commands it would execute to its standard output device, but doesn't execute them.

-f makefile Specifies the name of the makefile

-a Forces *make* to make all files upon which the specified target files directly or indirectly depend, and to make the target files, even those that it considers current.

-dMACRO=str Creates a macro named MACRO, and assigns *str* as its value.

2.6.2 make's standard output

make only uses its standard output device for printing error messages and, when *make* is started with the -N option, for logging commands. You can redirect *make*'s standard output device in the normal way.

When *make* is started without the -N option (that is, when you really want *make* to make something), commands are always logged to the console; you can't redirect them to another file or device.

The standard input and output devices of a program started by *make* are associated with the console, unless the command that started the program explicitly redirected one or both of them.

2.7 Executing commands

Throughout this document, we've implied that when *make* decides that a command needs to be executed, it executes it itself. Actually, *make* just builds an exec file of all the commands and transfers control to it. *make* itself doesn't execute any commands. When *make* decides that a command needs to be executed, it executes it immediately, and waits for the command to finish. It activates a command whose code is contained in a disk file by issuing an *fexec* function call. It activates DOS built-in commands and batch commands by calling the *system* function, which causes a new copy of the command processor to be

loaded. Thus, to use *make*, your system must have enough memory for DOS, *make*, and whatever programs are loaded by *make* to be in memory simultaneously.

2.8 Differences between the Manx and UNIX 'make' programs

The Manx *make* supports a subset of the features of the UNIX *make*. The following comments present features of the UNIX *make* that aren't supported by the Manx *make*.

* The UNIX *make* will let you make a file that isn't defined as a target in a makefile dependency entry, so long as a rule can be applied to create it. The Manx *make* doesn't allow this. For example, if you want to create the file *hello.o* from the file *hello.c* you could say, on UNIX

  make hello.o

  even if *hello.o* wasn't defined to be a target in a makefile dependency entry. With the Manx *make*, you would have to have a dependency entry in a makefile that defines *hello.o* as a target.

* The UNIX *make* supports the following options, which aren't supported by the Manx *make*:

  p, i, k, s, r, b, e, m, t, d, q

  The Manx *make* supports the option '-a', which isn't supported by the UNIX *make*.

* The special names .DEFAULT, .PRECIOUS, .SILENT, and .IGNORE are supported only by the UNIX *make*.

* Only the UNIX *make* allows the makefile to be read from *make*'s standard input.

* Only the UNIX *make* supports the special macros $<, $?, and $%, and allows an upper case D or F to be appended to the special macros, (which thus modifies the meaning of the macro).

* Only the UNIX *make* requires that the suffixes for additional rules be defined in a .SUFFIXES statement.

* Only the UNIX *make* allows macros to be defined on the command line that activates *make*.

* Only the UNIX *make* allows a target to depend on a member of a library or archive.

3. Examples

3.1 First example

This example shows a makefile for making several programs. Note the entry for *arc*. This doesn't result in the generation of a file called *arc*; it's just used so that we can generate *arcv* and *mkarcv* by entering

make arc.

```
#
# rules:
#
.c.o80:
        cc80 -DTINY -o $@ $*.c
#
# macros:
#
OBJ=make.o parse.o scandir.o dumptree.o rules.o command.o
#
# dependency entry for making make:
#

make: $(OBJ) cntlc.o envcopy.o
        ln -o make $(OBJ) envcopy.o cntlc.o -lc
#
# dependency entries for making arcv & mkarcv:
#

arc: mkarcv arcv
        @echo done

mkarcv: mkarcv.o
        ln -o mkarcv mkarcv.o -lc
arcv : arcv.o
        ln -o arcv  arcv.o -lc
#
# dependency entries for making CP/M-80 versions of arcv & mkarcv:
#
mkarcv80.com: mkarcv.o80
        ln80 -o mkarcv80.com mkarcv.o80 -lt -lc
arcv80.com: arcv.o80
        ln80 -o arcv80.com arcv.o80 -lt -lc

$(OBJ): libc.h make.h
```

3.2 Second example

This example uses *make* to make a library, my.lib. Three directories are involved: the directory *libc* and two of its subdirectories, *sys* and *misc*. The C and assembly language source files are in the two subdirectories. There are makefiles in each of the three directories, and this example makes use of all of them. With the current directory being *libc*, you enter

```
make my.lib
```

This starts *make*, which reads the makefile in the libc directory. *make* will change the current directory to *sys* and then start another *make* program.

This second *make* compiles and assembles all the source files in the *sys* directory, using the makefile that's in the *sys* directory.

When the 'sys' *make* finishes, the 'libc' *make* regains control, and then starts yet another *make*, which compiles and assembles all the source files in the *misc* subdirectory, using the makefile that's in the *misc* directory.

When the 'misc' *make* is done, the 'libc' *make* regains control and builds *my.lib*. You can then remove the object files in the subdirectories by entering

```
make clean
```

3.2.1 The makefile in the 'libc' directory

```
my.lib: sys.mk misc.mk
        rm my.lib
        libutil -o my.lib -f my.bld
        @echo my.lib done

sys.mk:
        cd sys
        make
        cd ..

misc.mk:
        cd misc
        make
        cd ..

clean:
        cd sys
        make clean
        cd ..
        cd misc
        make clean
        cd ..
```

3.2.2 Makefile for the 'sys' directory

```
REL=asctime.o bdos.o begin.o chmod.o croot.o csread.o ctime.o \
    dostime.o dup.o exec.o execl.o execlp.o execv.o execvp.o \
    fexec.o fexecl.o fexecv.o ftime.o getcwd.o getenv.o \
    isatty.o localtim.o mkdir.o open.o stat.o system.o time.o\
    utime.o wait.o dioctl.o ttyio.o access.o syserr.o

COPT=
HEADER=../header

.c.o:
        cc $(COPT) -I$(HEADER) $*.c -o $@
        sqz $@
.asm.o:
        as $*.asm -o $@
        sqz $@

all: $(REL)
        @echo sys done

clean:
        rm  *.o
```

3.2.3 Makefile for the 'misc' directory

```
REL=atoi.o atol.o calloc.o ctype.o format.o malloc.o qsort.o \
        sprintf.o sscanf.o fformat.o fscan.o

COPT=
HEADER=../header

.c.o:
        cc $(COPT) -I$(HEADER) $*.c -o $@
        sqz $@
.asm.o:
        as $*.asm -o $@
        sqz $@

all: $(REL)
        @echo misc done

fformat.o: format.c
        cc -I$(HEADER) -DFLOAT format.c -o fformat.o

fscan.o: scan.c
        cc -I$(HEADER) -DFLOAT scan.c -o fscan.o

clean:
        rm  *.o
```

NAME

mount, umount

SYNOPSIS

mount

umount volume

DESCRIPTION

mount displays information about the volumes having entries in the mounted volume table. *umount* removes an entry from this table.

This section contains detailed information about these two commands. For an overview of multivolume development, the mounted volume table, and so on, see the SHELL reference chapter.

The information displayed by *mount* for an entry in the mounted volume table has the form:

*d (-v) name x K used y K free # files

where:

*	Indicates that the volume contains the current directory. For other volumes, a space appears instead of '*';
d	The number of the drive containing the volume: 1 is the internal drive 2 is the external drive - means the volume isn't in a drive
v	The number of the volume's entry in the mounted volume table.
x	The amount of space, in kilobytes, occupied on the disk by files;
y	The amount of free space, in kilobytes, on the volume.
#	Is the number of files on the volume.

The *volume* argument to the *mount* command identifies the volume to be removed from the mounted volume table. It can be either the name of the volume, or the number of the drive containing it.

The code for these commands is contained in the SHELL.

EXAMPLES

umount data:

Remove the *data:* volume from the mounted volume table.

umount 2:

Remove the volume in the external drive from the mounted volume table.

umount 'another disk:'

Remove the volume named "another disk:" from the mounted volume table.

NAME

mv - move files

SYNOPSIS

mv [-f] infile outfile

mv [-f] file1 file2 ... directory

mv vol1: vol2:

DESCRIPTION

The first two forms of *mv* moves one or more files, and the original files then cease to exist. It moves any type file, and will move both the file's resource and data forks.

The first form of *mv* moves a single file, *infile*, to *outfile.*

The second format of *mv* copies *file1, file2, ...* to *directory.* The names of the files in the new directory have the same names as the original files. In this case, the name of the directory must be terminated with the character '/', unless the name is '.' or '..'.

With both forms, if an original and target file are on the same volume, *mv* simply changes the name of the original file to that of the target file.

If the two files are on different volumes, *mv* actually copies each original file to the target file and then removes the original file.

If the '-f' option is used, a pre-existing target file will be removed before the move occurs. If *-f* isn't specified and if a destination file already exists, *mv* will ask if you want to overwrite it.

The third form of *mv* changes the name of a volume. For this, a drive identifier (eg, 1:) can't be used.

The code for this command is contained in the SHELL.

EXAMPLES

mv hello.c test.c

Renames *hello.c* in the current directory to *test.c.*

mv hello.c ../test.c

Renames *hello.c* in the current directory to the name *test.c.* The new file is located in the parent directory of the current directory.

mv 1:/source/hello.c 2:/work/test.c

Copies *hello.c* from the directory */source* on drive 1: to the file *test.c* in the directory */work* on drive 2: and then removes the original file.

mv * /newdir/

Renames all files in the current directory so that they are in the directory */newdir* on the current volume. Note the terminating '/' on the name of the target directory.

mv sys:/work/*.c .

Assuming that the *sys:* and current volumes are different, this command copies all files in the */work* directory on the *sys:* volume having extension '.c' to the current directory. If the two volumes are the same, the original files are just renamed.

mv sys: newname:

This command changes the name of the mounted volume *sys:* to *newname:*.

NAME

prsetup - initialize printer

SYNOPSIS

prsetup [tabwidth]

DESCRIPTION

Initializes the printer so that output redirected from the SHELL will print correctly.

It sets tabs at the specified stops and tells the printer to output a line feed after a carriage return.

If tabwidth isn't specified, tab stops are set every four characters.

NAME

pwd - "print working directory"

SYNOPSIS

pwd

DESCRIPTION

pwd prints the name of the current directory.

The name is written to pwd's standard output device. Hence, the name is printed on the screen, by default, and can be redirected to another device or file, if desired.

The code for *pwd* is contained in the SHELL.

SEE ALSO

cd

NAME

MountRam - Ram Disk Utility

SYNOPSIS

MountRam

DESCRIPTION

MountRam causes some of the RAM memory on a 512K Macintosh to be reserved for use as a Ram Disk. Once *MountRam* has been run, it appears that there is a new drive on your Macintosh, whose number is 5 and whose name is *Ram:*. This drive behaves like a normal drive, except that (1) data transfer to it is faster than to a regular drive, and (2) the contents of the drive are destroyed when the Macintosh is turned off or rebooted.

Mountram leaves 128K bytes of RAM available for use as an application heap, and uses the rest of RAM for the RAM disk.

The file containing the *Mountram* program also contains a driver resource. This resource is called when a program attempts to access the ram disk. It is loaded into the system heap by *MountRam* and made non-purgable.

When a program terminates, the Macintosh normally searches the boot drive for the Finder or SHELL. Executing the SHELL from the RAM disk automatically makes the RAM disk the boot drive.

EXAMPLES

After the ram disk is created, you can copy files onto it. It's best to use it for holding programs, libraries, header files, and temporary files. This way, if the system crashes, you won't lose any files that aren't contained on regular disks.

The following exec file illustrates how a ram disk might be set up for development:

```
MountRam
cp /shell ram:
cp /bin/cc ram:bin/
cp /bin/as ram:bin/
cp /bin/ln ram:bin/
cp /bin/z ram:bin/
cp /include/* ram:include/
cp /lib/c.lib ram:lib/
set CLIB=ram:lib/
set INCLUDE=ram:include
set PATH=ram:bin;;sys:bin
ram:shell
```

This file first creates the ram disk. Then, it copies the SHELL, compiler, assembler, linker, z, all files in the *include* directory, and *c.lib* onto the ram disk. The next three lines set the environment variables to use the ram disk. Finally, the SHELL on the ram disk is executed, which changes the boot drive to the new ram disk.

The following exec file compiles, assembles, and links a program, using the ram disk to hold the assembly language source and the object module:

```
cc -o ram:temp.o $1.c
ln -o $1 ram:$1.o -lc
rm ram:$1.o
```

If you have a hard disk which already uses drive 5 as its number, edit the assembly language source and change the line "DRVNUM equ 5" to an appropriate number.

SEE ALSO

The Technical Information chapter has a section that discusses using Aztec C68K on a 512K Macintosh.

NAME

RGen - Resource generator

SYNOPSIS

RGen -f infile

DESCRIPTION

RGen is a resource compiler. It reads a text file that describes resources, generates the resources, and writes them to a file. If an error occurs, *RGen* will halt after first displaying an error message and the input line on which the error occurred.

RGen is very similar to the resource compiler in the Lisa Workshop described in *Inside Macintosh*. There are some differences in syntax, so care should be taken if converting files from the Lisa.

RGen doesn't support the following resource types:

CODE FWID
DRVR INIT
DSAT INTL
FRSV PREC
FONT

1. The input and output files

The *infile* parameter on the line that starts *RGen* is the name of the text file that defines the resources. The extension of this file is usually *.r*.

The first line of *infile* contains the name of the file to which *RGen* is to write resources. Normally, *RGen* creates a new output file, after first erasing an existing file of the same name, if necessary. However, if the first character of this line is an exclamation point (!), *RGen* will append resources to an existing file, and will create a new file only if the specified file doesn't already exist.

When *RGen* is being run on a Macintosh, the second line of *infile* defines the type and creator of the output file. The type is listed first on the line, followed by the creator. The type and creator names each have a maximum of four characters. If the type has less than four characters, a space must separate it and the creator names. If the type has exactly four characters, the creator name immediately follows the type name, with no intervening spaces. If the line is blank, both fields will be set to zero.

When *Rgen* is being run on a system other than a Macintosh (ie, it was supplied with a cross development version of C68), the second line of *infile* can be anything, because these versions of *RGen* simply read the line and ignore it. It must, however, be present, since *RGen* assumes that resource definitions begin on the third line.

Normally, when *RGen* encounters the definition of a resource that already exists in the output file, it will ask if you want to overwrite the existing resource. Alternatively, you can specify the *-f* option in the command that starts *RGen*, causing *RGen* to automatically overwrite existing resources without asking for your permission.

For example, if the following lines are the first two lines of the *RGen* input file, *RGen* will create a new file named *appres.rsc* in the current directory, with type *RSRC* and creator *TEMP*:

```
appres.rsc
RSRCTEMP
```

And if an exclamation mark began the first of the above lines (that is, the line was *!appres.rsc*), *RGen* would append resources to *appres.rsc*, without first deleting the file.

When *RGen* is executed on a Macintosh, it writes the resources to the resource fork of the output file.

When *RGen* is executed on another system (that is, it was provided as part of a cross development package, with the other system acting as the host and the Macintosh as the target), it writes the resources to a normal file; also, in this case the second line of the input file to *RGEN* has no effect, although it must be present. When the resulting file is transferred to the Macintosh using the *xfer* program, the *-p* option must be specified, causing the data to be written to the resource fork of a file on the Macintosh. *xfer* sets the type and creator of this file to *AZTC* and *Manx*, respectively. You can change the type and creator using the SHELL's *styp* command, if necessary.

2. Input File Syntax

In this section we want to describe the format of the input file lines that define resources. For a discussion of the first two lines of the input file, which define the output file, see above.

2.1 General Description of the Input File

Most blank lines and all comment lines are ignored. Some blank lines are required as separators. Comment lines are lines that begin with an asterisk. Comments at the ends of regular input lines are initiated by two consecutive semicolons (;;).

Blanks are generally ignored, except when used as a separator for different values on a line, or in strings. When a resource definition calls for several values on a line, they must all be on the same line. Numbers are interpreted as decimal, unless a particular instance is noted as otherwise.

Two special symbols may be used in the resource definitions. The continuation symbol, (++), is placed at the end of a line that is continued on the next line. This is usually used with long strings. The

second symbol, (\), specifies that the following two hexadecimal digits be interpreted as an ASCII character.

2.2 Resource Definitions

The general form of a resource definition is as follows:

```
TYPE type [= other type]
[name],ID [(attribute)]
type-specific data
```

The items in square brackets are optional. Note that both the resource name and resource attributes are optional, but the comma and resource ID are not. ID numbers should be unique within a resource type. The *type* field must be one of the predefined resource types that the compiler knows about. A list of those types is given below. New types can be defined from existing types by using the alternate form of the TYPE statement. The type-specific data which follows the TYPE and ID statements is described for each of the predefined types below.

RGen has 26 predefined resource types that it knows about. They are:

ALRT	ICON	PAT#
BNDL	ICN#	PICT
CDEF	KEYC	PROC
CNTL	GNRL	TEXT
CURS	MBAR	STR
DITL	MDEF	STR#
DLOG	MENU	WDEF
FKEY	PACK	WIND
FREF	PAT	

The type-specific data for each type is described below by example. For further information, refer to the appropriate section in the *Inside Macintosh* manual. Note that FKEY, KEYC, MDEF, PACK, and WDEF all have the same format as CDEF.

NOTE: Hex bytes are taken 2 at a time, hex words are taken 4 at a time. If there is an odd number, the number is scanned from right to left and a 0 is inserted in the beginning. Hex values may be in either upper or lower case.

Examples:

1. The values adAD, ADad, aDaD, ... are all the same.

2. Examples of various hex values:

Hex word: f0f1 is taken as F0F1
123 is taken as 0123

Hex byte: aD is taken as AD
1 is taken as 01

When keywords are required (i.e. Visible, Invisible, GoAway, NoGoAway) only enough characters are required to distinguish between the choices. These characters may be in either upper or lower case.

Examples:

1. For Visible or invisible, only a 'v' or an 'i' must be typed, the rest of the word is optional.

2. To specify a checkBox, the letters 'ch' are required, the rest of the word is optional in order to distinguish checkBox from ctrlItem ('ct').

3. Examples of Resource Definition

3.1 ALRT - Alert Template

```
TYPE ALRT
 ,128                ;; resource ID
50 50 250 250        ;; top left bottom right
1                    ;; resource ID of item list
7FFF                 ;; stages word in hexadecimal
```

3.2 BNDL - Application Bundle

```
TYPE BNDL
 ,128                ;; resource ID
MPNT 0               ;; bundle owner
ICN#                 ;; resource type
0 128 1 129          ;; local ID, 0 maps to 128, 1 to 129
FREF                 ;; resource type
0 128 1 129          ;; local ID, 0 maps to 128, 1 to 129
```

Note: the number of mappings from local ID to resource ID is variable. Simply include multiple mappings on a single line.

3.3 CDEF - Control definition function

```
TYPE CDEF
Myfile,156          ;; filename, resource ID
```

Note: This reads in the resource and id from the specified file. The specified file may only contain 1 resource. It is placed in the output file with the specified type and ID.

3.4 CURS - Cursor

```
TYPE CURS
,300                ;; resource ID
7FFC . . . 287F     ;; the data: 64 hex digits on one line.
0FC0 . . . 1FF8     ;; the mask: 64 hex digits on one line.
0008 0008           ;; the hotSpot in hexadecimal (v h)
```

3.5 CNTL - Control Template

```
TYPE CNTL
 ,130               ;; resource ID
Stop                ;; title
244 40 260 80       ;; top left bottom right
Invisible           ;; Visible or Invisible
0                   ;; ProcID (control definition ID)
0                   ;; RefCon (reference value)
0 1 0               ;; min, max, value
```

3.6 DITL - Dialog or Alert Item List

```
TYPE DITL
 ,129                    ;; resource ID
9                        ;; number of items in list

staticText               ;; static text dialog item
20 20 32 100             ;; top left bottom right
Name:                    ;; message

editText                 ;; editable text dialog item
20 120 32 200            ;; top left bottom right
your name here           ;; text itself

radioButton              ;; radio button dialog item
40 40 60 150             ;; top left bottom right
Choice                   ;; button label

checkBox Disabled        ;; disabled checkbox dialog item
75 40 95 150             ;; top left bottom right
Filter                   ;; checkbox label

button                   ;; button dialog item
75 160 95 200            ;; top left bottom right
Cancel                   ;; button label

iconItem                 ;; icon dialog item
40 40 60 150             ;; top left bottom right
128                      ;; resource ID

picItem Disabled         ;; disabled picture dialog item
75 75 160 160            ;; top left bottom right
130                      ;; resource ID

userItem                 ;; user dialog item
20 20 60 60              ;; top left bottom right

ctrlItem                 ;; control item
20 20 40 40              ;; top left bottom right
16                       ;; resource ID of control definition
```

Note: The item is assumed to be enabled unless followed by the keyword, Disabled.

3.7 DLOG - Dialog Template

```
TYPE DLOG
 ,3                    ;; resource ID
This is a dialog box   ;; title
100 100 190 250        ;; top left bottom right
Visible GoAway         ;; box status
0                      ;; procID (dialog definition ID)
0                      ;; refCon (reference value)
129                    ;; ID of item list (DITL above)
```

Note: The box status can be Visible or Invisible. The status also indicates whether a the box has a close control by specifying either GoAway or NoGoAway.

3.8 FREF - File Reference

```
TYPE FREF
 ,128                  ;; resource ID
APPL 0 filename        ;; file type, local ID of icon, filename
```

Note: The filename can be omitted if there is none.

3.9 ICON - Icon

```
TYPE ICON
,128                   ;; resource id
0380 0000              ;; The icon in hexadecimal - 64 hex values
...
1EC0 3180
```

3.10 ICN# - Icon list

```
TYPE ICN#
,128                   ;; resource id
2                      ;; Number of icons
0001 0000              ;; The icons in hexadecimal - 64 hex values
...                    ;; for each icon
0002 8000
```

3.11 MBAR - Menu bar

```
TYPE MBAR
,128                   ;; resource ID
3                      ;; Number of menus
128 130 156            ;; resource ID of the menus
```

3.12 MENU - Menu

```
TYPE MENU          ;; standard type
 ,3                ;; resource ID
Edit               ;; menu title
Undo               ;; item 1
(-                 ;; item 2
Copy               ;; item 3
Cut                ;; item 4
Paste              ;; item 5
Clear              ;; item 6
                   ;; MUST be followed by a blank line!!


TYPE MENU          ;; nonstandard type
 ,4                ;; resource ID
201                ;; resource ID of menu definition procedure
Patterns           ;; Menu title (may be followed by items)
```

3.13 PAT - Pattern

```
TYPE PAT
 ,200              ;; resource ID
55DD5566AA11AA66   ;; The pattern in 4 words of hexadecimal
```

3.14 PAT# - Pattern list

```
TYPE PAT#
 ,136              ;; resource ID
2                  ;; Number of patterns
5522552255225522   ;; The patterns in 4 words of hexadecimal, or
55DD5566AA11AA66   ;; per line
```

3.15 PICT - Picture

```
TYPE PICT
 ,130              ;; resource ID
5                  ;; Picture size (number of bytes) in decimal
50 50 300 300      ;; top left bottom right
4142434445         ;; The picture in hexadecimal
```

3.16 PROC - Procedure

```
TYPE PROC
 ,128              ;; resource ID
MyProcedure        ;; file name
```

Note: This type is used to create resources that contain code. It does so by reading the resource of type CODE and ID=1 from the specified file. It then strips the first four bytes off of it and saves it as a

resource of type PROC. This is useful for creating resource types such as WDEF, PACK and INIT.

3.17 STR - String

```
TYPE STR                ;; 'STR ' (space required)
 ,1                     ;; resource ID
This is a string ++     ;; the string itself, saved in Pascal string format
continued on a new line.
```

3.18 STR# - A Number of Strings

```
TYPE STR#
 ,1                     ;; resource ID
4                       ;; number of strings
This is string one      ;; and the strings themselves ...
And string two
Third string
Bench warmer
```

3.19 TEXT - Text

```
TYPE TEXT
,128                    ;; resource ID
16                      ;; Number of bytes in the text
This is the text        ;; The text
```

3.20 WIND - Window Template

```
TYPE WIND
 ,128                   ;; resource ID
Wonder Window           ;; window title
40 80 120 300           ;; top left bottom right
Invisible NoGoAway      ;; window status
0                       ;; procID (window definiton ID)
0                       ;; refCon (reference value)
```

Note: The box status can be Visible or Invisible. The status also indicates whether a the box has a close control by specifying either GoAway or NoGoAway.

4. Creating New Types

Creating types other than the predefined types can be accomplished in one of two ways. First, a new type can be based on an existing type, having the same structure, but a different type value. This is done by equating the new type to the existing type in the TYPE statement part of the definition. For example, to create a resource of type INIT, the following definition would be used.

```
TYPE INIT = PROC    ;; type INIT is just like PROC
 ,3                 ;; resource ID
progfile            ;; file containing CODE resource
```

The file, *progfile*, should be a file created by the linker with no overlays, no initialized data, and no relocatable references.

The second way of creating a new type allows the structure to be completely defined in the definition. For this purpose, equate using the GNRL resource type, which recognizes a set of element designators which can be used to define the structure. The element descriptors are defined as follows:

.P	Pascal string
.S	String without length byte
.I	Decimal integer
.L	Decimal long integer
.H	Hexadecimal value
.R	Load a resource from a file (filename type ID)

The following is a list of examples of new resource types created from the GNRL type.

```
TYPE CHRG = GNRL ;; define type CHRG
 ,200            ;; resource ID
 .I              ;; a decimal integer
57
 .P              ;; a Pascal string
Finance charges
                 ;; Must end with a blank line

TYPE ICN# = GNRL ;; icon list for an application
 ,128            ;; resource ID
 .H              ;; enter 2 icons in hexadecimal
0001 0002 0003 0004  ;; each is 32 bits by 32 bits
  ....
007d 007e 007f 0080  ;; for 128 words total
                 ;; Must end with a blank line

TYPE FONT = GNRL ;; define a new type
 ,200            ;; resource ID
 .R              ;; read from the System file the
System FONT 268  ;; type FONT with ID = 268
                 ;; Must end with a blank line
```

5. Including Resources

Just as resources may be appended to existing files, resources may be read from existing files and included as part of the new resource file. This process is not selective, and uses the entire resource file. This is done using a statement of the form:

INCLUDE filename

Later it will be shown how this statement is used to combine the output of the Aztec linker with resources produced by the resource compiler.

NOTE: Unlike *RMaker*, a given file name with no preceeding pathname is taken to be in the current directory. To get or put a file in another directory, the full pathname must preceed the file name.

6. Using RGen with Aztec C

During program development, it is possible to avoid using the resource compiler each time the program is changed. This is done by placing the resources produced by the resource compiler in a file with a fixed name. Then, in the *main()* routine of the program itself, the *OpenResFile()* routine should be used to open the resource file. This will make the resources available as though they were part of the program file itself.

When the program has been completed, the resource file can be combined with the program file for final use, and the *OpenResFile()* statement can be removed. To combine the output from the *RGen* with the output of LN, edit the *RGen* input file and add a statement:

INCLUDE linker.out

where *linker.out* is the name of the linker output file. The resulting file is the final form of the application being developed.

7. Explanation of RGen error messages

When started, *RGen* first displays a message on the screen which indicates that RGen is running. If everything goes well, RGen will print on the screen several messages listing the output file name, the data, map and total size. RGen may encounter an error while it is running, in which case it will send a message to the screen.

Following is a list of the messages which RGen will generate in response to an error.

1. Cannot open file: 'filename'.

The file does not exist. Make sure that the file name is entered correctly and that the file exists in the directory specified.

2. Cannot read from the input file: 'filename'.

2. Cannot read from the file.

For some reason, the file cannot be read. Check the file to make sure that the name and directory are correct.

3. Cannot create output file.

4. Cannot write the output file.

4. Cannot write to the output file: 'filename'.

Check there is enough room on the disk and that a correct path name and file name are given.

5. Not enough memory to create resource map.

Either the input file's resource map is too big to be read into memory or the output file's resource map is too big to be kept in memory.

6. Invalid dialog item type.

The type is not one of the given dialog item types, such as BtnItem, crtlItem, StatText, ...

7. Missing 'File name!'.

The type FONT requires a file name from which the resource is read in.

8. Missing font size.

The type FONT requires that a font resource has a size.

9. Missing '(' before resource attribute.

A resource attribute must be surrounded by parentheses.

10. Missing ID.

11. Missing comma before ID.

A resource ID must be preceeded by a comma.

12. Unexpectedly reached the end of the file.

The end of the input file occurred when there should have been more data. Check the input file with the RGen manual.

13. Looking for a number.

Something besides the expected number was found.

14. Incorrect keyword.

RGen was looking for the first character of a keyword (i.e. 'v' for visible or 'i' for invisible) and found a character that didn't match.

15. Missing an interpreting code.

The type GNRL requires an interpreting code (i.e. .P .S .I .H .B .R) so that it knows how to interpret the following data.

16. Missing definition data.

For the type GNRL, a .S, .I, .L, .H, or .B was found but no data was found after it.

17. Missing the input file name.

18. Missing the resource name.

19. Missing the ID.

For the type GNRL, a .R was found but either the input file name, resource name, or the ID was missing from the line following it.

20. 'Filename' is not a valid input text file.

The given file is not a valid one. RGen requires a text file.

21. Incorrect resource Type.

RGen requires one of the 26 resource types as listed in the RGen manual.

22. Invalid name.

Expecting the word "Type" or "INCLUDE" but found something else instead.

23. Output file name may not be the input file name.

Specify a different output file name as it may not be the same as the input file name.

24. Only allowed to have 1 resource in input file.

The specified file may only contain 1 resource that has 1 ID.

25. Special character style not found.

A character representing a character style in the menu manager was either not found or incorrect.

26. Illegal character found.

An incorrect meta-character was found in the MENU template.

27. 'Filename' is not of the correct format.

The specified input file is not of correct Macintosh format.

28. String is greater than 255 characters.

A string cannot be greater than 255 characters.

29. Illegal hex character.

Expecting a value between 0 and 9 or A and F but found something else.

30. Resource 'name' id 'number' already exists.

The resource and ID already exist. Check the input (.r) file to make sure that types and ID's are not duplicated. Also check any files that are being INCLUDED'ed.

31. Should be at the end of the line.

32. Should not be another number on this line.

33. Should be the last word on this line.

Another character was found when there should have been nothing else on the line. Check the RGen documentation for the correct template.

34. User terminated.

RGen was aborted by the user.

35. usage: rgen (-f) filename

To run the resource generator, type "rgen filename". The *-f* option will automatically remove a pre-existing resource of the specified TYPE and ID from the destination file, without asking your permission. This should be used if you know a resource already exists and are sure that it should be overwritten.

36. Invalid option specified.

An option other than *-f* was found. See message 35.

37. Missing an output file name.

A file name is needed to write the output to. This should be the first line in the .r file.

NAME

rm - remove files

SYNOPSIS

rm [-i] file [file] ...

DESCRIPTION

rm removes the specified files.

rm will not remove files that have been locked with the *lock* command. It will, however, remove files that have been locked by the Finder or by the SHELL's *flock* command. Also, *rm* will not remove directories.

If the '-i' (interactive) option is used, *rm* will ask the operator whether or not to remove each file. If 'y' is typed, the file will be removed. If anything else is typed, it won't be removed. Without this option, *rm* automatically removes the files, without questioning the operator.

The code for this command is contained within the SHELL.

EXAMPLES

rm *.bak

Remove all files having extension *.bak* from the current directory. The files are removed automatically, without querying the operator, except for files which were locked by the *lock* command.

rm -i data:/temp/*

Remove all files in the */temp* directory on the *data:* volume. For each file, *rm* asks the operator whether the file should be removed.

NAME

RMaker - Resource compiler

SYNOPSIS

RMaker

DESCRIPTION

RMaker is the resource compiler supplied with the Macintosh 68000 Development System(MDS) from Apple Computer. It is very similar to the resource compiler in the Lisa Workshop described in *Inside Macintosh.* There are some differences in syntax, so care should be taken if converting files from the Lisa. The resource compiler takes a text file which describes the individual resources as input and produces the appropriate resource file as output.

1. Input Format

RMaker input is in the form of a text file usually with an extension of ".r". The format of the text is fairly simple. Most blank lines and all comment lines are ignored. Some blank lines are required as separators. Comment lines are lines that begin with an asterisk. Comments at the ends of regular input lines are initiated by two consecutive semicolons (;;).

Blanks are generally ignored, except when used as a separator for different values on a line, or in strings. When a resource definition calls for several values on a line, they must all be on the same line. Numbers are interpreted as decimal, unless a particular instance is noted as otherwise.

Two special symbols may be used in the resource definitions. The continuation symbol, (++), is placed at the end of a line that is continued on the next line. This is usually used with long strings. The second symbol, (\), specifies that the following two hexadecimal digits be interpreted as an ASCII character.

2. File Information

The first true input line of the file is used to specify the name of the resource file to be created. Any name can be specified. The only restriction is that the file should not have an extension of ".rel", since that will cause the resource compiler to generate a file in a special format for the MDS system.

The line following the resource name should be used to specify the type and creator. The type is listed first on the line, followed by the creator. The type and creator names each have a maximum of four characters. If the type has less than four characters, a space must separate it and the creator names. If the type has exactly four characters, the creator name immediately follows the type name, with

no intervening spaces. If the line is blank, both fields will be set to zero.

For example, the following lines

```
appres.rsrc
RSRCTEMP
```

will create a resource file named *appres.rsrc* with type **RSRC** and creator **TEMP**.

The preceding example will create a new resource file. To append the resources to an existing resource file, simply precede the file name with an exclamation point. For example:

```
!oldres.rsrc
```

will append the resources to the file *oldres.rsrc*.

Just as resources may be appended to existing files, resources may be read from existing files and included as part of the new resource file. This process is not selective, and uses the entire resource file. This is done using a statement of the form:

```
INCLUDE filename
```

Later it will be shown how this statement is used to combine the output of the Aztec linker with resources produced by the resource compiler.

3. Resource Definitions

The remaining lines of the input file are the resource definition lines themselves. The general form of a resource definition is as follows:

```
TYPE type [= other type]
[name],ID [(attribute)]
type-specific data
```

The items in square brackets are optional. Note that both the resource name and resource attributes are optional, but the comma and resource ID are not. ID numbers should be unique within a resource type. The *type* field must be one of the predefined resource types that the compiler knows about. A list of those types is given below. New types can be defined from existing types by using the alternate form of the TYPE statement. The type-specific data which follows the TYPE and ID statements is described for each of the predefined types below.

RMaker has 12 predefined resource types that it knows about. They are:

ALRT	DLOG	PROC
BNDL	FREF	STR
CNTL	GNRL	STR#
DITL	MENU	WIND

The type-specific data for each type is described below by example. For further information, refer to the appropriate section in the *Inside Macintosh* manual.

4. Resource Examples

4.1 ALRT - Alert Template

```
TYPE ALRT
 ,128                ;; resource ID
50 50 250 250        ;; top left bottom right
1                    ;; resource ID of item list
7FFF                 ;; stages word in hexadecimal
```

4.2 BNDL - Application Bundle

```
TYPE BNDL
 ,128                ;; resource ID
MPNT 0               ;; bundle owner
ICN#                 ;; resource type
0 128 1 129          ;; local ID, 0 maps to 128, 1 to 129
FREF                 ;; resource type
0 128 1 129          ;; local ID, 0 maps to 128, 1 to 129
```

Note: the number of mappings from local ID to resource ID is variable. Simply include multiple mappings on a single line.

4.3 CNTL - Control Template

```
TYPE CNTL
 ,130                ;; resource ID
Stop                 ;; title
244 40 260 80        ;; top left bottom right
Invisible            ;; Visible or Invisible
0                    ;; ProcID (control definition ID)
0                    ;; RefCon (reference value)
0 1 0                ;; min, max, value
```

4.4 DITL - Dialog or Alert Item List

```
TYPE DITL
 ,129                    ;; resource ID
5                        ;; number of items in list

staticText               ;; static text dialog item
20 20 32 100             ;; top left bottom right
Name:                    ;; message

editText                 ;; editable text dialog item
20 120 32 200            ;; top left bottom right
your name here           ;; text itself

radioButton              ;; radio button dialog item
40 40 60 150             ;; top left bottom right
Choice                   ;; button label

checkBox Disabled        ;; disabled checkbox dialog item
75 40 95 150             ;; top left bottom right
Filter                   ;; checkbox label

button                   ;; button dialog item
75 160 95 200            ;; top left bottom right
Cancel                   ;; button label
```

Note: The five item types listed above are the only ones recognized. The item is assumed to be enabled unless followed by the keyword, Disabled.

4.5 DLOG - Dialog Template

```
TYPE DLOG
 ,3                      ;; resource ID
This is a dialog box     ;; title
100 100 190 250          ;; top left bottom right
Visible GoAway           ;; box status
0                        ;; procID (dialog definition ID)
0                        ;; refCon (reference value)
129                      ;; ID of item list (DITL above)
```

Note: The box status can be Visible or Invisible. The status also indicates whether a the box has a close control by specifying either GoAway or NoGoAway.

4.6 FREF - File Reference

```
TYPE FREF
 ,128                ;; resource ID
APPL 0 filename      ;; file type, local ID of icon, filename
```

Note: The filename can be omitted if there is none.

4.7 MENU - Menu

```
TYPE MENU
 ,3                  ;; resource ID
Edit                 ;; menu title
Undo                 ;; item 1
(-                   ;; item 2
Copy                 ;; item 3
Cut                  ;; item 4
Paste                ;; item 5
Clear                ;; item 6
                     ;; MUST be followed by a blank line!!
```

4.8 PROC - Procedure

```
TYPE PROC
 ,128                ;; resource ID
MyProcedure          ;; file name
```

Note: This type is used to create resources that contain code. It does so by reading the resource of type CODE and ID=1 from the specified file. It then strips the first four bytes off of it and saves it as a resource of type PROC. This is useful for creating resource types such as WDEF, PACK and INIT.

4.9 STR - String

```
TYPE STR             ;; 'STR ' (space required)
 ,1                  ;; resource ID
This is a string ++  ;; the string itself
continued on a new line.
```

4.10 STR# - A Number of Strings

```
TYPE STR#
 ,1                  ;; resource ID
4                    ;; number of strings
This is string one   ;; and the strings themselves ...
And string two
Third string
Bench warmer
```

4.11 WIND - Window Template

```
TYPE WIND
 ,128                ;; resource ID
Wonder Window        ;; window title
40 80 120 300        ;; top left bottom right
Invisible NoGoAway   ;; window status
0                    ;; procID (window definiton ID)
0                    ;; refCon (reference value)
```

Note: The box status can be Visible or Invisible. The status also indicates whether a the box has a close control by specifying either GoAway or NoGoAway.

5. Creating New Types

Creating types other than the predefined types can be accomplished in one of two ways. First, a new type can be based on an existing type, having the same structure, but a different type value. This is done by equating the new type to the existing type in the TYPE statement part of the definition. For example, to create a resource of type INIT, the following definition would be used.

```
TYPE INIT = PROC    ;; type INIT is just like PROC
 ,3                 ;; resource ID
progfile            ;; file containing CODE resource
```

The file, *progfile*, should be a file created by the linker with no overlays, no initialized data, and no relocatable references.

The second way of creating a new type allows the structure to be completely defined in the definition. For this purpose, equate using the GNRL resource type, which recognizes a set of element designators which can be used to define the structure. The element descriptors are defined as follows:

.P	Pascal string
.S	String without length byte
.I	Decimal integer
.L	Decimal long integer
.H	Hexadecimal value
.R	Load a resource from a file (filename type ID)

The following is a list of examples of new resource types created from the **GNRL** type.

```
TYPE CHRG = GNRL ;; define type CHRG
 ,200            ;; resource ID
.I               ;; a decimal integer
57
.P               ;; a Pascal string
Finance charges

TYPE ICN# = GNRL ;; icon list for an application
 ,128            ;; resource ID
.H               ;; enter 2 icons in hexadecimal
0001 0002 0003 0004  ;; each is 32 bits by 32 bits
 ....
007d 007e 007f 0080  ;; for 128 words total

TYPE FONT = GNRL ;; define a new type
 ,268            ;; resource ID
.R               ;; read from the System file
System FONT 268  ;; type FONT with ID = 268
```

To actually run the resource compiler, simply type its name to the SHELL. The program will load and display the standard file selection window. The window will show all text files with an extension of ".r". To see all text files, cancel the file selection window and select the **.R Filter** option in the **File** menu. Then select **Compile** from the same menu which will bring the file selection window back again.

To actually compile, select the file to be used for input and click **Open**. As each line is compiled, it is displayed in the left-hand window. The size of the resource file is displayed in the right hand window. If no errors occur, the program can be exited by clicking the window's close box or the **Quit** button. If an error occurs, the line containing the error is the last line on the screen. The program displays a box with an error message in it.

6. Using RMaker with Aztec C

During program development, it is possible to avoid using the resource compiler each time the program is changed. This is done by placing the resources produced by the resource compiler in a file with a fixed name. Then, in the *main()* routine of the program itself, the *OpenResFile()* routine should be used to open the resource file. This will make the resources available as though they were part of the program file itself.

When the program has been completed, the resource file can be combined with the program file for final use, and the *OpenResFile()* statement can be removed. To combine the output from the *RMaker* with the output of LN, edit the *RMaker* input file and add a statement:

```
INCLUDE linker.out
```

where *linker.out* is the name of the linker output file. The resulting

file is the final form of the application being developed.

NAME

set - environment variable and exec file utility

SYNOPSIS

set

set VAR=string

set [-+x] [-+e] [-+n]

set [-a]

DESCRIPTION

set is used to examine and set environment variables, to set exec file options, and to enable the trapping of errors by the SHELL.

set is a builtin command; that is, its code is contained in the SHELL.

Displaying and setting environment variables

The first form listed for *set* causes *set* to display the name and value of each environment variable.

The second form assigns *string* to the environment variable *VAR*.

Setting Exec file options

The third form, which can only be used within an exec file, sets options for the exec file. The options are associated with a character, as follows:

x	Command line logging. With this option enabled, before a command line in an exec file is executed, it's logged to the screen. By default, this option is disabled.
e	Exit prematurely. With this option enabled, a command which terminates with a non-zero return code causes the exec file to be aborted. By default, this option is enabled.
n	Non-execution. With this option enabled, commands in the exec file aren't executed. By default, this option is disabled.

Preceding an option's character with a minus sign enables the option, and preceding it with a plus sign disables it.

Enabling error trapping

The fourth form of the *set* command enables the trapping of the following Macintosh system errors:

LIBRARY FUNCTIONS OVERVIEW: MACINTOSH INFORMATION

Library Functions Overview: Macintosh Information

The *Library Functions Overview* chapter presented overview information that is independent of the system on which your programs run. This chapter presents overview information about the library functions that is specific to programs that run on a Macintosh.

The sections of this chapter are numbered; the information discussed in a section is related to the section in the *Library Functions Overview* chapter that has the same number.

1. Overview of I/O: Macintosh Information

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

1.1 Pre-opened devices and command line arguments

For programs running on a Macintosh, the program's name is pointed at by the first item in the array that is pointed at by the second argument of the of the program's *main* function. That is, if the *main* function begins

```
main(argc, argv)
int argc; char *argv[];
```

then *argv[0]* is a pointer to the program's name.

For programs that are activated by the SHELL, a command line argument can be a quoted string or a file name template. For details, see the SHELL reference chapter.

Programs that can be activated by the Finder can be passed command line arguments, but they must receive them using the Macintosh conventions, rather than the UNIX conventions. That is, they access them via a handle in the system area, rather than as arguments to the program's *main* function.

1.2 File I/O

1.2.3 Opening Files

When a SHELL-activated program wants to open a file, the file name has the standard SHELL format. That is, it consists of an optional volume name, an optional directory, and a filename. The

volume defaults to the volume containing the current directory, and the directory to the current directory.

For examples of file names, see the SHELL reference chapter.

A Macintosh file can have a data fork and/or a resource fork. The standard, UNIX-compatible functions for opening files open a file's data fork. Two special functions, *openrf* and *creatrf*, open a file's resource fork.

The functions that create files on the Macintosh (*open*, *fopen*, *creatrf*, etc) set the type and creator of the file to *TEXT* and *????*, respectively. These attributes can be changed, using the function *settyp*. This function is described in the Macintosh Functions chapter.

1.3 Device I/O

On the Macintosh, devices are accessed using the following names:

device	name
keyboard	.con
display	.con
printer	.bout
RS232 in	.ain
RS232 out	.aout

2. Overview of Standard I/O: Macintosh Information

2.1 Opening files on the Macintosh.

As mentioned in the I/O overview section, the standard, UNIX compatible functions for opening files on a Macintosh open a file's data fork.

Two special functions, *openrf* and *creatrf*, are provided for opening a file's resource fork for unbuffered i/o. To open a file's resource fork for standard i/o, first open it for unbuffered i/o, by calling *openrf* or *creatrf*. Then open it for standard i/o, by calling *fdopen*.

2.5 Buffering

On the Macintosh, the size of a buffer used for standard I/O is 512 bytes.

3. Overview of Unbuffered I/O: Macintosh Information

For the Macintosh, the *open* and *creat* functions open a file's data fork. Two special functions are provided which open a file's resource fork: *openrf* and *creatrf*.

4. Overview of Console I/O: Macintosh Information

On the Macintosh, the UNIX console i/o options are available. In addition, other options are available that aren't UNIX-compatible,

including the automatic expansion of tabs to spaces on output (the XTAB option) and whether or not the program will wait if it issues a read to the console when a key hasn't yet been depressed (the NODELAY option).

A program's default console mode on a Macintosh is line-oriented, with ECHO and CRMOD enabled, just as it would be on another system. In addition, a Macintosh program's default mode has the special options XTAB and ECHOE (defined below) enabled, NODELAY disabled, and tab stops set every four characters.

4.1 Line-oriented Input

On the Macintosh, all console options are program- selectable, even in line-oriented input mode.

Thus, line-oriented input doesn't automatically enable ECHO for a Macintosh program.

On the Macintosh, a non-UNIX option, NODELAY is available, which defines whether a program wants to wait if its read request can't be immediately satisfied. With NODELAY reset and with the console in line-oriented mode, a read request to the console will wait if an entire line hasn't been typed. With NODELAY set and with the console in line-oriented mode, a read request will always return immediately: if an entire line hasn't been typed, no characters will be returned to the program (even if some characters have been typed); if an entire line has been typed, the requested characters will be returned.

4.2 Character-oriented Input on the Macintosh

On the Macintosh, there is one exception to the rule that RAW mode resets all other options: with the console in RAW mode, a program still has control over the NODELAY option.

On the Macintosh, the console driver buffers typed characters, even when the console is in a character-oriented input mode. The console driver will scan the keyboard for input when ever an i/o operation is performed on the console. Thus, it is possible for the operator to enter characters while the program is sending information to the screen. The characters aren't returned to the program until it asks for them.

With the console in character-oriented input mode, the driver's treatment of a read request to the console depends on the console's NODELAY option: if this option is reset, the program will be suspended until at least one character has been received; then, the requested number of characters, up to the number in the internal buffer, are returned to the program. Thus, suppose a program issues the input call

```
read (0, buf, 80)
```

to the console, which is in a character-oriented mode with NODELAY reset. If there are characters in the driver's internal buffer, it will return the requested number of characters from this buffer, up to the number in the buffer; if 80 characters aren't already in the buffer, it won't wait for the operator to enter the remaining characters. If there are no characters in the driver's buffer, the driver will suspend the program until the operator types a character, and then return that character to the program.

If the console is in character-oriented mode with NODELAY set, a read request to the console will always return immediately: if no characters are in the driver's buffer, no characters are returned to the program; otherwise, the characters in the buffer are returned, up to the number requested.

4.4 The *sgtty* fields

4.4.1 The *sg_flags* field

On the Macintosh, the following non-UNIX flags for *sg_flags* are supported in addition to the UNIX-compatible flags:

XTAB — Convert tabs to spaces on output, with tab stops set as specified by TABSIZ. By default, XTAB is enabled.

TABSIZ — A mask for a four-bit field that defines the tabwidth to be used when XTAB is set. By default, TABSIZ is set to four.

ECHOE — When ECHO is set, and the 'erase' character is entered, output the 'erase' character, then a space, and then another 'erase' character (thus erasing the character from the screen); By default, ECHOE is enabled.

NODELAY — When a read is issued to the console and no keys have been typed, return immediately. By default, NODELAY is disabled.

CHKKEY — Defines whether the console driver is to check for output flow control. When this option is enabled, the operator can type ^S to suspend console output, and then type any character to allow it to continue. By default, CHKKEY is enabled.

ERASE — Defines whether the console driver should erase the character under the cursor before displaying the next output character. If this option is disabled, the new character's pattern is combined with that of the existing character. Not erasing speeds screen output considerably. By default, ERASE is disabled.

4.5 The *sg_erase* field

This field is supported on the Macintosh. It defines the character that, on console input, causes the previously-entered characters to not be returned to a program.

4.6 The *sg_kill* field

sg_kill is supported on the Macintosh. It defines the character that, on console input, causes the console driver to delete all the characters that are in its internal buffer and that haven't been returned to the program.

By default, this character is ^X, that is, control-X.

5. Overview of Dynamic Buffer Allocation: Mac Info

On the Macintosh, the non-UNIX memory allocation function *lmalloc* is provided. See the Macintosh Functions chapter for details.

6. Overview of Error Processing

On the Macintosh, the special values that toolbox and OS routines may return are defined in the file *syserr.h*.

MACINTOSH FUNCTIONS

Macintosh Functions

This chapter describes functions which are available only to programs which are running on a Macintosh.

As with the chapter describing system independent functions, this chapter is divided into sections, each of which describes a group of related functions.

The header to a section contains a letter in parentheses describing the library containing the section's functions. The codes and their related libraries are:

C	c.lib
S	s.lib

Following this introduction is an index to the functions, and then the functions themselves.

NAME

creatrf - create a new resource file

SYNOPSIS

creatrf(name, pmode)
char *name;
int pmode;

DESCRIPTION

creatrf creates a file and opens its resource fork 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).

creatrf is just like the UNIX function *creat* except that it opens a file's resource fork rather than its data fork.

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

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

The parameter *pmode* is optional. If specified, it is ignored. The pmode parameter should be included, however, for programs for which UNIX-compatibility is required, since the UNIX creat function requires it. In this case, pmode should have an octal value of 0666.

SEE ALSO

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

DIAGNOSTICS

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

NAME

execl, execv, execlp, execvp

SYNOPSIS

```
execl(name, arg0, arg1, arg2, ..., argn, (char *) 0)
char *name, *arg0, *arg1, *arg2, ...;

execv(name, argv)
char *name, *argv[];

execlp(name, arg0, arg1, arg2, ..., argn, (char *) 0)
char *name, *arg0, *arg1, *arg2, ...;

execvp(name, argv)
char *name, *argv[];
```

DESCRIPTION

These functions cause another program or an exec file to be executed. If a program is called, the called program is loaded on top of the calling program. If an exec file is called, the SHELL is loaded and told to execute it. Thus, in either case, if the exec function succeeds, it doesn't return to the caller.

These functions can be used within the SHELL or the Finder environment; that is, when either the SHELL or the Finder is being used as the command processor. The functions can activate programs of type 'AZTC' or 'APPL'.

When a program of type 'AZTC' is started, the exec function can specify parameters that are to be passed. The called program receives these parameters in the standard UNIX way; that is, as arguments to its *main* function.

When a program of type 'APPL' is started, the calling program can pass parameters to it; however, this is done using the standard Macintosh convention rather than as arguments to the exec function. That is, the caller must set up the argument list in the system heap, set the appropriate handle to it in the system area, and then issue the exec call. The called program then receives the arguments by fetching them from the system heap, rather than as arguments to its *main* function.

The following paragraphs first describe the parameters to the exec functions, then describe the differences between the functions, and finally discuss other features of the functions.

Parameters

name is the name of the file containing the program or exec file. *name* has the standard SHELL format for a file name; that is, it consists of an optional volume name, an optional directory (and the path to it), and the name of a file within the directory. The volume defaults to the volume containing the current directory,

NAME

exit, _exit

SYNOPSIS

exit(code)

_exit(code)

DESCRIPTION

These functions cause a program to terminate and control to return to the SHELL or to the Finder.

For an exec file-activated program, *code* is passed back to the exec file. If it is non-zero and if the exec file has set its 'abort' option, the exec file will be terminated.

exit and *_exit* differ in that *exit* closes all files opened for standard i/o, while *_exit* doesn't.

NAME

getenv

SYNOPSIS

char *getenv(name)
char *name;

DESCRIPTION

getenv returns a pointer to the character string associated with the environment variable *name*, or 0 if the variable isn't in the environment.

The character string is in a static buffer and will be overwritten when the next call is made to *getenv*.

NAME

openrf - open resource file

SYNOPSIS

#include "fcntl.h"

openrf(name, mode)
char *name;

DESCRIPTION

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

openrf is just like the UNIX function *open* except that it opens a file's resource fork instead of its data fork.

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

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

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

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

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

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

If the O_TRUNC option is specified, the file will be truncated so that nothing is in it. The truncation is performed by simply

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

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

NAME

screen manipulation functions:
scr_beep, scr_bs, scr_tab, scr_lf,
scr_cursup, scr_cursrt, scr_cr,
scr_clear, scr_home, scr_curs, scr_eol,
scr_linsert, scr_ldelete,
scr_cinsert, scr_cdelete

SYNOPSIS

```
scr_beep()
scr_bs()
scr_tab()
scr_lf()
scr_cursup()
scr_cursrt()
scr_cr()
scr_clear()
scr_home()
scr_eol()
scr_linsert()
scr_ldelete()
scr_cinsert()
scr_cdelete()
scr_curs(lin, col)
int lin, col;
```

DESCRIPTION

These functions can be called by command programs to manipulate screens of text. For example, there are functions to clear the screen, position the cursor, and insert and delete characters and lines.

These functions can be used in conjunction with the normal standard i/o and unbuffered i/o functions to display characters on the console.

A program that calls these functions must access the console using the Aztec console driver; that is, it must have been linked with *shcroot* or *mixcroot*.

scr_beep rings the keyboard bell.

scr_bs moves the cursor back one character space, without modifying the character that was backspaced over.

scr_tab moves the cursor right one tab stop.

scr_lf moves the cursor down one line, scrolling if at the bottom of the screen.

scr_cursup moves the cursor up without changing its column location.

NAME

settyp - set file fields

SYNOPSIS

```
settyp(filename, type, creator)
char *filename;
long type, creator;
```

DESCRIPTION

settyp sets the type and creator fields of the file named *filename.*

filename is a C format character string, and *type* and *creator* are longs.

For example,

```
settyp("myappl", 'APPL', 'HACK');
```

sets the file *myappl* to have type *APPL* and creator *HACK.*

NAME

monitor, _intr_sp - profiling functions

SYNOPSIS

int monitor(lowpc, highpc, buffer, size, numcalls)
int (*lowpc)(); /* start of area to be profiled, normally _Corg */
int (*highpc)(); /* end of area to be profiled, normally _Cend */
short *buffer; /* address of buffer to hold hits */
int size; /* size of buffer */
int numcalls; /* dummy argument as yet unimplemented */

DESCRIPTION

monitor is a function which sets up the Macintosh to perform a runtime analysis of where the user program is spending its execution time. This is accomplished by installing a task via the Vertical Retrace Manager to be executed every vertical retrace interrupt unless a speed other than default setting is used. This routine then records a tick if the current execution address at the time of the interrupt is in the address range being analyzed.

Once the analysis is complete, the tick summary is written to a file called mon.out which can be used as input to the *prof* utility to produce a report of runtime activity. *monitor* is called once with non-zero arguments to initiate analysis and once with all zero arguments to terminate analysis. A simple example of this is included in the file test.c which starts and stops the monitor in main() by using the macros MON_ON and MON_OFF set up in monitor.h. The test can be set up and run as follows:

```
cc test.c
ln -t test.o monitor.o -lc
test
prof -s test.sym
```

Any add-on boards or external monitors may cause problems.

Users may change the monitoring clock speed by using the intr_sp(speed) option. The default for speed is 60 which allows 60 interrupts per second. Speed may be slowed to one interrupt per second by passing a value of 1 to it.

EXAMPLE

The simplest way to describe the use of monitor is through an example.

Suppose there is a program *foo.c* for which analysis is desired. At the start of the main routine of *foo.c*, place the following code:

Linking a program with monitor calls
When linking a program containing monitor calls, the user should be careful to use the *-T or -W option*, which produces a symbol table for the program, as this is needed for running the *prof* utility which produces the report.

Notes:

The monitor() function currently will not profile a segment other than CODE 1. It will also not tell you any information about any calls you make to the Macintosh ROM.

SEE ALSO
prof

(Duplicate Page - First issued with release 1.06h)

NAME

mktemp - make a unique file name

SYNOPSIS

```
char *
mktemp (template)
char *template;
```

DESCRIPTION

mktemp replaces the character string pointed at by *template* with the name of a non-existent file, and returns as its value a pointer to the string.

The string pointed at by *template* should look like a file name whose last few characters are *X*s with an optional imbedded period.

mktemp replaces the *X*s with a letter followed by the least significant digits of the starting address of its program's data segment. The letter will be between 'A' and 'Z', and will be chosen such that the resulting character string isn't the name of an existing file.

DIAGNOSTICS

For a given character string, *mktemp* will try to convert the string into one of 26 file names. If all of these files exist, *mktemp* will replace the first character pointed at by *template* with a null character.

SEE ALSO

tmpfile, tmpnam

EXAMPLES

The following program calls *mktemp* to get a character string that it can use as a file name. If the program's data segment begins at the decimal address 123456, then the generated name will be one of the strings *abcA23.456*, *abcB23.456*, ..., *abcZ23.456*. If all these strings are the names of existing files, *mktemp* will replace the first character of the string passed to it, *a* in this case, with 0.

(Duplicate page - First issued with release 1.06h)

NAME

_newrom

SYNOPSIS

_newrom()

DESCRIPTION

_newrom returns a 0 if the machine has the old 64K ROM in it, a 1 if it has the Mac Plus Rom, or a 2 if the machine is a Mac II.

(Duplicate page - First issued with release 1.06h)

NAME

stat

SYNOPSIS

```
stat(name, buf)
char *name, *buf;
```

DESCRIPTION

stat returns the attribute byte, date and time, and size of the file *name*. This information is returned in *buf*, which has the following format:

```
struct stat {
    char st_attr;
    long st_mtime;
    long st_size;
        long st_rsize;
};
```

This structure, and the meaning of the bits in the attribute and time fields are defined in the header file *stat.h*, and in the TIME section.

name can optionally specify the full pathname where the file is located.

ERRORS

stat returns -1 if it fails, after setting a code in the global integer *errno*. The Errors section of the Library Overview chapter describes these codes.

(Duplicate page - First issued with release 1.06h)

NAME

screen manipulation functions:
scr_beep, scr_bs, scr_tab, scr_lf,
scr_cursup, scr_cursrt, scr_cr,
scr_clear, scr_home, scr_curs, scr_eol,
scr_linsert, scr_ldelete,
scr_cinsert, scr_cdelete, scr_echo, scr_getc

SYNOPSIS

scr_beep()

scr_bs()

scr_tab()

scr_lf()

scr_cursup()

scr_cursrt()

scr_cr()

scr_clear()

scr_home()

scr_eol()

scr_linsert()

scr_ldelete()

scr_cinsert()

scr_cdelete()

scr_curs(lin, col)
int lin, col

scr_echo(flg)
int flg;

scr_getc();

DESCRIPTION

These functions can be called by command programs to manipulate screens of text. For example, there are functions to clear the screen, position the cursor, insert and delete characters and lines, enable and disable echo mode, and get characters.

scr_getc reads a character from the keyboard, waiting if a key hasn't been depressed, and echoes it to the screen if the global integer *_echo* is non-zero. *scr_getc* returns one of the following values:

* For normal characters, its ASCII value (a number between decimal 0 and 127).
* For special characters, a number between 128 and 255.
* For control-break, -2.

(Duplicate page - First issued with release 1.06h)

NAME

time, ctime, localtime, gmtime, asctime

SYNOPSIS

```
long time(tloc)
long *tloc;

char *ctime(clock)
long *clock;

#include "time.h"

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;
```

DESCRIPTION

time returns the date and time, which it gets from the operating system. The other functions convert the date and time, which are passed as arguments, to another format.

time returns the current date and time packed into a long int. If its argument *tloc* is non-null, the return value is also stored in the field pointed at by the argument. The format of the value returned by *time* is described below.

ctime, *localtime*, and *gmtime* convert a date and time pointed at by their argument, which is in a format such as returned by *time*, to another format:

ctime converts the time to a 26-character ASCII string of the form

Mon Apr 30 10:04:52 1984\n\0

localtime and *gmtime* unpack the date and time into a structure and return a pointer to it. The structure, named *tm*, is described below and defined in the header file *time.h*.

asctime converts a date and time pointed at by its argument, which is in a structure such as returned by *localtime* and *gmtime*, to a 26-character ASCII string in the same form as returned by *ctime*.

The long int returned by *time* and passed to *ctime*, *localtime*, and *gmtime* has the following form (bit 0 is the least

(Duplicate page - First issued with release 1.06h)

NAME

tmpnam - create a name for a temporary file

SYNOPSIS

char *tmpnam (s)
char *s;

DESCRIPTION

tmpnam creates a character string that can be used as the name of a temporary file and returns as its value a pointer to the string. The generated string is not the name of an existing file.

s optionally points to an area into which the name will be generated. This must contain at least *L_tmpnam* bytes, where *L_tmpnam* is a constant defined in *stdio.h*.

s can also be a NULL pointer. In this case, the name will be generated in an internal array. The contents of this array are destroyed each time *tmpnam* is called with a NULL argument.

The generated name is prefixed with the string that is associated with the symbol *P_tmpdir*; this symbol is defined in *stdio.h*. In the distribution version of *stdio.h*, *P_tmpdir* is a null string; this results in the generated name specifying a file that will be located in the 'current area'. The location of this area is system dependent: on PC-DOS/MS-DOS 2.x and the Macintosh, it's the current directory on the default drive; on CP/M-86, it's the current user area on the default drive.

SEE ALSO

tmpfile, mktemp

TOOLBOX and OS FUNCTIONS

Chapter Contents

Toolbox Functions

This chapter describes how programs access the Macintosh toolbox and operating system routines. The information in this chapter supplements that presented in the Apple manual *Inside Macintosh*; to write C programs that access Macintosh routines, you will use information from the Apple manual and from this chapter.

The Macintosh routines that provide related functions are grouped together and called a "manager". In *Inside Macintosh*, a chapter is devoted to each of the Macintosh managers, with each chapter containing an overview section and a summary section. The overview section of an *inside Macintosh* chapter presents an overview of its manager, of course, and discusses the data structures passed to and returned from it, the values that may be set in the structures, and the manager's routines that can be called by user-written programs. The summary section of an *Inside Macintosh* chapter summarizes the information presented in the overview section, simply listing the constants, data structures, and routines related to it, without defining their meanings.

This chapter, then, is divided into sections, with each section summarizing the information needed for a program to access a particular Macintosh manager. The sections are titled, and a section's title identifies its corresponding Macintosh manager. The sections are sorted alphabetically, according to their titles.

As mentioned above, each section of this chapter summarizes the information that a programmer will need to write C programs that access a Macintosh manager. The information in each section is at the same level of detail as that contained in the summary section of its corresponding *Inside Macintosh* chapter. A section has the following organization:

1. Constants
   Lists symbols that have been declared within Manx-supplied header files that relate to the manager.
2. Data structures
   Lists the data structures that have been declared within Manx-supplied header files that relate to the manager.
3. Functions
   Describes the C functions that can be called to access the manager's routines. The description of each function includes the type of value, if any, that the function returns, the parameters that are passed to it,

and their types.

For each Macintosh manager, a header file is provided on the distribution disk, and should be included within a C program that accesses that manager's routines. The header file for a manager declares the constants and data structures related to the manager; it also declares the functions that C programs call to access the manager's routines, and the type of values they return.

Writing programs that call Macintosh routines

To write a C program that calls Macintosh routines, you should decide what toolbox routines and what Pascal variables a Pascal version of your program would call and declare, by reading the *Inside Macintosh* manual. Then you should translate these variable declarations and function calls to C, using the information in this chapter.

The names of the constants, data structures, and functions that a C program uses to access the Macintosh routines correspond very closely to those that a Pascal program uses to access the same routines. In most cases, they are identical. Thus, the variable declarations and function calls that a C program uses to access the Macintosh routines are frequently very similar to those used in a Pascal program. For example, the Pascal statements to call the QuickDraw routine *OpenPort* look like this:

```
VAR gp : GrafPtr;
OpenPort(gp);
```

The corresponding C statements would look like this:

```
#include <quickdraw.h>
...

GrafPtr gp;
OpenPort(gp);
```

The header file *quickdraw.h* declares the structure *GrafPtr* and the function *OpenPort*.

The following paragraphs discuss the translation of variable declarations and of function calls from Pascal to C.

Translating variable declarations

In most cases, the type name of a variable is the same in C as it is in Pascal. In the above example the type of the variable named *gp* was *GrafPtr* in both Pascal and C programs. The only differences are in the built-in Pascal types INTEGER and LongInt. These are translated to *short* and *long*, respectively.

Translating function calls

In many cases, the C form of a function call is identical to its Pascal equivalent, since the names of the functions are the same in both Pascal and C and since most types of variables are passed to a C routine just as they are to a Pascal routine. We presented an example of this above.

The main difference is in the passing of structures. In Pascal, a call that passes a structured variable to a routine passes the address of the variable, if the number of bytes in the variable is greater than four, and passes the contents of the variable, otherwise.

Thus, in Pascal, if *func* is a function and *tmp* a structured variable, the statement

```
func(tmp)
```

will pass the address of *tmp* to the function if *tmp* contains more than four bytes, and will pass the contents of *tmp*, otherwise.

In Aztec C, structured variables can't be passed to functions. Thus, if you attempted to compile the above statement, the compiler would generate an error message.

In Aztec C, instead of passing a structure to a function, you have to pass its address. This can be done using the 'address-of' operator, &. For example, the address of the structured variable *tmp* could be passed to *func* in a C program using the statement:

```
func(&tmp)
```

Thus, if *Inside Macintosh* says that a structure must be passed to a function, you must be careful in the translation of the Pascal to its C equivalent. Given the call to *func* shown above in a Pascal program, you would translate the call differently, depending on the size of the structured variable *tmp*.

If *tmp* contains more than four bytes, the call statement could be translated to

```
func(&tmp);
```

But if *tmp* contains four or fewer bytes, you must somehow generate code that passes the contents of *tmp*, and not its address.

Fortunately, of all the structures passed to the Macintosh routines, only one, Point, contains four or fewer bytes. Point contains exactly four bytes. Thus, translating Pascal statements that pass other types of structured variables to Macintosh routines is done as shown above: change *func(tmp)* to *func(&tmp)*.

The Manx-supplied header files declare a macro function, *pass*, that can be used to pass variables of type Point to Macintosh routines. If the variable *tmp* shown above is of type Point, the call to *func* can be

translated as

```
func(pass(tmp));
```

pass makes use of the facts that both the Point structure and a long variable contain four bytes, and that the contents of a long variable can be passed to a function. *pass* generates a value of type *long*; the value is the contents of the Point structure.

In summary, if *tmp* is a structure, the statement

```
func(tmp)
```

can be translated to

```
func(&tmp)
```

if *tmp* contains more than four bytes, and to

```
func(pass(tmp))
```

if *tmp* is of type *Point.*

Using Booleans

The typedef *Boolean* is defined in the file *quickdraw.h.* To simply and safely use *Booleans* in a program, follow these rules:

* Use the definitions *TRUE* and *FALSE*, which are defined in *quickdraw.h*, to assign a value to a *Boolean* variable or to pass a *Boolean* constant to a function.
* To test the value returned by a *Boolean* function or the value of a *Boolean* variable, use the fact that the value will be zero for false and non-zero for true.
* Don't directly assign the value of a *Boolean* function to a *Boolean* variable. Instead, test the value of the function, and set the variable to either *TRUE* or *FALSE*. For example, if *f()* and *var* are a *Boolean* function and variable, respectively, then assign the value of *f()* to *var* using a statement such as

```
var= f()? TRUE : FALSE;
```

* Don't immediately pass the value of a *Boolean* function to another function. Instead, test the value of the function, and pass either *TRUE* or *FALSE*. For example, if *f()* is a *Boolean* function and *g()* is another function, then pass the value of *f()* to *g()* using a statement such as

```
g(f()?TRUE:FALSE);
```

And that's all you need to know in order to successfully use *Booleans*. The following paragraphs discuss the implementation of *Booleans* in a C program in detail.

A *Boolean* is a Pascal data type. The size of a *Boolean* is implementation dependent; on the Macintosh, it's a single byte. We

have chosen to define a *Boolean* to be a signed *char*, using the *typedef* statement.

By definition, the least significant bit of a *Boolean* defines its value: 0 for false, 1 for true. In theory, the settings of the other bits of a *Boolean* are undefined. In practice, the other bits are 0 if the least significant bit is 0 and undefined if it's 1; it's this practical fact about the implementation of *Boolean*s that allows you to test a *Boolean* using 0 for false and non-zero for true.

In C, a *char* is passed to a function in two bytes. The least significant byte will contain the value. The most significant byte will contain either zero or a value generated by propagating the *char*'s sign bit, depending on whether the *char* is unsigned or signed.

In Macintosh Pascal, a *Boolean* is also passed to a function in two bytes; unlike C, however, the actual *Boolean* value is passed in the most significant byte, while the value in the least significant byte is undefined.

Because of this difference in the ways that a *char* is passed to a C function and that a *Boolean* is passed to a Pascal function, you could have problems in passing a value from a C program to a Pascal function that's expecting a *Boolean*. If you follow the rules defined above, however, you won't have problems: *TRUE* is defined to be -1 and *FALSE* to be 0, so if you directly pass one of these values to a Pascal function, the least significant bit of the most significant byte of the two-byte value that is passed will be set correctly. And if you pass the value of a *Boolean* variable to a Pascal function, where the variable was assigned the value *TRUE* or *FALSE*, the sign bit of the *Boolean* (ie, signed *char*) variable will define the value of the *Boolean* variable, and when it is propagated, the least significant bit of the most significant byte of the two-byte value that is passed will have the correct value.

In C, a function that is defined to return a *char* actually returns a two-byte value, with the value's least significant byte containing the *char* and its most significant byte containing a value generated by propagating the *char*'s sign bit.

In Macintosh Pascal, on the other hand, a function that is defined to return a *Boolean* also returns a two-byte value, with the *Boolean* in the value's most significant byte, and an undefined value in its least significant byte.

The compiler itself resolves this difference in the ways that a *char* value is returned by a C function and that a *Boolean* value is returned by a Pascal function: when the compiler encounters a statement that uses the value returned by a function that's defined to be of type *pascal Boolean* (or equivalently *pascal char*), it generates code that generates a two-byte value whose least significant byte is the *Boolean*

value returned by the function and whose most significant byte is generated by propagating the *Boolean* value's sign bit.

When a value is assigned to a *Boolean* variable, the sign bit must define the value, if the variable is to be correctly passed to a *Boolean* function. Since the sign bit that's returned by a Pascal *Boolean* function doesn't define the function's value, you can't directly assign the function's value to a *Boolean* variable. You must do it indirectly, as specified in the above rules.

Similar reasoning explains why you can't directly pass the value of a *Boolean* function to a Pascal function.

Pascal also has a *char* data type. On the Macintosh, a *char* is the same size as a *Boolean*, that is, one byte. Like *Booleans*, Pascal *char* data items are passed between functions in a two-byte field. However, a *char* value is in the field's least significant byte whereas a *Boolean* value is in the field's most significant byte. If you use the rules presented above for passing *Booleans* to Pascal functions, this difference is not a problem when a *Boolean* or *char* is passed to a Pascal function. But because the compiler doesn't know the difference between a Pascal function that returns a *Boolean* and one that returns a *char*, and since the compiler generates code that assumes that a Pascal *Boolean* or *char* function returns a *Boolean* value in the most significant byte, there is a potential problem in C programs calling a Pascal function that returns a *char* value. In practice, however, the problem never occurs, because there are no toolbox routines that return a *char*.

C in a Pascal world

Aztec C has made several extensions to the C language that facilitate the development of C programs on the Macintosh. These allow C programs to directly call Pascal programs, and vice versa, and allow a C program to define a character string constant that uses the Pascal format. For details, see the Programming Information section of the Compiler chapter.

The toolbox and the 128K Macintosh

The header files that are included in programs that call toolbox functions define a lot of constants and structures. These definitions use a lot of memory space, decreasing the size of the program that can be compiled.

With the limited amount of memory on a 128K Macintosh, this can be a problem. To help, the header files have surrounded infrequently-used definitions with statements of the form

```
#ifndef SMALL_MEM
  ...
#endif
```

Thus, if the compiler runs out of space when compiling a program, you can try recompiling with the symbol SMALL_MEM defined. This symbol can be defined using either the compiler's -D option or by explicitly #defining the symbol in the program.

Feel free to change the sections of the header files that are excluded from compilation on a 128K Macintosh. The choices we made are only a guideline.

For more discussion of software development on a 128K Macintosh, see the Technical Information chapter.

Calling Quickdraw from Drivers

A driver that calls a Quickdraw function should define the symbol _DRIVER. This can be done using either the compiler's -D option, or by explicitly defining the symbol in the program. This definition prevents the *quickdraw.h* header, when included in the driver program, from defining several fields that are defined in the programs that call the driver.

Control Manager Functions

The functions summarized in this section allow C programs to access routines that are part of the Macintosh Control Manager.

The constants, structures, and functions described in this section are defined in the header file *control.h.*

1. Constants

```
#define pushButProc         0
#define checkBoxProc        1
#define radioButProc        2
#define useWFont            8
#define scrollBarProc       16

#define inButton            10
#define inCheckBox          11

#define inUpButton          20
#define inDownButton        21
#define inPageUp            22
#define inPageDown          23
#define inThumb             129

#define drawCntl            0
#define testCntl            1
#define calcCRgns           2
#define initCntl            3
#define dispCntl            4
#define posCntl             5
#define thumbCntl           6
#define dragCntl            7
#define autoTrack           8
```

2. Data structures

```
struct ControlRecord **     ControlHandle;
```

```
struct ControlRecord {
        ControlHandle           nextControl;
        WindowPtr               contrlOwner;
        Rect                    contrlRect;
        char                    contrlVis;
        char                    contrlHilite;
        short                   contrlValue;
        short                   contrlMin;
        short                   contrlMax;
        Handle                  contrlProc;
        Handle                  contrlData;
        ProcPtr                 contrlAction;
        long                    contrlRfCon;
        Str255                  contrlTitle;
};

typedef struct ControlRecord    ControlRecord;
typedef struct ControlRecord *  ControlPtr;
```

3. Functions

3.1 Initialization and Allocation

pascal ControlHandle *NewControl* (theWindow, boundsRectPtr,
title, visible, value,
min, max, procID, refCon)
WindowPtr theWindow; Rect * boundsRectPtr;
Str255 title; Boolean visible;
short value, min, max, procID;
long refCon;

pascal ControlHandle *GetNewControl* (controlID, theWindow)
short controlID; WindowPtr theWindow;

pascal void *DisposeControl* (theControl)
ControlHandle theControl;

pascal void *KillControls* (theWindow)
ControlHandle theWindow;

3.2 Control Display

pascal void *SetCTitle* (theControl, theTitle)
ControlHandle theControl; Str255 theTitle;

pascal void *GetCTitle* (theControl, theTitle)
ControlHandle theControl;
Str255 theTitle;

```
pascal void HideControl ( theControl )
    ControlHandle theControl;

pascal void ShowControl ( theControl )
    ControlHandle theControl;

pascal void DrawControls ( theWindow )
    WindowPtr theWindow;

pascal void HiliteControl ( theControl, hiliteState )
    ControlHandle theControl;
    short hiliteState;

pascal void UpdtControls ( theWindow, update )
    windowPtr theWindow;
    RgnHandle update;
```

3.3 Mouse Location

```
pascal short TestControl ( theControl, pass(thePoint) )
    ControlHandle theControl;
    Point thePoint; /*** This Point must be cast to a long ***/

pascal short FindControl ( pass(thePoint), theWindow, theControlPtr )
    Point thePoint; /*** This Point must be cast to a long ***/
    WindowPtr theWindow;
    ControlHandle *theControlPtr;

pascal short TrackControl ( theControl, pass(startPt), actionProc )
    ControlHandle theControl;
    Point startPt; /*** This Point must be cast to a long ***/
    ProcPtr actionProc;
```

3.4 Control Movement and Sizing

```
pascal void MoveControl ( theControl, h, v )
    ControlHandle theControl; short h, v;

pascal void DragControl ( theControl, pass(startPt),
                          limitRect, slopeRect, axis )
    ControlHandle theControl;
    Point startPt; /*** This point must be cast to a long ***/
    Rect * limitRect, slopRect; short axis;

pascal void SizeControl ( theControl, w, h )
    ControlHandle theControl; short w, h;
```

3.5 Setting and Range of a Control

```
pascal void SetCtlValue ( theControl, theValue )
    ControlHandle theControl; short theValue;
```

pascal short *GetCtlValue* (theControl)
ControlHandle theControl;

pascal void *SetCtlMin* (theControl, minValue)
ControlHandle theControl; short minValue;

pascal short *GetCtlMin* (theControl)
ControlHandle theControl;

pascal void *SetCtlMax* (theControl, maxValue)
ControlHandle theControl; short maxValue;

pascal short *GetCtlMax* (theControl)
ControlHandle theControl;

3.6 Miscellaneous Utilities

pascal void *SetCRefCon* (theControl, refVal)
ControlHandle theControl; long refVal;

pascal long *GetCRefCon* (theControl)
ControlHandle theControl;

pascal void *SetCtlAction* (theControl, actionProc)
ControlHandle theControl; ProcPtr actionProc;

pascal ProcPtr *GetCtlAction* (theControl)
ControlHandle theControl;

Desk Manager Functions

This section describes functions that allow C programs to access Macintosh Desk Manager routines.

The constants, structures, and functions described in this section are defined in the header file *desk.h*.

1. Constants

```
#define undoCmd     0
#define cutCmd      2
#define copyCmd     3
#define pasteCmd    4
#define clearCmd    5
```

2. Functions

2.1 Opening and Closing Desk Accessories

```
pascal short OpenDeskAcc ( theAcc )
    Str255 theAcc;

pascal void CloseDeskAcc ( refNum )
    short refNum;
```

2.2 Handling Events in Desk Accessories

```
pascal void     SystemClick ( theEventPtr, theWindow )
    EventRecord * theEventPtr;  WindowPtr theWindow;

pascal Boolean  SystemEdit ( editCmd )
    short editCmd;
```

2.3 Performing Periodic Actions

```
pascal void     SystemTask ()
```

2.4 Advanced Routines

```
pascal Boolean  SystemEvent ( theEventPtr )
    EventRecord * theEventPtr;

pascal void SystemMenu ( menuResult )
    long menuResult;
```

Dialog Manager Functions

This section describes functions which allow C programs to access functions that are part of the Macintosh Dialog Manager.

The constants, structures, and functions described in this section are defined in the header file *dialog.h*.

1. Constants

```
#define crtlItem        0x04
#define btnCtrl         0x00
#define chkCtrl         0x01
#define radCtrl         0x02
#define resCtrl         0x03
#define statText        0x08
#define editText        0x10
#define iconItem        0x20
#define picItem         0x40
#define userItem        0x00
#define itemDisable     0x80

#define OK              1
#define Cancel          2

#define stopIcon        0
#define noteIcon        1
#define ctnIcon         2
```

2. Data Structures

```
struct DialogRecord {
        WindowRecord            window;
        Handle                  items;
        TEHandle                textH;
        short                   editField;
        short                   editOpen;
        short                   aDefItem;
};

typedef struct DialogRecord     DialogRecord;
typedef struct DialogRecord *   DialogPeek;
typedef WindowPtr               DialogPtr;
```

```
struct DialogTemplate {
        Rect                    boundsRect;
        short                   procID;
        char                    visible;
        char                    filler1;
        char                    goAwayFlag;
        char                    filler2;
        long                    refCon;
        short                   itemsID;
        Str255                  title;
};

typedef struct DialogTemplate           DialogTemplate;
typedef struct DialogTemplate *         DialogTPtr;
typedef struct DialogTemplate **        DialogTHandle;

struct StageList {
        char                    boldItem;
        char                    boxDrawn;
        char                    sound;
};

typedef struct StageList        StageList[4];

struct AlertTemplate {
        Rect                    boundsRect;
        short                   itemsID;
        StageList               stages;
};

#define volBits                 0x3
#define alBit                   0x4
#define OKDismissal             0x8
```

3. Functions

3.1 Initialization

pascal void *InitDialogs* (restartProc)
    ProcPtr restartProc;

pascal void *ErrorSound* (soundProc)
    ProcPtr soundProc;

pascal void *SetDAFont* (fontNum)
    short fontNum;

3.2 Creating and Disposing of Dialogs

pascal DialogPtr *NewDialog* (dStorage, boundsRectPtr, title,
visible, procID, behind,
goAwayFlag, refCon, items)
Ptr dStorage; Rect * boundsRectPtr; Str255 title;
Boolean visible, goAwayFlag; short procID; WindowPtr behind;
long refCon; Handle items;

pascal DialogPtr *GetNewDialog* (dialogID, dStorage, behind)
short dialogID; Ptr dStorage; WindowPtr behind;

pascal void *CloseDialog* (theDialog)
DialogPtr theDialog;

pascal void *DisposDialog* (theDialog)
DialogPtr theDialog;

pascal void *CouldDialog* (dialogID)
short dialogID;

pascal void *FreeDialog* (dialogID)
short dialogID;

3.3 Handling Dialog Events

pascal Boolean *IsDialogEvent* (theEventPtr)
EventRecord * theEventPtr;

pascalBoolean *DialogSelect* (theEventPtr, theDialogPtr,
itemHitPtr)
EventRecord * theEventPtr; DialogPtr * theDialogPtr;
short * itemHitPtr;

pascal void *ModalDialog* (filterProc, itemHitPtr)
ProcPtr filterProc; short * itemHitPtr;

pascal void *DlgCut* (theDialog)
DialogPtr thedialog;

pascal void *DlgCopy* (theDialog)
DialogPtr thedialog;

pascal void *DlgPaste* (theDialog)
DialogPtr thedialog;

pascal void *DlgDelete* (theDialog)
DialogPtr thedialog;

pascal void *DrawDialog* (theDialog)
DialogPtr theDialog;

3.4 Invoking Alerts

pascal short *Alert* (alertID, filterProc)
short alertID; ProcPtr filterProc;

pascal short *StopAlert* (alertID, filterProc)
short alertID; ProcPtr filterProc;

pascal short *NoteAlert* (alertID, filterProc)
short alertID; ProcPtr filterProc;

pascal short *CautionAlert* (alertID, filterProc)
short alertID; ProcPtr filterProc;

pascal void *CouldAlert* (alertID)
short alertID;

pascal void *FreeAlert* (alertID)
short alertID;

3.5 Manipulating Items in Dialogs and Alerts

pascal void *ParamText* (param0, param1, param2, param3)
Str255 param0, param1, param2, param3;

pascal void *GetDItem* (theDialog, itemNo, typePtr,
itemPtr, boxPtr)
DialogPtr theDialog; short itemNo, * typePtr; Handle * itemPtr;
Rect * boxPtr;

pascal void *SetDItem* (theDialog, itemNo, type, item, boxPtr)
DialogPtr theDialog; short itemNo, type; Handle item;
Rect * boxPtr;

pascal void *HideDItem* (dialog, itemNo)
DialogPtr dialog; short itemNo;

pascal void *ShowDItem* (dialog, itemNo)
DialogPtr dialog; short itemNo;

pascal short *FindDItem* (dialog, pass(thePoint))
DialogPtr dialog; Point thePoint;

pascal void *UpdtDialog* (dialog, updateRgn)
DialogPtr dialog; rgnHandle updateRgn;

pascal void *GetIText* (item, text)
Handle item; Str255 text;

pascal void *SetIText* (item, text)
Handle item; Str255 text;

pascal void *SelIText* (theDialog, itemNo, strtSel, endSel)
DialogPtr theDialog; short itemNo, strtSel, endSel;

short *GetAlrtStage* ( )

void *ResetAlrtStage* ( )

Disk Manager Functions

This section describes functions that allow C programs to access Macintosh Disk Manager routines.

The constants, structures, and functions described in this section are defined in the header file *disk.h*.

1. Constants

```
#define currPos         0x00
#define absPos          0x01
#define relPos          0x03
#define rdVerify        0x40
```

2. Data Structures

```
struct DrvSts {
    short           track;
    SignedByte      writeProt;
    SignedByte      diskInPlace;
    SignedByte      installed;
    SignedByte      sides;
    QElemPtr        qLink;
    short           qType;
    short           dQDrive;
    short           dQRefNum;
    short           dQFSID;
    SignedByte      twoSideFmt;
    SignedByte      needsFlush;
    short           diskErrs;
};
```

3. Functions

3.1 Disk Driver Routines

```
pascal short DiskEject ( drvNum )
    short drvNum;

pascal short SetTagBuffer ( buffPtr )
    Ptr buffPtr;

pascal short DriveStatus ( drvNum, status )
    short drvNum;
    DrvSts *status;
```

Event Manager Functions

The functions described in this section allow C programs to call the Macintosh Event Manager routines.

The constants, data structures, and functions described in This section are defined in the header file *event.h*.

1. Constants

```
#define nullEvent       0
#define mouseDown       1
#define mouseUp         2
#define keyDown         3
#define keyUp           4
#define autoKey         5
#define updateEvt       6
#define diskEvt         7
#define activateEvt     8
#define abortEvt        9
#define networkEvt      10
#define driverEvt       11
#define app1Evt         12
#define app2Evt         13
#define app3Evt         14
#define app4Evt         15

#define nullMask        0x0001
#define mDownMask       0x0002
#define mUpMask         0x0004
#define keyDownMask     0x0008
#define keyUpMask       0x0010
#define autoKeyMask     0x0020
#define updateMask      0x0040
#define diskMask        0x0080
#define activMask       0x0100
#define abortMask       0x0200
#define networkMask     0x0400
#define driverMask      0x0800
#define app1Mask        0x1000
#define app2Mask        0x2000
#define app3Mask        0x4000
#define app4Mask        0x8000

#define charCodeMask    0x000000ff
#define keyCodeMask     0x0000ff00
```

```
#define optionKey       0x0800
#define alphaLock       0x0400
#define shiftKey        0x0200
#define cmdKey          0x0100
#define btnState        0x0080
#define everyEvent      0xffff
```

2. Data Structures

```
struct EventRecord {
        short       what;
        long        message;
        long        when;
        Point       where;
        short       modifiers;
};
typedef struct EventRecord    EventRecord;
typedef long                  KeyMap[4];
```

3. Functions

3.1 Accessing Events

pascal Boolean *GetNextEvent* (eventMask, theEventPtr)
    short eventMask; EventRecord * theEventPtr;

pascal Boolean *EventAvail* (eventMask, theEventPtr)
    short eventMask; EventRecord * theEventPtr;

Boolean *OSEventAvail* (eventPtr, eventMask)
    EventRecord * eventPtr; short eventMask;

Boolean *GetOSEvent* (eventPtr, eventMask)
    EventRecord * eventPtr; short eventMask;

3.2 Posting and Removing Events

void *PostEvent* (eventCode, eventMsg)
    short eventCode; long eventMsg;

void *FlushEvents* (eventMask, stopMask)
    short eventMask, stopMask;

3.3 Reading the Mouse

pascal void *GetMouse* (mouseLocPtr)
    Point * mouseLocPtr;

pascal Boolean *Button* ()

pascal Boolean *StillDown* ()

pascal Boolean *WaitMouseUp* ()

3.4 Reading the Keyboard and Keypad

pascal void *GetKeys* (theKeys)
KeyMap theKeys;

3.5 Miscellaneous Utilities

pascal void *SetEventMask* (theMask)
short theMask;

pascal QHdrPtr *GetEvQHdr* ( )

pascal long *TickCount* ()

long *GetDbleTime* ()

long *GetCaretTime* ()

File Manager Functions

This section summarizes the information needed for C programs that want to access the Macintosh File Manager routines.

The constants, data structures, and functions are defined in the header file *pb.h*. This file makes references to information defined in the header file *types.h*. *pb.h* will automatically include *types.h* in a program if it hasn't yet been included.

1. Constants

```
#define fHasBundle          0x20
#define fInvisible          0x40

#define fTrash              -3
#define fDesktop            -2
#define fDisk               0

#define fsAtMark            0
#define fsFromStart         1
#define fsFromLEOF          2
#define fsFromMark          3
#define rdVerify            0x0040

#define fsCurPerm                       0
#define fsRdPerm                        1
#define fsWrPerm                        2
#define fsRdWrPerm                      3
#define fsRdWrShPerm        4
```

2. Data structures

```
struct Finfo {
        OSType          fdType;
        OSType          fdCreator;
        short           fdFlags;
        Point           fdLocation;
        short           fdFldr;
};
typedef struct Finfo    Finfo;
```

```
struct ioParam {
        short           ioRefNum;
        SignedByte      ioVersNum;
        SignedByte      ioPermssn;
        Ptr             ioMisc;
        Ptr             ioBuffer;
        long            ioReqCount;
        long            ioActCount;
        short           ioPosMode;
        long            ioPosOffset;
};
struct fileParam {
        short           ioFRefNum;
        SignedByte      ioFVersNum;
        SignedByte      filler1;
        short           ioFDirIndex;
        SignedByte      ioFlAttrib;
        SignedByte      ioFlVersNum;
        Finfo           ioFlFndrInfo;
        long            ioFlNum;
        unsigned short  ioFlStBlk;
        long            ioFlLgLen;
        long            ioFlPyLen;
        unsigned short  ioFlRStBlk;
        long            ioFlRLgLen;
        long            ioFlRPyLen;
        long            ioFlCrDat;
        long            ioFlMdDat;
};
```

```
struct hfileParam {
                        short           ioFRefNum;
                        SignedByte      ioFVersNum;
                        SignedByte      filler1;
                        short           ioFDirIndex;
                        SignedByte      ioFlAttrib;
                        SignedByte      ioFlVersNum;
                        FInfo           ioFlFndrInfo;
                        long            ioDirID;
                        unsigned short  ioFlStBlk;
                        long            ioFlLgLen;
                        long            ioFlPyLen;
                        unsigned short  ioFlRStBlk;
                        long            ioFlRLgLen;
                        long            ioFlRPyLen;
                        long            ioFlCrDat;
                        long            ioFlMdDat;
};
struct volumeParam {
        long            filler2;
        short           ioVolIndex;
        long            ioVCrDate;
        long            ioVLsBkUp;
        short           ioVAtrb;
        unsigned short  ioVNmFls;
        short           ioVDirSt;
        short           ioVBlLn;
        unsigned short  ioVNmAlBlks;
        long            ioVAlBlkSiz;
        long            ioVClpSiz;
        short           ioAlBlSt;
        long            ioVNextFNum;
        unsigned short  ioVFrBlk;
};
```

```
struct hvolumeParam {
                        long            filler4;
                        short           ioVolIndex;
                        long            ioVCrDate;
                        long            ioVLsMod;
                        short           ioVAtrb;
                        unsigned short  ioVNmFls;
                        short           ioVBitMap;
                        short           ioVAllocPtr;
                        unsigned short  ioVNmAlBlks;
                        long            ioVAlBlkSiz;
                        long            ioVClpSiz;
                        short           ioAlBlSt;
                        long            ioVNxtCNID;
                        unsigned short  ioVFrBlk;
                        short           ioVSigWord;
                        short           ioVDrvInfo;
                        short           ioVDRefNum;
                        short           ioVFSID;
                        long            ioVBkUp;
                        unsigned short  ioVSeqNum;
                        long            ioVWrCnt;
                        long            ioVFilCnt;
                        long            ioVDirCnt;
                        long            ioVFndrInfo[8];
};
struct hFileInfo {
                        FInfo           ioFlFndrInfo;
                        long            ioDirID;
                        unsigned short  ioFlStBlk;
                        long            ioFlLgLen;
                        long            ioFlPyLen;
                        unsigned short  ioFlRStBlk;
                        long            ioFlRLgLen;
                        long            ioFlRPyLen;
                        long            ioFlCrDat;
                        long            ioFlMdDat;
                        long            ioFlBkDat;
                        FInfo           ioFlXFndrInfo;
                        long            ioFlParID;
                        long            ioFlClpSiz;
};
```

```
struct DInfo {
                                Rect            FRect;
                                short           FRFlags;
                                Point           FRLocation;
                                short           FRView;
};
typedef struct DInfo            DInfo;
struct DXInfo {
                                Point           FRScroll;
                                long            FROpenChain;
                                short           FRUnused;
                                short           FRComment;
                                long            FRPutAway;
};
typedef struct DXInfo                           DXInfo;
struct dirInfo {
                                DInfo           ioDrUsrWds;
                                long            ioDrDirID;
                                unsigned short  ioDrNmFls;
                                short           filler3[9];
                                long            ioDrCrDat;
                                long            ioDrMdDat;
                                long            ioDrBkDat;
                                DXInfo          ioDrFndrInfo;
                                long            ioDrParID;
};
struct drvQElRec {
        struct drvQElRec *      drvLink;
        short                   drvFlags;
        short                   drvRefNum;
        short                   drvFSID;
        short                   drvBlkSize;
};
```

```
union OpParamType {
        struct {
                short                   sg_flags;
                char                    sg_erase;
                char                    sg_kill;
        } conCtl;
        short                           sndVal;
        short                           asncConfig;
        struct {
                Ptr                     asncBPtr;
                short                   asncBLen;
        } asyncInBuff;
        struct {
                unsigned char           fXOn;
                unsigned char           fCTS;
                char                    xon;
                char                    xoff;
                unsigned char           errs;
                unsigned char           evts;
                unsigned char           fInX;
                unsigned char           null;
        } asyncShk;
        struct {
                long                    param1;
                long                    param2;
                long                    param3;
        } printer;
        struct {
                Ptr                     fontRecPtr;
                short                   fontCurDev;
        } fontMgr;
        Ptr                             diskBuff;
        long                            asyncNBytes;
        struct {
                short                   asncS1;
                short                   asncS2;
                short                   asncS3;
        } asyncStatus;
        struct {
                short                   dskTrackLock;
                long                    dskInfoBits;
                struct drvQElRec        dskQElem;
                short                   dskPrime;
                short                   dskErrCnt;
        } diskStat;
};
typedef union OpParamType       OpParamType;
typedef union OpParamType *     OpParamPtr;
```

```
struct cntrlParam {
        short                   csRefNum;
        short                   csCode;
        OpParamType             csParam;
};
struct ParamBlkRec {
        struct ParamBlkRec *    ioLink;
        short                   ioType;
        short                   ioTrap;
        Ptr                     ioCmdAddr;
        ProcPtr                 ioCompletion;
        short                   ioResult;
        char *                  ioNamePtr;
        short                   ioVRefNum;
        union {
                struct ioParam      iop;
                struct fileParam    fp;
                struct volumeParam  vp;
                struct cntrlParam   cp;
        } u;
};
typedef struct ParamBlkRec      ParamBlkRec;
typedef struct ParamBlkRec *    ParmBlkPtr;
```

```
struct HPrmBlkRec {
                        struct HPrmBlkRec *qLink;
                        short           qType;
                        short           ioTrap;
                        Ptr             ioCmdAddr;
                        ProcPtr         ioCompletion;
                        short           ioResult;
                        char *          ioNamePtr;
                        short           ioVRefNum;
                        union {
                                        struct ioParam iop;
                                        struct hfileParam hfp;
                                        struct hvolumeParam hvp;
                                        struct cntrlParam cp;
                        } u;
};
typedef struct HPrmBlkRec               HPrmBlkRec;
typedef struct HPrmBlkRec *             HPrmBlkPtr;

struct CInfoPBRec {
                        struct CInfoPBRec *qLink;
                        short           qType;
                        short           ioTrap;
                        Ptr             ioCmdAddr;
                        ProcPtr         ioCompletion;
                        short           ioResult;
                        char *          ioNamePtr;
                        short           ioVRefNum;
                        short           ioFRefNum;
                        short           filler1;
                        short           ioFDirIndex;
                        SignedByte      ioFlAttrib;
                        SignedByte      filler2;
                        union {
                                        struct hFileInfo hfi;
                                        struct dirInfo di;
                        } u;
};
typedef struct CInfoPBRec       CInfoPBRec;
typedef struct CInfoPBRec *     CInfoPBPtr;
```

```
struct CMovePBRec {
                                struct CMovePBRec *qLink;
                                short           qType;
                                short           ioTrap;
                                Ptr             ioCmdAddr;
                                ProcPtr         ioCompletion;
                                short           ioResult;
                                char *          ioNamePtr;
                                short           ioVRefNum;
                                long            filler1;
                                char *          ioNewName;
                                long            filler2;
                                long            ioNewDirID;
                                long            filler3[2];
                                long            ioDirID;
};
typedef struct CMovePBRec       CMovePBRec;
typedef struct CMovePBRec *     CMovePBPtr;

struct WDPBRec {
                                struct WDPBRec *qLink;
                                short           qType;
                                short           ioTrap;
                                Ptr             ioCmdAddr;
                                ProcPtr         ioCompletion;
                                short           ioResult;
                                char *          ioNamePtr;
                                short           ioVRefNum;
                                short           filler;
                                short           ioWDIndex;
                                long            ioWDProcID;
                                short           ioWDVRefNum;
                                short           filler2[7];
                                long            ioWDDirID;
};
typedef struct WDPBRec          WDPBRec;
typedef struct WDPBRec *        WDPBPtr;
```

```
struct FCBPBRec {
                        struct FCBPBRec *qLink;
                        short           qType;
                        short           ioTrap;
                        Ptr             ioCmdAddr;
                        ProcPtr         ioCompletion;
                        short           ioResult;
                        char *          ioNamePtr;
                        short           ioVRefNum;
                        short           ioRefNum;
                        short           filler;
                        long            ioFCBIndx;
                        long            ioFCBFlNm;
                        short           ioFCBFlags;
                        unsigned short  ioFCBStBlk;
                        long            ioFCBEOF;
                        long            ioFCBPLen;
                        long            ioFCBCrPs;
                        short           ioFCBVRefNum;
                        long            ioFCBClpSiz;
                        long            ioFCBParID;
};
typedef struct FCBPBRec                 FCBPBRec;
typedef struct FCBPBRec *               FCBPBPtr;
```

```
struct VCB {
        struct VCB *    qLink;
        short           qType;
        short           vcbFlags;
        short           vcbSigWord;
        long            vcbCrDate;
        long            vcbLsMod;
        short           vcbAtrb;
        unsigned short  vcbNmFls;
        short           vcbVBMSt;
        short           vcbAllocPtr;
        unsigned short  vcbNmAlBlks;
        long            vcbAlBlkSiz;
        long            vcbClpSiz;
        short           vcbAlBlSt;
        long            vcbNxtCNID;
        unsigned short  vcbFreeBks;
        char            vcbVN[27];
        short           vcbDrvNum;
        short           vcbDRefNum;
        short           vcbFSID;
        short           vcbVRefNum;
        char *          vcbMAdr;
        char *          vcbBufAdr;
        short           vcbMLen;
        short           vcbDirIndex;
        short           vcbDirBlk;
        long            vcbVolBkUp;
        unsigned short  vcbVSeqNum;
        long            vcbWrCnt;
        long            vcbXTClpSiz;
        long            vcbCTClpSiz;
        unsigned short  vcbNmRtDirs;
        long            vcbFilCnt;
        long            vcbDirCnt;
        long            vcbFndrInfo[8];
        short           vcbVCSize;
        short           vcbVBMCSiz;
        short           vcbCtlCSiz;
        unsigned short  vcbXTAlBlks;
        unsigned short  vcbCTAlBlks;
        short           vcbXTRef;
        short           vcbCTRef;
        long            vcbCtlBuf;
        long            vcbDirIDM;
        short           vcbOffsM;
};
```

```
struct DrvQEl {
                        struct DrvQEl *qLink;
                        short          qType;
                        short          dQDrive;
                        short          dQRefNum;
                        short          dQFSID;
                        short          dQDrvSize;
};
```

3. Functions

3.1 High-Level Functions

3.1.1 Accessing Volumes

OSErr *GetVInfo* (drvNum, volName, vRefNumPtr, freeBytesPtr)
short drvNum; OSStrPtr volName; short * vRefNumPtr;
long * freeBytesPtr;

OSErr *GetVol* (volName, vRefNumPtr)
OSStrPtr volName; short * vRefNumPtr;

OSErr *SetVol* (volName, vRefNum)
OSStrPtr volName; short vRefNum;

OSErr *FlushVol* (volName, vRefNum)
OSStrPtr volName; short vRefNum;

OSErr *UnmountVol* (volName, vRefNum)
OSStrPtr volName; short vRefNum;

OSErr *Eject* (volName, vRefNum)
OSStrPtr volName; short vRefNum;

3.1.2 Changing file contents

OSErr *Create* (fileName, vRefNum, creator, fileType)
OSStrPtr fileName;
short vRefNum; OSType creator, fileType;

OSErr *FSOpen* (fileName, vRefNum, refNumPtr)
OSStrPtr fileName;
short vRefNum, * refNumPtr;

OSErr *FSClose* (refNum)
short refNum;

OSErr *OpenDriver* (name, refNum)
Str255 name; short refNum;

OSErr *CloseDriver* (refNum)
short refNum;

OSErr *FSRead* (refNum, countPtr, buffPtr)
short refNum; long * countPtr; Ptr buffPtr;

OSErr *FSWrite* (refNum, countPtr, buffPtr)
short refNum; long * countPtr; Ptr buffPtr;

OSErr *GetFPos* (refNum, filePosPtr)
short refNum; long *filePosPtr;

OSErr *SetFPos* (refNum, posMode, posOff)
short refNum, posMode; long posOff;

OSErr *GetEOF* (refNum, logEOF)
short refNum; long *logEOF;

OSErr *SetEOF* (refNum, logEOF)
short refNum; long logEOF;

OSErr *Allocate* (refNum, countPtr)
short refNum; long * countPtr;

OSErr *Control* (refNum, opCode, opParams)
short refNum, opCode; OpParamPtr opParams;

OSErr *Status* (refNum, opCode, opParamsptr)
short refNum, opCode; OpParamPtr * opParamsptr;

OSERR *KillIO* (refNum)
short refNum;

3.1.3 Changing Information about Files

OSErr *GetFInfo* (fileName, vRefNum, fndrInfoPtr)
OSStrPtr fileName; short vRefNum; FInfo * fndrInfoPtr;

OSErr *SetFInfo* (fileName, vRefNum, fndrInfo)
OSStrPtr fileName; short vRefNum; FInfo fndrInfo;

OSErr *SetFLock* (fileName, vRefNum)
OSStrPtr fileName; short vRefNum;

OSErr *RstFLock* (fileName, vRefNum)
OSStrPtr fileName; short vRefNum;

OSErr *Rename* (oldName, vRefNum, newName)
OSStrPtr oldName, newName; short vRefNum;

OSErr *FSDelete* (fileName, vRefNum)
OSStrPtr fileName; short vRefNum;

3.2 Low-level functions

3.2.1 Initialization

pascal void *InitQueue* ()

3.2.2 Accessing Volumes

pascal OSErr *PBMountVol* (paramBlock)
ParmBlkPtr paramBlock;

pascal OSErr *PBGetVInfo* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHGetVInfo* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBGetVol* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHGetVol* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBSetVol* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHSetVol* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBFlushVol* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBUnmountVol* (paramblock)
ParmBlkPtr paramBlock;

pascal OSErr *PBOffLine* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBEject* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

3.2.3 Changing File Contents

pascal OSErr *PBCreate* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHCreate* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBDirCreate* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBOpen* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHOpen* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBOpenRF* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHOpenRF* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBLockRange* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBUnlockRange* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBRead* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBWrite* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBGetFPos* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBSetFPos* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBGetEOF* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBSetEOF* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBAllocate* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBAllocContig* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBFlushFile* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBClose* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

3.2.4 Changing Information about Files

pascal OSErr *PBGetFInfo* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHGetFInfo* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBSetFInfo* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHSetFInfo* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBSetFLock* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHSetFLock* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBRstFLock* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHRstFLock* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBSetFType* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBSetFVers* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBRename* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHRename* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBDelete* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHDelete* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBControl* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBStatus* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBKillIO* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

3.2.5 Accessing Queues

pascal QHdrPtr *GetFSQHdr* ( )

pascal QHdrPtr *GetVCBQHdr* ( )

pascal QHdrPtr *GetDrvQHdr* ( )

3.2.6 Hierarchial-Only Routines

pascal OSErr *PBGetCatInfo* (paramBlock, async)
CInfoPBPtr paramBlock; Boolean async;

pascal OSErr *PBSetCatInfo* (paramBlock, async)
CInfoPBPtr paramBlock; Boolean async;

pascal OSErr *PBCatMove* (paramBlock, async)
CMovePBPtr paramBlock; Boolean async;

pascal OSErr *PBOpenWD* (paramBlock, async)
WDPBPtr paramBlock; Boolean async;

pascal OSErr *PBCloseWD* (paramBlock, async)
WDPBPtr paramBlock; Boolean async;

pascal OSErr *PBGetWDInfo* (paramBlock, async)
WDPBPtr paramBlock; Boolean async;

Font Manager Functions

The functions described in this section allow a C program to access Macintosh Font Manager routines.

The constants, data structures, and functions described in this section are defined in the header file *font.h*.

1. Constants

```
/* Font Numbers */

#define systemFont          0
#define applFont            1
#define newYork             2
#define geneva              3
#define monaco              4
#define venice              5
#define london              6
#define athens              7
#define sanFran             8
#define toronto             9

/* Font Types        */

#define propFont      0x9000
#define fixedFont     0xB000
#define fontWid       0xACB0
```

2. Data Structures

```
struct FMInput {
        short                   family;
        short                   size;
        char                    face;
        char                    needBits;
        short                   device;
        Point                   numer;
        Point                   denom;
};

typedef struct FMInput          FMInput;
```

```
struct FMOutput {
        short           errNum;
        Handle          fontHandle;
        Byte            bold;
        Byte            italic;
        Byte            ulOffset;
        Byte            ulShadow;
        Byte            ulThick;
        Byte            shadow;
        SignedByte      extra;
        Byte            ascent;
        Byte            descent;
        Byte            widMax;
        SignedByte      leading;
        Byte            unused;
        Point           numer;
        Point           denom;
};
typedef struct FMOutput     FMOutput;
typedef struct FMOutput *   FMOutPtr;
struct FontRec {
        short           fontType;
        short           firstChar;
        short           lastChar;
        short           widMax;
        short           kernMax;
        short           nDescent;
        short           fRectMax;
        short           chHeight;
        short           owTLoc;
        short           ascent;
        short           descent;
        short           leading;
        short           rowWords;
};
typedef struct FontRec      FontRec;
struct FontMetricRec {
                        Fixed       ascent;
                        Fixed       descent;
                        Fixed       leading;
                        Fixed       widMax;
                        Handle      WTabHandle;
};
typedef struct FontMetricRec        FontMetricRec;
```

3. Functions

3.1 Initializing the Font Manager

pascal void *InitFonts* ()

3.2 Getting Font Information

pascal void *GetFontName* (fontNum, theName)
short fontNum; Str255 theName;

pascal void *GetFNum* (fontName, theNumPtr)
Str255 fontName; short * theNumPtr;

pascal Boolean *RealFont* (fontNum, size)
short fontNum, size;

pascal void *FontMetrics* (theMetrics)
FontMetricRec * theMetrics;

3.3 Keeping Fonts in Memory

pascal void *SetFontLock* (lockFlag)
Boolean lockFlag;

3.4 Advanced Routine

pascal FMOutPtr *SwapFont* (inRecPtr)
FMInput * inRecPtr;

pascal void SetFScaleDisable (scaleDis)
Boolean scaleDis;

Memory Manager Functions

This section describes functions that allow C programs to access the Macintosh Memory Manager routines.

The constants, data structures, and functions described in this section are defined in the header file *memory.h*.

1. Constants

```
#define maxSize                 0x800000
```

2. Data structures

```
typedef long                    Size;
typedef int                     MemErr;
typedef struct Zone *           THz;

struct Zone {
        Ptr                     bkLim;
        Ptr                     purgePtr;
        Ptr                     hFstFree;
        long                    zcbFree;
        ProcPtr                 gzProc;
        short                   moreMast;
        short                   flags;
        short                   cntRel;
        short                   maxRel;
        short                   cntNRel;
        short                   maxNRel;
        short                   cntEmpty;
        short                   cntHandles;
        long                    minCBFree;
        ProcPtr                 purgeProc;
        Ptr                     sparePtr;
        Ptr                     allocPtr;
        short                   heapData;
};
typedef struct Zone             Zone;
```

3. Functions

3.1 Initialization and Allocation

void *InitAppleZone* ()

```
void SetApplBase ( startPtr )
    Ptr startPtr;

pascal void InitZone ( growProc, masterCount,
                       limitPtr, startPtr )
    ProcPtr growProc; short masterCount;
    Ptr limitPtr, startPtr;

void SetApplLimit ( zoneLimit )
    Ptr zoneLimit;

short MaxApplZone ()

long MaxBlock ()

void MoreMasters ()

long StackSpace ()
```

3.2 Heap Zone Access

```
THz GetZone ()

void SetZone ( hz )
    THz hz;

THz SystemZone ()

THz ApplicZone ()

void PurgeSpace ( total, contig)
                                long *total, *contig;
```

3.3 Allocating and Releasing Relocatable Blocks

```
Handle NewHandle ( logicalSize )
    Size logicalSize;

void DisposHandle ( h )
    Handle h;

Size GetHandleSize ( h )
    Handle h;

void SetHandleSize ( h, newSize )
    Handle h; Size newSize;

THz HandleZone ( h )
    Handle h;

Handle RecoverHandle ( p )
    Ptr p;

void ReallocHandle ( h, logicalSize )
    Handle h; Size logicalSize;
```

short *MoveHHi* (h)
Handle h;

3.4 Allocating and Releasing Nonrelocatable Blocks

Ptr *NewPtr* (logicalSize)
long logicalSize;

void *DisposPtr* (p)
Ptr p;

Size *GetPtrSize* (p)
Ptr p;

void *SetPtrSize* (p, newSize)
Ptr p; Size newSize;

THz *PtrZone* (p)
Ptr p;

3.5 Freeing space on the Heap

long *FreeMem* ()

Size *MaxMem* (growPtr)
Size * growPtr;

Size *CompactMem* (cbNeeded)
Size cbNeeded;

void *ResrvMem* (cbNeeded)
Size cbNeeded;

void *PurgeMem* (cbNeeded)
Size cbNeeded;

void *EmptyHandle* (h)
Handle h;

Handle *NewEmptyHandle* ( )

3.6 Properties of Relocatable Blocks

void *HLock* (h)
Handle h;

void *HUnlock* (h)
Handle h;

void *HPurge* (h)
Handle h;

void *HNoPurge* (h)
Handle h;

short *HSetRBit* (h)
Handle h;

short *HClrRBit* (h)
Handle h;

short *HGetState* (h)
Handle h;

pascal short *HSetState* (h, flg)
Handle h; short flg;

3.7 Grow Zone Functions

void *SetGrowZone* (growZone)
ProcPtr growZone;

Boolean *GZCritical* ()

Handle *GZSaveHnd* ()

3.8 Utility Routines

void *BlockMove* (sourcePtr, destPtr, byteCount)
Ptr sourcePtr, destPtr; Size byteCount;

Ptr *TopMem* ()

MemErr *MemError* ()

Menu Manager Functions

This section describes functions that allow C programs to call routines contained in the Macintosh Menu Manager.

The constants, data structures, and functions described in this section are defined in the header file *menu.h*.

1. Constants

```
#define noMark          0
#define commandMark     17
#define checkMark       18
#define diamondMark     19
#define appleMark       20

#define mDrawMsg        0
#define mChooseMsg      1
#define mSizeMsg        2

#define textMenuProc    0
```

2. Data Structures

```
struct MenuInfo {
        short           menuID;
        short           menuWidth;
        short           menuHeight;
        Handle          menuProc;
        unsigned long   enableFlags;
        Str255          menuData;
};


typedef struct MenuInfo     MenuInfo;
typedef struct MenuInfo *   MenuPtr;
typedef struct MenuInfo **  MenuHandle;
```

3. Functions

3.1 Initialization and Allocation

pascal void *InitMenus* ()

pascal MenuHandle *NewMenu* (menuID, menuTitle)
    short menuID; Str255 menuTitle;

```
pascal MenuHandle GetMenu ( menuID )
    short menuID;

pascal void DisposeMenu ( menuID )
    MenuHandle menuID;

pascal void AppendMenu ( menu, data )
    MenuHandle menu; Str255 data;

pascal void AddResMenu ( menu, theType )
    MenuHandle menu; ResType theType;

pascal  void InsertResMenu ( menu, theType, afterItem)
    MenuHandle menu; ResType theType; short afterItem;
```

3.2 Forming the Menu Bar

```
pascal void InsertMenu ( menu, beforeID )
    MenuHandle menu; short beforeID;

pascal void DrawMenuBar ()

pascal void DeleteMenu ( menuID )
    short menuID;

pascal void ClearMenuBar ()

pascal Handle GetNewMBar ( menuBarID )
    short menuBarID;

pascal Handle GetMenuBar ()

pascal void SetMenuBar ( menuBar )
    Handle menuBar;
```

3.3 Choosing from a Menu

```
pascal long MenuSelect ( pass(startPt) )
    Point startPt; /*** This Point must be cast to a long ***/

pascal long MenuKey ( ch )
    char ch;

pascal void HiliteMenu ( menuID )
    short menuID;
```

3.4 Controlling Items' Appearance

```
pascal void SetItem ( menu, item, itemString )
    MenuHandle menu; short item; Str255 itemString;

pascal void GetItem ( menu, item, itemString )
    MenuHandle menu; short item; Str255 itemString;
```

pascal void *DisableItem* (menu, item)
MenuHandle menu; short item;

pascal void *EnableItem* (menu, item)
MenuHandle menu; short item;

pascal void *CheckItem* (menu, item, checked)
MenuHandle menu; short item; Boolean checked;

pascal void *SetItemIcon* (menu, item, icon)
MenuHandle menu; short item; Byte icon;

pascal void *GetItemIcon* (menu, item, iconPtr)
MenuHandle menu; short item; Byte *iconPtr;

pascal void *SetItemStyle* (menu, item, chStyle)
MenuHandle menu; short item; Style chStyle;

pascal void *GetItemStyle* (menu, item, chStylePtr)
MenuHandle menu; short item; Style * chStylePtr;

pascal void *SetItemMark* (menu, item, markChar)
MenuHandle menu; short item; char markChar;

pascal void *GetItemMark* (menu, item, markCharPtr)
MenuHandle menu; short item; char * markCharPtr;

pascal void *InsMenuItem* (MenuHandle, itemstring, itemNum)
Handle MenuHandle; Str255 itemstring; short itemNum;

pascal void *DelMenuItem* (MenuHandle, itemNum)
Handle MenuHandle; short itemNum;

3.5 Miscellaneous Utilities

pascal void *SetMenuFlash* (count)
short count;

pascal void *CalcMenuSize* (menu)
MenuHandle menu;

pascal short *CountMItems* (menu)
MenuHandle menu;

pascal MenuHandle *GetMHandle* (menuID)
short menuID;

pascal void *FlashMenuBar* (menuID)
short menuID;

Operating System Utilities

This section describes functions that allow C programs to access Macintosh Operating System Utility functions, the sound functions, and the system error function.

The constants, structures, and functions described in this section are defined in the header file *osutil.h*.

1. Constants

```
#define vType          1
#define ioQType        2
#define drvQType       3
#define evType         4
#define fsQType        5
```

2. Data Structures

```
struct SysParamType {
    long            valid;
    short           portA;
    short           portB;
    long            alarm;
    short           font;
    short           kbdPrint;
    short           volClick;
    short           misc;
};

typedef struct SysParamType   SysParamType;
typedef struct SysParamType * SysPtr;

struct DateTimeRec {
    short           year;
    short           month;
    short           day;
    short           hour;
    short           minute;
    short           second;
    short           dayOfWeek;
};

typedef struct DateTimeRec    DateTimeRec;
```

3. Functions

3.1 Pointer and Handle Manipulation

OSErr *HandToHand* (theHandl)
Handle * theHandl;

OSErr *PtrToHand* (srcPtr, dstHandl, size)
Ptr srcPtr; Handle * dstHandl; long size;

OSErr *PtrToXHand* (srcPtr, dstHandl, size)
Ptr srcPtr; Handle * dstHandl; long size;

OSErr *HandAndHand* (aHandl, bHandl)
Handle aHandl, bHandl;

OSErr *PtrAndHand* (pntr, handl, size)
Ptr pntr; Handle handl; long size;

3.2 String Comparison

Boolean *EqualString* (aStr, bStr, case, marks)
Str255 aStr, bStr; Boolean case, marks;

void *UprString* (theString, marks)
Str255 theString; Boolean marks;

short *RelString* (str1, str2, caseSens, diacSens)
Str255 str1, str2; Boolean caseSens, diacSens;

3.3 Date and Time Operations

OSErr *ReadDateTime* (secsPtr)
long * secs;

void *GetDateTime* (secsPtr)
long * secs;

OSErr *SetDateTime* (secs)
long secs;

void *Date2Secs* (datePtr, secsPtr)
DateTimeRec * datePtr; long * secsPtr;

void *Secs2Date* (secs, datePtr)
long secs; DateTimeRec * datePtr;

void *GetTime* (datePtr)
DateTimePtr * datePtr;

void *SetTime* (datePtr)
DateTimePtr * datePtr;

3.4 Parameter RAM Operations

OSErr *InitUtil* ()

SysPtr *GetSysPPtr* ()

OSErr *WriteParam* ()

3.5 Queue Manipulations

void *Enqueue* (qElement, theQueue)
QElemPtr qElement; QHdrPtr theQueue;

OSErr *Dequeue* (qElement, theQueue)
QElemPtr qElement; QHdrPtr theQueue;

3.6 Dispatch Table Utilities

void *SetTrapAddress* (trapAddr, trapNum)
long trapAddr; short trapNum;

long *GetTrapAddress* (trapNum)
short trapNum;

3.7 Miscellaneous Utilities

void *Delay* (numTicks, finalTicks)
long numTicks, * finalTicks;

pascal void *SysBeep* (duration)
short duration;

pascal void *Restart* ( )

pascal void *Environs* (rom, machine)
short *rom, *machine;

3.8 The System Error Function

pascal void *SysError* (errorCode)
short errorCode;

Package Manager Functions

This section describes functions that allow C programs to access Macintosh Package Manager routines.

The constants, structures, and functions described in this section are defined in the header file *package.h.*

1. Constants

```
#define dskInit     2
#define stdFile     3
#define flPoint     4
#define trFunc      5
#define intUtil     6
#define bdConv      7

/* Constants for the Standard File Package */

#define putDlgID    -3999

#define putSave     1
#define putCancel   2
#define putEject    5
#define putDrive    6
#define putName     7

#define getDlgID    -4000

#define getOpen     1
#define getCancel   3
#define getEject    5
#define getDrive    6
#define getNmList   7
#define getScroll   8
```

2. Data Structures

```
struct SFReply {
    char        good;
    char        copy;
    OSType      fType;
    short       vRefNum;
    short       version;
    char        fName[64];
};
```

```
typedef struct SFReply      SFReply;
typedef OSType              SFTypeList[4];
typedef SFTypeList *        SFListPtr;
```

3. Functions

pascal void *InitPack* (packNumber)
    short packNumber;

pascal void *InitAllPacks* ()

/* Each of the following functions calls a package */
/* They require arguments - see *Inside Mac* for details */

pascal void *Pack0* ()

pascal void *Pack1* ()

pascal void *Pack2* ()

pascal void *Pack3* ()

pascal void *Pack4* ()

pascal void *FP68K* ()

pascal void *Pack5* ()

pascal void *Pack6* ()

pascal void *Pack7* ()

pascal void *Pack8* ()

pascal void *Pack9* ()

pascal void *Pack10* ()

pascal void *Pack11* ()

pascal void *Pack12* ()

pascal void *Pack13* ()

pascal void *Pack14* ()

pascal void *Pack15* ()

3.1 Standard File Package Functions

pascal void *SFPutFile* (pass(where), prompt, origName,
                dlgHook, replyPtr)
    Point where; /* This point must be cast to a long */
    Str255 prompt, origName; ProcPtr dlgHook; SFReply * replyPtr;

```
pascal void SFPPutFile ( pass(where), prompt, origName,
                         dlgHook, replyPtr, dlgID, filterProc )
    Point where; /* This point must be cast to a long */
    Str255 prompt, origName; ProcPtr dlgHook; SFReply * replyPtr;
    short dlgID; ProcPtr filterProc;

pascal void SFGetFile ( pass(where), prompt, fileFilter
                        numTypes, typeList, dlgHook, replyPtr )
    Point where; /* This point must be cast to a long */
    Str255 prompt; ProcPtr fileFilter, dlgHook;
    short numTypes; SFListPtr typeList;
    SFReply * replyPtr;

pascal void SFPGetFile ( pass(where), prompt, fileFilter
                         numTypes, typeList, dlgHook, replyPtr
                         dlgID, filterProc )
    Point where; /* This point must be cast to a long */
    Str255 prompt; ProcPtr fileFilter, dlgHook, filterProc;
    short numTypes, dlgID; SFListPtr typeList;
    SFReply * replyPtr;
```

3.2 Functions for the Disk Initialization Package

```
pascal void DILoad ()

pascal void DIUnload ()

pascal short DIBadMount ( pass(where), evtMessage )
    Point where; /* This point must be cast to a long */
    long evtMessage;

pascal short DIFormat ( drvNum )
    short drvNum;

pascal short DIVerify ( drvNum )
    short drvNum;

pascal short DIZero ( drvNum )
    short drvNum;
```

3.3 International Utility Constants

```
#define shortDate       0x000
#define longDate        0x100
#define abbrevDate      0x200

#define currSymLead     0x10
#define currNegSym      0x20
#define currTrailingZ   0x40
#define currLeadingZ    0x80
```

```
#define mdy             0
#define dmy             1
#define ymd             2

#define dayLdingZ       0x20
#define mntLdingZ       0x40
#define century         0x80

#define secLeadingZ     0x20
#define minLeadingZ     0x40
#define hrLeadingZ      0x80

#define verUS           0
#define verFrance       1
#define verBritain      2
#define verGermany      3
#define verItaly        4

typedef struct {
    char                decimalPt;
    char                thousSep;
    char                listSep;
    char                currSym1;
    char                currSym2;
    char                currSym3;
    Byte                currFmt;
    Byte                dateOrder;
    Byte                shrtDateFmt;
    char                dateSep;
    Byte                timeCycle;
    Byte                timeFmt;
    char                mornStr[4];
    char                eveStr[4];
    char                timeSep;
    char                time1Suff;
    char                time2Suff;
    char                time3Suff;
    char                time4Suff;
    char                time5Suff;
    char                time6Suff;
    char                time7Suff;
    char                time8Suff;
    Byte                metricSys;
    short               intl0Vers;
} Intl0Rec, *Intl0Ptr, **Intl0Hndl;
```

```
typedef struct {
    char        days[7][16];
    char        months[12][16];
    Byte        suppressDay;
    Byte        lngDateFmt;
    Byte        dayLeading0;
    Byte        abbrLen;
    char        st0[4];
    char        st1[4];
    char        st2[4];
    char        st3[4];
    char        st4[4];
    short       intl1vers;
    short       localRtn;
} Intl1Rec, *Intl1Ptr, **Intl1Hndl;
```

3.4 International Utility Functions

pascal Handle *IUGetIntl* (theID)
    short theID;

pascal void *IUSetIntl* (refNum, theID, intlParam)
    short refNum, theID; Handle intlParam;

pascal void *IUDateString* (dateTime, longFlag, result)
    long dateTime; DateForm longFlag; Str255 result;

pascal void *IUDatePString* (dateTime, longFlag, result)
    long dateTime; DateForm longFlag; Str255 result;

pascal void *IUTimeString* (dateTime, wantSeconds, result)
    long dateTime; Boolean wantSeconds; Str255 result;

pascal void *IUTimePString* (dateTime, wantSeconds,
                result, intlParam)
    long dateTime; Boolean wantSeconds;
    Str255 result; Handle intrlParam;

pascal Boolean *IUMetric* ()

pascal short *IUMagString* (aPtr, bPtr, aLen, bLen)
    Ptr aPtr, bPtr; short aLen, bLen;

pascal short *IUMagIDString* (aPtr, bPtr, aLen, bLen)
    Ptr aPtr, bPtr; short aLen, bLen;

pascal short *IUCompString* (aStr, bStr)
    Str255 aStr, bStr;

pascal short *IUEqualString* (aStr, bStr)
    Str255 aStr, bStr;

pascal void *StringToNum* (theString, theNumPtr)
Str255 theString; long * theNumPtr;

pascal void *NumToString* (theNum, theString)
long theNum; Str255 theString;

Print Manager Functions

The functions described in this section allow C programs to call Macintosh routines that are part of the Macintosh Print Manager.

The constants, structures, and functions described in this section are defined in the header file *print.h.*

The print manager functions can send information to either the printer or to the screen, depending on the version of the print manager functions with which a program is linked:

* *prlink*, in *c.lib*, sends output to the printer.
* *prscreen.o* sends output to the screen.

For the output of the print manager functions to be sent to the screen, a program must be linked with the *prscreen.o* file, with this file being specified before *c.lib.* Otherwise, the program will be linked with the *prlink* module, and hence send the output of the print manager functions to the printer.

1. Constants

```
#define bDraftLoop      0
#define bSpoolLoop      1
#define bUser1Loop      2
#define bUser2Loop      3

#define iPrBitsCtl      4
#define lScreenBits     0
#define lPaintBits      1
#define iprIOCtl        5
#define iPrEvtCtl       6
#define iPrEvtAll       0x0002fffd
#define iPrEvtTop       0x0001fffd
#define iPrDevCtl       7
#define lPrReset        0x00010000
#define lPrPageEnd      0x00020000
#define lPrLineFeed     0x00030000
#define iFMgrCtl        8

#define iPFMaxPgs       128
#define iPrPgFract      120
#define iPrAbort        128
#define iPrRelease      2
#define lPfType         'PFIL'
#define lPfSig          'PSYS'
```

```
#define sPrDrvr          "\P.Print"
#define iPrDrvrRef       -3
#define lPrintType       'PREC'
#define iPrintDef        0
#define iPrintLst        1
#define iPrintDrvr       2
#define iMyPrDrvr        0xe000
#define iPStrRFil        0xe000
#define iPStrPFil        0xe001
#define iPrStlDlg        0xe000
#define iPrJobDlg        0xe001
```

2. Data Structures

```
typedef char             TStr80[81];
typedef TStr80 *         TPStr80;
typedef Rect *           TPRect;

struct TPrPort {
        GrafPort         gPort;
        QDProcs          gProcs;
};
typedef struct TPrPort   TPrPort;
typedef struct TPrPort * TPPrPort;

union TPPort {
        GrafPtr          pGPort;
        TPPrPort         pPrPort;
};
typedef union TPPort     TPPort;

struct TPrInfo {
        short            iDev;
        short            iVRes;
        short            iHRes;
        Rect             rPage;
};
typedef struct TPrInfo   TPrInfo;

typedef unsigned char    TFeed;

#define feedCut          0
#define feedFanfold      1
#define feedMechCut      2
#define feedOther        3

typedef short            TWord;
```

```
struct TPrStl {
        TWord           wDev;
        short           iPageV;
        short           iPageH;
        SignedByte      bPort;
        TFeed           feed;
};
typedef struct TPrStl   TPrStl;

struct TPrJob {
        short           iFstPage;
        short           iLstPage;
        short           iCopies;
        SignedByte      bJDocLoop;
        char            fFromUsr;
        ProcPtr         pIdleProc;
        TPStr80         pFileName;
        short           iFileName;
        SignedByte      bFileVers;
        SignedByte      bJobX;
};
typedef struct TPrJob   TPrJob;
typedef unsigned char   TScan;

#define scanTB          0
#define scanBT          1
#define scanLR          2
#define scanRL          3

struct TPrXInfo {
        short           iRowBytes;
        short           iBandV;
        short           iBandH;
        short           iDevBytes;
        short           iBands;
        SignedByte      bPatScale;
        SignedByte      bUlThick;
        SignedByte      bUlOffset;
        SignedByte      bUlShadow;
        TScan           scan;
        SignedByte      bXInfoX;
};
typedef struct TPrXInfo TPrXInfo;
```

```
struct TPrint {
        short           iPrVersion;
        TPrInfo         prInfo;
        Rect            rPaper;
        TPrStl          prStl;
        TPrInfo         prInfoPT;
        TPrXInfo        prXInfo;
        TPrJob          prJob;
        short           printX[19];
};
typedef struct TPrint       TPrint;
typedef struct TPrint *     TPPrint;
typedef struct TPrint **    THPrint;

struct TPrStatus {
        short           iTotPages;
        short           iCurPage;
        short           iTotCopies;
        short           iCurCopy;
        short           iTotBands;
        short           iCurBand;
        char            fPgDirty;
        char            fImaging;
        THPrint         hPrint;
        TPPrPort        pPrPort;
        PicHandle       hPic;
};
typedef struct TPrStatus    TPrStatus;
```

3. Functions

3.1 Initialization and Termination

void *PrOpen* ();

void *PrClose* ();

3.2 Print records and dialogs

void *PrintDefault* (hPrint)
    THPrint hPrint;

Boolean *PrValidate* (hPrint)
    THPrint hPrint;

Boolean *PrStlDialog* (hPrint)
    THPrint hPrint;

Boolean *PrJobDialog* (hPrint)
THPrint hPrint;

void *PrJobMerge* (hPrintSrc, hPrintDst)
THPrint hPrintSrc, hPrintDst;

3.3 Document Printing

TPPrPort *PrOpenDoc* (hPrint, pPrPort, pIOBuf)
THPrint hPrint; TPPrPort pPrPort; Ptr pIOBuf;

void *PrCloseDoc* (pPrPort)
TPPrPort pPrPort;

void *PrOpenPage* (pPrPort, pPageFrame)
TPPrPort pPrPort; TPRect pPageFrame;

void *PrClosePage* (pPrPort)
TPPrPort pPrPort;

3.4 Spool Printing

void *PrPicFile* (hPrint, pPrPort, pIOBuf,
pDevBuf, prStatusPtr)
THPrint hPrint; TPPrPort pPrPort; Ptr pIOBuf;
Ptr pDevBuf; TPrStatus * prStatusPtr;

3.5 Handling Errors

short *PrError* ()

void *PrSetError* (iErr)
short iErr;

3.6 Low-Level Driver Access

void *PrDrvrOpen* ()

void *PrDrvrClose* ()

void *PrCtlCall* (iWhichCtl, lParam1, lParam2, lParam3)
short iWhichCtl; long lParam1, lParam2, lParam3;

Handle *PrDrvrDCE* ()

short *PrDrvrVers* ()

void *PrNoPurge* ()

void *PrPurge* ()

Quickdraw Functions

The functions described in this section allow a C program to call the Macintosh Quickdraw routines.

The constants, data structures, and functions described in this are defined in the header file *quickdraw.h*. The constants and data structures are also defined in the header file *qd.h*.

1. Constants

```
#define srcCopy          0
#define srcOr            1
#define srcXor           2
#define srcBic           3
#define notSrcCopy       4
#define notSrcOr         5
#define notSrcXor        6
#define notSrcBic        7
#define patCopy          8
#define patOr            9
#define patXor           10
#define patBic           11
#define notPatCopy       12
#define notPatOr         13
#define notPatXor        14
#define notPatBic        15

#define normalBit        0
#define inverseBit       1
#define redBit           4
#define greenBit         3
#define blueBit          2
#define cyanBit          8
#define magentaBit       7
#define yellowBit        6
#define blackBit         5

#define blackColor       33
#define whiteColor       30
#define redColor         205
#define greenColor       341
#define blueColor        409
#define cyanColor        273
#define magentaColor     137
#define yellowColor      69
```

```
#define picLParen       0
#define picRParen       1
```

2. Data Structures

```
typedef char            QDByte;
typedef QDByte *        QDPtr;
typedef QDPtr *         QDHandle;
typedef unsigned char   Pattern[8];
typedef int             Bits16[16];

#define frameMode       0
#define paintMode       1
#define eraseMode       2
#define invertMode      3
#define fillMode        4

typedef unsigned short  Style;
#define boldStyle       0x01
#define italicStyle     0x02
#define underlineStyle  0x04
#define outlineStyle    0x08
#define shadowStyle     0x10
#define condenseStyle   0x20
#define extendStyle     0x40

struct FontInfo {
        short           ascent;
        short           descent;
        short           widMax;
        short           leading;
};

typedef struct FontInfo FontInfo;

struct Point {
        short           v;
        short           h;
};
typedef struct Point    Point;

#define vh(x)   ((int *)(&(x).v))
```

```
struct Rect {
        short                   top;
        short                   left;
        short                   bottom;
        short                   right;
};
typedef struct Rect             Rect;

#define topLeft(x)              (*(struct Point *)(&(x).top))
#define botRight(x)             (*(struct Point *)(&(x).bottom))

struct BitMap {
        QDPtr                   baseAddr;
        short                   rowBytes;
        Rect                    bounds;
};
typedef struct BitMap           BitMap;

struct Cursor {
        Bits16                  data;
        Bits16                  mask;
        Point                   hotSpot;
};
typedef struct Cursor           Cursor;

struct PenState {
        Point                   pnLoc;
        Point                   pnSize;
        short                   pnMode;
        Pattern                 pnPat;
};
typedef struct PenState         PenState;

struct Region {
        short                   rgnSize;
        Rect                    rgnBBox;
};
typedef struct Region           Region;
typedef struct Region *         RgnPtr;
typedef struct Region **        RgnHandle;

struct Picture {
        short                   picSize;
        Rect                    picFrame;
};

typedef struct Picture          Picture;
typedef struct Picture *        PicPtr;
typedef struct Picture **       PicHandle;
```

```
struct Polygon {
        short                   polySize;
        Rect                    polyBBox;
        Point                   polyPoints[1];
};

typedef struct Polygon          Polygon;
typedef struct Polygon *        PolyPtr;
typedef struct Polygon **       PolyHandle;

struct QDProcs {
        QDPtr                   textProc;
        QDPtr                   lineProc;
        QDPtr                   rectProc;
        QDPtr                   rRectProc;
        QDPtr                   ovalProc;
        QDPtr                   arcProc;
        QDPtr                   polyProc;
        QDPtr                   rgnProc;
        QDPtr                   bitsProc;
        QDPtr                   commentProc;
        QDPtr                   txMeasProc;
        QDPtr                   getPicProc;
        QDPtr                   putPicProc;
};

typedef struct QDProcs          QDProcs;
typedef struct QDProcs *        QDProcsPtr;
```

```
struct GrafPort {
        short           device;
        BitMap          portBits;
        Rect            portRect;
        RgnHandle       visRgn;
        RgnHandle       clipRgn;
        Pattern         bkPat;
        Pattern         fillPat;
        Point           pnLoc;
        Point           pnSize;
        short           pnMode;
        Pattern         pnPat;
        short           pnVis;
        short           txFont;
        Style           txFace;
        short           txMode;
        short           txSize;
        long            spExtra;
        long            fgColor;
        long            bkColor;
        short           colorBit;
        short           patStretch;
        QDHandle        picSave;
        QDHandle        rgnSave;
        QDHandle        polySave;
        QDProcsPtr      grafProcs;
};

typedef struct GrafPort         GrafPort;
typedef struct GrafPort *       GrafPtr;

#ifndef __DRIVER
GrafPtr                         thePort;
Pattern                         white;
Pattern                         black;
Pattern                         gray;
Pattern                         ltGray;
Pattern                         dkGray;
Cursor                          arrow;
BitMap                          screenBits;
long                            randSeed;
#endif
```

3. Functions

3.1 GrafPort Routines

pascal void *InitGraf* (globalPtr)
QDPtr globalPtr;

pascal void *OpenPort* (gp)
GrafPtr gp;

pascal void *InitPort* (gp)
GrafPtr gp;

pascal void *ClosePort* (gp)
GrafPtr gp;

pascal void *SetPort* (gp)
GrafPtr gp;

pascal void *GetPort* (gp)
GrafPtr *gp;

pascal void *GrafDevice* (device)
short device;

pascal void *SetPortBits* (bmPtr)
BitMap * bmPtr;

pascal void *PortSize* (width, height)
short width, height;

pascal void *MovePortTo* (leftGlobal, topGlobal)
short leftGlobal, topGlobal;

pascal void *SetOrigin* (h, v)
short h, v;

pascal void *SetClip* (rgn)
RgnHandle rgn;

pascal void *GetClip* (rgn)
RgnHandle rgn;

pascal void *ClipRect* (rPtr)
Rect * rPtr;

pascal void *BackPat* (pat)
Pattern pat;

3.1.1 Cursor Handling

pascal void *InitCursor* ()

pascal void *SetCursor* (crsrPtr)
Cursor * crsrPtr;

pascal void *HideCursor* ()

pascal void *ShowCursor* ()

pascal void *ObscureCursor* ()

3.1.2 Pen and Line Drawing

pascal void *HidePen* ()

pascal void *ShowPen* ()

pascal void *GetPen* (pt)
Point * pt;

pascal void *GetPenState* (pnStatePtr)
PenState * pnStatePtr;

pascal void *SetPenState* (pnStatePtr)
PenState * pnStatePtr;

pascal void *PenSize* (width, height)
short width, height;

pascal void *PenMode* (mode)
short mode;

pascal void *PenPat* (pat)
Pattern pat;

pascal void *PenNormal* ()

pascal void *MoveTo* (h, v)
short h, v;

pascal void *Move* (dh, dv)
short dh, dv;

pascal void *LineTo* (h, v)
short h, v;

pascal void *Line* (dh, dv)
short dh, dv;

3.2 Text Drawing

pascal void *TextFont* (font)
short font;

pascal void *TextFace* (face)
Style face;

pascal void *TextMode* (mode)
short mode;

pascal void *TextSize* (size)
short size;

pascal void *SpaceExtra* (extra)
short extra;

pascal void *DrawChar* (ch)
char ch;

pascal void *DrawString* (s)
Str255 s;

pascal void *DrawText* (textBuf, firstByte, byteCount)
QDPtr textBuf; short firstByte, byteCount;

pascal short *CharWidth* (ch)
char ch;

pascal short *StringWidth* (s)
Str255 s;

pascal short *TextWidth* (textBuf, firstByte, byteCount)
QDPtr textBuf; short firstByte, byteCount;

pascal void *GetFontInfo* (infoPtr)
FontInfo * infoPtr;

pascal void *MeasureText* (count, textAddr, charLocs)
short count; Ptr textAddr, charLocs;

3.3 Drawing in Color

pascal void *ForeColor* (color)
long color;

pascal void *BackColor* (color)
long color;

pascal void *ColorBit* (whichBit)
short whichBit;

3.4 Calculations with Rectangles

pascal void *SetRect* (rPtr, left, top, right, bottom)
Rect * rPtr; short left, top, right, bottom;

pascal void *OffsetRect* (rPtr, dh, dv)
Rect * rPtr; short dh, dv;

pascal void *InsetRect* (rPtr, dh, dv)
Rect * rPtr; short dh, dv;

pascal Boolean *SectRect* (srcRectAptr, srcRectBptr, dstRectPtr)
Rect * srcRectAptr, * srcRectBptr, * dstRectPtr;

pascal void *UnionRect* (srcRectAptr, srcRectBptr, dstRectPtr)
Rect * srcRectAptr, * srcRectBptr, *dstRectPtr;

```
pascal Boolean PtInRect ( pass(pt), rPtr )
    Point pt; /*** This Point must be cast to a long ***/
    Rect * rPtr;

pascal void Pt2Rect ( pass(ptA), pass(ptB), dstRectPtr )
   Point ptA, ptB; /*** This Point must be cast to a long ***/
    Rect * dstRectPtr;

pascal void PtToAngle ( rPtr, pass(pt), angle )
    Rect * rPtr; short * angle;
    Point pt; /*** This Point must be cast to a long ***/

pascal Boolean EqualRect ( rectAptr, rectBptr )
    Rect * rectAptr, * rectBptr;

pascal Boolean EmptyRect ( rPtr )
    Rect * rPtr;
```

3.5 Graphic Operations on Rectangles

```
pascal void FrameRect ( rPtr )
    Rect * rPtr;

pascal void PaintRect ( rPtr )
    Rect * rPtr;

pascal void EraseRect ( rPtr )
    Rect * rPtr;

pascal void InvertRect ( rPtr )
    Rect * rPtr;

pascal void FillRect ( rPtr, pat )
    Rect * rPtr; Pattern pat;
```

3.6 Graphic Operations on Ovals

```
pascal void FrameOval ( rPtr )
    Rect * rPtr;

pascal void PaintOval ( rPtr )
    Rect * rPtr;

pascal void EraseOval ( rPtr )
    Rect * rPtr;

pascal void InvertOval ( rPtr )
    Rect * rPtr;

pascal void FillOval ( rPtr, pat )
    Rect * rPtr; Pattern pat;
```

3.7 Graphic Operations on Round-Corner Rectangles

pascal void *FrameRoundRect* (rPtr, ovalWidth, ovalHeight)
Rect * rPtr; short ovalWidth, ovalHeight;

pascal void *PaintRoundRect* (rPtr, ovalWidth, ovalHeight)
Rect * rPtr; short ovalWidth, ovalHeight;

pascal void *EraseRoundRect* (rPtr, ovalWidth, ovalHeight)
Rect * rPtr; short ovalWidth, ovalHeight;

pascal void *InvertRoundRect* (rPtr, ovalWidth, ovalHeight)
Rect * rPtr; short ovalWidth, ovalHeight;

pascal void *FillRoundRect* (rPtr, ovalWidth, ovalHeight, pat)
Rect * rPtr; short ovalWidth, ovalHeight; Pattern pat;

3.8 Graphic Operations on Arcs and Wedges

pascal void *FrameArc* (rPtr, startAngle, arcAngle)
Rect * rPtr; short startAngle, arcAngle;

pascal void *PaintArc* (rPtr, startAngle, arcAngle)
Rect * rPtr; short startAngle, arcAngle;

pascal void *EraseArc* (rPtr, startAngle, arcAngle)
Rect * rPtr; short startAngle, arcAngle;

pascal void *InvertArc* (rPtr, startAngle, arcAngle)
Rect * rPtr; short startAngle, arcAngle;

pascal void *FillArc* (rPtr, startAngle, arcAngle, pat)
Rect * rPtr; short startAngle, arcAngle; Pattern pat;

3.9 Calculations with Regions

pascal RgnHandle *NewRgn* ()

pascal void *DisposeRgn* (rgn)
RgnHandle rgn;

pascal void *CopyRgn* (srcRgn, dstRgn)
RgnHandle srcRgn, dstRgn;

pascal void *SetEmptyRgn* (rgn)
RgnHandle rgn;

pascal void *SetRectRgn* (rgn, left, top, right, bottom)
RgnHandle rgn; short left, top, right, bottom;

pascal void *RectRgn* (rgn, rPtr)
RgnHandle rgn; Rect * rPtr;

pascal void *OpenRgn* ()

pascal void *CloseRgn* (dstRgn)
RgnHandle dstRgn;

pascal void *OffsetRgn* (rgn, dh, dv)
RgnHandle rgn; short dh, dv;

pascal void *InsetRgn* (rgn, dh, dv)
RgnHandle rgn;

pascal void *SectRgn* (srcRgnA, srcRgnB, dstRgn)
RgnHandle srcRgnA, srcRgnB, dstRgn;

pascal void *UnionRgn* (srcRgnA, srcRgnB, dstRgn)
RgnHandle srcRgnA, srcRgnB, dstRgn;

pascal void *DiffRgn* (srcRgnA, srcRgnB, dstRgn)
RgnHandle srcRgnA, srcRgnB, dstRgn;

pascal void *XorRgn* (srcRgnA, srcRgnB, dstRgn)
RgnHandle srcRgnA, srcRgnB, dstRgn;

pascal Boolean *PtInRgn* (pass(pt), rgn)
Point pt; /*** This Point must be cast to a long ***/
RgnHandle rgn;

pascal Boolean *RectInRgn* (rPtr, rgn)
Rect * rPtr; RgnHandle rgn;

pascal Boolean *EqualRgn* (rgnA, rgnB)
RgnHandle rgnA, rgnB;

pascal Boolean *EmptyRgn* (rgn)
RgnHandle rgn;

3.10 Graphic Operations on Regions

pascal void *FrameRgn* (rgn)
RgnHandle rgn;

pascal void *PaintRgn* (rgn)
RgnHandle rgn;

pascal void *EraseRgn* (rgn)
RgnHandle rgn;

pascal void *InvertRgn* (rgn)
RgnHandle rgn;

pascal void *FillRgn* (rgn, pat)
RgnHandle rgn; Pattern pat;

3.11 Bit Transfer Operations

pascal void *ScrollRect* (rPtr, dh, dv, updateRgn)
Rect * rPtr; short dh, dv; RgnHandle updateRgn;

```
pascal void CopyBits ( srcBitsPtr, dstBitsPtr,
                       srcRectPtr, dstRectPtr,
                       mode, maskRgn )
    BitMap * srcBitsPtr, * dstBitsPtr;
    Rect * srcRectPtr, * dstRectPtr;
    short mode; RgnHandle maskRgn;

pascal void CopyMask ( srcBits, maskBits, dstBits,
                                          srcRect, maskRect, dstRect)
    BitMap * srcBits, * maskBits, * dstBits;
    Rect   * srcRect, * maskRect, * dstRect;
```

3.12 Pictures

```
pascal PicHandle OpenPicture ( picFramePtr )
    Rect * picFramePtr;

pascal void PicComment ( kind, datasize, dataHandle )
    short kind, datasize; QDHandle dataHandle;

pascal void ClosePicture ()

pascal void DrawPicture ( myPicture, dstRectPtr )
    PicHandle myPicture; Rect * dstRectPtr;

pascal void KillPicture ( myPicture )
    PicHandle myPicture;
```

3.13 Calculations with Polygons

```
pascal PolyHandle OpenPoly ()

pascal void ClosePoly ()

pascal void KillPoly ( poly )
    PolyHandle poly;

pascal void OffsetPoly ( poly, dh, dv )
    PolyHandle poly; short dh, dv;
```

3.14 Graphic Operations on Polygons

```
pascal void FramePoly ( poly )
    PolyHandle poly;

pascal void PaintPoly ( poly )
    PolyHandle poly;

pascal void ErasePoly ( poly )
    PolyHandle poly;

pascal void InvertPoly ( poly )
    PolyHandle poly ;
```

```
pascal void FillPoly ( poly, pat )
    PolyHandle poly; Pattern pat;
```

3.15 Calculations with Points

```
pascal void AddPt ( pass(srcPt), dstPtPtr )
    Point srcPt;  /*** This Point must be cast to a long ***/
    Point * dstPtPtr;

pascal void SubPt ( pass(srcPt), dstPtPtr )
    Point srcPt; /*** This Point must be cast to a long ***/
    Point  * dstPtPtr;

pascal void SetPt ( ptPtr, h, v )
    Point * ptPtr; short h, v;

pascal Boolean EqualPt ( pass(ptA), pass(ptB) )
    Point ptA, ptB; /*** These Points must be cast to longs ***/

pascal void LocalToGlobal ( ptPtr )
    Point * ptPtr;

pascal void GlobalToLocal ( ptPtr )
    Point * ptPtr;
```

3.16 Miscellaneous Utilities

```
pascal short Random ()

pascal Boolean GetPixel ( h, v )
    short h, v;

pascal void StuffHex ( thingPtr, s )
    QDPtr thingPtr; Str255 s;

pascal void ScalePt ( ptPtr, srcRectPtr, dstRectPtr )
    Point *ptPtr;  Rect * srcRectPtr, * dstRectPtr;

pascal void MapPt ( ptPtr, srcRectPtr, dstRectPtr )
    Point *ptPtr;  Rect * srcRectPtr, * dstRectPtr;

pascal void MapRect ( rPtr, srcRectPtr, dstRectPtr )
    Rect * rPtr, * srcRectPtr, * dstRectPtr;

pascal void MapRgn ( rgn, srcRectPtr, dstRectPtr )
    RgnHandle rgn, Rect * srcRectPtr, * dstRectPtr;

pascal void MapPoly ( poly, srcRectPtr, dstRectPtr )
    PolyHandle poly;  Rect * srcRectPtr, * dstRectPtr;
```

3.17 Customizing QuickDraw Operations

```
pascal void SetStdProcs ( procsPtr )
    QDProcs *procsPtr;
```

```
pascal void StdText ( byteCount, textPtr, pass(numer), pass(denom))
    short byteCount; QDPtr textPtr;
    Point numer, denom; /*** These Points must be cast to longs ***/

pascal void StdLine ( pass(newPt) )
    Point newPt; /*** This Point must be cast to a long ***/

pascal void StdRect ( verb, rPtr )
    GrafVerb verb; Rect * rPtr;

pascal void StdRRect ( verb, rPtr, ovalwidth, ovalHeight )
    GrafVerb verb; Rect * rPtr; short ovalWidth, ovalHeight;

pascal void StdOval ( verb, rPtr )
    GrafVerb verb; Rect * rPtr;

pascal void StdArc ( verb, rPtr, startAngle, arcAngle )
    GrafVerb verb; Rect * rPtr; short startAngle, arcAngle;

pascal void StdPoly ( verb, poly )
    GrafVerb verb; PolyHandle poly;

pascal void StdRgn ( verb, rgn )
    GrafVerb verb; RgnHandle rgn;

pascal void StdBits ( srcBitsPtr, srcRectPtr, dstRectPtr,
                      mode, maskRgn )
    BitMap * srcBitsPtr; Rect *srcRectPtr, * dstRectPtr;
    short mode; RgnHandle maskRgn;

pascal void StdComment ( kind, dataSize, dataHandle )
    short kind, dataSize; QDHandle dataHandle;

pascal short StdTxMeasure (byteCount, textPtr
                  numer, denom, infoPtr )
    short byteCount; QDPtr textPtr;
    Point numer, denom; /*** These Points must be cast to longs ***/
    FontInfo * infoPtr;

pascal void StdGetPic ( dataPtr, byteCount )
    QDPtr dataPtr; short bytecount;

pascal void StdPutPic ( dataPtr, byteCount )
    QDPtr dataPtr; short bytecount;

pascal void SeedFill ( srcPtr, dstPtr, srcRow, dstRow, height, words,
                                                    seedH, seedV)
    Ptr srcPtr, dstPtr;          short srcRow, dstRow, height, words, seed

pascal void CalcMask ( srcPtr, dstPtr, srcRow, dstRow, height, words)
    Ptr srcPtr, dstPtr;          short srcRow, dstRow, height, words;

pascal void GetMaskTable ()
```

Resource Manager Functions

This section describes functions that allow C programs to call the Macintosh Resource Manager Routines.

The constants, data structures, and functions described in this section are defined in the header file *resource.h.*

1. Constants

```
#define resSysRef          0x80
#define resSysHeap         0x40
#define resPurgeable       0x20
#define resLocked          0x10
#define resProtected       0x08
#define resPreload         0x04
#define resChanged         0x02
#define resUser            0x01

#define mapReadOnly        0x80
#define mapCompact         0x40
#define mapChanged         0x20

typedef long ResType;
```

2. Functions

2.1 Initializing the Resource Manager

pascal short *InitResources* ()

pascal void *RsrcZoneInit* ()

2.2 Opening and Closing Resource Files

pascal void *CreateResFile* (filename)
    Str255 filename;

pascal short *OpenResFile* (filename)
    Str255 filename;

pascal short *OpenRFPerm* (filename, VRefNum, permission)
    Str255 filename; short VRefNum; Byte permission;

pascal void *CloseResFile* (refNum)
    short refNum;

2.3 Checking for errors

pascal short *ResError* ()

2.4 Setting the Current Resource File

pascal short *CurResFile* ()

pascal short *HomeResFile* (theResource)
Handle theResource;

pascal void *UseResFile* (refNum)
short refNum;

pascal short *CountTypes* ()

pascal short *Count1Types* ()

pascal void *GetIndType* (theType, index)
ResType * theType; short index;

pascal void *Get1IndType* (theType, index)
ResType * theType; short index;

2.5 Getting and Disposing of Resources

pascal void *SetResLoad* (load)
Boolean load;

pascal short *CountResources* (theType)
ResType theType;

pascal short *Count1Resources* (theType)
ResType theType;

pascal Handle *GetIndResource* (theType, index)
ResType theType; short index;

pascal Handle *Get1IndResource* (theType, index)
ResType theType; short index;

pascal Handle *GetResource* (theType, theID)
ResType theType; short theID;

pascal Handle *Get1Resource* (theType, theID)
ResType theType; short theID;

pascal Handle *GetNamedResource* (theType, name)
ResType theType; Str255 name;

pascal Handle *Get1NamedResource* (theType, name)
ResType theType; Str255 name;

pascal void *LoadResource* (theResource)
Handle theResource;

pascal void *ReleaseResource* (theResource)
Handle theResource;

pascal void *DetachResource* (theResource)
Handle theResource;

2.6 Getting Resource Information

pascal short *UniqueID* (theType)
ResType theType;

pascal short *Unique1ID* (theType)
ResType theType;

pascal void *GetResInfo* (theResource, theID, theType, name)
Handle theResource; short *theID;
ResType *theType; Str255 name;

pascal short *GetResAttrs* (theResource)
Handle theResource;

pascal long *SizeResource* (theResource)
Handle theResource;

pascal long *MaxSizeRsrc* (theResource)
Handle theResource;

pascal long *RsrcMapEntry* (theResource)
Handle theResource;

2.7 Modifying Resources

pascal void *SetResInfo* (theResource, theID, name)
Handle theResource; short theID; Str255 name;

pascal void *SetResAttrs* (theResource, attrs)
Handle theResource; short attrs;

pascal void *ChangedResource* (theResource)
Handle theResource;

pascal void *AddResource* (theData, theType, theID, name)
Handle theData; ResType theType; short theID; Str255 name;

pascal void *RmveResource* (theResource)
Handle theResource;

pascal void *UpdateResFile* (refNum)
short refNum;

pascal void *WriteResource* (theResource)
Handle theResource;

pascal void *SetResPurge* (install)
Boolean install;

2.8 Advanced Routines

pascal short *GetResFileAttrs* (refNum)
short refNum;

pascal void *SetResFileAttrs* (refNum, attrs)
short refNum, attrs;

2.9 Modifying System References

pascal void *AddReference* (theResource, theID, name)
Handle theResource; short theID; Str255 name;

pascal void *RmveReference* (theResource)
Handle theResource;

Vertical Retrace Manager Functions

This section describes functions that allow C programs to access Macintosh Vertical Retrace Manager routines.

The constants, structures, and functions described in this section are defined in the header file *retrace.h*.

1. Data Structures

```
typedef struct {
    QElemPtr        *qLink;
    short           qType;
    ProcPtr         vblAddr;
    short           vblCount;
    short           vblPhase;
} VBLTask;
```

2. Functions

2.1 Vertical Retrace Routines

pascal short *VInstall* (vblTaskPtr)
    QElemPtr vblTaskPtr;

pascal short *VRemove* (vblTaskPtr)
    QElemPtr vblTaskPtr;

pascal QHdrPtr *GetVBLQHdr* ()

Scrap Manager Functions

The functions described in this section allow C programs to call routines that are part of the Macintosh Scrap Manager.

The constants, structures, and functions described in this section are defined in the header file *scrap.h*.

1. Data Structures

```
struct ScrapStuff {
        long            scrapSize;
        Handle          scrapHandle;
        short           scrapCount;
        short           scrapState;
        StringPtr       scrapName;
};

typedef struct ScrapStuff       ScrapStuff;
typedef struct ScrapStuff *     PScrapStuff;
```

2. Functions

2.1 Getting Scrap Information

pascal PScrapStuff *InfoScrap* ()

2.2 Keeping the Scrap on the Disk

pascal long *UnloadScrap* ()

pascal long *LoadScrap* ()

2.3 Reading from the Scrap

pascal long *GetScrap* (hDest, theType, offset)
    Handle hDest; ResType theType; long *offset;

2.4 Writing to the Scrap

pascal long *ZeroScrap* ()

pascal long *PutScrap* (length, theType, source)
    long length; ResType theType; Ptr source;

Segment Loader Functions

This section describes functions that allow C programs to call Macintosh Segment Loader routines.

The functions described in this section are defined in the header file *segment.h.*

1. Constants

```
#define appOpen   0
#define appPrint  1
```

2. Functions

pascal void *LoadSeg* (segID)
    short segID;

pascal void *UnloadSeg* (routineAddr)
    Ptr routineAddr;

void *CountAppFiles* (messagePtr, countPtr)
    short * messagePtr, *countPtr;

void *GetAppFiles* (index, theFilePtr)
    short index; AppFile *theFilePtr;

void *ClrAppFiles* (index)
    short index;

pascal void *GetAppParms* (apName, apRefNumPtr, apParamPtr)
    Str255 apName; short * apRefNumPtr; Handle * apParamPtr;

pascal void *ExitToShell* ()

void *Launch* (name, sound)
    char *name; short sound;

void *Chain* (name, sound)
    char *name; short sound;

Serial Driver Functions

This section describes functions that allow C programs to access Macintosh Serial Driver routines.

The RAM serial driver is loaded into memory and installed from a resource on disk by the function *RamSDOPen*, if the system driver is version 0; it is deleted from memory by *RamSDClose*. Before running a program that calls *RamSDOpen*, you must move the resources that contain the serial driver from the distribution disk to your own disk, using either *RGen* or *cprsrc*. The resource with type=SERD and ID=1 contains the driver for the Mac; that with type=SERD and ID=2 contains the driver for the MacXL. The file on the distribution disks that contain these resources is *serial/SERD*.

The constants, structures, and functions described in this section are defined in the header file *serial.h*.

1. Constants

```
#define baud300          380
#define baud600          189
#define baud1200         94
#define baud1800         62
#define baud2400         46
#define baud3600         30
#define baud4800         22
#define baud7200         14
#define baud9600         10
#define baud19200        4
#define baud57600        0

#define stop10           0x4000
#define stop15           0x8000
#define stop20           0xc000

#define noParity         0x2000
#define oddParity        0x1000
#define evenParity       0x3000

#define data5            0x0000
#define data6            0x0800
#define data7            0x0400
#define data8            0x0c00
```

```
#define swOverrunErr        0x01
#define parityErr           0x10
#define hwOverrunErr        0x20
#define framingErr          0x40

#define ctsEvent            0x20
#define breakEvent          0x80

#define xOffWasSent         0x80

#define sPortA              0x000
#define sPortB              0x100
```

2. Data Structures

```
typedef struct {
    char            fXOn;
    char            fCTS;
    unsigned char   xOn;
    unsigned char   xOff;
    char            errs;
    char            evts;
    char            fInX;
    char            null;
} SerShk;

typedef struct {
    char            cumErrs;
    char            xOffSent;
    char            rdPend;
    char            wrPend;
    char            ctsHold;
    char            xOffHold;
} SerStaRec;
```

3. Functions

3.1 Opening and Closing the RAM Serial Driver

```
pascal short RamSDOpen ( whichPort )
  short whichPort;  /* either SPortA or SPortB */

pascal void RamSDClose ( whichPort )
  short whichPort;  /* either SPortA or SPortB */
```

3.2 Changing Serial Driver Information

```
pascal short SerReset ( refNum, serConfig )
  short refNum;
  short serConfig;

pascal short SerSetBuf ( refNum, serBPtr, serBLen)
  short refNum;
  Ptr serBPtr;
  short serBLen;

pascal short SerHShake ( refNum, flgs )
  short refNum;
  SerShk * flgs;

pascal short SerSetBrk ( refNum )
  short refNum;

pascal short SerClrBrk ( refNum )
  short refNum;
```

3.3 Getting Serial Driver Information

```
pascal short SerGetBuf ( refNum, count )
  short refNum;
  long * count;

pascal short SerErrFlg ( refNum, serSta )
  short refNum;
  SerStaRec * serSta;
```

Sound Driver Functions

This section describes functions that allow C programs to access Macintosh Sound Driver routines.

The constants, structures, and functions described in this section are defined in the header file *sound.h.*

1. Constants

```
#define swMode          -1
#define ftMode          1
#define ffMode          0
```

2. Data Structures

```
typedef char FreeWave[30001];

typedef struct {
    short               mode;
    Fixed               count;
    FreeWave            waveBytes;
} FFSynthRec, *FFSynthPtr;

typedef struct {
    short               count;
    short               amplitude;
    short               duration;
} Tone, Tones[5001];

typedef struct {
    short               mode;
    Tones               triplets;
} SWSynth, SWSynthPtr;

typedef char  Wave[256];
typedef Wave  *WavePtr;
```

```
typedef struct {
    short           duration;
    Fixed           sound1Rate;
    long            sound1Phase;
    Fixed           sound2Rate;
    long            sound2Phase;
    Fixed           sound3Rate;
    long            sound3Phase;
    Fixed           sound4Rate;
    long            sound4Phase;
    WavePtr         sound1Wave;
    WavePtr         sound2Wave;
    WavePtr         sound3Wave;
    WavePtr         sound4Wave;
} FTSoundRec, *FTSndRecPtr;

typedef struct {
    short           mode;
    FTSndRecPtr     sndRec;
} FTSynthRec, *FTSynthPtr;
```

3. Functions

3.1 Sound Functions

pascal void *SetSoundVol* (level)
    short level;

pascal void *GetSoundVol* (&levelPtr)
    short levelPtr;

pascal Boolean *SoundDone* ()

pascal void *StopSound* ()

pascal void *StartSound* (synthRec, numbytes, doneRtn)
    Ptr synthRec; long numBytes; ProcPtr doneRtn;

System Error Codes

The constants described in this section are defined in the header file *syserr.h.*

1. Constants

```
/* Macintosh OS system errors: */
#define noErr       0       /* All is well */

/*  File system error codes: */
#define qErr        (-1)
#define vTypErr     (-2)

#define dirFulErr   (-33)   /* Directory full */
#define dskFulErr   (-34)   /* disk full */
#define nsvErr      (-35)   /* no such volume */
#define ioErr       (-36)   /* I/O error */
#define bdNamErr    (-37)   /* bad name */
#define fnOpnErr    (-38)   /* File not open */
#define eofErr      (-39)   /* End of file */

#define posErr      (-40)   /* tried to position before
                               file origin */
#define mFulErr     (-41)   /* memory full (open)
                               or file won't fit (load) */

#define tmfoErr     (-42)   /* too many files open */
#define fnfErr      (-43)   /* File not found */

#define wPrErr      (-44)   /* diskette is write protected */
#define fLckdErr    (-45)   /* file is locked */
#define vLckdErr    (-46)   /* volume is locked */
#define fBsyErr     (-47)   /* File is busy (delete) */
#define dupFNErr    (-48)   /* duplicate filename (rename) */
#define opWrErr     (-49)   /* file already open with
                               with write permission */
```

```
#define paramErr        (-50)   /* error in user parameter list */
#define rfNumErr        (-51)   /* refnum error */
#define gfpErr          (-52)   /* get file position error */
#define volOffLinErr    (-53)   /* volume not on line error
                                   (was Ejected) */
#define permErr         (-54)   /* permissions error
                                   (on file open) */
#define volOnLinErr     (-55)   /* drive volume already on-line
                                   at MountVol */
#define nsDrvErr        (-56)   /* no such drive
                                   (tried to mount a bad drive #) */
#define noMacDskErr     (-57)   /* not a mac diskette
                                   (sig bytes are wrong) */
#define extFSErr        (-58)   /* volume in question belongs
                                   to an external fs */
#define fsRnErr         (-59)   /* Problem during rename */

#define badMDBErr       (-60)   /* bad master directory block */
#define wrPermErr       (-61)   /* write permissions error */

#define clkRdErr        (-85)
#define clkWrErr        (-86)
#define prWrErr         (-87)
#define prInitErr       (-88)

#define PortInUse       (-97)   /* some other driver
                                   is currently using this port */
#define PortNotCf       (-98)   /* parameter ram is set
                                   for some other type use */

#define noScrapErr      (-100)
#define noTypeErr       (-102)

#define memFullErr      (-108)  /* Not enough room in heap zone */
#define nilHandleErr    (-109)  /* Master Pointer was
                                   NIL in HandleZone */
#define memWZErr        (-111)  /* WhichZone failed
                                   (applied to free block) */
#define memPurErr       (-112)  /* trying to purge a locked or
                                   non-purgeable block */

#define resNotFound     (-192)
#define resFNotFound    (-193)
#define addResFailed    (-194)
#define addRefFailed    (-195)
#define rmvResFailed    (-196)
#define rmvRefFailed    (-197)
```

TextEdit Functions

This section describes functions that allow a program to access routines in the the Macintosh TextEdit package.

The constants, data structures, and functions described in this section are defined in the header file *textedit.h.*

1. Constants

```
#define teJustLeft      0
#define teJustCenter    1
#define teJustRight     -1
```

2. Data Structures

```
typedef char        Chars[32001];
typedef Chars *     CharsPtr;
typedef Chars **    CharsHandle;
```

```
struct TERec {
        Rect            destRect;
        Rect            viewRect;
        Rect            selRect;
        short           lineHeight;
        short           fontAscent;
        Point           selPoint;
        short           selStart;
        short           selEnd;
        short           active;
        long            wordBreak;
        long            clikLoop;
        long            clickTime;
        short           clickLoc;
        long            caretTime;
        short           caretState;
        short           just;
        short           length;
        Handle          htext;
        short           recalBack;
        short           recalLines;
        short           clikStuff;
        short           crOnly;
        short           txFont;
        short           txFace;
        short           txMode;
        short           txSize;
        GrafPtr         inPort;
        Ptr             highHook;
        Ptr             caretHook;
        short           nLines;
        short           lineStarts[32001];
};

typedef struct TERec        TERec;
typedef struct TERec *      TEPtr;
typedef struct TERec **     TEHandle;
```

3. Functions

3.1 Initialization

pascal void *TEInit* ()

pascal TEHandle *TENew* (destRectPtr, viewRectPtr)
    Rect * destRectPtr, * viewRectPtr;

pascal void *TEDispose* (hTE)
TEHandle hTE;

3.2 Manipulating Edit Records

pascal void *TESetText* (text, length, hTE)
Ptr text; long length; TEHandle hTE;

pascal CharsHandle *TEGetText* (hTE)
TEHandle hTE;

3.3 Editing

pascal void *TEKey* (key, hTE)
char key; TEHandle hTE;

pascal void *TECut* (hTE)
TEHandle hTE;

pascal void *TECopy* (hTE)
TEHandle hTE;

pascal void *TEPaste* (hTE)
TEHandle hTE;

pascal void *TEDelete* (hTE)
TEHandle hTE;

pascal void *TEInsert* (text, length, hTE)
Ptr text; long length; TEHandle hTE;

3.4 Selection Range and Justification

pascal void *TESetSelect* (selStart, selEnd, hTE)
long selStart, selEnd; TEHandle hTE;

pascal void *TESetJust* (j, hTE)
short j; TEHandle hTE;

pascal void *TESelView* (hTE)
TEHandle hTE;

pascal void *TEAutoView* (auto, hTE)
Boolean auto; TEHandle hTE;

3.5 Mice and Carets

pascal void *TEClick* (pass(pt), extend, hTE)
Point pt; /*** This Point must be cast to a long ***/
Boolean extend; hTE TEHandle;

pascal void *TEIdle* (hTE)
TEHandle hTE;

pascal void *TEActivate* (hTE)
TEHandle hTE;

pascal void *TEDeactivate* (hTE)
TEHandle hTE;

3.6 Text Display

pascal void *TEUpdate* (rUpdatePtr, hTE)
Rect * rUpdatePtr; TEHandle hTE;

pascal void *TextBox* (text, length, boxPtr, j)
Ptr text; long length; Rect * boxPtr; short j;

3.7 Advanced Routines

pascal void *TEScroll* (dh, dv, hTE)
short dh, dv; TEHandle hTE;

pascal void *TEPinScroll* (dh, dv, hTE)
short dh, dv; TEHandle hTE;

pascal void *TECalText* (hTE)
TEHandle hTE;

3.8 Scrap-related Functions

pascal short *TEFromScrap* ( )

pascal short *TEToScrap* ( )

Handle *TEScrapHndl* ()

long *TEGetScrpLen* ()

pascal void *TESetScrpLen* (len)
long len;

Toolbox Utility Functions

The functions described in this section allow a C program to access Macintosh Toolbox utility routines.

The constants, structures, and functions described inthis section are defined in the header file *toolutil.h*.

1. Constants

```
#define sysPatListID        0
```

2. Data structures

```
typedef long                Fixed;

struct Int64Bit {
        long                hiLong;
        long                loLong;
};

typedef struct Int64Bit     Int64Bit;

typedef struct Cursor *     CursPtr;
typedef struct Cursor **    CursHandle;

typedef struct Pattern *    PatPtr;
typedef struct Pattern **   PatHandle;
```

3. Functions

3.1 Fixed-Point Arithmetic

```
pascal Fixed FixRatio ( numerator, denominator )
    short numerator, denominator;

pascal Fixed FixMul ( a, b )
    Fixed a, b;

pascal short FixRound ( x )
    Fixed x;

pascal Fixed Long2Fix ( x )
    long x;

pascal long Fix2Long ( x )
    Fixed x;

pascal Fract Fix2Fract ( x )
    Fixed x;
```

pascal Fixed *Fract2Fix* (x)
Fract x;

pascal Fract *FracCos* (x)
Fixed x;

pascal Fract *FracSin* (x)
Fixed x;

pascal Fract *FracSqrt* (x)
Fract x;

pascal Fract *FracMul* (x, y)
Fract x, y;

pascal Fixed *FracDiv* (x, y)
Fract x, y;

pascal Fixed *FixAtan2* (x, y)
long x, y;

pascal Fixed *FixDiv* (x, y)
Fixed x, y;

3.2 String Manipulation

pascal StringHandle *NewString* (s)
Str255 s;

pascal void *SetString* (h, s)
StringHandle h; Str255 s;

pascal StringHandle *GetString* (stringID)
short stringID;

pascal void *GetIndString* (theString strListID, index)
Str255 theString; short strListID, index;

3.3 Byte Manipulation

pascal long *Munger* (h, offset, ptrl, lenl, ptr2, len2)
Handle h; Ptr ptrl, ptr2;
long offset, lenl, len2;

pascal Boolean *BitTst* (bytePtr, bitNum)
Ptr bytePtr; long bitNum;

pascal void *BitSet* (bytePtr, bitNum)
Ptr bytePtr; long bitNum;

pascal void *BitClr* (bytePtr, bitNum)
Ptr bytePtr; long bitNum;

3.4 Logical Functions

pascal long *BitAnd* (long1, long2)
long long1, long2;

pascal long *BitOr* (long1, long2)
long long1, long2;

pascal long *BitXor* (long1, long2)
long long1, long2;

pascal long *BitNot* (long1, long2)
long long1, long2;

pascal long *BitShift* (long1, count)
long long1; short count;

pascal void *PackBits* (srcPtr, dstPtr, srcBytes)
Ptr * srcPtr, * dstPtr; short srcBytes;

pascal void *UnPackBits* (srcPtr, dstPtr, dstBytes)
Ptr * srcPtr, * dstPtr; short srcBytes;

3.5 Other Operations on Long Integers

pascal short *HiWord* (x)
long x;

pascal short *LoWord* (x)
long x;

pascal void *LongMul* (a, b, destPtr)
long a, b; Int64Bit * destPtr;

3.6 Graphics Utilities

pascal Handle *GetIcon* (iconID)
short iconID;

pascal void *PlotIcon* (theRectPtr, theIcon)
Rect * theRectPtr; Handle theIcon;

pascal PatHandle *GetPattern* (patID)
short patID;

pascal void *GetIndPattern* (thePattern, patID, index)
Pattern thePattern;
short patID, index;

pascal CursHandle *GetCursor* (cursorID)
short cursorID;

pascal void *ShieldCursor* (left, top, right, bottom)
short left, top, right, bottom;

```
pascal PicHandle GetPicture ( pictureID )
    short pictureID;
```

3.7 Miscellaneous Utilities

```
pascal long DeltaPoint ( pass(ptA), pass(ptB) )
    Point ptA, ptB;

pascal Fixed SlopeFromAngle ( angle )
    short angle;

pascal short AngleFromSlope ( slope )
    Fixed slope;
```

Types

This section describes definitions that are common to most C programs.

The constants, structures, and functions described in this section are defined in the header file *types.h.*

1. Constants

```
#define TRUE                    (-1)
#define FALSE                   0

typedef unsigned                charByte;
typedef char                    SignedByte;
typedef char                    Ptr;
typedef Ptr *                   Handle;
typedef short                   (*ProcPtr)();
typedef char                    Boolean;

typedef unsigned char           Str255[256];
typedef Str255 *                StringPtr;
typedef Str255 **               StringHandle;

typedef short                   OSErr;
typedef long                    OSType;
```

Window Manager Functions

This section describes functions that allow C programs to call the Macintosh Window Manager routines.

The constants, data structures, and functions described in this section are defined in the header file *window.h.*

1. Constants

```
#define documentProc        0
#define dBoxProc            1
#define plainDBox           2
#define altDBoxProc         3
#define noGrowDocProc       4
#define rDocProc            16

#define dialogKind          2
#define userKind            8

#define inDesk              0
#define inMenuBar           1
#define inSysWindow         2
#define inContent           3
#define inDrag              4
#define inGrow              5
#define inGoAway            6

#ifndef hAxisOnly
#define noConstraint        0
#define hAxisOnly           1
#define vAxisOnly           2

#define wDraw               0
#define wHit                1
#define wCalcRgns           2
#define wNew                3
#define wDispose            4
#define wGrow               5
#define wDrawGIcon          6

#define wNoHit              0
#define wInContent          1
#define wInDrag             2
#define wInGrow             3
#define wInGoAway           4
```

2. Data structures

```
typedef struct WindowRecord *WindowPeek;

struct WindowRecord {
        GrafPort            port;
        short               windowKind;
        char                visible;
        char                hilited;
        char                goAwayFlag;
        char                spareFlag;
        RgnHandle           strucRgn;
        RgnHandle           contRgn;
        RgnHandle           updateRgn;
        Handle              windowDefProc;
        Handle              dataHandle;
        StringHandle        titleHandle;
        short               titleWidth;
        Handle              controlList;
        WindowPeek          nextWindow;
        PicHandle           windowPic;
        long                refCon;
};

typedef struct WindowRecord  WindowRecord;
typedef GrafPtr              WindowPtr;
```

3. Functions

3.1 Initialization and Allocation

```
pascal void InitWindows ()

pascal void GetWMgrPort ( wPortPtr )
    GrafPtr * wPortPtr;

pascal WindowPtr NewWindow ( wStorage, boundsRectPtr, title
                             visible, procID, behind,
                             goAwayFlag, refCon )
    Ptr wStorage; Rect *boundsRectPtr; Str255 title;
    Boolean visible, goAwayFlag; short procID; WindowPtr behind;
    long refCon;

pascal WindowPtr GetNewWindow ( windowID, wStorage, behind )
    short windowID; Ptr wStorage; WindowPtr behind;

pascal void CloseWindow ( theWindow )
    WindowPtr theWindow;
```

pascal void *DisposeWindow* (theWindow)
WindowPtr theWindow;

3.2 Window Display

pascal void *SetWTitle* (theWindow, title)
WindowPtr theWindow; Str255 title;

pascal void *GetWTitle* (theWindow, title)
WindowPtr theWindow; Str255 title;

pascal void *SelectWindow* (theWindow)
WindowPtr theWindow;

pascal void *HideWindow* (theWindow)
WindowPtr theWindow;

pascal void *ShowWindow* (theWindow)
WindowPtr theWindow;

pascal void *ShowHide* (theWindow, showFlag)
WindowPtr theWindow; Boolean showFlag;

pascal void *HiliteWindow* (theWindow, fHiLite)
WindowPtr theWindow; Boolean fHiLite;

pascal void *BringToFront* (theWindow)
WindowPtr theWindow;

pascal void *SendBehind* (theWindow, behindWindow)
WindowPtr theWindow, behindWindow;

pascal WindowPtr *FrontWindow* ( )

pascal void *DrawGrowIcon* (theWindow)
WindowPtr theWindow;

3.3 Mouse Location

pascal short *FindWindow* (pass(thePt), whichWindowPtr)
Point thePt; /*** This Point must be cast to a long ***/
WindowPtr * whichWindowPtr;

pascal Boolean *TrackGoAway* (theWindow, pass(thePt))
WindowPtr theWindow;
Point thePt; /*** This Point must be cast to a long ***/

3.4 Window Movement and Sizing

pascal void *MoveWindow* (theWindow, hGlobal, vGlobal, front)
WindowPtr theWindow; short hGlobal, vGlobal; Boolean front;

TECHNICAL INFORMATION

Chapter Contents

Technical Information

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

It's divided into the following sections:

1. *Memory Organization.* Describes how RAM memory is used on the Macintosh, and the organization of a command program that has been created by the Aztec linker.

2. *Command Programs.* Describes the different types of command programs that can be created using Aztec C68K: their features, how they are created, and how they are used.

3. *Drivers and Desk Accessories.* Describes how to create and use drivers and desktop accessories.

4. *The Console Driver.* Describes the Aztec console driver: what it is, and how the operator and programs use it.

5. *Using Aztec C68K with a 128K Macintosh.*

6. *Using Aztec C68K with a 512K Macintosh.*

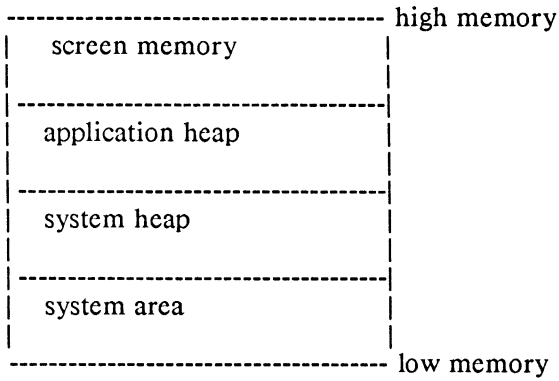
7. *Using Aztec C68K with a hard disk.*

8. *Using Aztec C68K on a single-drive Macintosh.*

9. *Data formats.* Describes the format of the data items supported by Aztec C68K.

1. Memory organization

In the Macintosh's address space, RAM memory is organized as follows:



The system area contains interrupt vectors and data used by the operating system, Finder, and SHELL.

The system heap contains the operating system; drivers and desktop accessories which you create can also be loaded here, if you want them to be permanently resident.

The application heap is the area designated for use by application programs. Drivers and desktop accessories can also be loaded into it. The area is described in detail below.

The Macintosh's screen is accessed by setting and resetting bits in the screen memory area.

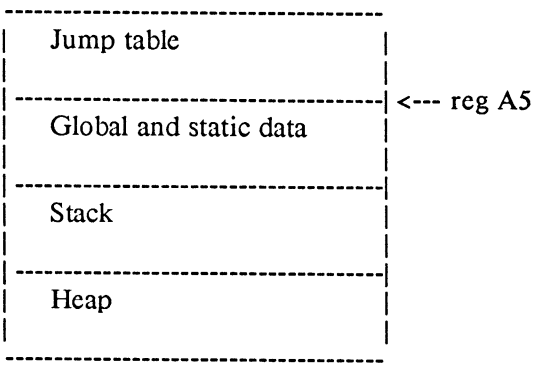
The application heap

The SHELL, Finder, and all programs activated by them reside in the application heap area, as do their stack, global and static data, and dynamically allocated buffers. The SHELL and Finder also store data in the system area and system heap.

When a program running in this area terminates, the area is reinitialized; hence it's not possible for programs to store data in this area for later use by other programs.

Drivers and desktop accessories can be loaded into the application heap or the system heap. They can only be permanently located in the system heap, since each time an application program terminates, the application heap is deallocated.

The application heap is organized as follows:



Jump table

The jump table provides the mechanism which allows a code segment to be automatically loaded when one of its functions is called, and which allows a single code segment to be any size.

A program's jump table is located at the high end of the application heap. Register A5 points to its first entry, plus 32.

Automatic loading of segments

The jump table contains a set of entries. Each function which is called from another segment has an entry in the table.

When a call is made to a function located in another segment, the call is actually made to the function's jump table entry. The entry causes the function's segment to be loaded, if it's not already loaded, and then jumps to it.

To start a program, the SHELL opens the resource file containing it, loads its jump table from the resource of type 'CODE' and id 0, and jumps to its first entry. Since no code segments have been loaded, this forces the segment containing the function corresponding to the table's first entry to be loaded. This function is then called.

The function corresponding to the jump table's first entry thus performs the program's initialization activities; when done, it then calls the program's *main* function.

The next section of this chapter describes the Manx-supplied startup functions.

Large sized code segments

When a call is made to a function located in the same segment as the caller, and the addresses of the caller and the function are within 32K bytes of each other, the function is called directly.

If the two addresses are more than 32K bytes apart, the caller will call a jump table entry created especially for this call, which will call

the function.

Global and static data area

This area contains a program's initialized and uninitialized global and static data.

When a program starts, the Manx-supplied startup code for the program clears its uninitialized data area and loads the program's initialized data from the resource having type CODE and id 256. This resource resides in the resource file containing the program's code.

The size of this area is set to the sum of the sizes of a program's initialized and uninitialized data, and can be up to 32K bytes long.

The area begins just below the program's jump table. Entries in it are accessed using a negative offset from register A5.

The stack area

A program's stack resides in the application heap's stack area. By default, the stack area is 8K bytes. This can be overridden for a particular program by using the -S option when linking the program.

The size of one program's stack area has no relationship to that of another. Thus, if a program is executed which has a stack area other than 8k bytes, and if the next program executed doesn't specify its stack size, the stack area for the second program will be the default size, 8K bytes.

The stack area begins at the base of a program's global and static data area.

The heap

The heap is the area into which code segments are loaded and from which buffers are dynamically allocated. Drivers and desktop accessories are also loaded into this area.

The heap occupies the area between the beginning of the application heap and the base of the stack area.

2. Command programs

This section discusses command programs. It's divided into three subsections: the first describes the different types of command programs, and how they are created. The second describes how you can substitute your own startup routines for ours in a command program. The third describes how open files and devices are passed between programs.

2.1 Creating command programs

2.1.1 Introduction

Three types of command programs can be created with the standard Aztec software. The type of a particular commmand program is determined by the version of the function *Croot* that it contains. This function performs initialization activities that selects the program's characteristics, and then calls the program's *main* function.

The versions of the *Croot* functions, and the principal characteristics that they give to a command program are:

shcroot Program can be activated by the SHELL, but not by the Finder. The program can use many features that are available to programs on a UNIX system. The name *shcroot* is an acronym for "root function for SHELL-activated C program".

sacroot Program can be activated by the Finder as well as the SHELL. It can't perform UNIX-style I/O to the console. The name *sacroot* is an acronym for "root function for stand-alone C program".

mixcroot Program can be activated by the Finder as well as the SHELL. It can perform UNIX-style I/O to the console. The name *mixcroot* is an acronym for "root function for mixed-mode C programs".

To write command programs containing *shcroot*, you don't need any special knowledge about the Macintosh. Thus, when you are first learning C or getting familiar with the Aztec C68K package, you'll probably want to create programs that contain this version of *Croot*.

Programs containing *sacroot* or *mixcroot* are expected to be Macintosh-specific. These versions of *Croot* give fewer UNIX features to a program, which makes it easier for a program to get at the special features of the Macintosh.

The different versions of *Croot* are provided in both source and object form. The object code for *shcroot* is contained in *c.lib*; the object code for *sacroot* and *mixcroot* are in *sacroot.o* and *mixcroot.o*, respectively. If you don't explicitly request the linker to include a module containing *Croot*, it will include *shcroot* from *c.lib*. That is, by

default, a program will have the characteristics determined by *shcroot.*

You can also include your own version of *Croot* instead of ours, if desired, to give your program characteristics not provided by our *Croot* functions.

2.1.1.1 Special linker options for command programs

The following special linker options can be used when linking command programs:

-M Causes the file containing the program to be of type 'APPL', thus allowing it to be started by either the Finder or the SHELL. If *-M* isn't specified (and if the -D and -A options, which are used when creating drivers and desktop accessories, are also not specified) the type of the file containing the program is 'AZTC'; programs of this type can be started by the SHELL but not by the Finder.

+O Used to segment a command program's code. Without this option, a command program's code is unsegmented. Creation of segmented command programs is discussed below.

2.1.1.2 Global and static data

A command program can contain up to 32 Kbytes of global and static data. The data can be uninitialized or initialized. For example, the following is an uninitialized global variable:

```
int a;
```

The following is an initialized global variable:

```
int a=1;
```

Uninitialized global and static variables are automatically cleared. Initialized variables can be initialized to constants or to the addresses of memory locations whose addresses depend on the address at which a program is loaded.

Each of a command program's code segments can be any size. Segment 0 can contain references to memory locations whose addresses aren't known until the program is loaded.

Memory references in a program's initialized data or its first code segment are automatically adjusted when the program is started.

2.1.1.3 The console driver

As mentioned above, a command program that has been linked with *shcroot* or *mixcroot* can talk to the screen or keyboard using UNIX I/O functions. When such a command program issues a UNIX I/O call to the screen or keyboard, the Aztec console driver performs the I/O

for the program. The console driver is a Macintosh resource, and is independent of the command programs that call it. That is, the console driver is not linked into the command programs that call it.

For more information, see the section on the console driver in the Technical Information chapter and the overview of console I/O in the Library Functions chapter.

2.1.2 Features of the different types of command programs

2.1.2.1 Command programs containing *shcroot*

A command program containing *shcroot* has the following features:

* It can be activated by the SHELL, but not by the Finder. It can also be activated by another command program; in this case, the SHELL must have been the last command-processor-type program to have been executed.
* It can be passed arguments when started by either the SHELL or by another program's *exec* call. It receives the arguments in the standard UNIX way; that is, as arguments to its *main* function.
* It can access the standard i/o devices standard in, standard out, and standard error. By default, these are connected to the console, and can be redirected to another device or file, if desired;
* The screen is automatically initialized for the program: *InitGraf* is called, and the entire screen is made the current window. The screen isn't cleared;
* Console i/o is handled by the Manx-supplied driver, *.con*. This driver is automatically loaded into memory when needed;
* It supports the SHELL's hierarchical file system. For example, it can access files in the current directory without having to specify the path to the directory; if a file is located on the current volume, the program needn't specify the volume; and so on.

To create a command program that contains *shcroot*, just link the program (1) without specifying the -M, -D, or -A options, (2) with *c.lib*, and (3) without specifying a module containing another *Croot*. If, in addition, you link the program without specifying any '+O' options, the program will contain a single code segment.

For example, the following command links the "hello, world" program, whose object module is in the file *hello.o* that is in the current directory, writing the executable code to the file *hello* that is also in the current directory:

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

This command has a single code segment and can be activated by the SHELL, but not by the Finder.

2.1.2.2 Command programs containing *sacroot*

A command program containing *sacroot* is generally linked with the '-M' option, which allows it to be activated by the Finder. Such a program has the following features:

* It can be activated by the Finder, SHELL, or any command program;
* It can be passed arguments. It receives them using the standard Macintosh conventions, and not as arguments to its *main* function.
* No unexpected screen initialization is automatically done for it: neither *InitGraf* nor *InitWindow* is called, and the contents of the screen are not modified.
* The program can't access the console using UNIX i/o functions;
* The standard i/o devices aren't supported;
* It can access files using UNIX i/o functions, but the SHELL's hierarchical file system isn't supported. Thus, it must specify the complete, Macintosh, name for a file in order to access it.

2.1.2.3 Command programs containing *mixcroot*

Command programs that contain *mixcroot* are generally linked with the '-M' option, thus allowing them to be activated by the Finder.

The features that *mixcroot* and *sacroot* give to a program differ only in the area of console i/o. A program linked with *mixcroot* has the following special features:

* It can access the console using UNIX i/o functions;
* It can access the standard i/o devices, which are automatically connected to the console.
* The screen is automatically initialized for it by calling *InitGraf*, by making the entire screen the current window, and by clearing the screen;
* The Manx-supplied console driver *.con* is used to access the console. This driver is automatically loaded into memory when needed.

A program linked with *sacroot*, on the other hand, must access the console using Macintosh functions; it can't access it using UNIX i/o functions; it doesn't support the standard i/o devices; and it itself must

initialize the screen;

The features that both *mixcroot* and *sacroot* give to a command program are:

* The program can be activated by the SHELL, Finder, or any command program;
* The program can be passed arguments. It receives them using the standard Macintosh conventions, and not as arguments to its *main* function.
* The program can access files using UNIX i/o functions, but the SHELL's hierarchical structure isn't supported.

For example, the following will create a "hello, world" program which can be activated by the SHELL or the Finder:

```
ln -M hello.o mixcroot.o -lc
```

This program contains a single code segment. It differs visibly from the program linked with *shcroot*, in that it automatically clears the screen and its standard output device can't be redirected. Also, it must receive any arguments passed to it using the standard Macintosh conventions. (though this program doesn't use them anyway).

If this program is going to be activated by the Finder, it needs to be able to access the console driver. This can be done by placing the resource containing the driver in the file that contains the executable 'hello' program. There are two commands that will do this: *InstallConsole* and *cprsrc*. The first command was designed to just copy the console driver into another file, while the second command is more general, being able to copy any resource from one file to another. Using *cprsrc*, the console driver could be copied into the *hello* file with the command:

```
cprsrc DRVR 30 sys:system hello
```

The console driver has type DRVR and ID 30; it's in the sys:system file.

2.1.3 Command programs having multiple code segments

This section discusses command programs having segmented code. It's divided into the following paragraphs:

Writing segmented programs: describes how a program loads and unloads its code segments and how it accesses global data;

Linking segmented programs: describes how a program creates command programs having segmented code.

As mentioned above, the code for any command program can be segmented.

The technique by which a command program loads and unloads its code segments is the same for both types of command programs; hence the programmer information section doesn't differentiate between the two types of command programs.

The procedure for linking a command program having segmented code is the same as for linking a command program whose code is unsegmented, with additional options interspersed showing where the segmentation is to occur. Thus, the operator information section doesn't make a big deal about the creating of SHELL-activated command programs having segmented code versus the creation of Finder-activated command programs having segmented code.

2.1.3.1 Writing segmented programs

There are two areas of concern for programs whose code is segmented: how segments are loaded and unloaded, and how programs access global data.

Loading and unloading code segments

A function in a command program whose code is segmented can call any other function in the program just as if the program's code was unsegmented. If the called function is in a loaded segment, control of the processor is simply passed to it; otherwise, the segment containing it is loaded into the application heap area of memory and then control is passed to it.

Within the application heap is an area from which buffers are dynamically allocated. It is in this area that code segments are loaded. Code segments and dynamically allocated buffers can be, and frequently are, interspersed in this area.

A code segment can be loaded anywhere within this area. The program has no control over where a segment is loaded, but an attempt is made to load code segments together in the low end of the area, to avoid memory fragmentation problems.

When a segment is loaded, it is 'locked' in memory; this means that it can't be moved around when the system wants to collect all free space together in the area, and that its section of memory can't be reallocated or used for other purposes.

Thus, once a program no longer needs a loaded code segment, it should 'unload' the segment, to allow the memory which it occupied to be reused for either the loading of other code segments or for use as a dynamically allocated buffer.

A program must explicitly request that a loaded segment be unloaded, by calling the function *UnloadSeg*, passing to it the address of any function in the segment.

UnloadSeg informs the system that the memory occupied by the segment is available for reallocation. If a function in a segment is called and the segment's code is in memory in an unloaded state, the memory will be simply reallocated to the segment, and the function called, without reloading the segment from disk.

Global and static data

There is only one global and static data segment for a program, regardless of the number of code segments it has. This segment is in a separate area of the application heap from that in which code segments are loaded.

The initialized global and static data for a program is initially, and automatically, loaded when the program is started, and remains loaded during the entire execution of the program, independent of the state of the program's code segments.

A program can contain up to 32K bytes of global and static data, and can contain both uninitialized and initialized variables. Uninitialized variables are automatically set to 0 when the program is started. Initialized variables can contain addresses or constants; in the former case the addresses will be adjusted automatically when the program is started.

The name of each global variable is unique: if several segments declare a global variable having the same name, they will both access the same variable.

2.1.3.2 Linking segmented programs

All the code segments of a command program are created during a single activation of the linker.

The code for a command program can be divided into a maximum of 256 segments, each of which has an identifying number between 0 and 255. All command programs must have a code segment 0, which is the first segment loaded for the program.

The linker command which creates a command program whose code is segmented looks like the command which links an unsegmented program, except that the list of files are interspersed with '+O' options. This option causes the object modules which follow it to be placed in a selected code segment.

A segment number can optionally be appended to a '+O' option, to explicitly select the segment into which the following modules will be placed. If a segment number isn't specified for a '+O' option, the modules will be placed in the next available segment.

An example

For example, the following command creates the SHELL- activated command program *prog*, which has three code segments. Segment 0

contains the code for the modules *menu.o, subs.o,* and any needed modules from *c.lib.* Segment 1 contains the code for the modules *mod1.o* and *mod2.o.* Segment 2 contains the code for *mod3.o* and *mod4.o,* and any *c.lib* modules referenced by segments 1 and 2 which aren't in segment 0:

```
ln -f prog.lnk
```

where *prog.lnk* contains:

```
menu.o subs.o -lc
+o
mod1.o mod2.o
+O
mod3.o
mod4.o
-lc
```

All the files for this example could have been specified on the command line which activated the linker. We didn't do this for two reasons: first, the entire command wouldn't have fit on one line of this page. Second, you'll also use -F files to link command programs having segmented code, since such programs tend to have many modules, making it impractical to specify all the file and segmentation information on one line.

Including modules from Libraries

This example illustrates a point about library searches during the linking of command programs having segmented code. As the linker includes object modules in segments, it builds a list of global symbols which are called or referenced but haven't been found yet . When a library is searched during the linking of a segment, and a module is found that contains a needed global symbol, the module is included in the segment, regardless of the segment which referenced it.

Thus, in the above example, when *c.lib* is searched during the linking of segment 0, the only modules in it that are included in segment 0 are those referenced by segment 0. When *c.lib* is searched during the linking of segment 2, modules from it are included that contain global symbols referenced by both segment 1 and 2 but that aren't in segment 0.

Segment 0 must contain the startup code for a program. This code is in *c.lib;* hence *c.lib* must always be first searched while segment 0 is selected. It would not be correct, for example, to modify the above example so that *c.lib* was searched only during the linking of segment 2, since this would force the startup code to be placed in segment 2.

Reselecting segments

The linker allows segments to be selected once, and later be reselected. In this case, the modules specified following the reselection

are appended to the code that's already in the segment. A segment can be reselected any number of times.

For example, the above example can be modified so that all *c.lib* modules referenced by all segments are included in segment 0, by modifying *prog.lnk* as follows:

```
menu.o subs.o
+O
mod1.o mod2.o
+O
mod3.o mod4.o
+O0
-lc
```

The '+O0' reselects segment 0, so that the modules pulled from *c.lib* are included in segment 0.

Linking Finder-activated command programs

Finder-activated command programs containing segmented code are created in the same way that an unsegmented version is created, with '+O' options interspersed in the list of file names to specify the segmentation.

For example, we can modify the above example so that the resulting command program can be executed by the Finder and supports UNIX i/o calls to the console by simply changing the command line:

```
ln -M mixcroot.o -f prog.lnk
```

In this case, *prog.lnk* doesn't need to be modified.

2.1.4 Technical information about command programs

This section presents information about command programs which is not discussed above.

2.1.4.1 Command file types

The file created by the linker for a command program is a 'resource file'; that is, a file all of whose data is in its resource fork. The type of the file is either 'AZTC' or 'APPL', depending on whether the program was linked without or with the '-M' option, respectively.

In fact, the only function of the -M option is to set the type of the file containing the command program to 'APPL'.

The SHELL will activate command programs contained in resource files whose type is either 'AZTC' or 'APPL'; the Finder will only activate resource files whose type is 'APPL'.

When the SHELL activates a command program, it looks at the type of the program's resource file: if the type is 'AZTC' it will place

command line arguments for it in the system area and redirect the program's standard i/o devices, if requested. When the program is started, the startup code in the Manx-supplied module *shcroot* will get the arguments from the system area and pass them to the program as parameters to the program's *main* function.

If the type of the program's resource file is 'APPL' the SHELL will place the command line arguments for it in the system area and then activate the program, without redirecting the program's standard i/o devices.

2.1.4.2 Resources for command programs

The resource fork of a file contains items called 'resources', each of which has a type and an identifying number associated with it.

For a command program, the linker creates several resources, each having the type 'CODE'. The resources and their ids are:

id	*contents*
0	jump table
1	code segment 0
2	code segment 1
...	...
256	initialized data
257	relocation info
	& optional alternate jump table

2.1.4.3 Segment information

Only the initialized data segment and code segment 0 can contain pointer fields whose values must be adjusted when the program is started. This requirement is always satisfied by programs created from C source.

Code segments can be any size; however, functions within a segment that are called from other segments must be within the first 64 Kbytes of the start of the segment. The startup routine for a program must be in the first 64 Kb of segment 0.

2.2 Customizing startup routines for command programs

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 provided with this package. Next, we describe what the Aztec startup routines do. Finally, we discuss different ways that you can modify the standard startup procedure.

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 first code segment.

Execution of a command program normally begins at the label *.begin* in the module *crt0*, which is in *c.lib*. The following facts account for this:

* *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 *Croot*.

As you already know, three modules are supplied that contain a Croot function: *shcroot*, *sacroot*, and *mixcroot*. A Croot function performs additional initialization activities as described below and then calls the program's function *main*.

2.2.2 What *crt0* and the *Croot* routines do

This section describes, in detail, what happens when the operator tells the SHELL or Finder to activate a command program.

The SHELL will only activate command programs contained in files whose type is *AZTC* or *APPL*; the Finder will only activate programs in files of type *APPL*. Both the SHELL and the Finder activate a command program by calling the operating system function *launch*.

Before calling *launch* to activate a program, the SHELL will move the command arguments for it into the system area, where they can be fetched by the program.

launch initializes the application heap, thus essentially removing the SHELL or Finder from memory, opens the resource file containing the program, loads its jump table from the resource having type CODE and id 0 into the top of the application heap, sets register A5 to the base of the jump table, and jumps to the first entry in the jump table. Since no code segments have yet been loaded for the program, the segment containing the function corresponding to the first jump table entry isn't loaded either, so this jump forces the function's segment to

be loaded into the application heap. Once the segment is loaded, a jump is made to the function.

2.2.2.1 The startup function

Thus, the function corresponding to the first entry in a program's jump table is the startup function for the program, and is the first function to be executed when the program is started. This function must be located in the program's code segment 0.

The way that the linker selects the entry point for a program was described above. In the remainder of this section, we want to describe the Manx-supplied startup routines: the routine that contains the entry point *.begin*, *crt0*, and the three versions of the *Croot* function, *shcroot*, *sacroot*, and *mixcroot*.

2.2.2.2 The *crt0* module

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

* It loads the program's initialized global and static data from the resource having type CODE and id 256. This data is loaded just below the program's jump table in the application heap.
* It allocates space for the program's uninitialized global and static data and clears it. This space is located just below the program's initialized data.
* It relocates fields within the initialized data area and code segment 0 which contain addresses. For this, it uses information in the program's resource whose type is CODE and id is 257.
* If an alternate jump table is contained in the program's resource whose type is CODE and id is 257, the startup code adjusts the jump table which is already loaded using this information.
* It allocates space for the program's stack. This area is located just below the program's uninitialized data area. By default, this area is 8k bytes, and can set to other values using the linker's -S option when linking a program.

The *.begin* routine then calls the function *Croot*.

2.2.2.3 The *Croot* functions

Three *Croot* modules are provided with the Aztec C package:

* shcroot
* mixcroot
* sacroot

One of these modules must be included in a program which also includes the *crt0* module. If desired, you can also include your own version of *Croot* instead of ours in your programs.

These functions perform additional initialization actions and then call the program's *main* function.

shcroot

This version of *Croot* performs the following activities:

* Verifies that the SHELL was the last command-processor-type program to be executed;
* Sets a pointer which causes the program to use the unbuffered i/o table that is located in the system area;
* Sets a pointer which causes calls to the function *fixname* to be processed by the function *shfixname*;
* Calls the SHELL's version of *InitWindow*, which calls *InitGraf*, sets the current window to be the entire screen. This version of *InitWindow* doesn't do anything physical to the screen (like clear it);
* Copies arguments from the system area to the application heap;
* Calls the program's *main* function, passing it the arguments which were moved from the system area, in the standard UNIX manner.

A program which includes *shcroot* uses a standard i/o table located in its own space and the unbuffered i/o table which is located in the system area. Because of this, files which are open for unbuffered i/o in the calling program are also open for unbuffered i/o in the called program, and are accessed using the same file descriptors. The only streams open for standard i/o in the called program are the standard i/o devices stdin, stdout, and stderr.

The function *fixname* translates a file name to Macintosh format, if necessary. There are two functions which may be called to process calls to *fixname*. *shfixname*, the one called for programs linked with *shcroot*, translates names from the SHELL format to Macintosh format. For example, if passed a file name which doesn't contain a volume or path to a directory, *fixname* prepends the name of the current volume and the path to the current directory to the name.

safixname, the other function called to process calls to *fixname* doesn't do anything. *safixname* is called for programs linked with *sacroot* and *mixcroot*, and hence requires programs linked with these two modules to specify complete Macintosh file names before accessing a file.

sacroot.o

This version of *Croot* performs the following activities:

* It sets a global pointer that causes the program to use an unbuffered i/o table which is contained in its own space, rather than the table in the system area; The table is cleared;

* It sets a global pointer that causes a call to *fixname* to be processed by *safixname*;
* It calls the program's *main* function with *argc*=0 and **argv*=0; thus, if the program is passed arguments, it must receive them using the standard Macintosh conventions, and not the UNIX conventions.

Unlike *shcroot*, *sacroot* doesn't care what command processor type program was last executed.

sacroot doesn't initialize the screen at all. The program itself will have to call *InitGraf*, *InitWindow*, and so on.

A program which includes *sacroot* uses a standard i/o table and an unbuffered i/o table which are located in its own space. Because of this, a program which contains this Croot and which is activated by another program doesn't 'inherit' the files and devices which the calling program left open. Initially, no devices are open for the program.

mixcroot.o

This version of *Croot* performs the following activities:

* It sets a global pointer which causes the program to use the unbuffered i/o table that is located in the program's space.
* It clears the unbuffered i/o table and attempts to open file descriptors 0, 1, and 2 to the device *.con*; if this fails, *Croot* exits;
* It calls the SHELL's version of *InitWindow* (described above for *shcroot*);
* It calls the program's *main* function with *argc*=0 and **argv*=0; thus, if the program is passed arguments, it must receive them using the standard Macintosh conventions, and not the UNIX conventions.

Unlike *shcroot*, *mixcroot* doesn't care what command-processor-type program was last executed.

A program which includes *mixcroot* uses a standard i/o table and an unbuffered i/o table which are located in its own space. Because of this, a program which contains this Croot and which is activated by another program doesn't 'inherit' the files and devices which the calling program left open. Initially, only the console and keyboard are open for the program, as the standard i/o devices.

2.2.3 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 *Croot*.

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

Another possibility is to remove *crt0* from *c.lib*, so that execution begins with the first statement of a command program.

2.3 Passing open files and devices to command programs

When a program which has been linked with the module *shcroot* is started, either by the SHELL or by another program which has also been linked with *shcroot*, the program 'inherits' the files and devices which were left open for unbuffered i/o by the caller. 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.

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. In programs linked with *shcroot* this table is located in the system area; for other programs it's in the program's own space.

The unbuffered i/o table that is located in the system area 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 'launches' 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.

The SHELL can only preopen the standard i/o devices for programs linked with *shcroot*, since this is the only type program that uses the unbuffered i/o table that is in the system area.

Programs linked with *mixcroot* also have the standard i/o devices preopened to the console when they start, but in this case, the opening of these devices is done by *mixcroot* and not by the SHELL, Finder, or whatever program activated it.

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 the standard i/o devices are preopened for a program, and 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 the standard i/o devices can be done by simply redirecting the first three entries in the unbuffered i/o table.

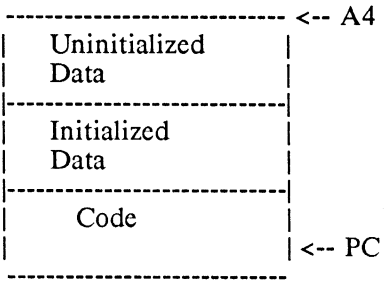
Since the standard i/o table is located in a program's own space, files opened for standard i/o in one program aren't open for standard i/o in the called program.

3. Drivers and desktop accesories

With the Aztec development software, you can create drivers and desktop accessories, as well as command programs. A driver is a program which command programs call to access a device; a desktop accessory is a program that the operator can activate via the Finder's apple menu.

Characteristics of drivers and desktop accessories

While in memory, a driver or desktop accessory occupies a single, contiguous block of memory. It is organized into three segments, as shown below:



The code segment contains all the executable code for the program. The initialized and uninitialized data segments contain the program's static and global variables.

The program is position independent, and hence can be loaded anywhere in memory. To accomplish position independence, instructions access information contained in the code segment using an offset from the current program counter. And to access information contained in a data segment, instructions use an offset from register A4, which must point to the first byte beyond the uninitialized data segment.

A driver or desktop accessory has the following restrictions:

* It can't contain any absolute memory references, since, unlike command programs, it doesn't have a startup routine to adjust such references;
* Its code segment must be less than 32K bytes long;
* The sum of the sizes of its two data segments must be less than 32K bytes.

On disk, a driver or desktop accessory occupies a single resource of a file's resource fork. This resource contains the code, global data, and static data for the driver or desktop accessory. The resource has type *DRVR*; its ID number and name are specified when it is linked.

3.1 Writing drivers and desktop accessories

3.1.1 Initializing register A4

A driver or desktop accessory must explicitly initialize register A4 before it can access any global or static data. This can be done with assembly language statements of the form:

```
public   __Cend__, __Uend__, __Dorg__
lea      clab+(__Cend__-clab)+(__Uend__-__Dorg__),A4
```

where *clab* is a label located in the code segment. The symbols declared to be public in the first statement are created by the linker, and have the following values:

__Cend__ Offset of the first byte following the code segment
__Uend__ Offset of the first byte after the uninitialized data segment.
__Dorg__ Offset of the first byte of the initialized data segment

All offsets are relative to the first byte of the code segment.

The first operand of the *lea* instruction adds, to the address of the code segment label *clab* , the distance from that label to the end of the uninitialized data segment. Since this operand uses a code segment label, the assembler and linker turn the instruction into a PC-relative reference, which will be valid wherever the program is loaded in memory.

3.1.2 Calling Quickdraw from a driver or desktop accessory

Driver or desktop accessory modules that call Quickdraw should be compiled with the symbol *__DRIVER* defined. Variables definitions in the file *quickdraw.h* that aren't needed by drivers and desktop accessories are surrounded by "#ifndef __DRIVER ... #endif" statements. So defining __DRIVER will prevent these variables from being defined, which would cause them to take up space in the program's data segment.

3.2 Compiling, assembling and linking

Once you have written the code for a driver or desktop accessory, you must compile, assemble, and link it.

3.2.1 Compiling

When compiling, you must specify the *-B* and *-U* options.

The -B option

The -B option prevents the compiler from writing the statement "public .begin" to the assembly language file. This statement is needed when a command program is linked, because it causes the linker to include the standard startup routine, *crt0*, and indirectly causes execution of the command program to begin at *.begin*.

This statement is not needed when a driver or desktop accessory is linked, because it begins execution at the beginning of the code segment, and performs its own startup activities.

The -U option

This option causes the compiler to generate code that uses register A4 as a base register when accessing global or static data, and prevents the compiler from generating code that uses A4 for holding register variables or temporary values.

If this option isn't specified, the compiler generates code that is appropriate for a command program.

3.2.2 Assembling

To assemble the output of the compiler, when a driver or desktop accessory is being created, nothing special needs to be done: the compiler has taken care of everything.

If you've written an assembly language module for the driver or desktop accessory, you must either include the statment

```
USEA 4
```

in the source code, or specify the option '-U4' when you start the assembler. For example,

```
as -U4 subs.asm
```

The assembler by default generates position independent code. When it finds a reference to a symbol in a data segment, it translates the instruction to a base-register relative form. The base register it uses by default is A5, which is appropriate when generating a command program, but not when generating a driver or desktop accessory.

3.2.3 Linking

The command to have the linker create a driver or desktop accessory has the same format as that to create a command program:

```
ln [-options] file1.o file2.o ...
```

The options which can be used when linking a command program can also be used when linking a driver or desktop accessory, with the exception of the -M and +O options. Two options cause the linker to create drivers and desktop accessories:

-D	create a driver;
-A	create a desktop accessory

The -N option

The option

```
-N name
```

defines the name of the resource containing the program and the name by which programs will access a driver or by which the operator will access a desktop accessory. The actual name is created from the specified name by prefixing it with the character '.' or '\0', depending on whether a driver or desktop accessory is being created.

If this option isn't specified, the program name defaults to *Test.*

The -I option

The option

-I id

causes the ID of the resource containing the driver or desktop accessory to be set to the decimal value *id.* If this option isn't used, the ID is set to 31.

The -R option

The option

-R attr

causes the attributes of the resource containing the driver or desktop accessory to be set to the hexadecimal value *attr.* If this option isn't used, the attributes are set to 0x30.

More on linking

The linker places the driver or desktop accessory that it creates in the resource fork of the output file.

The resource file contains a single resource, which contains the code, global data, and static data for the driver or desktop accessory. The resource has type *DRVR*, ID number as specified by the -I option, and name as defined by the -N option.

3.3 Examples

The C source for a desktop accessory, *explorer*, is provided with Aztec C68K. This program is described in the Examples chapter.

The source for the Aztec console driver, *.con*, is provided with Aztec C68K.

See the release document to find out the names of the files containing these programs.

4. The console driver

The console driver is a program that makes the Macintosh screen and keyboard look like a CRT connected to a UNIX system to command programs and to the operator. It's a Macintosh resource, having type=DRVR and ID=30.

Program's interface to the console driver

With the assistance of the console driver, a command program can access the screen or keyboard as if it was a file named *.con*, and access it using the standard UNIX I/O functions. For example, it can prepare the console for I/O by calling *fopen*, read and write characters from and to it by calling *getchar* and *putchar*, and terminate console I/O by calling *fclose*.

The console driver allows the console to set in several different modes. For example, the echoing of characters that the operator types can be enabled or disabled, and keyboard input can be either line- or character-oriented. A command program selects the mode for console I/O by calling the function *ioctl*. This function is described in the overview section of the Library Functions chapter.

The console driver displays characters in the current window, using that window's current text attributes, and displaying the characters in the current window's font, size and face.

The console driver singles out one character position, which we'll call the current character position, in the current window. This is the location at which it will display the next character that a program sends to it. This is also the position at which it will display the next character that the operator enters, if echoing is enabled. When it receives the character from the program, the driver then draws the character at this location, and decides where the next character will be displayed. Normally, this new location is the next character position to the right of the newly-written character. If that position falls over the right-hand edge of the current window, the next character position is set to the beginning of the next line. If that position falls over the bottom edge of the current window, the screen is scrolled up one line (that is, all lines are moved up one line and the bottom line is made blank), and sets the next character position to the beginning of the newly cleared bottom line.

When the command program has issued an input request to the console, the console driver will display a solid block, called the cursor, at the current character position.

Most characters that a command program sends to the console driver are simply written to the screen. Some, however, are control codes that cause the console driver to perform special functions. The

control codes (in hex) and their functions are:

code	*function*
07	beep
08	non-destructive backspace
09	tab character
0a	cursor down/linefeed (scroll if at bottom)
0b	cursor up
0c	non-destructive cursor right
0d	return to beginning of line
1a	home and clear screen
1e	home the cursor
1b 45	insert blank line at cursor
1b 51	insert blank character at cursor
1b 52	delete line at cursor
1b 54	clear to end of line from cursor
1b 57	delete character at cursor
1b 59	clear to end of screen
1b 3d y+20 x+20	move cursor to x,y position

The control codes for moving the cursor are useful only with fonts having a fixed pitch; that is, with fonts whose characters all have the same width. If a program tries to backspace with a non-proportional font, the cursor may be left positioned incorrectly, since the console driver normally moves the cursor in units corresponding to the width of an 'n'.

Operator's interface to the console driver

When a command program has issued an input request to the keyboard, the console driver normally just returns to the program the characters that the operator enters. Some characters that the operator enters, however, have special meaning to the console driver:

key	*Meaning*
backspace	Delete the last character entered by the operator, if it hasn't yet been returned to the program;
^D	Return EOF to the program.
^X	Delete all characters that haven't yet been returned to the program.
^S	Suspend screen output until a character is depressed.
^1	Eject the disk in drive 1: (The internal drive).
^2	Eject the disk in drive 2: (The external drive).

In the above list, the character ^ stands for the clover key; for example, to enter ^X you hold down the clover key and then type X.

The console driver allows programs to redefine the keys that are used for backspace and line-delete, to enable and disable the erasing of the previously entered character on the screen in response to the 'backspace' key, and to enable and disable the support of flow control.

Type-ahead

The console driver buffers keyboard input, and looks for a typed key whenever it's called - even if it was called to send characters to the screen. Thus, the console driver supports 'type-ahead' in a limited way, allowing the operator to enter characters before the command program has issued a read request, so long as the command program periodically writes to the console when it isn't reading from it.

If the operator does enter characters when the program is writing to the screen, the console driver won't echo the typed characters until the program issues a read request to the console. This feature prevents echoed characters from being mixed up with program output.

Creating programs that use the console driver

The console driver is a Macintosh resource, and isn't linked into the command programs that call it. The version of the Aztec startup routine *Croot* with which a command program is linked determines whether or not the program will use the console driver. Three versions of *Croot* are available:

shcroot	Program will use the console driver. Program can be started by the SHELL, but not by the Finder.
sacroot	Program won't use the console driver. Program can be started by either the SHELL or the Finder.
mixcroot	Program will use the console driver. Program can be started by either the SHELL or the Finder.

shcroot is included in the library *c.lib*, while the other two are supplied in separate object modules. Thus, by default a command program will use the console driver and can only be started by the SHELL. These versions of *Croot* give other characteristics to command programs in addition to the type of console I/O it supports; see the Command Program description in the Technical Information chapter for more details.

Loading the console driver into the application heap

The console driver can be loaded into either the application heap or the system heap. If it's loaded into the application heap, it's loaded automatically when the command program issues its first call to the console driver. However, since the application heap is cleared whenever a command program terminates, the console driver must be reloaded into the application heap for each command program that calls it.

When a command program issues an I/O request to the console driver and the driver isn't in memory, the Macintosh operating system will look for it in the file from which the command program was loaded and then in the *system* file which was on the disk from which the system was initially loaded. When found, the driver is loaded into

the application heap.

Loading the console driver into the system heap

The system heap is not cleared when a program running in the application heap terminates. Thus, it's possible to load the console driver into the system heap and make it resident, so that it doesn't have to be reloaded for each application program. The Aztec command program *FixAttr* can be used for this purpose: it sets the attributes of the console driver resource (along with those of some commonly used font tables) in the disk file containing this resource so that whenever power is turned on or the reset button is hit, the console driver is loaded into the system heap and made resident.

Loading the console driver into the system heap can be done only for Macintosh systems that have 512K bytes of memory, since the system heap on other systems isn't large enough to hold the console driver.

See the section entitled *FixAttr* in the Utility Programs chapter for more details.

Moving the console driver into files

Programs that are intended to be started by the SHELL, and the SHELL itself, usually don't contain the console driver resource: when linking a command program, the linker doesn't put a copy of the console driver resource in the file to which the program is written. Thus, when the SHELL is active, the console driver must be contained in the System file. The System file that is provided with the Aztec software contains the console driver, but standard Macintosh System files don't. Thus, if you must use the SHELL with a System file other than the one provided with the Aztec software, you must first install the console driver in the System file.

Programs that are to be started by the Finder and that are to use the console driver (that is, that have been linked with *mixcroot*) are frequently expected to be run with a standard Macintosh System file; that is, with a System file that doesn't contain the console driver resource. In this case, the console driver resource must be explicitly copied into the file containing the command program before the program can be run.

The Aztec utility program *InstallConsole* can be used to copy the console driver resource into another file. See its description in the Utility Program chapter.

Source for the console driver

The source for the console driver is provided with commercial versions of the Aztec software. See the release document for the name of the files containing it.

5. Using Aztec C68K with 128K Macintoshes

Normal operation of the Aztec C68K system should be unaffected when running on at least a 128K Macintosh. However, there are circumstances where the memory size constitutes a problem. In particular, compiling stand-alone Macintosh applications where many of the Macintosh header files are included can overflow the symbol table of the compiler. Two steps have been taken to lessen the likelihood of this occurring. First, the header files have been split into distinct groupings of functions. This allows some header files to be excluded from a particular compilation.

Secondly, the header files have been edited to include a number of statements of the form:

```
#ifndef SMALL_MEM
  ....
#endif
```

Under normal compilation, these statements will be ignored. However, if the macro variable, SMALL_MEM, is defined, these statements will prevent sections of the header files from being compiled. The sections were selected by choosing data structures and definitions whose exact nature is not necessarily significant. For example, there are a number of structures which are always referred to by handles which are returned from routines that create them and passed to routines that manipulate them. The choices made will certainly not meet the needs of every programmer. Feel free to add sections that aren't being used to the blocked off areas and to remove sections that are necessary. The choices made are only a guideline.

Another technique that can be used to free up memory for more symbols, is to adjust the size of the compiler's tables below the default values. In particular, the static string area defaults to being 2000 bytes in length. If only a few or no strings are used in a particular module, compiling it with a **-z100** option will free up a lot of space.

The second area of memory limitation is in the editor. Since Z is a memory based editor, it can only edit what it can hold in memory at one time. Currently on a 128K Macintosh, this is about 41K.

For 512K Macintosh's, the compiler memory limitations and the editor limitations become much less severe, since both programs take full advantage of the extra memory available.

6. Using Aztec C68K on 512K Macintoshes

There are several things you can do with the 512K bytes of memory that is in a fat Mac, and we're going to discuss them in this section:

* You can create large programs;
* Commonly-used resources can be loaded into the system heap;
* Part of RAM can be turned into a RAM disk.

6.1 Large programs

The primary size-related features of a command program created by Aztec C68K are independent of the amount of memory available on the Macintosh on which the program is to run. That is,

* A command program created by Aztec C68K is organized into one or more segments containing executable code, and one segment containing global and static data.
* Each code segment can be any size; the data segment can contain up to 32K bytes of global and static data.
* Only code segments containing functions being executed need be in memory; the others can be loaded and unloaded as necessary.
* The segments for a program can occupy as much of the Mac's application heap as desired.

Thus, for example, if you have a command program with 320K bytes of code, you could create a program having a single, 320K-byte code segment. Or you could partition the code into segments so that not all the 320K bytes of code need be occupying the application heap at one time.

6.2 Putting resources in the system heap

On a 128K-byte Macintosh, the system heap doesn't have much free space. Thus, resources that are needed by application programs must be loaded into the application heap. Since this space is cleared whenever an application program terminates, the resources used by a program must be loaded for that program. This constant reloading of resources slows down the system.

The 512K-byte Macintosh has a large system heap, with lots of free space. The Aztec program *FixAttr* can be used to make some frequently-used resources resident in the system heap, so that they don't have to be reloaded for each program. These resources are the console driver, and the Monaco 9- and 12-point fonts. See the description of *FixAttr* in the Utility Programs chapter for more details.

6.3 Creating a RAM disk

Aztec C68K includes software that turns part of the RAM on a 512K Macintosh into a RAM disk. This emulated disk is just like a normal disk drive, except that data transfer is much faster, and the contents of the drive are destroyed when the Macintosh is turned off or is rebooted. For details, see the section RAMDISK in the Utility Programs chapter.

7. Using Aztec C68K with a hard disk

The Aztec C68K development software will work on a Macintosh that has a hard disk. First, though, you must install the Aztec console driver in the System file on the hard disk. The following steps describe how this is done.

1) Boot the hard disk as you normally would.
2) Insert the distribution disk (or copy) marked 2 of 3.
3) Double click the application program named *InstallConsole*.
4) When the Mini-finder window comes up, click the "Drive" button till the hard disk volume name is displayed.
5) Select the file *System* in the window, and click "Open".
6) When the completion message appears, click "Ok".
7) When the Mini-finder window comes back, click "Cancel".
8) Now copy the files from both distribution disks and double-click the SHELL application.

See the Console Driver section in the Technical Information chapter for more information on the console driver. The *InstallConsole* program is also described in the Utility Programs chapter.

8. Using Aztec C68K on Single-drive Macintoshes

It should be noted that this system has not been designed with a single drive in mind. For those who, for whatever reason, like to juggle disks, the following comments have been included to provide some assistance.

There are two approaches to using a single drive. The first approach is to trim as much as possible and get everything possible onto one disk. The second approach is to swap disks, but to minimize the swapping of disks as much as possible. In both approaches, a copy is made of a bootable disk which contains data files. More than likely, a different boot disk should be used for each project being worked on to maximize available space.

If one is adopting the first approach, the following is a list of ways in which to free up space on the disk.

1) If no floating point is being done:
   a) Remove the *lib/m.lib* file
   b) Using the Apple Resource Mover program or by modifying *cprsrc.c*, remove the floating point and transcendental math packages from the System resource file. These are packages 4 and 5.
2) Remove header files from the *include/* directory that are not used in the current program being developed.
3) Using the *libutil* program, remove unused functions from the library.
4) Remove the compiler error message file in *include/cc.msg*.
5) Combine header files that are always included in the current program into a single file.
6) Remove other packages from the System file, like the Disk Initialization package, the International Utilities package and the Standard File package.

After removing the appropriate elements, use this disk as the base disk. Copy it and create source files on the copy. For each additional project, copy the original base disk and use the copy.

If the second approach is being used, then the best way to use the system is to reorganize the disks. The basic idea here is to minimize disk swapping. The best way to do that is to only swap disks for files that are used once. In particular, program loads cause the data disk to be ejected, the disk containing the program is inserted, the program is loaded, and the data disk reinserted. If the data and header files are on different disks, the disks must be swapped once (at least) for each header file. By having the System, SHELL, library and header files on

the data disk, the only swapping necessary is to load the compiler, assembler, linker or editor. This can free up to 250K of disk space if some of the items from the previous approach are adopted as well.

Make two copies of the *sys:* disk. Label one *bin:* and the second *sys:*. Boot the *bin:* disk and use the *mv* command to rename the disk to *bin:*.

```
mv sys: bin:
```

Now boot the *sys:* disk. Using Z edit the *.profile* file and include the following line:

```
set PATH=$PATH;bin:bin
```

Now, the SHELL will check the current disk, and then ask for the *bin:* disk to check for the file there. To free up the space, remove the files in the *bin/* directory by typing:

```
rm bin/*
```

In addition, if the System file and SHELL are removed from the *bin:* disk, the utility programs in the *sys2:bin/* directory on the second distribution disk may be copied to the *bin:bin/* directory.

Using this approach, there are a number of different ways that combinations of files can be split between disks to meet the needs of the individual. For example, the editor could be kept on the data disk, or the library could be placed on the *bin:* disk. Remember, that the PATH and INCLUDE variables can be used to indicate the path to be searched and allow sets of files to be split up. Experiment, and if you find a combination that you find particularly useful, write us a letter.

9. Data Formats

char variables

Variables of type *char* can be either signed or unsigned, and default to signed. When a signed char variable is used in an expression, it's converted to a 16-bit integer by propagating the most significant bit. Thus, a char variable whose value is between 128 and 255 will appear to be a negative number if used in an expression.

When an unsigned char variable is used in an expression, it's converted to a 16-bit integer in the range 0 to 255.

char variables on Aztec C for other systems can also be signed or unsigned, but the default type (signed or unsigned) differs from system to system. On 8086- and 8088-based systems, **char**s are signed by default. On other systems, *char*s are unsigned by default. Thus, the *char* variables in programs requiring system independence should be unsigned, if those variables are used in expressions.

int and *short* variables

int and *short int* variables are two bytes long, and can be either signed or unsigned. By default, variables of these types are signed. A negative value is stored in two's complement format. An int or short int is stored in memory with its least significant byte at the highest numbered address. A -2 stored at location 100 would thus look like:

location	contents in hex
100	FF
101	FE

long variables

Variables of type *long* occupy four bytes, and can be either signed or unsigned. Negative values are stored in two's complement representation. Variables of type *long* are stored sequentially in memory, with the least significant byte stored at the highest memory address and the most significant byte at the lowest memory address.

float and *double* variables

Variables of type *float* are represented in 32 bits, and those of type *double* are represented in 64 bits. They are in standard IEEE format.

Such variables are stored with the byte containing the sign and exponent at the lowest address, and the mantissa bytes at the highest-addressed bytes.

pointer variables

Pointers are four bytes (32 bits) long. Pointers are stored in memory with the least significant byte at the highest numbered address. The internal representation of the address 0x12345678 stored

in location 0x100 would thus be:

location	contents in hex
100	12
101	34
102	56
103	78

LIBRARIES SUMMARY - AZTEC/MPW

1. INTRODUCTION

As stated, Aztec C68K Release 3.4 includes two complete sets of header (*include*) files and two different *c.lib* library files. The first set is simply an updated version of the headers and the library we have supplied in the past (information from the new Inside Macintosh volume V is added to support the Macintosh II and Macintosh SE). The second set provides a closer match to the C compiler distributed by Apple Computer for use with the Macintosh Programmer's Workshop (MPW).

This appendix describes the differences included with this release and also discusses our plans for the future. We need your feedback to supply the programming development that you want to use!

After you read the section and send us your feedback, file this section at the end of the "Technical Information" documentation in your manual.

2. DIFFERENCES

The distributed headers and libraries reflect the following:

2.1. 31-Character Names

Global names in Aztec C have 31 significant characters (formerly eight) and have a leading underscore (formerly a trailing underscore).

2.2 Naming Conventions

In the past, there was no "standard" way to organize the information in Inside Macintosh into header files and libraries suitable for use with C. Inside Macintosh uses Pascal calling conventions, so C compiler vendors had to translate as best they could. Obviously, significant differences occurred in the choices made.

With the MPW C release, Apple Computer provides a *de facto* standard for organizing this information and our new set of headers follows that scheme. Manx did this for two important reasons:

1. It allows our present customers to get more closely in step with new Apple developments, for example, the rising importance of C language.
2. It permits users of competing products to gracefully upgrade to Aztec C68K.

Additionally, Manx upgraded our existing headers to support our customers. However, to maintain two sets of headers could cause

2.3 Glue Routines

MPW C introduces a somewhat different convention for passing arguments to the Macintosh ROM: character strings are assumed to be C-style (null- terminated), but "glue routines" translate such strings to pascal STR255 strings (leading length byte) before passing them to the Toolbox. STR255s returned by the Toolbox are converted by the glue routines into C strings.

The second set of headers declares the glue routines for the MPW convention, and the second c.lib contains all the necessary glue routines.

CAUTION: Strings passed or received as part of structures are *not* converted. For example, if you use SFGetFile to get a file name from the user in an SFReply struct, use *ptoc* to convert the file name to a C string before you call the FSOpen File Manager so that the FSOpen "glue" can turn the name back into a STR255 before passing it to the ROM.

2.4 Integer Sizes

MPW C also differs from Aztec C68K in the choice of the size of an integer (int)--MPW uses 32 bits, Aztec uses 16 bits. The new headers and libraries, like the old ones, continue to assume 16-bit ints. This becomes a problem only if the program you are trying to port from MPW C to Aztec C makes unwarranted assumptions about how much data an int can hold. A common pitfall is to assume that an int can hold a pointer value, which is not true in the 16-bit int model. And you cannot simply turn all the int variables to "long int." Doing so destroys calls that the library and Toolbox routines expect to be in 16-bit quantities. The next section describes some solutions to this problem.

3. HOW TO USE THIS RELEASE

3.1 Pick a Naming Convention

In your SHELL .profile, select one set of headers as the default by setting your INCLUDE variable:

```
set INCLUDE=sys2:include/
```

or

```
set INCLUDE=sys3:include/
```

On a program-by-program basis, override your default choice by using the *-I* option for cc:

```
cc -Isys2:include/ ...
cc -Isys3:include/ ...
```

3.2 Pick a Library

3.2 Pick a Library

If you use the revised Aztec (sys3:include/) headers, you must link with sys3:lib/ libraries. To choose this as your default, set the CLIB SHELL variable to:

```
set CLIB=sys3:lib/
```

or to the place on your hard disk where you installed the files from sys3:lib/.

If you use the new MPW-compatible (sys2:include/) headers, you have a choice, depending upon whether you want the MPW-style glue or not.

To use the MPW-style glue,

```
set CLIB=sys2:lib/;sys3:lib/
```

in your .profile.

If you want the new headers but do not want to use MPW-style glue, set your CLIB environment variable to pick up the Aztec libraries as usual:

```
set CLIB=sys3:lib/
```

You also need to use an extra option in your *cc* commands:

```
cc ... -D__INLINE ...
```

You do not need to use sys2:lib/ at all. The -D__INLINE causes the compiler to generate ROM traps as inline code, bypassing the glue routines. This option gives you the best of both words--Apple naming compatibility with the speed and compactness of direct ROM traps.

One possible problem exists: if *ln* complains that it cannot resolve references to certain Toolbox routines, recheck your *make* files to make sure that you used -D__INLINE in all *cc* commands, or that you have chosen the c.lib that includes the MPW-style glue.

3.3 Using 32-Bit Integers

A new *cc* option, *+L*, makes int equivalent to long. For example, a long int always has 32 bits, a short int always has 16 bits, but a variable declared int has 32 bits when *+L* is used, and 16 bits when *+L* is not used.

This option solves some of the problems of porting programs that assume that long is the same as int (i.e., from VAX or MPW). However, the Aztec runtime libraries and the Macintosh ROM continue to use 16-bit ints, and there are NO libraries or glue routines in the C68K package that assume 32-bit ints.

Manx does not plan to make 32-bit libraries available but we will, as usual, respond to market demand. Commercial users may, at their discretion, use *+L* to recompile selected library routines to use 32-bit ints.

Version 4.1 of C68K--planned for late 1987--will address this difficulty.

4. A LOOK AHEAD

4.1 MPW Shell Support

C68K, Version 3.5, which will be available later this summer, will provide full compatibility with the MPW shell--the compiler, linker, etc. and all utilities will be available as MPW "tools," and Aztec customers will be able to construct MPW tools of their own. Most programs that run under the Aztec SHELL may be turned into MPW tools with just a relink. Please let us know if this new capability is something you are interested in.

Manx continues to support the Aztec SHELL and Z editor, which are somewhat more "lean and mean" than the MPW facilities.

4.2 Aztec Headers

Manx will support the MPW-style headers from now on. If you have a problem with the conversion, let us know. Manx will continue to support the older header scheme as long as our customers have a need for it, but we would like to hear from you.

4.3 MPW Glue

The MPW-style glue routines are somewhat at variance with the usual "just do what I ASK for" style of C programming. Manx will continue to support the routines if our customers find them useful, otherwise Manx will drop them in future releases. Please let us know what you want!

Memory Models

The memory model used by a program determines how the program's executable code makes references to code and data. This in turn indirectly determines the amount of code and data that the program can have, the size of the executable code, and the program's execution speed.

Before getting into the details of memory models, we want to describe the sections into which a C68k-generated program is organized. The sections of a program are these:

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

There are two attributes to a program's memory model: one attribute specifies whether the program uses the *large data* or the *small data* memory model; the other attribute specifies whether the program uses the *large code* or *small code* memory model.

1. Large Data Versus Small Data

The fundamental difference between a *large data* and a *small data* program concerns the way that instructions access data segment data: a *large data* program accesses the data using position-dependent instructions; a *small data* program accesses the data using position-independent instructions. An instruction makes position-dependent reference to data in the data segment by specifying the absolute address of the data; it makes a position-independent reference to data in the data segment by specifying the location as an offset from a reserved address register. On the Macintosh, the small data area comprises the 32k of memory at negative offsets from the *A5* register. Other differences in *large data* and *small data* programs result from this fundamental difference; these other differences are:

* There is no limit to the amount of global and static data that a *large data* program can have. A *small data* program, on the other hand, can have at most 64k bytes of global and static data.

* For a *small data* program, an address register must be reserved to point into the middle of the data segment. For a *large data* program, an instruction that wants to access data in the data segment contains the absolute address of the data, and hence doesn't need this address register.

* It takes more time to load a code segment for a *large data* program than for a *small data* program. The reason for this is that the absolute address of data segment data isn't known until a program is loaded. Thus, instructions that access data segment data using absolute addresses must be adjusted when the code segment containing the instructions is loaded, whereas instructions that access data segment data in a position-independent way don't need to be adjusted.

* A code segment is larger when its program uses *large data* than when it uses *small data*, because a reference to data in a data segment occupies a 32-bit field in a *large data* instruction, and occupies a 16-bit field in a *small data* instruction.

* A program is slower when it uses *large data* than when it uses *small data*, because it takes more time for an instruction to access data when it specifies the absolute address of the data than when it specifies the data's offset from an address register.

2. Large Code Versus Small Code

The fundamental difference between a *large code* and a *small code* program concerns the way that instructions in the program refer to locations that are located in the code segment: for a *large code* program the reference is made using position-dependent instructions; for a *small code* program, the reference is made using position-independent instructions. An instruction makes position-dependent reference to a code segment location by specifying the absolute address of the location; it makes a position-independent reference to a code segment location by specifying the location as an offset from the current program counter. Other differences in *large data* and *small data* programs result from this fundamental difference; these other differences are:

* The size of a code segment is unlimited for both *large code* and *small code* programs. An instruction in a *large code* program can directly call or jump to the location, regardless of its location in the code segment.

  An instruction in a *small code* program can only directly call or jump to locations that are within 32k bytes of the instruction. To allow instructions in *small code* programs to transfer control to any location, regardless of its location in the code segment, a "jump table", which is located in the program's data segment, is used. If a location to which an instruction wants to transfer control is more than 32k bytes from the instruction, the transfer is made indirectly, via the jump table: the instruction calls or jumps to an entry in the jump table, which in turn jumps to the desired location. A jump instruction in a jump table entry refers to a code segment location using an absolute, 32-bit address, and hence can directly access any location in the program's code segment.

  When a *small code* program is linked, the linker automatically builds the jump table: if the location to which an instruction wants to transfer control is outside the instruction's range, the linker creates a jump table entry that jumps to the location and transforms the pc-relative instruction into a position-independent call or jump to the jump table entry.

* A code segment can contain data as well as executable code. An instruction in a *large code* program can access data located anywhere in the code segment, because it accesses code segment data using position-dependent instructions, in which the location is referred to using a 32-bit, absolute address. An instruction in a *small code* program can only access code segment data that is located within 32k bytes of the instruction.

* For a *small code* program to access the jump table, an address register needs to be reserved and set up to point into the middle of the program's data segment; if the program also uses *small data*, the same address register is used for both jump table accesses and normal accesses of data segment data. On the Macintosh, the jump table is at positive offsets from

the *A5* register. For a *large code* program, this address register is not needed for the referencing of locations in the code segment.

* A program takes longer to load if it uses *large code* than if it uses *small code*. Instructions in a *large code* program that reference a code segment location must be adjusted when the program is loaded, since such instructions must contain the absolute address of the location and since this isn't known until the program is loaded. Instructions in a *small code* program that reference code segment locations need not be adjusted, since they are always independent of the location at which the code segment is loaded: if the location is within 32k of the referencing instruction, the instruction is pc-relative; and if it's outside this range, the instruction is a position-independent jump to a jump table entry.

  When a *small code* program that contains a jump table is loaded, its jump table entries must be adjusted, since these are jump instructions to code segment locations, where each instruction must contain the absolute address of the destination address. However, it should take less time to adjust the jump table for a *small code* program than to adjust the code segment of a *large code* version of the same program, since for any destination of a jump or call instruction a *small code* version of a program will have at most one jump table entry needing adjustment, whereas a *large code* version of the program may have many jump or call instructions to the same location that need to be adjusted.

* A code segment is larger when its program uses *large code* than when it uses *small code*, because instructions that reference code segment locations by specifying an absolute address use a 32-bit field to define the location, whereas instructions that reference data by specifying a pc-relative address or an offset from an index register use a 16-bit field to define the location.

* A program is usually slower when it uses *large code* than when it uses *small code*, because it takes more time for an instruction to reference a code segment location when it specifies the absolute address of the data than when it specifies the location in a pc-relative form.

A large *small code* program that has lots of indirect transfers of control via the jump table may not differ much in execution time from a *large code* version of the same program, since the *small code* indirect transfer via the jump table will take more time than the *large code* direct transfer.

3. Selecting A Module's Memory Model

You define the memory model to be used by a module when you compile the module, by specifying or not specifying the following options:

+C	Module uses *large code*. If this option isn't specified, the module will use *small code*.
+D	Module uses *large data*. If this option isn't specified, the module will use *small data*.

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

cc prog	small code, small data
cc +C prog	large code, small data
cc +D prog	small code, large data
cc +C +D	large code, large data

Generating Libraries

This section describes the procedures for constructing versions of the libraries for special needs. Source for the libraries is distributed only with the *Commercial* version of Aztec C68K.

1. Object Libraries

The libraries distributed in object form with Aztec C68K are compiled and assembled assuming that an int is 16 bits in size and using the *small code* and *small data* models (see the appendix included with this release entitled "Memory Models," which contains a tutorial description of small versus large models).

There are two sets of libraries distributed in object form and these are also discussed in an appendix to this release entitled "Libraries Summary - Aztec/MPW." The libraries are:

* The set in sys2:lib/ that uses MPW-compatible headers (in sys2:include/) and contains MPW-style glue
* The set in sys3:lib/ that uses the Aztec headers (in sys3:include/).

2. Which Libraries?

The sys4: disk contains two sets of archives--sys4:sys2_arc/ and sys4:sys3_arc/--corresponding to the sys2: and sys3: libraries, respectively. The following discussion applies to either library set.

Each set of archives is set up to generate four sets of libraries, identified by keywords, as follows:

s	Small code,small data, 16-bit int. Used to build the libraries distributed in object form with Aztec C68K.
s32	Small code, small data, 32-bit int.
ld	Small code, large data, 16-bit int.
ld32	Small code, large data, 32-bit int.

Note that all sets use the *small code* model. There is no real need to resort to the *large code* model: The Aztec C68K linker can build code segments far in excess of 32Kb without using the *large* model.

Warning: At present, there is NO set of libraries that includes glue for the Macintosh Toolbox and presumes 32-bit ints. s32 and ld32 are provided to assist in porting applications from 32-bit machines that expect portable C library support.

3. Source Code

In each of sys4:sys2_arc/ and sys4:sys3_arc/ you will find the archive files for the libraries. The archives were built with mkarcv and may be unpacked with arcv (see the "Utilities section of the manual for details).

4. Setting up to Build Libraries

Create a new directory in which to build the libraries, change directory to it (we call the new directory libgen), and unpack the master makefile:

```
mkdir libgen
cd libgen
arcv sys4:sys2_arc/inp.arc
```

(you could use sys4:sys3_arc/ instead to work with the Aztec libraries). Use the editor of your choice to familiarize yourself with the makefile if you wish. The next step is to construct the subdirectories and unpack the archives by entering:

```
make ARC= sys4:sys2_arc unpack
```

This creates the directories and unpacks the archives into them. Use:

```
make ARC= sys4:sys3_arc unpack
```

if you wish to work with the Aztec libraries. You can also create directories and unpack archives selectively to build only one or two libraries--the unpack target tells make to unpack them all.

Before proceeding further, make sure that the SHELL variable *INCLUDE* names the appropriate directory:

```
set INCLUDE= sys2:include/
```

for the MPW-compatible headers, or

```
set INCLUDE= sys3:include/
```

for the Aztec headers.

To rebuild all libraries for every combination of small/large data and 16/32-bit ints, simply enter:

make LIBTYPE= all new

A more selective approach is to construct just the libraries you want, just for the small/large 16/32 model that you want.

make LIBTYPE= *libtype* remake *target...*

where *libtype* is one of s, s32, ld, or ld32. For example, if you want c.lib and m.lib using small data and 32-bits ints, use the command:

make LIBTYPE= s32 remake c32lib m32lib

Note that the targets named do NOT have an embedded period, but the libraries that actually result will be named c32.lib and m32.lib. Refer to the makefile itself if you are unsure of the exact target name to use.

5. Updating Existing Libraries

Sometimes you may want to modify just one routine and replace it in an existing library. It is not necessary to remake the entire library to do this. The following example shows a step-by-step replacement of the getc routine.

First, go back to your build directory, e.g.,

cd libgen

and, if necessary, create the directory and unpack the appropriate archive, as follows:

mkdir stdio
cd stdio
arcv sys4:sys2_arc/stdio.arc

Now edit getc.c as desired, and type

make getc.o

Note that, if you want large data, 32-bit *ints*, or both, make getc.do, getc.32o, or getc.d32o instead.

To install the new version, use the lb command:

lb $(LIB)/c.lib -r getc getc

(you may want to apply the lb command to a copy of c.lib rather than your production version for testing).

Finally, you may want to put the new source code back in the archive, as follows:

```
cd libgen/stdio
mkarcv sys4:sys2_arc/stdio.arc < arc.inp
```

(Duplicate page - First issued with release 1.06i)

This is a description of how to include the AppleTalk resource in your program. First, make sure that the AppleTalk library - *a.lib* - is linked in with your application. Next, the AppleTalk file - *ABPackage* - must be included as a resource in your program. The following is an example of a file that when input to *RGen*, includes *ABPackage* as a resource of type 'atpl' in your program:

```
!program

TYPE atpl=GNRL
        ,0
        .R
ABPackage atpl 0
```

(Duplicate pages - First issued with release 1.06h)

AppleTalk Manager Functions

The functions described in this section allow a C program to access Macintosh AppleTalk Manager routines.

The constants, data structures, and functions described in this section are defined in the header file *appletalk.h.*

1. Constants

```
#define lapSize                 20
#define ddpSize                 26
#define nbpSize                 26
#define atpSize                 56

#define tLAPRead                0
#define tLAPWrite               1
#define tDDPRead                2
#define tDDPWrite               3
#define tNBPLookup              4
#define tNBPConfirm             5
#define tNBPRegister            6
#define tATPSndRequest          7
#define tATPGetRequest          8
#define tATPSdRsp               9
#define tATPAddRsp              10

#define lapProto                0
#define ddpProto                1
#define nbpProto                2
#define atpProto                3

typedef short                   ABCallType;
typedef short                   ABProtoType;
typedef unsigned char           ABByte;
typedef String(32)              Str32;
```

2. Data Structures

```
typedef struct {
        Byte                    dstNodeId;
        Byte                    srcNodeId;
        Byte                    LAPProtType;
} LAPAdrBlock;

typedef struct {
        short                   aNet;
        Byte                    aNode;
        Byte                    aSocket;
} AddrBlock;

typedef struct {
        Str32                   objStr;
        Str32                   typeStr;
        Str32                   zoneStr;
} EntityName, *EntityPtr;

typedef struct {
        Byte                    retransInterval;
        Byte                    retransCount;
} RetransType;

typedef struct {
        short                   buffSize;
        Ptr                     buffPtr;
        short                   dataSize;
        long                    userBytes;
} BDSElement;
typedef BDSElement              BDSType[8];
typedef BDSType *               BDSPtr;

typedef struct {
        ABCallType              abOpcode;
        short                   abResult;
        long                    abUserReference;
        LAPAdrBlock             lapAddress;
        short                   lapReqCount;
        short                   lapActCount;
        Ptr                     lapDataPtr;
} LAPRecord, *LAPRecPtr, **LAPRecHdl;
```

```
typedef struct {
        ABCallType          abOpcode;
        short               abResult;
        long                abUserReference;
        short               ddpType;
        short               ddpSocket;
        AddrBlock           ddpAddress;
        short               ddpReqCount;
        short               ddpActCount;
        Ptr                 ddpDataPtr;
        short               ddpNodeID;
} DDPRecord, *DDPRecPtr, **DDPRecHdl;

typedef struct {
        ABCallType          abOpcode;
        short               abResult;
        long                abUserReference;
        EntityPtr           nbpEntityPtr;
        Ptr                 nbpBufPtr;
        short               nbpBufSize;
        short               nbpDataField;
        AddrBlock           nbpAddress;
        RetransType         nbpRetransmitInfo;
} NBPRecord, *NBPRecPtr, **NBPRecHdl;
```

```
typedef struct {
        ABCallType          abOpcode;
        short               abResult;
        long                abUserReference;
        short               atpSocket;
        AddrBlock           atpAddress;
        short               atpReqCount;
        Ptr                 atpDataPtr;
        BDSPtr              atpRspBDSPtr;
        unsigned char       atpBitMap;
        short               atpTransId;
        short               atpActCount;
        long                atpUserData;
        Boolean             atpXO;
        Boolean             atpEOM;
        short               atpTimeOut;
        short               atpRetries;
        short               atpNumBufs;
        short               atpNumRsp;
        short               atpBDSSize;
        long                atpRspUData;
        Ptr                 atpRspBuf;
        short               atpRspSize;
} ATPRecord, *ATPRecPtr, **ATPRecHdl;
```

3. Functions

3.1 Opening and Closing AppleTalk

pascal short *MPPOpen* ( );

pascal short *MPPClose* ( );

3.2 AppleTalk Link Access Protocol

pascal short *LAPOpenProtocol* (theLAPType, protoPtr)
    ABByte theLAPType; Ptr protoPtr;

pascal short *LAPCloseProtocol* (theLAPType)
    ABByte theLAPType;

pascal short *LAPWrite* (abRecord, async)
    LAPRecHdl abRecord; Boolean async;

pascal short *LAPRead* (abRecord, async)
    LAPRecHdl abRecord; Boolean async;

pascal short *LAPRdCancel* (abRecord)
    LAPRecHdl abRecord;

3.3 Datagram Delivery Protocol

pascal short *DDPOpenSocket* (theSocket, sktListener)
Byte *theSocket; Ptr sktListener;

pascal short *DDPCloseSocket* (theSocket)
Byte theSocket;

pascal short *DDPWrite* (abRecord, doChecksum, async)
DDPRecHdl abRecord; Boolean doChecksum, async;

pascal short *DDPRead* (abRecord, retCksumErrs, async)
DDPRecHdl abRecord; Boolean retCksumErrs, async;

pascal short *DDPRdCancel* (abRecord)
DDPRecHdl abRecord;

3.4 AppleTalk Transaction Protocol

pascal short *ATPLoad* ( )

pascal short *ATPUnload* ( )

pascal short *ATPOpenSocket* (addrRcvd, atpSocket)
AddrBlock abRecord; Byte *atpSocket;

pascal short *ATPCloseSocket* (atpSocket)
Byte atpSocket;

pascal short *ATPSndRequest* (abRecord, async)
ATPRecHdl abRecord; Boolean async;

pascal short *ATPRequest* (abRecord, async)
ATPRecHdl abRecord; Boolean async;

pascal short *ATPReqCancel* (abRecord, async)
ATPRecHdl abRecord; Boolean async;

pascal short *ATPGetRequest* (abRecord, async)
ATPRecHdl abRecord; Boolean async;

pascal short *ATPSndRsp* (abRecord, async)
ATPRecHdl abRecord; Boolean async;

pascal short *ATPAddRsp* (abRecord)
ATPRecHdl abRecord;

pascal short *ATPResponse* (abRecord, async)
ATPRecHdl abRecord; Boolean async;

pascal short *ATPRspCancel* (abRecord, async)
ATPRecHdl abRecord; Boolean async;

3.5 Name-Binding Protocol

pascal short *NBPRegister* (abRecord, async)
NBPRecHdl abRecord; Boolean async;

pascal short *NBPLookup* (abRecord, async)
NBPRecHdl abRecord; Boolean async;

pascal short *NBPExtract* (theBuffer, numInBuf, whichOne,
abEntity, address)
Ptr theBuffer; short numInBuf, whichOne;
EntityName *abEntity; AddrBlock address;

pascal short *NBPConfirm* (abRecord, async)
NBPRecHdl abRecord; Boolean async;

pascal short *NBPRemove* (abEntity)
EntityPtr abEntity;

pascal short *NBPLoad* ( )

pascal short *NBPUnload* ( )

3.6 Miscellaneous Routines

pascal short *GetNodeAddress* (myNode, myNet)
short *myNode, *myNet;

pascal short *IsMPPOpen* ( )

pascal short *IsATPOpen* ( )

(Duplicate page - First issued with release 1.06h)

File Manager Functions

This section summarizes the information needed for C programs that want to access the Macintosh File Manager routines. The constants, data structures, and functions are defined in the header file *pb.h*. This file makes references to information defined in the header file *types.h*. *pb.h* will automatically include *types.h* in a program if it hasn't yet been included.

1. Constants

```
#define fHasBundle          0x20
#define fInvisible          0x40

#define fTrash              -3
#define fDesktop            -2
#define fDisk               0

#define fsAtMark            0
#define fsFromStart         1
#define fsFromLEOF          2
#define fsFromMark          3
#define rdVerify            0x0040

#define fsCurPerm           0
#define fsRdPerm            1
#define fsWrPerm            2
#define fsRdWrPerm          3
#define fsRdWrShPerm        4
```

2. Data structures

```
struct Finfo {
        OSType          fdType;
        OSType          fdCreator;
        short           fdFlags;
        Point           fdLocation;
        short           fdFldr;
};

typedef struct Finfo    Finfo;
```

```
struct ioParam {
        short           ioRefNum;
        SignedByte      ioVersNum;
        SignedByte      ioPermssn;
        Ptr             ioMisc;
        Ptr             ioBuffer;
        long            ioReqCount;
        long            ioActCount;
        short           ioPosMode;
        long            ioPosOffset;
};
struct filcParam {
        short           ioFRefNum;
        SignedByte      ioFVersNum;
        SignedByte      filler1;
        short           ioFDirIndex;
        SignedByte      ioFlAttrib;
        SignedByte      ioFlVersNum;
        Finfo           ioFlFndrInfo;
        long            ioFlNum;
        unsigned short  ioFlStBlk;
        long            ioFlLgLen;
        long            ioFlPyLen;
        unsigned short  ioFlRStBlk;
        long            ioFlRLgLen;
        long            ioFlRPyLen;
        long            ioFlCrDat;
        long            ioFlMdDat;
};
```

```
struct hfileParam {
        short               ioFRefNum;
        SignedByte          ioFVersNum;
        SignedByte          filler1;
        short               ioFDirIndex;
        SignedByte          ioFlAttrib;
        SignedByte          ioFlVersNum;
        FInfo               ioFlFndrInfo;
        long                ioDirID;
        unsigned short      ioFlStBlk;
        long                ioFlLgLen;
        long                ioFlPyLen;
        unsigned short      ioFlRStBlk;
        long                ioFlRLgLen;
        long                ioFlRPyLen;
        long                ioFlCrDat;
        long                ioFlMdDat;
};
struct volumeParam {
        long                filler2;
        short               ioVolIndex;
        long                ioVCrDate;
        long                ioVLsBkUp;
        short               ioVAtrb;
        unsigned short      ioVNmFls;
        short               ioVDirSt;
        short               ioVBlLn;
        unsigned short      ioVNmAlBlks;
        long                ioVAlBlkSiz;
        long                ioVClpSiz;
        short               ioAlBlSt;
        long                ioVNextFNum;
        unsigned short      ioVFrBlk;
};
```

```
struct hvolumeParam {
        long            filler4;
        short           ioVolIndex;
        long            ioVCrDate;
        long            ioVLsMod;
        short           ioVAtrb;
        unsigned short  ioVNmFls;
        short           ioVBitMap;
        short           ioVAllocPtr;
        unsigned short  ioVNmAlBlks;
        long            ioVAlBlkSiz;
        long            ioVClpSiz;
        short           ioAlBlSt;
        long            ioVNxtCNID;
        unsigned short  ioVFrBlk;
        short           ioVSigWord;
        short           ioVDrvInfo;
        short           ioVDRefNum;
        short           ioVFSID;
        long            ioVBkUp;
        unsigned short  ioVSeqNum;
        long            ioVWrCnt;
        long            ioVFilCnt;
        long            ioVDirCnt;
        long            ioVFndrInfo[8];
};
struct hFileInfo {
        FInfo           ioFlFndrInfo;
        long            ioDirID;
        unsigned short  ioFlStBlk;
        long            ioFlLgLen;
        long            ioFlPyLen;
        unsigned short  ioFlRStBlk;
        long            ioFlRLgLen;
        long            ioFlRPyLen;
        long            ioFlCrDat;
        long            ioFlMdDat;
        long            ioFlBkDat;
        FInfo           ioFlXFndrInfo;
        long            ioFlParID;
        long            ioFlClpSiz;
};
```

```
struct DInfo {
        Rect                    FRect;
        short                   FRFlags;
        Point                   FRLocation;
        short                   FRView;
};
typedef struct DInfo            DInfo;

struct DXInfo {
        Point                   FRScroll;
        long                    FROpenChain;
        short                   FRUnused;
        short                   FRComment;
        long                    FRPutAway;
};
typedef struct DXInfo           DXInfo;

struct dirInfo {
        DInfo                   ioDrUsrWds;
        long                    ioDrDirID;
        unsigned short          ioDrNmFls;
        short                   filler3[9];
        long                    ioDrCrDat;
        long                    ioDrMdDat;
        long                    ioDrBkDat;
        DXInfo                  ioDrFndrInfo;
        long                    ioDrParID;
};

struct drvQElRec {
      struct drvQElRec *        drvLink;
      short                     drvFlags;
      short                     drvRefNum;
      short                     drvFSID;
      short                     drvBlkSize;
};
```

```
union OpParamType {
        struct {
                short           sg_flags;
                char            sg_erase;
                char            sg_kill;
        } conCtl;
        short                   sndVal;
        short                   asncConfig;
        struct {
                Ptr             asncBPtr;
                short           asncBLen;
        } asyncInBuff;
        struct {
                unsigned char   fXOn;
                unsigned char   fCTS;
                char            xon;
                char            xoff;
                unsigned char   errs;
                unsigned char   evts;
                unsigned char   fInX;
                unsigned char   null;
        } asyncShk;
        struct {
                long            param1;
                long            param2;
                long            param3;
        } printer;
        struct {
                Ptr             fontRecPtr;
                short           fontCurDev;
        } fontMgr;
        Ptr                     diskBuff;
        long                    asyncNBytes;
        struct {
                short           asncS1;
                short           asncS2;
                short           asncS3;
        } asyncStatus;
        struct {
                short           dskTrackLock;
                long            dskInfoBits;
                struct drvQElRec dskQElem;
                short           dskPrime;
                short           dskErrCnt;
        } diskStat;
};
typedef union OpParamType   OpParamType;
typedef union OpParamType * OpParamPtr;
```

```
struct cntrlParam {
        short                   csRefNum;
        short                   csCode;
        OpParamType             csParam;
};
struct ParamBlkRec {
        struct ParamBlkRec *    ioLink;
        short                   ioType;
        short                   ioTrap;
        Ptr                     ioCmdAddr;
        ProcPtr                 ioCompletion;
        short                   ioResult;
        char *                  ioNamePtr;
        short                   ioVRefNum;
        union {
                struct ioParam      iop;
                struct fileParam    fp;
                struct volumeParam  vp;
                struct cntrlParam   cp;
        } u;
};
typedef struct ParamBlkRec      ParamBlkRec;
typedef struct ParamBlkRec *    ParmBlkPtr;
```

```
struct HPrmBlkRec {
        struct HPrmBlkRec *     qLink;
        short                   qType;
        short                   ioTrap;
        Ptr                     ioCmdAddr;
        ProcPtr                 ioCompletion;
        short                   ioResult;
        char *                  ioNamePtr;
        short                   ioVRefNum;
        union {
                                struct ioParam iop;
                                struct hfileParam hfp;
                                struct hvolumeParam hvp;
                                struct cntrlParam cp;
        } u;
};
typedef struct HPrmBlkRec       HPrmBlkRec;
typedef struct HPrmBlkRec *     HPrmBlkPtr;

struct CInfoPBRec {
        struct CInfoPBRec *     qLink;
        short                   qType;
        short                   ioTrap;
        Ptr                     ioCmdAddr;
        ProcPtr                 ioCompletion;
        short                   ioResult;
        char *                  ioNamePtr;
        short                   ioVRefNum;
        short                   ioFRefNum;
        short                   filler1;
        short                   ioFDirIndex;
        SignedByte              ioFlAttrib;
        SignedByte              filler2;
        union {
                                struct hFileInfo hfi;
                                struct dirInfo di;
        } u;
};
typedef struct CInfoPBRec       CInfoPBRec;
typedef struct CInfoPBRec *     CInfoPBPtr;
```

```
struct CMovePBRec {
        struct CMovePBRec *  qLink;
        short                qType;
        short                ioTrap;
        Ptr                  ioCmdAddr;
        ProcPtr              ioCompletion;
        short                ioResult;
        char *               ioNamePtr;
        short                ioVRefNum;
        long                 filler1;
        char *               ioNewName;
        long                 filler2;
        long                 ioNewDirID;
        long                 filler3[2];
        long                 ioDirID;
};
typedef struct CMovePBRec    CMovePBRec;
typedef struct CMovePBRec *  CMovePBPtr;

struct WDPBRec {
        struct WDPBRec *     qLink;
        short                qType;
        short                ioTrap;
        Ptr                  ioCmdAddr;
        ProcPtr              ioCompletion;
        short                ioResult;
        char *               ioNamePtr;
        short                ioVRefNum;
        short                filler;
        short                ioWDIndex;
        long                 ioWDProcID;
        short                ioWDVRefNum;
        short                filler2[7];
        long                 ioWDDirID;
};
typedef struct WDPBRec       WDPBRec;
typedef struct WDPBRec *     WDPBPtr;
```

```
struct FCBPBRec {
        struct FCBPBRec *       qLink;
        short                   qType;
        short                   ioTrap;
        Ptr                     ioCmdAddr;
        ProcPtr                 ioCompletion;
        short                   ioResult;
        char *                  ioNamePtr;
        short                   ioVRefNum;
        short                   ioRefNum;
        short                   filler;
        long                    ioFCBIndx;
        long                    ioFCBFlNm;
        short                   ioFCBFlags;
        unsigned short          ioFCBStBlk;
        long                    ioFCBEOF;
        long                    ioFCBPLen;
        long                    ioFCBCrPs;
        short                   ioFCBVRefNum;
        long                    ioFCBClpSiz;
        long                    ioFCBParID;
};
typedef struct FCBPBRec         FCBPBRec;
typedef struct FCBPBRec *       FCBPBPtr;
```

```
struct VCB {
        struct VCB *            qLink;
        short                   qType;
        short                   vcbFlags;
        short                   vcbSigWord;
        long                    vcbCrDate;
        long                    vcbLsMod;
        short                   vcbAtrb;
        unsigned short          vcbNmFls;
        short                   vcbVBMSt;
        short                   vcbAllocPtr;
        unsigned short          vcbNmAlBlks;
        long                    vcbAlBlkSiz;
        long                    vcbClpSiz;
        short                   vcbAlBlSt;
        long                    vcbNxtCNID;
        unsigned short          vcbFreeBks;
        char                    vcbVN[27];
        short                   vcbDrvNum;
        short                   vcbDRefNum;
        short                   vcbFSID;
        short                   vcbVRefNum;
        char *                  vcbMAdr;
        char *                  vcbBufAdr;
        short                   vcbMLen;
        short                   vcbDirIndex;
        short                   vcbDirBlk;
        long                    vcbVolBkUp;
        unsigned short          vcbVSeqNum;
        long                    vcbWrCnt;
        long                    vcbXTClpSiz;
        long                    vcbCTClpSiz;
        unsigned short          vcbNmRtDirs;
        long                    vcbFilCnt;
        long                    vcbDirCnt;
        long                    vcbFndrInfo[8];
        short                   vcbVCSize;
        short                   vcbVBMCSiz;
        short                   vcbCtlCSiz;
        unsigned short          vcbXTAlBlks;
        unsigned short          vcbCTAlBlks;
        short                   vcbXTRef;
        short                   vcbCTRef;
        long                    vcbCtlBuf;
        long                    vcbDirIDM;
        short                   vcbOffsM;
};
```

```
struct DrvQEl {
        struct DrvQEl *     qLink;
        short               qType;
        short               dQDrive;
        short               dQRefNum;
        short               dQFSID;
        short               dQDrvSize;
};
```

3. Functions

3.1 High-Level Functions

3.1.1 Accessing Volumes

OSErr *GetVInfo* (drvNum, volName, vRefNumPtr, freeBytesPtr)
short drvNum; OSStrPtr volName; short * vRefNumPtr;
long * freeBytesPtr;

OSErr *GetVol* (volName, vRefNumPtr)
OSStrPtr volName; short * vRefNumPtr;

OSErr *SetVol* (volName, vRefNum)
OSStrPtr volName; short vRefNum;

OSErr *FlushVol* (volName, vRefNum)
OSStrPtr volName; short vRefNum;

OSErr *UnmountVol* (volName, vRefNum)
OSStrPtr volName; short vRefNum;

OSErr *Eject* (volName, vRefNum)
OSStrPtr volName; short vRefNum;

3.1.2 Changing file contents

OSErr *Create* (fileName, vRefNum, creator, fileType)
OSStrPtr fileName;
short vRefNum; OSType creator, fileType;

OSErr *FSOpen* (fileName, vRefNum, refNumPtr)
OSStrPtr fileName;
short vRefNum, * refNumPtr;

OSErr *FSClose* (refNum)
short refNum;

OSErr *OpenDriver* (name, refNum)
Str255 name; short refNum;

OSErr *CloseDriver* (refNum)
short refNum;

OSErr *FSRead* (refNum, countPtr, buffPtr)
short refNum; long * countPtr; Ptr buffPtr;

OSErr *FSWrite* (refNum, countPtr, buffPtr)
short refNum; long * countPtr; Ptr buffPtr;

OSErr *GetFPos* (refNum, filePosPtr)
short refNum; long *filePosPtr;

OSErr *SetFPos* (refNum, posMode, posOff)
short refNum, posMode; long posOff;

OSErr *GetEOF* (refNum, logEOF)
short refNum; long *logEOF;

OSErr *SetEOF* (refNum, logEOF)
short refNum; long logEOF;

OSErr *Allocate* (refNum, countPtr)
short refNum; long * countPtr;

OSErr *Control* (refNum, opCode, opParams)
short refNum, opCode; OpParamPtr opParams;

OSErr *Status* (refNum, opCode, opParamsptr)
short refNum, opCode; OpParamPtr * opParamsptr;

OSERR *KillIO* (refNum)
short refNum;

3.1.3 Changing Information about Files

OSErr *GetFInfo* (fileName, vRefNum, fndrInfoPtr)
OSStrPtr fileName; short vRefNum; FInfo * fndrInfoPtr;

OSErr *SetFInfo* (fileName, vRefNum, fndrInfo)
OSStrPtr fileName; short vRefNum; FInfo fndrInfo;

OSErr *SetFLock* (fileName, vRefNum)
OSStrPtr fileName; short vRefNum;

OSErr *RstFLock* (fileName, vRefNum)
OSStrPtr fileName; short vRefNum;

OSErr *Rename* (oldName, vRefNum, newName)
OSStrPtr oldName, newName; short vRefNum;

OSErr *FSDelete* (fileName, vRefNum)
OSStrPtr fileName; short vRefNum;

3.2 Low-level functions

3.2.1 Initialization

pascal void *InitQueue* ()

3.2.2 Accessing Volumes

pascal OSErr *PBMountVol* (paramBlock)
ParmBlkPtr paramBlock;

pascal OSErr *PBGetVInfo* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHGetVInfo* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBGetVol* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHGetVol* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBSetVol* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHSetVol* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBFlushVol* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBUnmountVol* (paramblock)
ParmBlkPtr paramBlock;

pascal OSErr *PBOffLine* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBEject* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

3.2.3 Changing File Contents

pascal OSErr *PBCreate* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHCreate* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBDirCreate* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBOpen* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHOpen* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBOpenRF* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHOpenRF* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBLockRange* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBUnlockRange* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBRead* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBWrite* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBGetFPos* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBSetFPos* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBGetEOF* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBSetEOF* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBAllocate* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBAllocContig* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBFlushFile* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBClose* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

3.2.4 Changing Information about Files

pascal OSErr *PBGetFInfo* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHGetFInfo* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBSetFInfo* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHSetFInfo* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBSetFLock* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHSetFLock* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBRstFLock* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHRstFLock* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBSetFType* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBSetFVers* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBRename* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHRename* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBDelete* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBHDelete* (hparamBlock, async)
HPrmBlkPtr hparamBlock; Boolean async;

pascal OSErr *PBControl* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBStatus* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

pascal OSErr *PBKillIO* (paramBlock, async)
ParmBlkPtr paramBlock; Boolean async;

3.2.5 Accessing Queues

pascal QHdrPtr *GetFSQHdr* ( )

pascal QHdrPtr *GetVCBQHdr* ( )

pascal QHdrPtr *GetDrvQHdr* ( )

3.2.6 Hierarchial-Only Routines

pascal OSErr *PBGetCatInfo* (paramBlock, async)
CInfoPBPtr paramBlock; Boolean async;

pascal OSErr *PBSetCatInfo* (paramBlock, async)
CInfoPBPtr paramBlock; Boolean async;

pascal OSErr *PBCatMove* (paramBlock, async)
CMovePBPtr paramBlock; Boolean async;

pascal OSErr *PBOpenWD* (paramBlock, async)
WDPBPtr paramBlock; Boolean async;

pascal OSErr *PBCloseWD* (paramBlock, async)
WDPBPtr paramBlock; Boolean async;

pascal OSErr *PBGetWDInfo* (paramBlock, async)
WDPBPtr paramBlock; Boolean async;

(Duplicate pages - First issued with 1.06h)

List Manager Functions

The functions described in this section allow a C program to access Macintosh List Manager routines.

The constants, data structures, and functions described in this section are defined in the header file *list.h*.

1. Constants

```
/* Automatic scrolling */

#define lDoVAutoscroll      2
#define lDoHAutoscroll      1

#define lOnlyOne            0x0080
#define lExtendDrag         0x0040
#define lNoDisjoint         0x0020
#define lNoExtend           0x0010
#define lNoRect             0x0008
#define lUseSense           0x0004
#define lNoNilHilite        0x0002
#define lInitMsg            0
#define lDrawMsg            1
#define lHiliteMsg          2
#define lCloseMsg           3

typedef struct Point        Cell;
typedef char                DataArray[32001];
typedef char *              DataPtr;
typedef char **             DataHandle;
```

2. Data Structures

```
struct ListRec {
        Rect            rView;
        GrafPtr         port;
        Point           indent;
        Point           cellSize;
        Rect            visible;
        ControlHandle   vScroll;
        ControlHandle   hScroll;
        Byte            selFlags;
        Boolean         lActive;
        Byte            lReserved;
        Byte            listFlags;
        long            clikTime;
        Point           clikLoc;
        Point           mouseLoc;
        Ptr             lClikLoop;
        Cell            lastClick;
        long            refCon;
        Handle          listDefProc;
        Handle          userHandle;
        Rect            dataBounds;
        DataHandle      cells;
        short           maxIndex;
        short           cellArray[1];
};

typedef struct ListRec          ListRec;
typedef struct ListRec *        ListPtr;
typedef struct ListRec **       ListHandle;
```

3. Functions

3.1 Creating and Disposing of Lists

```
pascal ListHandle LNew  (rView, dataBounds, pass(cSize), Point,
                                theWindow, drawIt, hasGrow,
                                scrollHoriz, scrollVert);
    Rect *rView, *dataBounds;  Point cSize;
        short theProc;  WindowPtr theWindow;
        Boolean drawIt, hasGrow, scrollHoriz, scrollVert;

pascal void LDispose ( lHandle)
    ListHandle lHandle;
```

3.2 Adding and Deleting Rows and Columns

pascal short *LAddColumn* (count, colNum, lHandle)
short count, colNum; ListHandle lHandle;

pascal short *LAddRow* (count, rowNum, lHandle)
short count, rowNum; ListHandle lHandle;

pascal void *LDelColumn* (count, colNum, lHandle)
short count, colNum; ListHandle lHandle;

pascal void *LDelRow* (count, rowNum, lHandle)
short count, rowNum; ListHandle lHandle;

3.3 Operations on Cells

pascal void *LAddToCell* (dataPtr, dataLen, pass(theCell), lHandle)
Ptr dataPtr; short dataLen; Cell theCell; ListHandle lHandle;

pascal void *LClrCell* (pass(theCell), lHandle)
Cell theCell; ListHandle lHandle;

pascal void *LGetCell* (dataPtr, dataLen, pass(theCell), lHandle)
Ptr dataPtr; short *dataLen; Cell theCell; ListHandle lHandle;

pascal void *LSetCell* (dataPtr, dataLen, pass(theCell), lHandle)
Ptr dataPtr; short dataLen; Cell theCell; ListHandle lHandle;

pascal void *LCellSize* (pass(cSize), lHandle)
Point cSize; ListHandle lHandle;

pascal short *LGetSelect* (next, theCell, lHandle)
Boolean next; Cell *theCell; ListHandle lHandle;

pascal void *LSetSelect* (setIt, pass(theCell), lHandle)
Boolean setIt; Cell theCell; ListHandle lHandle;

3.4 Mouse Location

pascal short *LClick* (pass(pt), modifiers, lHandle)
Point pt; short modifiers; ListHandle lHandle;

pascal long *LLastClick* (lHandle)
ListHandle lHandle;

3.5 Accessing Cells

pascal void *LFind* (offset, len, pass(theCell), lHandle)
short *offset, *len; Cell theCell; ListHandle lHandle;

pascal short *LNextCell* (hNext, vNext, theCell, lHandle)
Boolean hNext, vNext; Cell *theCell; ListHandle lHandle;

pascal void *LRect* (cellRect, pass(theCell), lHandle)
Rect *cellRect; Cell theCell; ListHandle lHandle;

pascal short *LSearch* (dataPtr, dataLen, searchProc,
pass(theCell), lHandle)
Ptr dataPtr; short dataLen; Cell theCell; ListHandle lHandle;

pascal void *LSize* (listWidth, listHeight, lHandle)
short listWidth, listHeight; ListHandle lHandle;

3.6 List Display

pascal void *LDraw* (pass(theCell), lHandle)
Cell theCell; ListHandle lHandle;

pascal void *LDoDraw* (drawIt, lHandle)
Boolean drawIt; ListHandle lHandle;

pascal void *LScroll* (dCols, dRows, lHandle)
short dCols, dRows; ListHandle lHandle;

pascal void *LAutoScroll* (lHandle)
ListHandle lHandle;

pascal void *LUpdate* (theRgn, lHandle)
RgnHandle theRgn; ListHandle lHandle;

pascal void *LActivate* (act, lHandle)
Boolean act; ListHandle lHandle;

3.7 List Definition Procedure

pascal void *MyListDef* (lMessage lSelect, lRect, pass(lCell),
lDataOffset, lDataLen, lHandle)
short lMessage; Boolean lSelect;
Rect lRect; Cell lCell;
short lDataOffset, lDataLen; ListHandle lHandle;

Memory Manager Functions

This section describes functions that allow C programs to access the Macintosh Memory Manager routines.

The constants, data structures, and functions described in this section are defined in the header file *memory.h*.

1. Constants

```
#define maxSize        0x800000
```

2. Data structures

```
typedef long            Size;
typedef int             MemErr;
typedef struct Zone *   THz;

struct Zone {
        Ptr             bkLim;
        Ptr             purgePtr;
        Ptr             hFstFree;
        long            zcbFree;
        ProcPtr         gzProc;
        short           moreMast;
        short           flags;
        short           cntRel;
        short           maxRel;
        short           cntNRel;
        short           maxNRel;
        short           cntEmpty;
        short           cntHandles;
        long            minCBFree;
        ProcPtr         purgeProc;
        Ptr             sparePtr;
        Ptr             allocPtr;
        short           heapData;
};
typedef struct Zone     Zone;
```

3. Functions

3.1 Initialization and Allocation

void *InitAppleZone* ()

void *SetApplBase* (startPtr)
Ptr startPtr;

pascal void *InitZone* (growProc, masterCount,
limitPtr, startPtr)
ProcPtr growProc; short masterCount;
Ptr limitPtr, startPtr;

Ptr *GetApplLimit* ();

void *SetApplLimit* (zoneLimit)
Ptr zoneLimit;

short *MaxApplZone* ()

long *MaxBlock* ()

void *MoreMasters* ()

long *StackSpace* ()

3.2 Heap Zone Access

THz *GetZone* ()

void *SetZone* (hz)
THz hz;

THz *SystemZone* ()

THz *ApplicZone* ()

void *PurgeSpace* (total, contig)
long *total, *contig;

3.3 Allocating and Releasing Relocatable Blocks

Handle *NewHandle* (logicalSize)
Size logicalSize;

void *DisposHandle* (h)
Handle h;

Size *GetHandleSize* (h)
Handle h;

void *SetHandleSize* (h, newSize)
Handle h; Size newSize;

THz *HandleZone* (h)
Handle h;

Handle *RecoverHandle* (p)
Ptr p;

void *ReallocHandle* (h, logicalSize)
Handle h; Size logicalSize;

short *MoveHHi* (h)
Handle h;

3.4 Allocating and Releasing Nonrelocatable Blocks

Ptr *NewPtr* (logicalSize)
long logicalSize;

void *DisposPtr* (p)
Ptr p;

Size *GetPtrSize* (p)
Ptr p;

void *SetPtrSize* (p, newSize)
Ptr p; Size newSize;

THz *PtrZone* (p)
Ptr p;

3.5 Freeing space on the Heap

long *FreeMem* ()

Size *MaxMem* (growPtr)
Size * growPtr;

Size *CompactMem* (cbNeeded)
Size cbNeeded;

void *ResrvMem* (cbNeeded)
Size cbNeeded;

void *PurgeMem* (cbNeeded)
Size cbNeeded;

void *EmptyHandle* (h)
Handle h;

Handle *NewEmptyHandle* ( )

3.6 Properties of Relocatable Blocks

void *HLock* (h)
Handle h;

void *HUnlock* (h)
Handle h;

void *HPurge* (h)
Handle h;

void *HNoPurge* (h)
Handle h;

short *HSetRBit* (h)
Handle h;

short *HClrRBit* (h)
Handle h;

short *HGetState* (h)
Handle h;

pascal short *HSetState* (h, flg)
Handle h; short flg;

3.7 Grow Zone Functions

void *SetGrowZone* (growZone)
ProcPtr growZone;

Boolean *GZCritical* ()

Handle *GZSaveHnd* ()

3.8 Utility Routines

void *BlockMove* (sourcePtr, destPtr, byteCount)
Ptr sourcePtr, destPtr; Size byteCount;

Ptr *TopMem* ()

MemErr *MemError* ()

Sane Manager Functions

The functions described in this section allow a C program to access the *S*tandard *A*pple *N*umeric *E*nvironment (SANE) Manager.

Note: SANE is documented in the *Apple Numerics Manual.*

The constants, data structures, and functions described in this section are defined in the header file *sane.h.*

1. Constants

```
#define SIGDIGLEN        20  /* significant decimal digits */
#define DECSTROUTLEN     80  /* max length for decimal string */

/* Decimal Formatting Styles */

#define FLOATDECIMAL      0
#define FIXEDDECIMAL      1

/* Exceptions */

#define INVALID           1
#define UNDERFLOW         2
#define OVERFLOW          4
#define DIVBYZERO         8
#define INEXACT          16

/* Ordering Relations */

#define GREATERTHAN       0
#define LESSTHAN          1
#define EQUALTO           2
#define UNORDERED         3

/* Inquiring Classes */

#define SNAN              0
#define QNAN              1
#define INFINITE          2
#define ZERONUM           3
#define NORMALNUM         4
#define DENORMALNUM       5

/* Rounding Directions */

#define TONEAREST         0
#define UPWARD            1
#define DOWNWARD          2
#define TOWARDZERO        3
```

```
/* Rounding Precisions */

#define EXTPRECISION        0
#define DBLPRECISION        1
#define FLOATPRECISION      2

/* Type Definitions */

typedef short exception;        /* sum of INVALID...INEXACT */
typedef short relop;            /* relational operator */
typedef short numclass;         /* inquiry class */
typedef short rounddir;         /* rounding direction */
typedef short roundpre;         /* rounding precision */
typedef short environment;
```

2. Data Structures

```
typedef struct decimal {
  char   sgn, unused;     /* sign 0 for +, 1 for - */
  short                   exp;     /* decimal exponent */
  struct {unsigned char length, text[SIGDIGLEN], unused} sig;
                          /*significant digits */
} decimal;


typedef struct decform {
  char   sgn, unused;     /* FLOATDECIMAL or FIXEDDECIMAL */
  short  digits;
} decform;


typedef void (*haltvector) ();
```

3. Functions

3.1 Conversions Between Binary and Decimal Records

void *num2dec* (f, x, d)
  decform *f; extended x; decimal *d;

extended *dec2num* (d)
  decimal *d;

3.2 Conversions Between Decimal Records and ASCII Strings

void *dec2str* (f, d, s)
  decform *f; decimal *d; char *s;

void *str2dec* (s, ix, d, vp)
  char *s; short *ix, *vp; decimal *d;

3.3 Arithmetic, Auxiliary, and Elementary Functions

extended *remainder* (x, y, quo)
extended x,y; short *quo;

extended *rint* (x)
extended x;

extended *scalb* (n, x)
short n; extended x;

extended *logb* (x)
extended x;

extended *copysign* (x, y)
extended x,y;

extended *nextfloat* (x, y)
extended x,y;

extended *nextdouble* (x, y)
extended x,y;

extended *nextextended* (x, y)
extended x,y;

extended *log2* (x)
extended x;

extended *log1* (x)
extended x;

extended *exp2* (x)
extended x;

extended *exp1* (x)
extended x;

extended *power* (x, y)
extended x,y;

extended *ipower* (x, i)
extended x; short i;

extended *compound* (r, n)
extended r, n;

extended *annuity* (r, n)
extended r, n;

extended *randomx* (x)
extended *x;

3.4 Inquiry Routines

numclass *classfloat* (x)
extended x;

numclass *classdouble* (x)
extended x;

numclass *classcomp* (x)
extended x;

numclass *classextended* (x)
extended x;

long *signnum* (x)
extended x;

3.5 Environment Access Routines

An exception variable encodes the exceptions whose sum is its value.

void *setexception* (e, s)
exception e; long s;

long *testexception* (e)
exception e;

void *sethalt)* (e, s)
exception e; long s;

long *testhalt* (e)
exception e;

void *setround* (r)
rounddir r; rounddir getround();

void *setprecision* (p)
roundpre p; roundpre getprecision();

void *setenvironment* (e)
environment (e)

void *getenvironment* (e)
environment *e;

void *procentry* (e)
environment *e;

void *procexit* (e)
environment e; haltvector gethaltvector();

void *sethaltvector* (v)
haltvector v;

3.6 Comparison Routine

relop *relation* (x y)
extended x,y

3.7 NaNs and Special Constants

extended *nan* (c)
unsigned char c;

extended *inf* ();

extended /fIpi ();

extended *fabs* (x)
extended x;

extended *sqrt* (x)
extended x;

extended *exp* (x)
extended x;

extended *log* (x)
extended x;

extended *tan* (x)
extended x;

extended *sin* (x)
extended x;

extended *cos* (x)
extended x;

extended *atan* (x)
extended x;

(Duplicate pages - First issued with release 1.06h)

SCSI Manager Functions

The functions described in this section allow a C program to access Macintosh SCSI Manager routines.

The constants, data structures, and functions described in this section are defined in the header file *scsi.h*.

1. Constants

```
#define scInc           1
#define scNoInc         2
#define scAdd           3
#define scMove          4
#define scLoop          5
#define scNOp           6
#define scStop          7
#define scComp          8

#define badParmsErr     4
#define CommErr         2
#define compareErr      6
#define phaseErr        5
```

2. Data Structures

```
struct SCSIInstr {
        short           opcode;
        long            param1;
        long            param2;
};
typedef struct SCSIInstr    SCSIInstr;
```

3. Functions

pascal OSErr *SCSIReset* ( )

pascal OSErr *SCSIGet* ( )

pascal OSErr *SCSISelect* (target)
    short target;

pascal OSErr *SCSICmd* (buffer, count)
    Ptr buffer; short count;

pascal OSErr *SCSIRead* (tibPtr)
    Ptr tibPtr;

pascal OSErr *SCSIRBlind* (tibPtr)
    Ptr tibPtr;

pascal OSErr *SCSIWrite* (tibPtr)
    Ptr tibPtr;

pascal OSErr *SCSIComplete* (stat, message, wait)
        short *stat, *message; long wait;

pascal OSErr *SCSIStat* ( )

EXAMPLES

Chapter Contents

Examples

This chapter describes some sample programs, which illustrate how C programs can access the special features of the Macintosh.

The *Explorer* Desk Accessory

This is a description of the *Explorer* desk accessory written with the Aztec C68K compiler system. The information provided is accurate to the best of our knowledge. The best way to read this description is with a printed copy of the source to the program which is located in *sys2:example/explor.c.*

The *Explorer* desk accessory is a program which creates a window that displays, in either hexadecimal or ASCII, the contents of ram memory. The window comes complete with a scrollbar on the right side of the window which allows the user to scroll through memory. In addition, a small box is displayed in the window where the user can type a new start address. When Return or Enter is pressed, the display is changed to start with the address in the box.

The first part of the program is almost the only assembly language part. It is placed at the beginning of the program so that it ends up being the first thing in the resource. The first four words define information used by the Desk Manager when the desk accessory is installed.

The first word signifies that the accessory will respond to control calls and needs to be updated once in a while. The second word tells how often to update the accessory. The number specified is in sixtieths of a second, so an update every five seconds is being asked for. The third word signifies which types of events the desk accessory is intending to handle. This mask is generated using the same bit masks used for the event manager in general. Refer to the *Event Manager* section of *Inside Macintosh* for more information. Finally, the ID number of the menu used by the desk accessory is specified. Desk accessory menu ID's must be negative numbers.

The next five words are used by the Desk Manager to access the different functions of the desk accessory. For this accessory, only the *open(), control()* and *close()* functions actually do anything. The *prime* and *status* entry points are directed to a routine which performs *No OP*eration. Each word is an offset from the beginning of the driver to the function itself. This is specified by subtracting the *main* label which was placed at the beginning of the table entries from each of the function names. Note that function names in Aztec C are denoted by appending an underscore to the name.

Finally, the title of the desk accessory is defined as a Pascal string. All of the components thus far are considered standard parts of any desk accessory and any device driver as well. Refer to the sections *Desk Manager* and *Device Manager* in the *Inside Macintosh* manual for more details of these components.

The next part of the assembly language is part of the magic that allows a desk accessory to access its own global data without

overwriting the global data of the current application program. Normally, the compiler, assembler and linker use register A5 to reference global data. However, as shall be seen later, this program was compiled to use register A4 instead. The line of code after the *save__* label sets up register A4 to point to the end of the code and data for the desk accessory. It does so by adding the size of the code and the size of the data to the *main* label and placing the resulting address into the A4 register. Now any references to global data will occur as desired. The symbols *__Uend, __Dorg,* and *__Cend* are automatically generated by the linker.

The first use of this is in the next two lines. When the desk accessory is called by the Desk Manager, the registers A0 and A1 contain pointers to a ParamBlkRec and a Device Control Entry. These structures contain information indicating what function the accessory is to perform. To access them from C, they are stored in two global pointers, *Pbp* and *Dp.* Then control is returned to the calling routine. This routine is called *save().*

The *restore()* routine restores the A0 register which is not saved by the Desk Manager when calling the accessory. The Desk Manager saves registers A1-A6 and D3-D7 whenever it calls an accessory.

That's it for assembly language till a bit later. The rest is almost entirely C code. It begins by including a whole set of header files. This program uses quite a few different functions from the Macintosh toolbox and requires all those listed. Preceding the includes are two #define's. The first, **__DRIVER** is used in the *quickdraw.h* file to keep from defining a set of global variables which are normally used by an application. It wouldn't hurt to have them defined, but it would make the accessory bigger than it needs to be.

The second #define, **SMALL__MEM,** is used to keep the amount of symbol space used by the compiler down. On a 128K Macintosh this is necessary because of the huge number of symbols which are a part of the toolbox. The #undef is used later because those files following it need to be used in their entirety. Since almost all other header files include *quickdraw.h,* it should be defined before the others. If the last four files had been specified first, then *quickdraw.h* would have been processed with SMALL__MEM undefined and used up more symbol table space than necessary.

The next set of lines are a set of macro definitions for use later in the program. The first two define the Boolean results TRUE and FALSE. TRUE is defined as 0x100, because of the way Pascal defines the Boolean when passed to or returned from a function. NLINES is the number of lines we will be displaying in the window. MENUID is our familiar ID we saw in the assembly language part. It must be the same.

MEMTOP is an example of how one easily refers to some absolute memory location using Aztec C. In this case, location 0x108 is treated as having a long value stored in it. When defined this way, MEMTOP can be used as a regular global variable, and may be read or written.

The last #define is a short-hand we will use later on. Dp is the pointer to the Device Control Entry that is passed by the Desk Manager. It points to a structure which contains a field, dCtlStorage, which will hold a handle to some data used by the accessory. The macro is casting the general handle, dCtlStorage, to a handle to a structure of type *storage*. Then, it dereferences the handle once to get a pointer to a structure of that type. We'll see this used later as,

```
SP->xxxx
```

to reference a member of the structure. Otherwise we would have to always write:

```
(*(struct storage **)Dp->dCtlStorage)->xxxx
```

Next, we have the global data definitions. First, the two pointers, Dp and Pbp, are defined. The DCE structure is defined in the header file *desk.h*. The ParamBlkRec structure is, of course, in *pb.h*. These are followed by a set of rectangle definitions. Wind_rect is the initial window position and size in global coordinates. Scrl_rect is the rectangle defining the right-hand scroll bar of the window. Cont_rect defines the content region of the window. Full_rect is the sum of the content region and the scroll bar region. Finally, Edit_rect defines the rectangle used for editing the starting address and changing the cursor.

The Ibeam Cursor structure defines the basic cursor shape used when editing. Whenever the cursor enters the Edit_rect, the cursor will be changed to the I-beam shape defined here.

The storage structure holds information used by the desk accessory. It will be discussed more when talking about the *open()* routine. The first entry is a handle to the menu item which will be created by the accessory. The second item is a handle to the scroll bar control. *where* is the pointer into memory that will be used to display its contents. Next is a flag used to indicate that automatic updating has been enabled. *size* indicates the number of elements displayed on a single line. It has a value of 8 when displaying in hexadecimal and 16 when displaying in ASCII. It is also used as a flag when displaying the startup message. *incr* is used to scale the motion of the thumb in the scroll bar. Finally, *hte* is a handle to the TextEdit record that will be used.

So much for the preliminaries. The first four routines correspond to the four entries in the offset table at the beginning of the file. The *open()* and *close()* routines are normally called once each when the desk accessory is initialized and when it is closed. *open()* may be

called even while the desk accessory is active. In that case it performs no function. The Desk Manager will simply make its window the active window. The *control()* routine does the bulk of the work responding to events and actions of the user.

The *open()* routine's primary function is to get things set up. The first thing it does is to call the *save()* routine, which is the assembly language routine which sets up register A4, and the pointers, Dp and Pbp. Then, it copies the Dp pointer into a register version. While not necessary, since we will be accessing the pointer a lot, it will generate smaller faster code. Next, we check if this is the first time the driver has been opened. If the window field of the DCE structure is empty, it is the first time, else we drop to the bottom, restore register A0, and return. If it is not the first time, we get to work.

First, we allocate some storage. You might well ask, why not just make it a global structure?? Well, the way desk accessories work, is that they share the application heap space with the application currently running. While the accessory's window is active, the accessory is locked in memory. However, when the accessory's window is not active, the memory occupied by the accessory's code and data may be needed by the active process. In that case, the code and data will be purged from memory. Anything stored there will be lost. Now, a desk accessory could be set up to be non-purgeable, and in that case, the data would not be lost. However, that might cut down on the usefulness of the program and is not a good general practice.

Instead, we will allocate our data as a relocatable chunk of memory which constitutes our minimum needs. In this case, about 22 bytes. We will keep the handle in the DCE structure which will not be purged even if the code is. At the same time, we temporarily lock the structure so that anything we do here won't move it. This is necessary because of the next line, where we set a structure pointer to point to the structure itself. If we didn't do this, one of the later calls might cause the structure to move, in which case the *sp* pointer would be pointing to the wrong location.

Now we set up the window. First we call *NewWindow()* with the predefined Wind_rect rectangle and the appropriate arguments. The window will be visible with a goaway box and will be the frontmost window. The type of window is a regular document window without a grow zone. The title of the window will be *Explorer*, using the name defined earlier. After saving the window pointer, we change the windowKind field to be the RefNum of the desk accessory. This is necessary so that the application program will be able to determine that the window is a desk accessory. Then, we create the scroll bar control as part of the window.

After initializing our storage variables to zero, we set the increment of the scroll bar. Since the scroll bar can have values from 0 to 0x7fff,

we divide the size of ram memory which is always located in location 0x108 of low memory by 0x8000. This gives us the amount of each click of the scroll bar. We will be using this value later.

Lastly, we set up the menu. First, we get a new menu item with id MENUID and title *Explorer*. Then, we place the menu items in the menu. There are five items, four are real, and one is a separator. The fourth item is initialized with a check mark in front of it. This is indicated by the !\x12 following the item. The menu will appear and disappear from the menu bar as the accessory's window becomes active and inactive. Finally, we unlock the handle, restore register A0, and return.

The *close()* routine must release all the storage used by the accessory. It disposes of the TextEdit record, the window, the menu, and the storage used by the accessory. It sets the dCtlWindow field to zero, so the next *open()* will work correctly. After restoring register A0, the routine returns.

The remaining routines are all involved in handling events generated by the user and by the Desk Manager. The information is passed to the accessory through the ParamBlkRec structure pointer. The csCode element of the structure contains the type of action to be performed. Refer to *Inside Macintosh* for details of the different actions. The *control()* routine handles all the different actions directly, with the exception of event type actions. They are handled by a separate function.

First, the routine sets up a pointer to the storage information. Then, it makes the accessory's window the active port using the *SetPort()* call. Then it determines what kind of action to perform. If the action is an event action, the function, *doevent()*, is called with a pointer to the storage, and a pointer to the event record. The different actions interpret the csParam field of the ParamBlkRec individually. For events, it is a pointer to the event record. However, since csParam is defined as a union, to get a pointer to an event record, the address of csParam is cast as a handle to an event record, and the handle dereferenced once to get a pointer.

The *accRun* action is specified whenever the specified 300 clock ticks have expired. In this case, we check to see if the automatic update flag is set. If so, we redraw the window.

The *accCursor* action is similar to the *accRun* action, but is called almost continuously when the accessory's window is active. It is used to give the accessory the ability to change the cursor's shape based on its position. In this case, if the TextEdit record is active, we call *TEIdle()*, which will display the blinking line used when editing text. Then, the current position of the mouse is placed in the *pt* variable. Next, we check to see if the point specified is in the rectangle we are using for editing. Notice that we check the Boolean result from the

Pascal function by bitwise anding it with 0x100. This is necessary because of the way Pascal defines the Boolean type as being the low order bit of the high order byte. The contents of the low order byte are not defined, so a simple test against zero may not always work.

If the cursor is in the rectangle, the cursor is changed to the I-beam shape defined before. Otherwise, *InitCursor()* will restore it to the normal arrow shape.

The next case deals with menu actions. For menu items, csParam is treated as an array of two integers, the first containing the menu number, and the second the item number. Since we only have one menu, we can assume that that is the one selected. So, we will just switch on the item number by casting the csParam address to be a pointer to an int and then indexing to the second element.

The first item is the automatic update item. First we toggle the storage value, and then set the check mark according to the new value of the storage value. The second item is manual refresh and we just draw the window for that one. The third item was the line of dashes and we should never see that one. The fourth and fifth items act like radio buttons. If one is selected, the other is deselected. In this case, the size element of the storage structure is used to track which item is currently active. Size is set to 8 or 16 depending on which item has been selected. Then the check mark is removed or added accordingly to both menu items. Finally, the window content area is erased and then redrawn in the new format.

The last five actions support the standard TextEdit menu selections of the Edit menu. All use the corresponding TextEdit function with the exception of Undo, which is not supported by this accessory.

After the appropriate action is performed, the storage is unlocked, the register A0 is restored, and control is returned to the Desk Manager.

The next routine to look at is the event handler, *doevent()*. The first type of event handled are keyboard events. The keyboard is used to type in a new start address for the window display. The first thing that is checked is whether there were any modifier keys being held down at the same time as the key. If so, the key is ignored, and a beep is sounded.

Otherwise, the character is checked to see if it falls in the range of valid hexadecimal values. If so, the *TEKey()* procedure is used to act upon the key press. If not, the character is checked to see if it is either the return or the enter key. The enter key returns a value of 3. If it is one of the two, then the routine, *xtol()*, is called to evaluate the value in the current TextEdit record. We'll check that routine out later.

If the value is a valid one, then the where field of the storage structure is set to the new value. Then, the TextEdit record is set so that all the text is selected. This is done so that a new number can be typed without deleting the old one. The first character typed will delete the whole record unless the mouse is used to change the selection range first. Finally, the window is redrawn at the new position. It has been assumed here, that the *draw_wind()* routine will fix the position of the scroll bar thumb.

The next event that is handled is the mouse down event. There are two places where a mouse down is important. First, if it is in the TextEdit area, the *TEClick()* routine takes the appropriate action for setting the selection range. There are actually two checks, first for the content area, and then for the TextEdit record area. This allows for future expansion of doing something with mouse events in the rest of the content area.

If the mouse button is pressed in the scroll bar area, the function, *FindControl()*, is used to determine which part of the contol is being accessed. There are five parts to a scroll bar. The first two are the line at a time buttons at the top and the bottom of the bar. For each of these, the routine, *TrackControl()*, is called with the appropriate parameters. The last parameter passed is the address of a routine to call for each press of the mouse button or for continued pressing of the button. Notice that the two routines which are passed have been declared as being of type *pascal void*. This declaration indicates that it is a function which follows the Pascal calling conventions and does not return a value. Each routine will scroll the window one line in the proper direction each time it is called. We'll look at these routines later.

The next two routines deal with scrolling the window a page at a time. In this case, a common subroutine is called with the control handle, the part code and the number of lines to scroll. The sign of the line number will indicate the direction. The last part is the thumb itself. This part is accessed when the mouse is clicked in the box itself and is dragged to a new absolute location. This is also handled by *TrackControl()*, but this time no action is taken and no function address is passed.

TrackControl() will drag an outline of the box around and return when the mouse button is released. At that point, the thumb will have a new value. This value is retrieved using the *GetCtlValue()* function and is multiplied by the scaling increment to calculate a new address in memory to display. Thus, when the thumb is dragged to the top, the value of the thumb is 0 and the address displayed is zero. When the thumb is dragged to the middle, its value is half of its maximum value or 0x4000. On a 128K Macintosh, the increment is 4, so the new address will be 4 * 0x4000 or 0x10000, which is 64K, halfway to 128K. On a 512K Macintosh, the increment is 16, so the new address will be

16 * 0x4000 or 0x40000, which is 256K, halfway to 512K.

The other two events handled by the accessory are the activation event and the update event. The activation event occurs when the accessory's window changes from active to inactive or from inactive to active. The direction is determined by the low order bit of the *modifiers* element of the event record. Shortly after opening the desk accessory, an activate event will occur. This is used to display the startup message by calling the *signature()* function. This is signalled by the size field of the storage structure being zero. It is then initialized to 8 which is the hexadecimal display format.

There are three other actions performed by the activate event. First, the menu for the desk accessory is added to or removed from the menu bar as appropriate. Next, the scroll bar control is displayed or hidden. And finally, the TextEdit activation or deactivation routine is called. Then, in either case, the menu bar is redrawn.

An update event occurs when a part of the window must be redrawn, usually because another window was obscuring a part of it and was then moved or deleted. In this case, the standard *BeginUpdate()* routine is called, then the window is redrawn, the scroll bar is redrawn, and finally *EndUpdate()* is called.

And that's all the events handled by the accessory. Now we will look at the subroutines which perform some of the actions previously described.

The first function is a relatively unimaginative routine to display the title, author, and company for a few seconds. It mostly consists of a set of calls to move to a position and draw the appropriate string there. Then, it waits 100 clock ticks before erasing the window. Finally, the font and size are set to Monaco 9 and the TextEdit record handle in the storage structure is initialized. The rectangle passed to *TENew()* is inset because it is going to be framed later, and this way when the text is highlighted, there is a gap between the frame and the inverted text.

Now, for the real workhorse routine, *draw_wind()*. This routine draws all the contents of the window each time it is called, as well as setting the value of the thumb based on the start address. The first thing it does is to set a register pointer to the storage structure. Next, it copies the start address into an unsigned long for some work. First it performs some bounds checks to make sure that the start address is not less than 0. If so, it is set to 0. The next check is to make sure that the start address does not display any data beyond the end of ram memory. It does this by calculating the number of items that could be displayed at one time in the window. This is simply the number of items on one line, *sp->size*, times the number of lines, *NLINES*. It then subtracts this number from the maximum address to achieve the maximum start address. If the start address is beyond this, it is set to

it. Then, the storage value is reset. Next, the thumb is set using the *SetCtlValue()* routine with the current start address scaled by the increment field of the storage structure.

The display area is then set to be only the content rectangle by using the *RectRgn()* routine to set the clipRgn of the current window to that rectangle. The clipRgn is used to limit where things are displayed. This prevents the display of the characters from overwriting the scroll bar control. The title to the TextEdit record is displayed, and the TextEdit record itself is displayed. Finally, the TextEdit record is framed with a box.

Each line is then drawn one at a time by working through the loop. First, the variable, *k*, is set to the offset from the start address for this line. Then, the pen is moved to the appropriate position for this line. The character array, *buf*, is used to temporarily hold the line while it is being formatted. The character pointer, *cp*, is used to add characters to the buffer. First, the address is put in the beginning of the buffer. Only the last six digits of the address are displayed. Each digit is calculated by taking the remainder after division by 16 and using it as an index into an array of characters representing the 16 hexadecimal digits. This is done for either display format.

If the format is hexadecimal, the next eight values in memory are displayed using a similar technique for each digit. The hex values are separated by a blank. The string is terminated by a null, and then drawn after being converted to Pascal format. If the format is ASCII, only the address is displayed. Then, the buffer is filled with the actual values from memory, with the exception that any value that falls outside of the ASCII range is replaced by a '.'. That buffer is then drawn.

After all NLINES lines have been drawn, the clipRgn is set back to the full content region so that the scroll bars can be painted as well.

The next three routines handle the line by line scrolling when the mouse is in the arrow box of the scroll bar with the mouse down. Both *scrlup()* and *scrldn()* are declared as being of type *pascal void*. As mentioned before, this means that the arguments passed to each is the reverse of the order normally used for C. It also causes the compiler to generate code that saves the extra registers D3 and A2 which are not normally saved by a C function. Finally it generates code at the return from the routine which pops all the arguments off the stack. Both routines call the common handler *scroll()* with the control handle, the code passed to the routine and a flag indicating which routine is calling the common routine.

The common routine begins and ends with a small amount of assembly language code. This is necessary since it is possible that the Pascal routine, *TrackControl()*, may have changed the value in the A4 register. So, we save the register value on the stack and reload the A4

register using the same statement as before. Next, we check to see if the mouse is still in the control that it started in. If so, we adjust the start address by the size of one line in the appropriate direction and then redraw the window. Note that this means that we aren't doing any fancy scrolling. The time to redraw the window is so short, that it is not worth the extra code to get fancy. Finally, the register is popped back off the stack and control is returned to the calling routine.

The semi-colon before the **#asm** was placed there because of a bug in the compiler. When the compiler reaches the right curly bracket at the end of the *if* statement, it checks for an *else* statement. Part of the check involves processing lines for macros and pre-processor directives. As a result, the *#asm* is parsed and sent to the output file before the terminating label is sent. This bug was found while writing this program. Without the semi-colon, the register was only popped back off the stack if the *if* statement evaluated as true. This means that if you put the mouse in the scroll arrow and held down the button, it worked just fine. However, if while holding the button down, the mouse was moved outside the arrow, then the condition was false, and the program blew up spewing garbage all over the screen, ejecting disks and doing other vile, nasty things. Remove the semi-colon and recompile it to see what we mean.

The next routine sort of does what the *TrackControl()* routine does for the up and down line arrows for the up and down page controls. It sits in a loop, while the mouse button is still down, and checks to see where the mouse is positioned. If the mouse is in the control's area, then the start address is adjusted by one full page size, which is the size of one line times the number of lines. Then the *draw_wind()* routine is called to update the display and the thumb.

The last routine is the routine called by *doevent()* to evaluate the TextEdit record as a new start address. First it picks up the length from the TextEdit record. Notice, that to use a handle to a structure to reference an element of the structure, the handle must be dereferenced once, in parentheses, to get a pointer which can then be used. Next it picks up a pointer to the text. Since hText is a handle as well, it is dereferenced once to get a pointer to the text. Then, each character of the text is looked at. If it is in the hexadecimal range, the value is added to the running total kept in the long variable, *l*. If it is not a hexadecimal digit, then the selection range is set to that character and a minus one returned. After all the characters have been processed, the calculated value is returned.

That's it for the code itself, just a few words about compiling and linking. To compile the program use the command:

```
cc -abu -z200 explor.c
as -s -ZAP explor.asm
```

The -a option prevents the automatic start of the assembler. The -b

option inhibits the generation of the

public .begin

statement which is used by general application programs. The -u option causes the compiler to generate code which leaves the A4 register free. The -z200 reduces the size of the compiler's string buffer, which allows the program to be compiled on a 128K Macintosh. The -s option to the assembler, tells it to do multiple passes which allows it to generate short branches and eliminate some instructions where possible. The -ZAP tells it to remove the input file when finished.

To link the program type:

ln -an Explorer explor.o -lc

which links it as a desk accessory whose name is *Explorer*, whose resource type is DRVR and whose ID is 31. To actually use the program install it in the System file with a command of the form:

sys2:bin/cprsrc DRVR 31 explor sys:System

which will copy the resource from the file *explor* into the System file.

Menu Definition Example

The files in the *mdef/* directory demonstrate two methods of creating and using a menu definition procedure (proc). A menu definition proc is used if one wants to change the way a standard menu appears. One of the fields of the MenuRecord structure is a handle to a procedure which defines draws and responds to mouse activity.

There are two ways to have the menu definition proc take effect. The first way is to actually have the procedure as part of the main program. In this case, just change the appropriate field of the structure to point to the new proc. The second way is to have the procedure be a resource and define a menu resource that refers to it. The first way is advantageous if the menu proc needs to access data that is part of the application. The second way is useful for creating a general purpose menu definition proc which may be placed in the system file and used by different programs.

In the examples presented here, there is a single routine which performs the menu definition. It is in the file *mymenu.c*. To illustrate the first method, it is linked with a slightly modified version of the *edit.c* program. A fourth menu is added called *Test* that uses the menu definition proc. See the file *mkall* for the compiling and linking procedure. See the *setup()* function in *edit.c* for the calling sequence.

The second method is demonstrated by using the *grow.c* program. In this case, the file *mymenu.o* is linked separately, and the output is used with the resource compiler to create a resource of type MDEF. The changes to *grow.c* are also in the *setup()* function. The changes to *grow.r* define the MDEF resource and also the MENU resource which uses the MDEF resource had to be specified in detail and could not use the predefined type.

The menu definition procedure itself just sets the size of the menu, highlights or dehighlights the entire menu if the mouse is moved into the rectangle, and initializes it to light gray if the menu is not enabled.

MENU

NAME

db - symbolic debugger

SYNOPSIS

db [options] [progfile] [arg1 arg2 ...]

DESCRIPTION

db is used to debug programs which have been created using the Aztec C compiler, assembler, and linker.

db has all the standard features of an assembly language debugger. It also has features not found in all debuggers, such as the ability to reference memory locations by name as well as by address, the ability to define sequences of commands to be macros, which can then be activated by entering a single letter, and a flexible mechanism for handling breakpoints.

In addition, *db* has features specifically tailored to its use with Aztec C, such as the ability to list the name and parameters of the currently executing function, and the function that called it, and so on, back to the initial function. Another special feature is the ability to display, on entry and exit from each function, the function's parameters and return value.

Requirements

A Macintosh with at least 512K of memory is recommended for use with *db*. The debugger itself uses about 96K.

Preview

The remainder of this description of *db* is in three sections: *overview*, which describes *db* features in more detail and introduces the commands; *usage*, which describes in full detail how to use *db*; and a *command summary*.

DEBUGGING UTILITIES

The operator can also define names to *db* using the *v* command, and the 'clear symbols' command, *cs*, will remove symbols from the memory-resident symbol table.

1.2.1 Code and Data symbols.

db classifies symbols as being either code or data symbols. All symbols in the program's symbol table resource which occur between the special linker symbols __Corg__ and __Cend__ are considered to be code symbols, and all others are data symbols.

There are two commands for viewing the symbols which are known to *db*: *dc* and *dd*, which display code and data symbols, respectively.

1.3 Loading programs and symbols

A program and its symbols can be loaded into memory when *db* is started; in this case, the command line defines the program to be loaded. The *db* 'load program' command, *lp*, can also be used. When told to load a program, *db* automatically tries to load the program's symbol table, too.

The 'load symbols' command, *ls*, can be used when *db* did not start the current program.

When the *lp* command finds a symbol table resource, it clears the symbol table of all symbols before loading the new symbols.

Only one user program can be in memory at once. If an *lp* command is entered before a currently loaded program has exited, the current program is terminated by the debugger before the new program is loaded.

When a program exits, it must be reloaded with the *lp* command before execution can begin again.

1.4 Breakpoints

Before transferring control of the processor to a user's program in response to a *g* command, *db* can set "breakpoints" at specified locations in the code. When the user's program reaches a breakpoint, *db* regains control.

A breakpoint has a 'skip count' associated with it, which allows a breakpoint to be passed several times before actually taking the breakpoint and returning control to *db* and the user. When a breakpoint is reached, *db* is always activated; it increments a counter associated with the breakpoint. When the counter's value is greater than the breakpoint's skip count, the breakpoint is taken; that is, *db* retains control of the processor. Otherwise, *db* returns control of the processor to the user's program after the breakpoint. By default, a breakpoint's skip count is 0; thus, each time the breakpoint is reached, it's taken.

Debugging Utilities

This chapter describes the debugger utility *db* that is provided with some versions of Aztec C68K.

meets the specified condition.

When an *s* command is used to single-step a program and a memory-change breakpoint is set, *db* will examine the specified memory location after each instruction is executed, and take a breakpoint when appropriate.

The *bb* and *bw* commands are used to set and remove memory-change breakpoints.

1.6 Trace mode

db supports a 'trace mode', which displays information whenever a function or Macintosh system trap is entered or exited.

With this mode enabled, on entry to a function or Macintosh system trap, the function or trap name and its arguments are displayed, and, optionally, on exit from a function, its return value is displayed. The return value is not displayed if a Macintosh system trap was entered.

The commands *bt* and *bT* affect trace mode: *bt* enables and disables trace mode, and *bT* enables and disables the display of function exit information.

1.7 Backtracing

When *db* regains control from an executing program (for example, because a breakpoint was taken), it has the ability to display information on how the program got to its current location: the *ds* command will display information about the currently executing function, and the function which called it, and so on, back to the Manx function *Croot*, which called the user's function *main*.

ds displays, for each function, its name, arguments which were passed to it, and the address to which it will return.

1.8 Macros

db allows the user to define and execute 'macros'; that is, a sequence of *db* commands.

A macro is associated with a single alphabetical character, so up to 26 macros can be known to *db* at any time.

The *db* command *x* is used both to define and execute a macro.

1.9 Displaying source files

db allows the user to display source files, thus providing a convenient means to examine the source of a program being debugged.

Only a single source file can be examined at a time. The 'load source file' command, *lf*, defines the source file to be displayed, and the 'display source lines' command, *df*, displays its lines.

1. Overview

db commands consist of one or two characters, the first of which identifies the command category. If there's only one command in the category, then the command has just this one letter; otherwise, the command has a second letter which identifies the specific operation to be performed.

1.1 Basic commands

db has two types of commands for examining memory: display and print, whose first characters are *d* and *p*, respectively. The 'display' commands *db* and *dw* simply display hexadecimal bytes and words.

The 'print' command, *p*, is more powerful, being able to convert a sequence of one or more possibly different types of data items to ASCII. For example, you can tell it that beginning at the location *var* is a sequence of the following items: an *int*, a *float*, and a pointer to a *char* string. The *p* command will convert the two binary items to ASCII and print them, and display the referenced character string.

The 'register' command, *r*, displays and modifies the 68000 registers.

The 'memory modify' commands, *m*, modify memory.

The *u* commands 'unassemble' code; that is, display it symbolically, in a form similar to its appearance in an assembly language source file.

The *s* and *g* commands cause the user's program to be executed. *s* commands "single step" the user's program; that is, execute a specified number of instructions in the user's program and then return control to *db*. *g* commands transfer control of the processor unconditionally to the user's program. In this case, *db* regains control when the user's program terminates, when an error occurs (such as division by zero), or when a "breakpoint" is taken. Breakpoints are discussed below.

? is the *help* command: it causes *db* to display a summary of all *db* commands. For some command categories, you can get information about the commands in a category by typing the first letter of the category's commands followed by a *?*. For example, typing *m?* gets you information about the memory modification commands (all of whose first letter is *m*).

1.2 Names

db allows memory locations to be referenced by name as well as by location. It learns a program's global names by reading the resource SYMS from the program's resource file. and placing them in a memory-resident symbol table. The linker generates a symbol table resource for a program in response to the -w option.

db only allows global symbols to be accessed by name; automatic variables and static variables can't be accessed by name.

2. Using DB

2.1 Starting DB

db is started with a command of the form:

db [options] [progfile] [arg1 arg2 ...]

where the options are:

-s#	Set the symbol table up to hold # symbols (default is 300). The linker reports the number of symbols put in the 'SYMS' resource when the program has been linked with the *-w* option.
-a	Use the 'printer' port (port A) for input/output.
-b	Use the 'modem' port (port B) for input/output.
-r#	Set the baud rate for input/output to #. The default baud rate is 9600 baud.

Note that the options *-a*, *-b*, and *-r* only apply when using an attached external terminal for input and output.

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

If the program file name specifies a drive or directory, *db* searches for the program file in just that particular area. Otherwise, it searches the current directory.

The "arg" parameters are passed to the program using the *argv* parameter of the program's *main* function: *arg1* is pointed at by *argv[1]*, *arg2* by *argv[2]*, and so on. *argv[0]* always contains zero.

db must be invoked under the SHELL.

2.2 Using DB with an external terminal

db can be used with an external terminal for input and output. Debugging messages can be viewed on the external terminal's screen while the program's output appears on the Macintosh screen. A cable is needed to connect the port on the terminal with either port A or port B on the back of the Macintosh and requires the following pin setting. The pin setting on the external terminal side assumes a 25 pin RS232 port.

Note: On some external terminals, the settings for pins 2 and 3 may need to be switched.

A breakpoint can also have a sequence of *db* commands associated with it. When a breakpoint is taken, these commands will be executed before *db* allows the operator to enter commands. For example, if you just want to examine a variable each time a certain location in the code is reached and then have the program continue execution, you could define a breakpoint at the location, and specify a list of commands to do just that: the first command in the sequence would be a *d* command to display memory, and the second would be a *g* command to continue execution of the program.

There are two ways to define breakpoints: with the *g* command, and with special breakpoint commands, whose first letter is *b*.

The breakpoint commands manipulate a table of breakpoints: there are commands for entering breakpoints into the table, displaying the entries, reseting their counters, and removing them from the table.

There's a difference between a breakpoint defined in a *g* command and those in the breakpoint table: the *g* command breakpoint is temporary, while a breakpoint table is more permanent (it exists until removed from the table). Before transferring control to the user's program in response to a *g* command, *db* sets all breakpoints that are in the breakpoint table and that are specified in the *g* command itself. When a breakpoint is taken, *db* removes all breakpoints from the code and forgets all about the *g* command breakpoint. The breakpoint table breakpoints, however, are still in the table and will be set back in memory when control is again returned to the user's program.

db remembers the skip counter associated with a breakpoint which is in the breakpoint table: when it sets breakpoints in memory, the count for such a breakpoint is set to its remembered value (that is, its value in the table); and when a breakpoint is taken, the accumulated count for the breakpoints in memory are saved in the breakpoint table.

1.5 Memory-change breakpoints

The breakpoints described above are taken when a program reaches a specified point in the code. A second type of breakpoint, called a memory-change breakpoint, is taken when a specified memory location is changed from or set to a particular value.

With a memory-change breakpoint set, *db* will detect either the function or the instruction which modifies the specified memory location, depending on whether the user's program was activated using a *g* command or is being single-stepped using an *s* command, respectively.

When the user's program is activated with a *g* command and a memory-change breakpoint is set, *db* will examine the specified memory location on entry to, and exit from, each function. It will take a breakpoint, that is, interrupt execution of the program and return control to the operator, when the contents of the memory location

An EXPR has a 16-bit value. The operators that are applied to the TERMS out of which the EXPR is built affect just this 16-bit value.

When an EXPR refers to a memory location (that is, it is built up from an ADDR), the 16-bit value is the offset of the location from the beginning of the segment containing it. In this case, the EXPR can also specify the beginning paragraph number of the segment containing the location. For more discussion about this, see the description of ADDR below.

2.3.1.2 The Definition of TERM

A TERM always resolves to a numeric value, and can be one of the following:

REGISTER
CONSTANT
-TERM
ADDR
*ADDR
#ADDR
.
@ [function]
(EXPR)

These names are defined in the following paragraphs.

REGISTER

Registers are specified by their standard names; that is, A0, D0, PC and so on. The value of the TERM is the contents of the register.

CONSTANT

A CONSTANT can be a decimal, hexadecimal, or octal number, or a character.

A sequence of digits preceded by '*0x*' is taken to be a hexadecimal number and those preceded by '*0b*' are binary numbers. A sequence of digits with a leading *0* is taken to be an octal value. Digit strings ending in . are taken to be decimal values. If none of these prefixes or suffixes are present, the radix of the value is taken from the current radix. The default radix is hexadecimal. Note that if the current radix is set to hexadecimal, numbers must start with a digit to distinguish them from symbols (i.e. *fabc* will be taken to be a symbol where *0fabc* will be taken to be a hexadecimal number.

A character is represented by the character, surrounded by single quotes, as in 'x'. The value of a character constant is its ASCII value.

Certain characters, the single quote ', and the \ may also be defined within the single quotes. These are identified by a leading backslash character, and are:

The 'find string' command, *f*, will find a character string in the source file.

1.10 Other features

Some other features of *db* which haven't yet been discussed are:

* The 'evaluate expression' command, =, does just that.
* The 'help' command, *?*, lists commands.
* The 'exit' command allows the user to exit to the SHELL without removing *db* from the system.
* The 'input radix' command changes the default radix for input and display.

command never modifies its associated '.'.

@ [function]

The @ symbol has as its value the return address of the specified function. The function name is optional, and defaults to the current function. The main use for @ is in the *g* command.

For example,

```
g @
```

transfers control to the user's program, and sets a breakpoint at the return address of the current function.

As another example,

```
g @putc
```

transfers control to the user's program. When the function *putc* is reached, a breakpoint will be set at the address to which it will return.

2.3.1.3 The Definition of ADDR

An ADDR defines the address of a location in memory, and has the form:

```
EXPR
```

Here are some examples of ADDR:

```
pc
main+10
.-40
*sp+8                  Reference to location on the stack.
data+*(a6+6)
```

2.3.1.4 The Definition of RANGE

A RANGE defines a block of memory. It has one of the following forms:

```
ADDR,CNT
ADDR>ADDR
ADDR
,CNT
```

The form ADDR,CNT specifies the starting address, ADDR, and a number, CNT. CNT is interpreted differently by different commands. For example, the 'disassemble code' command, *u*, will display CNT lines, while the 'display bytes' command, *db*, will display CNT bytes.

The form ADDR>ADDR specifies the starting and ending addresses of the range.

A full range need not be explicitly specified, because *db* remembers the last-used range and will set unspecified RANGE parameters from

external terminal	MacPort A (or B)
2	5
3	9
7	3
20	6

To start, on the Macintosh, type *db* followed by any options. Option *-a* or *-b* must be selected. Option *-r* must be selected if the baud rate is other than 9600 baud, the default baud rate.

For example, if the cable is connected to port A on the Macintosh and the external terminal port is set up for 4800 baud, start *db* to run the program *hello* by specifying on the Macintosh:

```
db -a -r4800 hello
```

From here, all input is supplied from the external terminal's keyboard and all output is sent to the external terminal's screen.

2.3 Commands

This section describes in detail the *db* commands. It first defines some terms that are used in the command descriptions. These terms are *expr*, *term*, *addr*, *range*, and *cmdlist*.

2.3.1 Definitions

2.3.1.1 The Definition of EXPR

An EXPR has the following form:

TERM [binop TERM ...]

That is, an EXPR can be a single TERM or a series of TERMS separated by binary operators. The binary operators are:

+ -	addition
- -	subtraction
* -	multiplication
/ -	division
% -	modulus
& -	bitwise and
| -	bitwise inclusive or
^ -	bitwise exclusive or

All operators have the same precedence, and an unparenthesized EXPR is evaluated left to right. If you want to override the default order of evaluation of an expression, you can parenthesize the relevant parts of the expression.

In the parameterized form of the commands, ADDR specifies the field to be monitored.

With the '==' form, the breakpoint will be triggered when the debugger detects that the field is equal to the specified value, VAL.

With the '!=' form, the breakpoint will be triggered when the debugger detects that the field is different from the specified value.

The VAL parameter is optional. If not specified, it defaults to the current value at the ADDR.

---

bc - Clear a single breakpoint
bC - Clear all breakpoints

Syntax:

bc ADDR
bC

Description:

These commands delete breakpoints from the breakpoint table.

bc deletes the single breakpoint specified by the address ADDR, and *bC* deletes all breakpoints from the table. If ADDR is a trap name preceded by an '!', the breakpoint for this trap is cleared.

---

bd - Display breakpoints

Syntax:

bd

Description:

bd displays all entries in the breakpoint table.

For each breakpoint, the following information is displayed:

* Its address, using a symbolic name, if possible; the name will start with an '!' if it is a trap breakpoint.
* The number of times it's been 'hit' without a breakpoint being taken.
* The skip count for it;
* The command list for it, if any.

For example, a *bd* display might be:

char	*hex value*	*db notation*
newline	0a	\n
horizontal tab	09	\t
backspace	08	\b
carriage return	0d	\r
form feed	0c	\f
backslash	5c	\\
single quote	27	\'
bit pattern	ddd	\ddd

ADDR

A TERM can be an ADDR; that is, a reference to a location in memory. See the definition of ADDR, below, for more details.

*ADDR

When a TERM consists of a * followed by an ADDR, the value of the TERM is the contents of the 32-bit field referred to by the ADDR. For example,

*VAR	The contents of the VAR field;
*A7	The contents of the 32-bit field in the data segment pointed at by A7;
*SP	The contents of the 32-bit field on the top of the stack;
*(LBL+2)	The contents of the 32-bit field referred to by LBL+2;

Because an ADDR can itself be an EXPR, the *ADDR term may require extra parentheses. For example,

```
*sp+2
```

is equivalent to *(sp+2) and not (*sp)+2. The value of the first interpretation is the contents of the second word on the stack, while the value of the second is two plus the contents of the first word on the stack.

period(.)

The value of a TERM consisting of a period, '.', is the starting address ADDR of the last similar command. For example, if ten bytes of memory were displayed using the *db* command, as in

```
db 0x100,10
```

then '.' would be set to 0x100 for the next db or dw command. If the next db or dw command is

```
dw .
```

the same 10 bytes would be displayed as words.

The '.' has a separate value for the *u* command, for the *db*, *dw*, and *m* commands, for the *p* command, and for the *df* command. An *m*

---

bt - Toggle the trace mode flag
bT - Toggle the return trace mode flag

Syntax:

bt
bT

Description:

bt and *bT* toggle the trace mode and return trace mode flags, respectively.

The state of the trace mode flag determines whether trace mode is enabled or disabled.

The state of the return trace mode flag determines whether the tracing of a function's return is enabled or disabled. If trace mode is disabled, the return trace mode flag has no effect.

2.4.2 The Clear Commands

cs - Clear symbol table

Syntax:

cs

Description:

cs removes all symbols from the debugger's memory-resident symbol table.

When a program is loaded using the *lp* command, the *cs* command is automatically called.

2.4.3 The Display Commands

db - Display memory in bytes
dw - Display memory in words
dl - Display memory as longs
d - Display memory in last format

Syntax:

db [RANGE]
dw [RANGE]
dl [RANGE]
d [RANGE]

Description:

The *db*, *dw* and *dl* commands display successive bytes, words and double words of memory, respectively. *d* displays memory using the last format specified; for example, if *d* is entered, and *db* was the last 'display memory' command, then *d* will display bytes,

the remembered values:

* When a RANGE is specified which consists of a single ADDR, the last used CNT is used.
* When a RANGE is specified which consists of ',CNT', the next consecutive address is used, and the remembered count is changed to the new value.
* When nothing is specified as the RANGE, the next consecutive address is used as the starting ADDR, and the CNT is set to the remembered value.

2.3.1.5 The Definition of CMDLIST

A CMDLIST is a list of commands. It consists of a sequence of commands or macros separated by semicolons:

COMMAND [;COMMAND ...]

If a macro is in a CMDLIST, it must be the last command in the list.

2.4 Command descriptions

The following descriptions of debugger commands uses terms and concepts which were presented in the preceding sections.

The commands are listed alphabetically. For an index, see the command summary which follows the descriptions.

2.4.1 The Breakpoint Commands

bb - Set Byte Memory-Change Breakpoint
bw - Set Word Memory-Change Breakpoint
bl - Set Long Memory-Change Breakpoint

Syntax:

bb
bw
bl
bb ADDR == [VAL]
bb ADDR != [VAL]
bw ADDR == [VAL]
bw ADDR != [VAL]
bl ADDR == [VAL]
bl ADDR != [VAL]

Description:

These commands are used to set and clear a memory-change breakpoint, with the parameterized versions used to set breakpoints and the parameter-less version to clear them. The *bb* command is used to monitor a one-byte field, the *bw* command to monitor a two-byte (word) field and the *bl* command is used to monitor a four-byte field.

The starting line number is optional; if not specified, the display starts with the "current" line.

The current line in a source file is set by the source file commands *lf*, *df*, and *f*, as follows:

* When the file is first loaded with the *lf* command, the first line in the file is the current line;
* When the last source file command was 'display source', *df*, the current line is the line following the last one displayed;
* When the last source file command was 'find string', *f*, the current line is the line in which the string was found.

df also sets the "F-dot" for the source file to the number of the first line displayed. The F-dot is the line referred to when the starting line number of the range in a *df* command specifies a period (.). Also, source string searches begin at the line following the F-dot line.

Each displayed line is preceded with a line number in decimal, a colon, and the line itself.

---

dg - Display global values

Syntax:

dg

Description:

For each data symbol in the debugger's symbol table, *dg* displays the contents of the 16-bit field referenced by that symbol.

---

dha - Display application heap structure

Syntax:

dha

Description:

This command displays the application heap structure. The address and size of each block is shown as well as any attributes associated with it such as relocatable, locked, purgable or free.

address	hits	skip	command
printf__	1	2	
putc__	0	0	db __Cbuffs

In this example, two breakpoints are in the table. The first is at the beginning of the function *printf__*; a breakpoint will be taken for it every third time it is reached, and no command will be executed. Given its current hit count, a breakpoint will be taken the second time *printf__* is reached.

The second breakpoint is at the function *putc__*; a breakpoint will be taken each time the function is reached, and will display memory, in bytes, starting at __Cbuffs.

---

br - Reset breakpoint counters

Syntax:

br [ADDR]

Description:

br resets the 'hit' counter for the specified breakpoint which is at the address, ADDR. If ADDR isn't given, the 'hit' counters for all breakpoints in the breakpoint table are reset. If ADDR is '!' followed by a trap name, the trap counter is reset.

---

bs - Set or modify a breakpoint

Syntax:

[#] bs ADDR [;CMDLIST]

Description:

bs enters a breakpoint into the breakpoint table, or modifies an existing entry.

The optional parameter # is the skip count for the breakpoint. If not specified, the skip count is set to 0, meaning that each time the breakpoint is reached it will be taken.

The optional parameter CMDLIST is a list of debugger commands to be executed when the breakpoint is taken. If ADDR is '!' followed by a trap name, this trap is breakpointed.

2.4.5 The 'Find Source String' Command

f - find string in source file

Syntax:

fSTRING

Description:

This command searches the current source file (that is, the one specified by the last *lf* command) for a specified string.

The search begins at the line following the current line.

If the string is found, the current line and the F-dot line of the source file is set to the line containing the string; otherwise, these values are unchanged.

STRING is the character string to be located, and consists of all characters following the *f* and preceeding the carriage return.

If the first character of the string is '^', the search will begin with the first character on a line. In this case, '^' isn't part of the search string.

The 'current line' and F-dot line for a source file are defined in the description of the *df* command.

2.4.6 The Go commands

g - Execute the program
G - Execute the program, without setting table breakpoints

Syntax:

[#]g [@ <function>] [ADDR] [;CMDLIST]
[#]G [@ <function>] [ADDR] [;CMDLIST]

Description:

The *g* commands transfer control of the processor to the user's program, at the address specified by PC. The user's program then executes until it terminates, an error such as division by zero occurs, or a breakpoint is taken; control then returns to the debugger program.

The parameters to the 'g' commands allow one or two temporary breakpoints to be set in memory before the user's program is executed.

The difference between the 'g' and the 'G' command is that the 'G' command sets in memory just the breakpoints specified in the command itself, while the 'g' command also sets the breakpoints specified in the breakpoint table.

too.

The starting address of the RANGE parameter is optional; if not specified, it defaults to the ending address of the last display's RANGE, plus one.

Each line of the display begins with the address, followed by a hexadecimal display of 16 bytes, 8 words or 4 double words, followed by an ASCII display, by bytes, of the same data. For the ASCII display, values falling outside the range 0x20 to 0x7f are displayed as a period.

If the ending address does not fall on a multiple of 16 bytes, only the number of bytes or words specified in the last line will be displayed.

---

dc - Display all code symbols

Syntax:

dc

Description:

dc lists all the code symbols in the memory-resident symbol table and all user-defined symbols.

For each symbol, its name and address are displayed.

---

dd - Display all data symbols

Syntax:

dd

Description:

dd lists all the data symbols in the memory-resident symbol table.

For each symbol, its name and address are displayed.

---

df - Display source file lines

Syntax:

df [RANGE]

Description:

df displays lines from the source file which was specified in the last *lf* command.

The RANGE parameter specifies the numbers of the lines to be displayed.

---

lp - Load program

Syntax:

lp
lp progfile [arg1 arg2 ...]

Description:

lp loads a program into memory. If a symbol table file can be found for the program, it will be loaded, too.

If the *lp* command is given without parameters, the last *lp* command is re-executed. The following comments describe the parameterized version of *lp*.

Loading the program

The parameter *progfile* specifies the file containing the program.

If the program file name specifies a drive or path, the file is searched for in just that location; otherwise, it's searched for on the current directory.

If an attempt is made to load a program before a currently loaded program has terminated, the current program will be terminated by the debugger before the new program is loaded.

Loading the symbol table

After the program is loaded, the memory-resident symbol table is cleared of all symbols except for those defined with the *v* command, and an attempt is made to locate and load the program's symbol table from the resource 'SYMS' in the current resource file.

The U-dot (that is, the value of the period parameter associated with the *u* commands) is set to the PC. The D-dot (the value of the period parameter for the *d* and *m* commands) is set to 0.

Once a program exits, it must be reloaded with an *lp* command before it can begin again.

---

ls - load symbols

Syntax:

ls

Description:

ls loads symbols from the specified resource 'SYMS' in the current resource file into the debugger's memory-resident symbol table, after first clearing the memory-resident table of all but

---

dhs - Display system heap structure

Syntax:

dhs

Description:

This command displays the system heap structure. The address and size of each block is shown as well as any attributes associated with it such as relocatable, locked, purgable or free.

---

ds - Display Stack Backtrace

Syntax:

ds

Description:

ds displays information about the current function or trap, the function which called it, and so on, back to Croot, the Manx function which called the user's function *main*.

For each function, the information consists of the function's name, the parameters passed to it, and the address to which it will return.

The arguments are displayed as a series of 16-bit hex values. If an argument is actually of type long or double, it will be displayed as separate words.

ds determines the number of parameters by looking at the instructions which follow the address to which the function will return.

ds assumes that the A6 register points to the C stack frame for the current function, unless the current instruction is within 4 bytes of the start of the function.

2.4.4 The Exit Command

e - exit

Syntax:

e

Description:

This command permits the user to exit from the debugger without removing it from memory. The user can then re-enter the debugger at any later time simply by pressing the Interrupt button on the left-hand side of the Macintosh.

---

mm - Move memory

Syntax:

mm RANGE = ADDR

Description:

mm copies one block of memory to another.

The RANGE parameter specifies the source block and ADDR the starting address of the block to be modified.

---

ms - Search memory

Syntax:

ms RANGE = EXPR1 [EXPR2 ...]

Description:

ms searches a block of memory for a sequence of bytes having specified values. For each match, the corresponding address of the start of the string is displayed.

RANGE specifies the block of memory. The EXPR parameters are expressions, each of whose resulting values is one byte of the search sequence.

2.4.9 The Radix Command

n - Change radix

Syntax:

nX

Description:

This command changes the default radix (hexadecimal) for user input or debugger display. X is a single character 'b', 'd', 'o', or 'x' which changes the default radix to binary, decimal, octal, or hexadecimal respectively.

The radix can be forced to a given value by specifying one of the following prefixes before the desired number:

0x	hex
0o	octal
0b	binary

To display a number in decimal, the number must have a . (period) appended to it.

The '#' and 'ADDR' parameters define one of the temporary breakpoints that a *G* or *g* command can set:

* # is the skip count for the breakpoint; it defaults to zero, meaning that the breakpoint is taken every time it's reached;
* ADDR is the address for the breakpoint;

If ADDR is '!' followed by a trapname, the debugger will set a temporary breakpoint on this trap.

The '@ <function>' parameter specifies that a temporary breakpoint is to be set at the return address of the specified function. If the function isn't specified, it defaults to the current function. If a function is specified, the breakpoint is set to the address to which the function will return. In this case, the breakpoint isn't set until the function is entered; thus, in programs which call the function from several different places, the breakpoint will be set at the actual address to which the function will return.

The ';CMDLIST' parameter defines a sequence of debugger commands, separated by semicolons, that the debugger is to execute once a breakpoint which is specified in the 'go' command is taken. If this parameter isn't specified, it defaults to the command list used for the last temporary breakpoint.

Before setting breakpoints and transferring control to the user's program, the debugger single-steps the user's program, (that is, causes it to execute one instruction). This allows the operator to transfer control to a location in the program at which there is a breakpoint, without immediately triggering a breakpoint and re-entry to the debugger.

2.4.7 The Load Commands

lf - Load a source file

Syntax:

lf filename

Description:

lf opens the specified source file for subsequent examination by the *df* command.

If a file has already been opened by a previous *lf*, it's closed before the new file is opened.

pd var

The code *x* says "take the two-byte binary value at the current address, convert it to hexadecimal, and print the result". So the hexadecimal value of *var* could be printed with the command:

px var

* *indir* is a string of zero or more * characters, which are indirection indicators specifying that the value at the current data item is a pointer to a chain of zero or more pointers, the last of which points to an object whose type and requested conversion are defined by *desc_code*.

To find the data object corresponding to a format item that has indirection indicators, *p* begins by setting its idea of the address of the data object to the current address. It then works its way from left to right through the indirection indicators; for each indicator it replaces its current idea of the data object address with the pointer that is in the field at this address. The data object address is distinct from the current address: at the end of this process, the *p* command's current address is simply incremented past the first pointer.

A * specifies that the pointer within the field referenced by the current data object address is four bytes long. This pointer is the offset component of the new data object address from the last segment referenced.

For example, if the variable *cp* is a pointer to a character string (that is, its declaration is *char *cp*), then the string pointed at by *cp* could be printed by the command

p*s cp

Here we have made use of the *s desc_code*, which specifies that the data object is a character string, and that the string's characters are to be printed, with possible modifications as noted below, up to a terminating null character. After this command, the *p* command's current address is set to the byte immediately following *cp*.

As another example, if *cpp* is a pointer to an array of pointers to character strings (that is, the declaration of *cpp* is *char **cpp*), then the string pointed at by the first element of the array could be displayed with the command

p**s cpp

those symbols defined with the *v* command.

2.4.8 The Memory Modification Commands

mb - Modify bytes of memory
mw - Modify words of memory
ml - Modify double words of memory

Syntax:

mb ADDR EXPR1 [EXPR2 ...]
mw ADDR EXPR1 [EXPR2 ...]
ml ADDR EXPR1 [EXPR2 ...]

Description:

mb, *mw* and *ml* modify bytes and words of memory, respectively.

The parameter ADDR specifies the address of the first byte or word to be modified.

The EXPR parameters are expressions, whose resulting values are set in memory, with EXPR1 set in the first byte or word specified, EXPR2 set in the next higher byte or word, and so on.

The EXPR parameters can be separated by spaces or commas.

---

mc - Compare memory

Syntax:

mc RANGE = ADDR

Description:

mc compares two blocks of memory and, for each comparison which fails, displays the corresponding address, and value.

RANGE specifies one of the blocks of memory. The second begins at ADDR and has the same length as the first block.

---

mf - Fill memory

Syntax:

mf RANGE = EXPR

Description:

mf sets each byte in a block of memory to a specified value.

The RANGE parameter specifies the memory block, and EXPR an expression whose resulting value is the value to be set in the range.

sixteen.

The second item causes the print command to again take the item at the current address as a pointer, increment the current address by four, and then convert to decimal and print the four successive two-byte values that begin at the address defined by the pointer. At the end of the process, the current address has been advanced by four.

As an example of the use of format strings containing several format items, consider the following code:

```
struct {
   int *ip;
   float flt;
   char *cp;
} var = {&i, 3.14159, "ralph"};
int i=2;
```

The command

```
p*d4-xf*s4-x var
```

will print

```
2 xxxxxxxx 3.14159 ralph yyyyyyyy
```

where *xxxxxxxx* is the hexadecimal address of *i* and *yyyyyyyy* is the hexadecimal address of the string.

A complete list of the desc_codes

We have introduced some of the *desc_code*s above. Here is a list of the basic *desc_code*s:

b	Convert to hexadecimal and print a byte.
d	Convert to decimal and print a two-byte signed binary value.
D	Convert to decimal and print a four-byte signed binary value.
f	Convert and print a four-byte *float.*
F	Convert and print an eight-byte *double.*
o	Convert to octal and print a two-byte field.
O	Convert to octal and print a four-byte field.
x	Convert to hexadecimal and print a 2-byte field.
X	Convert to hexadecimal and print a 4-byte field.
u	Convert to decimal and print an unsigned, two-byte value.
U	Convert to decimal and print an unsigned, four-byte value.
p	Print a pointer in address form with translation.
P	Print a pointer in address form without translation.
c	Print a character with translation.

2.4.10 The 'Print' Command

p - formatted print

Format:

p[format] [ADDR][,COUNT]

Description:

p generates a formatted display of memory of a section of memory, by converting data items in memory to a displayable form as directed by the format conversion string *format.*

format is a list of format specifications, each of which defines the type of a data item and the conversion to be performed on it.

p works its way through the *format* string, converting and displaying data items in memory as requested by the format string items. When *p* reaches an item in the *format* string, it converts the data item at its 'current address' as directed by the format item. When it finishes processing a format string item, it increments its current address by the size of the data item that it just processed, so as to be ready to process the next data item as directed by the next format string item.

The *format* string is optional; if not specified, the *format* string used by the previous *p* command is used.

ADDR specifies the address of the first data item that *p* is to convert and display. If ADDR is not entered, the starting address is assumed to be the print command's 'current address'. Normally, this is the address of the first byte beyond the last data item converted by the last *p* command. However, there is a *format* item that causes *p* to remember the address contained in the current data item, and then make that the current address after it finishes processing the entire format string.

COUNT specifies the number of times that *p* is to work its way through the *format* string. Each time through, *p* begins at the current address that was left by the last time through. If COUNT isn't specified, it defaults to one time.

The format items have the form

[rpt][indir_flgs][size]desc_code

where

* *desc_code* is a single-letter code that defines the type of the data item and the conversion to be performed upon it. For example, the code *d* says 'take the two-byte binary value at the current address, convert it to decimal, and print it'. So if *var* is an *int*, the following command could be used to print its value in decimal:

pointer to this structure. The program that uses this structure and field will chain symbol table items together, and set a pointer to the head of the chain in *sym_head.*

```
struct symbol {
   struct symbol * sym_next;
   char *sym_name;
   unsigned sym_val;
} *sym_head;
```

The following command would display the symbol table item pointed at by *sym_head* and then set the *p* command's current address to the next symbol table item, which is pointed at by the *sym_next* field in the first item:

```
pA"symbol name="*snt"value="x sym_head
```

After this command is entered, you can display successive symbol table items by simply entering

```
p
```

The *p* command's current address is correctly set to the next table item, and since a format string isn't specified, the *p* command will use the one that it last used.

You can print out multiple symbol table items by entering a single *p* command. To do this, place a comma and the maximum number of items to be printed after the command's starting address. The command will follow the chain, printing symbol table items until it either prints the specified number of items or it prints an item whose *sym_next* pointer is null. In the latter case, it will terminate and leave the *p* command's current address set to the address of the last symbol table item. For example, entering

```
pA"symbol name="*snt"value="x sym_head,100
```

will print symbol table items until it either prints 100 items or it prints an item having a null *sym_next* pointer.

2.4.11 The Quit command

q - Quit the debugger

Syntax:

q

Description:

q terminates the program being debugged, restores any modified interrupt vectors, and returns control to the operating system.

Following this command, the *p* command's current address is set to the byte following *cpp*.

* The *rpt* parameter of a format item defines the number of times that the item is to be processed. It allows a sequence of *rpt* identical format items to be abbreviated by just one such item with a leading *rpt* count.

  For example, if *a* is an array of *float*s, then the first five items in this array could be displayed with the command

  ```
  p5f a
  ```

  This command uses the fact that the *desc_code* to convert a four-byte floating point value at the current address to a displayable value is *f*. This command is equivalent to the command *pfffff a*. At the end of this command, the *p* command's current address is set to the address of the byte following the last displayed *float*.

* The *size* parameter of a format item defines the number of data items that are to be converted and printed. When the format item doesn't use indirection, *size* has the same effect as *rpt*; for example, in the *p5f a* command above, the 5 could be interpreted as being a *size* parameter instead of a *rpt* parameter.

  When the format item does use indirection, then the *size* parameter defines the number of data items to be converted and printed at the end of the indirection chain. For example, if a module defines *lpp* as a pointer to an array of pointers to array of *long*s (that is, the declaration of *lpp* is *long **ip*), then the following command would display the first four *long*s pointed at by the first element of the pointer array:

  ```
  p**4D lpp
  ```

  Here we have used the *D desc_code*, which specifies that a four-byte signed binary value is to converted to decimal and printed. The following command would display the first three *long*s pointed at by the first three elements of the pointer array:

  ```
  p3*4D *lpp
  ```

To demonstrate further the difference between the *rpt* and *size* fields in a format item, consider the format items *4*d* and **4d*. The first causes the print command to take the item at the current address as a pointer, increment the current address by four, convert to decimal and print the two-byte value referenced by the pointer, and then repeat the process three more times. At the end of the process, the current address has been advanced by

Chapter Contents

2.4.14 The Unassemble commands

u - Unassemble memory, with symbols
U - Unassemble memory, without symbols

Syntax:

u RANGE
U RANGE

Description:

These commands 'disassemble' a range of memory; that is, display the assembly language instructions in the range.

The *u* and *U* commands differ in that the *u* command will make use of the symbol table during disassembly and the *U* command won't. Also, the *U* command displays, for each instruction, the hex value of each byte of the instruction, whereas the *u* command won't.

With the *u* command, the disassembly of an instruction which references memory displays the location as the symbol nearest to the location plus an offset, if possible. With the *U* command, the location is displayed as a hexadecimal value.

The RANGE parameter specifies the area of memory to be disassembled. It gives the starting address, and either the number of instructions to be disassembled, or the ending address of the area.

2.4.15 The Variable commands

v - Create a new symbol
V - Modify the value of an existing symbol

Syntax:

v SYMBOL = ADDR
V SYMBOL = ADDR

Description:

The *v* and *V* commands are used to create a new symbol or modify the value for an existing symbol, respectively, in the debugger's memory resident symbol table.

SYMBOL is the name of the symbol being created or modified, and ADDR is its address.

The symbol will be classified as a code symbol.

C	Print a character without translation.
s	Print a string up to a terminating null byte with translation.
S	Print a string up to a terminating null byte without translation.

For the C and S *desc_codes*, each character is printed "as is", with no translations.

For the c and s codes, printable ASCII characters (that is, whose hex value is between 0x20 and 0x7f) are printed "as is". A character whose hex value is less than 0x20 is printed as two characters: ^ followed by the printable character whose hex value equals the original character's value plus 0x40. A character whose hex value is 0x80 or greater is displayed as a ' character followed by the one or two characters that would be printed for the character whose hex value equals that of the original character less 0x80. For example, 0x41, 0x1, and 0x81 would be printed as A, ^A, and '^A, respectively.

The following *desc_codes* can be used to assist in the formatting of the *p* output:

character	*output*
N or n	Output a newline character
R or r	Output a blank character
T or t	Output a tab character
"string"	output "string"

These characters can be preceded by a count specifying the number of characters or strings to be output.

The next group of *desc_codes* change the *p* command's notion of the current address. They don't cause any printing.

^	Back up the current address by the size of the last data item.
- or +	Back up or advance, respectively, the current address by *size* bytes, where *size* is a decimal value preceding the - code. If *size* isn't specified, it defaults to one byte.
A or a	Remember the pointer that is contained in the current data object; If this pointer is not null, set the *p* command's current address to this value after the entire format string has been processed. If the pointer is null, set the *p* command's current address to the value it had before the entire format string was processed.

The A and a *desc_codes* are useful for printing the elements of a linked list. For example, consider the following code, which defines the structure for a symbol table item, and declares *sym_head* to be a

symbol table has been loaded, the closest symbol is displayed as well.

2.4.19 The Help command

? - list commands

Syntax:

?

Description:

This command lists the debugger commands. For groups of related commands, the listing usually lists the first letter of the commands followed by a ?. You can get a listing of all the commands in such a group by typing the the letter, the ?, and return. For example, the listing for the 'display' commands is *d?*; thus you can type *d?* followed by return to get a listing of all the 'display' commands.

2.4.12 The Register command

r - Register display

Syntax:

r
r <reg>=EXPR

Description:

r displays and modifies the registers, including the status registers, of the program being debugged.

The parameter-less version displays the registers.

The parameterized version modifies the contents of a register, with <reg> being the name of the register to be modified, and EXPR an expression whose resulting value is to be set into the register.

2.4.13 The Single Step commands

s - Single step with display
S - Single step without display
t - Single step with display through traps
T - Single step without display through traps

Syntax:

[#] s [;CMDLIST]
[#] S [;CMDLIST]
[#] t [;CMDLIST]
[#] T [;CMDLIST]

Description:

These commands 'single step' the user's program; that is, execute its instructions one by one. The 's' versions of the command treat the Macintosh system calls as a single function. The 't' versions allow the user to single step through such traps.

The optional '#' parameter specifies the number of instructions to be executed; it defaults to one instruction.

The optional CMDLIST parameter is a list of debugger commands to be executed after each single step.

The commands differ in that *s* and *t* display information after each single step, whereas *S* and *T* only display information after the last single step.

The displayed information consists of the registers and a disassembly of the next instruction to be executed.

When single-stepping, breakpoints aren't enabled.

formatted print commands

p	generate formatted print

quit command

q	quit debugger

register command

r	register display

single step commands

s/S	single step with/without display
t/T	single step with/without display through traps

unassembly commands

u/U	unassemble memory

variable command

v/V	create/modify symbol

macro command

x	define or modify a command macro

swap screen command

‘	display the other screen

display expression

=	display value of an expression

help command

?	list debugger commands

2.4.16 The Macro command

x - Macro command

Syntax:

xc
xc = CMDLIST
x?

Description:

The *x* command defines or executes a sequence of debugger commands, called a 'macro'. It can also list the defined macros.

A macro is associated with a letter of the alphabet, so up to 26 macros can be known to the debugger at one time. Case is not significant.

A macro is defined by typing the letter 'x', followed by the letter with which the macro is to be associated. Then follows an '=' character and the macro's list of debugger commands, with the commands separated by semicolons.

A macro is executed by typing 'x', followed by the letter with which the macro is associated, followed by a carriage return.

The macros which have been defined can be listed using the command *x?*.

2.4.17 The Swap Screen command

‘ - swap screens

Syntax:

‘

Description:

The back quote (‘) command allows either the debugger's output screen or the user's screen to be on display at any given time. It toggles between the screens and allows the user to examine either one.

2.4.18 The 'Display expression' Command

= - Display the value of an expression

Syntax:

= *EXPR*

Description:

This command displays the value of an expression.

The expression is displayed in several formats: hexadecimal, signed decimal, unsigned decimal, octal, binary, and ASCII. If a

3. Command Summary

breakpoint commands

bb/bw/bl	set byte/word/long memory-change breakpoint
bc/bC	clear one/all breakpoints
bd	display the breakpoint table
br	reset the breakpoint counters
bs	set or modify a breakpoint
bt/bT	enable/disable trace mode

clear commands

cs	clear all symbols

display commands

db/dw/dl/d	display memory in bytes/words/longs/last format
dc/dd	display code/data symbols
df	display source file lines
dg	display global values
dha	display application heap structure
dhs	display system heap structure
ds	display stack backtrace

exit command

e	exit to shell without removing debugger

find source string

f	find string in source file

go commands

g/G	execute user's program

load commands

lf	load a source file
lp	load program
ls	load symbols

memory modification commands

mb/mw/ml	modify bytes/words/longs of memory
mc	compare areas of memory
mf	fill memory
mm	move memory
ms	search memory

radix command

n	change the default radix for input and display

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;
    int size = 2, count = 10;

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

NAME

frexp, ldexp, modf - build and unbuild real numbers

SYNOPSIS

```
#include <math.h>

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

double ldexp(value, exp)
double value;

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

DESCRIPTION

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

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

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

NAME

fseek, ftell - reposition a stream

SYNOPSIS

```
#include "stdio.h"

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

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

DESCRIPTION

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

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

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

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

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

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

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

SEE ALSO

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

DIAGNOSTICS

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

EXAMPLE

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

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

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

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

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

NAME

getc, agetc, getchar, getw

SYNOPSIS

```
#include "stdio.h"

int getc(stream)
FILE *stream;

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

int getchar()

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

DESCRIPTION

getc returns the next character from the specified input stream.

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

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

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

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

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

SEE ALSO

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

DIAGNOSTICS

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

integer *errno*.

NAME

gets, fgets - get a string from a stream

SYNOPSIS

```
#include "stdio.h"

char *gets(s)
char *s;

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

DESCRIPTION

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

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

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

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

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

SEE ALSO

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

DIAGNOSTICS

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

NAME

ioctl, isatty - device i/o utilities

SYNOPSIS

```
#include "sgtty.h"

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

isatty(fd)
```

DESCRIPTION

ioctl sets and determines the mode of the console.

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

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

SEE ALSO

Console I/O (O)

NAME

lseek - change current position within file

SYNOPSIS

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

DESCRIPTION

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

fd is the file descriptor associated with the file.

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

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

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

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

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

SEE ALSO

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

DIAGNOSTICS

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

EXAMPLES

1. To seek to the beginning of a file:

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

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

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

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

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

3. To seek backward five bytes:

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

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

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

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

NAME

malloc, calloc, realloc, free - memory allocation

SYNOPSIS

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

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

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

free(ptr)
char *ptr;
```

DESCRIPTION

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

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

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

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

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

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

SEE ALSO

Memory Usage (O), break (S)

DIAGNOSTICS

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

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

NAME

movmem, setmem, swapmem

SYNOPSIS

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

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

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

DESCRIPTION

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

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

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

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

NAME

open

SYNOPSIS

```
#include "fcntl.h"

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

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

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

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

SEE ALSO

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

DIAGNOSTICS

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

EXAMPLES

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

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

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

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

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

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

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

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

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

NAME

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

SYNOPSIS

```
#include "stdio.h"

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

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

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

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

DESCRIPTION

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

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

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

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

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

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

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

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

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

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

The conversion characters are:

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

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

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

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

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

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

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

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

% Output a %. No argument is converted.

SEE ALSO

Standard I/O (O)

EXAMPLES

1. The following program fragment:

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

   will print a message of the form

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

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

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

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

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

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

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

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

NAME

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

SYNOPSIS

```
#include "stdio.h"

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

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

putchar(c)
char c;

putw(w,stream)
FILE *stream;

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

DESCRIPTION

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

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

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

SEE ALSO

Standard I/O

DIAGNOSTICS

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

NAME

puts, fputs - put a character string on a stream

SYNOPSIS

```
#include "stdio.h"

puts(s)
char *s;

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

DESCRIPTION

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

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

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

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

SEE ALSO

Standard I/O (O), putc

DIAGNOSTICS

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

NAME

qsort - sort an array of records in memory

SYNOPSIS

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

DESCRIPTION

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

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

EXAMPLE

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

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

main()
{
    int i,numlines, cmp();
    char buf[LINESIZE];
    for (numlines=0; numlines<MAXLINES; ++numlines){
        if (gets(buf) == NULL)
            break;
        lines[numlines] = malloc(LINESIZE);
        strcpy(lines[numlines], buf);
    }
    qsort(lines, numlines, 2, cmp);
    for (i = 0; i <numlines; ++i)
        printf("%s\n", lines[i]);
}
cmp(a,b)
char **a, **b;
{
    return strcmp(*a, *b);
}
```

NAME

ran - random number generator

SYNOPSIS

double ran()

DESCRIPTION

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

NAME

read - read from device or file without buffering

SYNOPSIS

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

DESCRIPTION

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

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

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

bufsize is the number of characters to be transferred.

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

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

SEE ALSO

Unbuffered I/O (O), open, close

DIAGNOSTICS

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

NAME

rename - rename a disk file

SYNOPSIS

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

DESCRIPTION

rename changes the name of a file.

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

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

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

NAME

scanf, fscanf, sscanf - formatted input conversion

SYNOPSIS

```
#include "stdio.h"

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

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

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

DESCRIPTION

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

The functions get the text from different places:

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

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

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

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

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

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

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

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

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

Matching 'white space' characters

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

Matching ordinary characters

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

Matching conversion specifications

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

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

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

Details of input conversion

A conversion specification consists of:

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

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

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

The following conversion characters are supported:

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

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

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

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

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

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

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

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

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

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

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

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

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

SEE ALSO

Standard I/O (O)

EXAMPLES

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

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

The input line

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

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

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

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

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

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

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

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

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

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

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

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

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

   32 rufus 75.36e-1

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

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

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

   or

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

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

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

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

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

7. Consider the following program fragment:

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

   When given the following input:

   12345 678 90a65

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

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

NAME

setbuf - assign buffer to a stream

SYNOPSIS

```
#include "stdio.h"

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

DESCRIPTION

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

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

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

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

SEE ALSO

Standard I/O (O), malloc

NAME

setjmp, longjmp - non-local goto

SYNOPSIS

```
#include "setjmp.h"

setjmp(env)
jmp_buf env;

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

DESCRIPTION

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

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

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

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

WARNING

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

EXAMPLE

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

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

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

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

```
#include "setjmp.h"

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

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

Here's one of the low level functions:

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

NAME

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

SYNOPSIS

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

double tan(x)
double x;

double cotan(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

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

DESCRIPTION

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

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

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

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

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

SEE ALSO

Errors (O)

DIAGNOSTICS

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

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

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

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

NAME

sinh, cosh, tanh

SYNOPSIS

```
#include <math.h>

double sinh(x)
double x;

double cosh(x)
double x;

double tanh(x)
double x;
```

DESCRIPTION

These functions compute the hyperbolic functions of their arguments.

SEE ALSO

Errors (O)

DIAGNOSTICS

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

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

NAME

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

SYNOPSIS

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

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

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

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

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

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

strlen(s)
char *s;

char *index(s, c)
char *s;

char *rindex(s, c)
char *s;
```

DESCRIPTION

These functions operate on null-terminated strings, as follows:

strcat appends a copy of string *s2* to string *s1*. *strncat* copies at most *n* characters. Both terminate the resulting string with the null character (\0) and return a pointer to the first character of the resulting string.

strcmp compares its two arguments and returns an integer greater than, equal, or less than zero, according as *s1* is lexicographically greater than, equal to, or less than *s2*. *strncmp* makes the same comparison but looks at *n* characters at most.

strcpy copies string *s2* to *s1* stopping after the null character has been moved. *strncpy* copies exactly *n* characters: if *s2* contains less than *n* characters, null characters will be appended to the resulting string until *n* characters have been moved; if *s2* contains *n* or more characters, only the first *n* will be moved, and the resulting string will not be null terminated.

strlen returns the number of characters which occur in *s* up to the first null character.

index returns a pointer to the first occurrance of the character *c* in string *s*, or zero if *c* isn't in the string.

rindex returns a pointer to the last occurrance of the character *c* in string *s*, or zero if *c* isn't in the string.

NAME

toupper, tolower

SYNOPSIS

toupper(c)

tolower(c)

#include "ctype.h"

_toupper(c)

_tolower(c)

DESCRIPTION

toupper converts a lower case character to upper case: if *c* is a lower case character, *toupper* returns its upper case equivalent as its value, otherwise *c* is returned.

tolower converts an upper case character to lowr case: if *c* is an upper case character *tolower* returns its lower case equivalent, otherwise *c* is returned.

toupper and *tolower* do not require the header file *ctype.h.*

_toupper and *_tolower* are macro versions of *toupper* and *tolower*, respectively. They are defined in *ctype.h.* The difference between the two sets of functions is that the macro versions will sometimes translate non-alphabetic characters, whereas the function versions don't.

NAME

ungetc - push a character back into input stream

SYNOPSIS

```
#include "stdio.h"

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

DESCRIPTION

ungetc pushes the character *c* back on an input stream. That character will be returned by the next *getc* call on that stream. *ungetc* returns *c* as its value.

Only one character of pushback is guaranteed. EOF cannot be pushed back.

SEE ALSO

Standard I/O (O)

DIAGNOSTICS

ungetc returns EOF (-1) if the character can't be pushed back.

NAME

unlink

SYNOPSIS

unlink(name)
char *name;

DESCRIPTION

unlink erases a file.

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

unlink returns 0 if successful.

DIAGNOSTICS

unlink returns -1 if it couldn't erase the file and places a code in the global integer *errno* describing the error.

NAME

write

SYNOPSIS

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

DESCRIPTION

write writes characters to a device or disk which has been previously opened by a call to *open* or *creat.* The characters are written to the device or file directly from the caller's buffer.

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

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

bufsize is the number of characters to be written.

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

SEE ALSO

Unbuffered I/O (O) , open, close, read

DIAGNOSTICS

If the operation is unsuccessful, *write* returns -1 and places a code in the global integer *errno.*

STYLE

Chapter Contents

Style

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 *case*s in a *switch* statement, it is advisable to end each *case* with a *break*, including the last *case* in the *switch*.

* **strange loops**

The behavior of loops can be explored by inserting *printf* statements in the body of the loop. Obviously, this will indicate if the loop has even been entered at all in course of a run. A counter will show just how many times the loop was executed; a small slip-up will cause a loop to be run through once too often or seldom. The condition for leaving the loop should be doublechecked for accuracy.

* **use of strings**

All strings must be terminated by a null character in memory. Thus, the string, "hello", will occupy a six-element array; the sixth element is ' '. This convention is essential when passing a string to a standard library function. The compiler will append the null character to string constants automatically.

* **pointer vs object of a pointer**

The greatest difficulty in using pointers is being sure of what is needed and what is being used. Functions which take a pointer argument require an address in memory. The best way to ensure that the correct value is being passed is to keep track of what is being pointed to by which pointer.

* **array subscripting**

The first element in a C array has a subscript of zero. The array name without a subscript is actually a pointer to this element. Obviously, many problems can develop from an incorrect subscript. The most damaging can be subscripting out of bounds, since this will access memory above the array and overwrite any data there. If array elements or data stored with arrays are being lost, this error is a good candidate.

* **function interface**

During the design stage, the components of a program should be associated with functions. It is important that the data which is passed among or shared by these functions be explicitly defined in the preliminary design of the program. This will greatly facilitate the coding of the program since the interface between functions must be precise in several respects.

First of all, if the parameters of a function are established, a call can be made without the reservation that it will be changed later. There is less chance that the arguments will be of the wrong type or specified in the wrong order.

A function is given only a private copy of the variables it is passed. This is a good reason to decide while designing the program how functions should access the data they require. You will be able to detail the arguments to be passed in a function call, the global data which the function will alter, the value which the function will return and what declarations will be appropriate-- all without concern for how the function will be coded.

Argument declarations should be a fairly simple matter once these things are known. Note that this declaration list must stand before the left brace of the function body.

The type of the function is the same as the type of the value it returns. Functions must be declared just like any variable. And just like variables, functions will default to type int, that is, the compiler will assume that a function returns an integer if you do not tell it otherwise with a declaration. Thus if function f calls function g which returns a variable of type double, the following declaration is needed:

```
function f()
{
        double g(), bigfloat;

        g(bigfloat);
}
double g(arg)
double arg;
{
        return(arg);
}
```

* **be sure of what a function returns**

You will probably know very well what is returned by a function you have written yourself. But care should be taken when using functions coded by someone else. This is especially true of the standard library functions. Most of the supplied library functions will return an int or a char pointer where you might expect a char. For instance, getchar() returns an int, not a char. The functions supplied by Manx adhere to the UNIX model in all but a few cases.

Of course, the above applies to a function's arguments as well.

* **shared data**

Variables that are declared globally can be accessed by all functions in the file. This is not a very safe way to pass data to functions since once a global variable is altered, there is no returning it to its former state without an elaborate method of saving data. Moreover, global data must be carefully managed; a function may process the wrong variable and consequently inhibit any other function which depends on that data.

Since C provides for and even encourages private data, this definitely should not be a common bug.

COMPILER ERROR MESSAGES

Chapter Contents

Compiler Error 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.

INDEX

Order of chapters in manual

System Dependent Chapters

System Independent Chapters

Index

S

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)
1-(415)-339-2427 also at 300/1200 bps.(MAC & AMIGA only)

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.

MANX PROBLEM REPORT

Date:_____/_____/____

Name:_____

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

Company :_____

Address :_____

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

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

Send this form to :

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

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

or call tech support at 1-201-542-1795 between 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.)

MANX PROBLEM REPORT

Date:_____/_____/_____

Name:_____

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

Company :_____

Address :_____

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

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

Send this form to :

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

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

or call tech support at 1-201-542-1795 between 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.)