REGISTRATION FORM

To qualify for maintenance and update information you must return your registration card.

The **Mark Williams Company** is pleased to provide our customers with maintenance and update information but you must be a registered customer. Please complete the form below, detach it, and return it to us within 30 days. No postage is required.

When you complete your registration card, please be sure to let us know where you learned about our product. This will allow us to spend less money on advertising and more money developing quality software for you, our valued customer.

Mark Williams Company **REGISTRATION CARD**

Please complete and return this card within 30 days of purchase to qualify for maintenance and updates.

Customer Data

Name/Title ________________

Company Name ________________

Address ________________

City ________ State or Country ________ Zip ________

Daytime Telephone ________________

Product Data

Purchased from ________ Date Purchased ________

Product ________________

Version Number ________________

CAG-1487-2.0.1

Computer ________________

Hard Disk Name ________ Hard Disk Memory Size ________

How did you first hear about this product?

- ☐ Saw your ad in ________________ (please specify)
- ☐ Other ________________ (please specify)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS • PERMIT NO. 10820 • CHICAGO, IL

POSTAGE WILL BE PAID BY ADDRESSEE

Mark Williams Company
1430 Wrightwood Avenue
Chicago, Illinois 60614

Mark Williams C

for the Atari ST

Mark Williams Company

Revision 3 Printing 5 4 3 2 1

Published by Mark Williams Company, 1430 West Wrightwood Avenue, Chicago, Illinois 60614.

Table of Contents

Cconin	Read a character from the standard input
Cconis	Find if a character is waiting at standard input
Cconos	Check if console is ready to receive characters
Cconout	Write a character onto standard output
Cconrs	Read and edit a string from the standard input
Cconws	Write a string onto standard output
cd	Change directory
ceil	Numeric ceiling function
char	
character constant	
clearerr	Present stream status
CLK_TCK	
clock	Get number of clock ticks since system boot
close	Close a file
cmp	Compare bytes of two files
Cnecin	Perform modified raw input from standard input
commands	
compound number	
con	
cos	Calculate cosine
cosh	Calculate hyperbolic cosine
cp	Copy a file
cpp	C preprocessor
Cprnos	Check if printer is ready to receive characters
Cprnout	Send a character to the printer port
Crawcin	Read a raw character from standard input
Crawio	Perform raw I/O with the standard input
creat	Create/truncate a file
crts0.o	
crtsd.o	
crtsg.o	
cshconv	Run a Mark Williams C program under the Beckemeyer C she
ctime	Convert system time to an ASCII string
ctype	
ctype.h	Header file for data tests
cursconf	Set the cursor's configuration
Cursconf	Get or set the cursor's configuration
daemon	
data formats	
data types	
date	Print/set the date and time
dayspermonth	Return number of days in a given month
db	Assembler-level symbolic debugger
Dcreate	Create a directory
Ddelete	Delete a directory

fabs	Compute absolute value
Fattrib	Get and set file attributes
fclose	Close stream
Fclose	Close a file
Fcreate	Create a file
fcvt	Convert floating point numbers to ASCII strings
Fdatime	Get or set a file's date/time stamp
Fdelete	Delete a file
Fdup	Generate a substitute file handle
feof	Discover stream status
ferror	Discover stream status
fflush	Flush stream output buffer
Fforce	Force a file handle
fgetc	Read character from stream
Fgetdta	Get a disk transfer address
fgets	Read line from stream
fgetw	Read integer from stream
field	
file	Name a file's type
file	
FILE	Descriptor for a file stream
file descriptor	
fileno	Get file descriptor
flexible arrays	
float	
floor	Set a numeric floor
Flopfmt	Format tracks on a floppy disk
Floprd	Read sectors on a floppy disk
Flopver	Verify a floppy disk
Flopwr	Write sectors on a floppy disk
fopen	Open a stream for standard I/O
Fopen	Open a file
form_alert	Display an alert box
form_center	Center an object on the screen
form_dial	Reserve/free screen space for dialogue
form_do	Handle user input in form dialogue
form_error	Display a DOS error alert
fprintf	Format output
fputc	Write character to stream
fputs	Write string to stream
fputw	Write an integer to a stream
fread	Read data from stream
Fread	Read a file
free	Return dynamic memory to free memory pool
Frename	Rename a file

graf_watchbox	Draw a watched box
handle	
header file	
help	Print concise description of command
hidemouse	Hide the mouse pointer
HOME	
horizontal tab	
htom	Redraw screen from high to medium resolution
hypot	Compute hypotenuse of right triangle
Ikbdws	Write a string to the intelligent keyboard device
INCDIR	
#include	
include file	
index	Find a character in a string
Initmous	Initialize the mouse
int	
interrupt	
Iorec	Set the I/O record
isalnum	Check if a character is a number or letter
isalpha	Check if a character is a letter
isascii	Check if a character is an ASCII character
iscntrl	Check if a character is a control character
isdigit	Check if a character is a numeral
isleapyear	Indicate if a year was a leap year
islower	Check if a character is a lower-case letter
isprint	Check if a character is printable
ispunct	Check if a character is a punctuation mark
isspace	Check if a character prints white space
isupper	Check if a character is an an upper-case letter
j0	Compute Bessel function
j1	Compute Bessel function
jday_to_time	Convert Julian date to system time
jday_to_tm	Convert Julian date to system calendar format
Jdisint	Disable interrupt on muli-function peripheral device
Jenabint	Enable a multi-function peripheral port interrupt
jn	Compute Bessel function
Kbdvbase	Return a pointer to the keyboard vectors
kbrate	Reset the keyboard's repeat rate
Kbrate	Get or set the keyboard's repeat rate
keyboard	
Keytbl	Set the keyboard's translation table
keyword	
Kgettime	Read time from intelligent keyboard's clock
kick	Force OS to reread the disk cache
Ksettime	Set time in intelligent keyboard's clock

metafile	
mf	Measure space left in RAM
Mfpint	Initialize the MFP interrupt
Mfree	Free allocated memory
Midiws	Write a string to the MIDI port
mkdir	Create a directory
mktemp	Generate a temporary file name
modf	Separate integral part and fraction
modulus	
msh	
Mshrink	Shrink amount of allocated memory
msleep	Stop executing for a specified time
mtoh	Redraw the screen from medium to high resolution
mtol	Redraw the screen from medium to low resolution
mtype.h	
mv	
nested comments	
newline	
nm	Print a program's symbol table
notmem	Check if memory is allocated
n.out	
NUL	
NULL	
nybble	
obdefs.h	
objc_add	Redefine a child object within an object tree
objc_change	Change an object's state within a clipping rectangle
objc_delete	Delete an object from an object tree
objc_draw	Draw an object
objc_edit	Edit a text object
objc_find	Find if mouse pointer is over particular object
objc_order	Reorder a child object within the object tree
objc_set	Calculate an object's absolute screen position
object	
object format	
od	
Offgibit	Clear a bit in the sound chip's A port
Ongibit	Turn on a bit in the sound chip's A port
open	Open a file
operator	
osbind.h	
path	
PATH	
patterns	
peekb	Extract a byte from memory

record	
Rect	
register	
register variable	
rewind	Reset file pointer
rindex	Find a character in a string
rm	Remove files
rmdir	Remove a directory
rsconf	Reconfigure the serial port
Rsconf	Configure the serial port
rsrc_free	Free memory allocated to a set of resources
rsrc_gaddr	Get the address of a resource object
rsrc_load	Load a resource file into memory
rsrc_obfix	Change the form of an object's coordinates
rsrc_saddr	Store address of a free string or a bit image
runtime startup	
rvalue	
Rwabs	Read or write data on a disk drive
scanf	Format input
Scrdmp	Print a dump of the screen
screen control	
scrp_read	Read the scrap directory
scrp_write	Write to the scrap directory
set	
setbuf	Set alternative stream buffers
setcol	Reset a color
Setcolor	Set one color
setenv	Set an environmental variable
Setexc	Get or set an exception vector
setjmp	Perform non-local goto
setjmp.h	Header file for setjmp and longjmp functions
setpal	Reset the color palette
Setpallete	Set the screen's color palette
setphys	Reset physical screen's display space
setprt	Reset the printer port
Setprt	Get or set the printer's configuration
setrez	Reset the screen resolution
Setscreen	Set the video parameters
Settime	Set the current time
shel_envrn	Search for an environmental variable
shel_find	Search PATH for file name
shel_read	Let an application identify the program that called it
shel_write	Run another application
shellsort	Sort arrays in memory
short	

system	
system variables	
tail	Print the end of a file
tan	Calculate tangent
tanh	Calculate hyperbolic cosine
tempnam	Generate a unique name for a temporary file
tetd_to_tm	Convert IKBD time to system calendar format
Tgetdate	Get the current date
Tgettime	Get the current time
Tickcal	Return system timer's calibration.
time	Get current time
time	
time_to_jday	Convert system time to Julian date
time.h	Header file with time-description structure
timezone	
TIMEZONE	Time zone environmental parameter
tm_to_jday	Convert calendar format to Julian time
tm_to_tetd	Convert system calendar format to IKBD time
TMPDIR	
tmpnam	Generate a unique name for a temporary file
toascii	Convert characters to ASCII
tolower	Convert characters to lower case
_tolower	Convert letter to lower case
tos	Execute GEM-DOS program
TOS	
touch	Update modification time of a file
toupper	Convert characters to upper case
_toupper	Convert letter to upper case
Tsetdate	Set a new date
Tsettime	Set a new time
type promotion	
type checking	
typedef	
ungetc	Return character to input stream
union	
uniq	Remove/count repeated lines in a sorted file
UNIX routines	
unlink	Remove a file
unset	Discard a shell variable
unsetenv	Discard an environmental variable
unsigned	
v_arc	Draw a circular arc
v_bar	Draw a rectangle
v_bit_image	Print a bit image file
v_cellarray	Draw a table of colored cells

vertical tab	
vex_butv	Set new button interrupt routine
vex_curv	Set new cursor interrupt routine
vex_motv	Set new mouse movement interrupt routine
vex_timv	Set new timer interrupt routine
vm_filename	Rename a metafile
void	
vq_cellarray	Return information about cell arrays
vq_chcells	Find how many characters virtual device can print
vq_color	Check/set color intensity
vq_curaddress	Get the text cursor's current position
vq_extnd	Perform extend inquire of VDI virtual device
vq_key_s	Check control key status
vq_mouse	Check mouse position and button state
vq_tabstatus	Find if graphics tablet is available
vqf_attributes	Read the area fill's current attributes
vqin_mode	Determine mode of a logical input device
vql_attributes	Read the polyline's current attributes
vqm_attributes	Read the marker's current attributes
vqp_error	Inquire if an error occurred with the Polaroid Palette
vqp_films	Get films supported by driver for Polaroid Palette
vqp_state	Read current settings of the Polaroid Palette driver
vqt_attributes	Read the graphic text's current attributes
vqt_extent	Calculate a string's length
vqt_fontinfo	Get information about special effects for graphics text
vqt_name	Get name and description of graphics text font
vqt_width	Get character cell width
vr_recfl	Draw a rectangular fill area
vr_trnfm	Transform a raster image
vro_cpyfm	Copy raster form, opaque
vrq_choice	Return status of function keys when any key is pressed
vrq_locator	Find location of mouse cursor when a key is pressed
vrq_string	Read a string from the keyboard
vrq_valuator	Return status of shift and cursor keys
vrt_cpyfm	Copy raster form, transparent
vs_clip	Set the virtual device's clipping rectangle
vs_color	Set color intensity
vs_curaddress	Move alphabetic cursor to specified row and column
vs_palette	Select color palette on medium-resolution screen
vsc_form	Draw a new shape for the mouse pointer
vsf_color	Set a polygon's fill color
vsf_interior	Set a polygon's fill type
vsf_perimeter	Set whether to draw a perimeter around a polygon
vsf_style	Set a polygon's fill style
vsf_udpat	Define a fill pattern

1. A Tutorial Introduction

Congratulations on choosing the Mark Williams C compiler. Mark Williams C has the state-of-the-art power and flexibility that the professional programmer needs, but is simple enough for the beginner to learn quickly.

Mark Williams C uses the latest advancements in compiler design. It parses programs by recursive descent and uses table-driven code generation to produce fast, dense code. Then it performs extensive optimization to make the code even better. Ease of use, full documentation, and compact generated code make Mark Williams C the right tool for the rapid development of your programs.

Mark Williams C for the Atari ST is a member of the Mark Williams Company family of C compilers. Mark Williams Company compilers support many different environments and processors. The environments include the following:

COHERENT
CP/M-68K
ISIS-II
MS-DOS
RMX
TOS
VAX/VMS

In addition to the 68000 family, the processors supported include:

PDP-11
Z8001
Z8002
8086
80186
80286

What is in Mark Williams C?

Mark Williams C is a uniquely powerful C programming system designed for the Atari ST. It consists of the following:

1. The Mark Williams C compiler, plus an assembler, a linker, a preprocessor, and other tools.
2. A set of commands selected from the COHERENT operating system, including the MicroEMACS screen editor and the **make** programming discipline.

3. A full set of libraries, including the standard C library, a mathematics library, plus libraries that implement the Atari AES, VDI, and Line A routines.

4. A set of sample programs, including full source code for the MicroEMACS editor, and text files to be used with the tutorials included your documentation.

5. The Mark Williams micro-shell **msh**, a command processor designed to control the operation of the compiler and its commands.

Mark Williams C is designed to work through **msh**, a command processor that combines aspects of the Bourne and Berkeley C shells into a small but powerful program. With **msh**, you can perform numerous tasks to speed program development. It gives Mark Williams C unique power in developing programs for the Atari ST.

Hardware requirements

Mark Williams C is designed to be used on the Atari ST, either the 520 or 1040 models. It can be used with the following hardware configurations:

* An Atari ST with two disk drives, single- or double-sided.
* An Atari ST with one floppy disk drive and a hard disk.
* An Atari ST with one double-sided floppy disk drive, one megabyte of RAM, and a 500-kilobyte RAM disk.

How to use this manual

This manual is in four sections. Section 1, which you are now reading, is a tutorial introduction to Mark Williams C. It will show you how to use the micro-shell **msh**, how to use its commands, how to compile programs, and how to use the various libraries available with Mark Williams C.

Section 2 is a table of all error messages produced by the compiler, the assembler, and the linker.

Section 3 is the Lexicon. This is by far the largest part of the manual. The Lexicon contains several hundred entries; each describes a command, a function, defines a C technical term, or otherwise gives you useful information.

All of the Lexicon's entries are in alphabetical order, and are designed to be easily used. For example, if you want information on how to use the STDIO routines, simply turn to the entry in the Lexicon on **STDIO**; there, you will find a list of all the STDIO routines, a description of each, and instructions on how to use them. Or, if you want information on how Mark Williams C encodes floating point numbers, simply turn to the entry on **float**. There, you will find a full description of floating point numbers. Many Lexicon entries have full C programs as examples; all have full cross-references to related entries.

This tutorial will refer constantly to the Lexicon. If you are unfamiliar with a technical term used in this manual, look it up in the Lexicon. Chances are, you will find a full explanation. If you are not sure how to use the Lexicon, look up the entry for **Lexicon** within the Lexicon. This will help you get started.

Finally, the back of this manual has a tutorial for the **make** program building discipline and the MicroEMACS screen editor. If you are unfamiliar with either of these tools, you will find that these tutorials will give you a good beginning in using them.

User registration and reaction report

Before you go any further, fill out the User Registration Card that came with your copy of Mark Williams C. Returning this card will make you eligible for direct telephone support from the Mark Williams Company technical staff, and ensures that you will automatically receive information about all new releases and updates. Many interesting developments and additions are planned for Mark Williams C.

If you have comments or reactions to the Mark Williams C software or documentation, please fill out and mail the User Reaction Report included at the end of the manual. We especially wish to know if you found errors in this manual. Mark Williams Company needs your comments to continue to improve Mark Williams C.

Getting started

The rest of this tutorial assumes that you have installed Mark Williams C on your Atari ST. If you have not yet done so, turn to the Release Notes that are included with this manual. There you will find directions on how to install Mark Williams C on your system; how to set up your system to run Mark Williams C properly; how to install the Mark Williams rebootable RAM disk; and how to run this product with unusual configurations of hardware.

If you wish to continue with this tutorial, return here after you have installed Mark Williams C on your system.

Introducing the Mark Williams C micro-shell

Mark Williams C is designed to run under a micro-shell, called **msh**. **msh** allows the passing of commands that are longer and more complex than can be handled easily through the GEM desktop; it also gives you an easy way to *redirect* the output of commands, *pipe* output to other commands, build and access tree-structured directories, and perform many other tasks to speed program development. **msh** comes with a full complement of utilities and tools, to increase its usefulness.

What is msh?

msh is a *command processor*. It reads and interprets commands, which can either be typed directly into **msh**, or stored in files, called *scripts*, that **msh** opens and reads. **msh** differs from icon-driven or menu-driven systems in that you type words into it rather than clicking items on the screen. If you have used COHERENT or UNIX, you will find that **msh** combines aspects of the Bourne shell and the Berkeley C shell to create a command processor that is simple yet powerful.

How to enter msh

Entering **msh** is easy. If you have a two-floppy disk system, just place your installed disk that is labelled "compiler" into drive A:, then use the mouse to open drive A: and display the contents of **bin**. If you have a hard disk, use the mouse to display the contents of **bin** on the logical drive on which you have stored the compiler. Point to the icon labelled **MSH.PRG** and click the left button twice.

The screen clears and a dollar sign '$' appears in the upper left-hand corner. The dollar sign is a *prompt*: it means that **msh** is ready to accept a command.

To test **msh**, type the following command:

```
echo foo
```

Press the carriage return key. **echo** is a command that repeats all of the words, or *arguments*, that follow it. You will find that this command is quite useful in certain programming situations. As you can see, the argument **foo** appeared on the next line of the screen; then another dollar sign appeared, which signals that **msh** is ready to accept another command.

Introducing MicroEMACS, the screen editor

Mark Williams C includes a full screen editor, called MicroEMACS. MicroEMACS allows you to divide the screen into windows and edit different files simultaneously. It has a full search-and-replace function, allows you to define keyboard macros; and has a large set of commands for killing and moving text.

For a list of the MicroEMACS commands, see the Lexicon entry for **me**, the MicroEMACS command. At the back of this manual is a full tutorial that shows you how to use most of its commands, and contains a number of exercises to help you sharpen your skill.

You can begin to use the editor immediately, however, by remembering a half-dozen or so commands. To see how MicroEMACS works, do the following exercise: First, make sure that MicroEMACS is available to your system; if you have a system with only floppy disks or with only a floppy disk and a RAM disk, make sure that the "compiler" disk is in a disk drive. Now, type the following command:

```
me hello.c
```

As you can see, the screen clears, and an information line appears at the bottom. Type the following text, as it is shown here. If you make a mistake, simply backspace over it and type it correctly; the backspace key will wrap around lines:

```
main() {
        printf("hello, world\n");
}
```

When you have finished, save the file by typing **<ctrl-X><ctrl-S>** (that is, hold down the control key and type 'X', then hold down the control key and type 'S'). MicroEMACS will tell you how many lines of text it just saved. Now, exit from the editor by typing **<ctrl-X><ctrl-C>**.

Now, type **ls -l**. In a second, the *list* command **ls** will print some information about the file, including its size, the date and time it was created, and its name. This proves that the file has been created and stored on disk.

If you wish to change a file, just type the **me** command, as before:

```
me hello.c
```

The text of the file you just typed is now displayed on the screen. Now, try changing the word **hello** to **Hello**, as follows: First, type **<ctrl-N>**. That moves you to the *next* line. (The command **<ctrl-P>** would move you to the previous line, if there were one.) Now, type the command **<ctrl-F>**. As you can see, the cursor moved forward one space. Continue to type **<ctrl-F>** until the cursor is located over the letter 'h' in **hello**. If you overshoot the character, move the cursor backwards by typing **<ctrl-B>**. When the cursor is correctly positioned, delete the 'h' by typing the *delete* command **<ctrl-D>**; then type a capital 'H' to take its place. Now, again save the file by typing **<ctrl-X><ctrl-S>**, and exit from MicroEMACS by typing **<ctrl-X><ctrl-C>**.

With these few commands, you can load files into memory, edit them, create new files, save them to disk, and exit. This just gives you a sample of what MicroEMACS can do, but it is enough to get you started.

Setting the shell's internal variables

msh is designed to allow you to alter the way it operates. In effect, you can customize **msh** to suit your own needs. One way to do so is by using the **set** command.

For example, you may wish to change the prompt from the dollar sign to something else. You can do this with the **set** command. To change the prompt to **st>**, type the following command:

```
set prompt="st> "
```

Try it.

As you can see, the prompt changed as soon as you pressed the carriage return key. If you type **set** by itself, a list of variables will appear. **set** allows you to define new variables, which are read by **msh** and interpreted.

Try using **set** to create a "quick and dirty" command to clear the screen. As shown in the Lexicon article on **screen control**, the escape sequence that clears the screen on the Atari ST is **<esc>E**—that is, the escape character followed by a capital 'E'. Note that ^[is the way the Atari ST echoes the escape character on the screen. To create your new command, just type the following into **msh**:

```
set cls="echo -n ^[E"
```

Now, try typing:

```
$cls
```

The dollar sign tells **msh** that the following string is a variable rather than a command. As you can see, the screen cleared and the cursor is now in the upper left-hand corner of the screen. **msh** replaces **cls** with its defined value, and executes **echo** as if it has been typed in from the keyboard.

To erase a variable, use the command **unset**. For example, to erase the variable **cls**, type:

```
unset cls
```

Try typing **$cls** again. The shell sends you the message

```
variable 'cls' is not set
```

which shows that **cls** has been erased.

Setting the environment

msh manages a set of *environmental variables*. These can be used by programs that run under **msh**. For example, when the compiler driver **cc** begins its work, it looks for an environmental variable called **LIBPATH**, which tells **cc** which directories hold libraries. This system was designed to spare you the trouble of constantly giving programs the same information. For example, you need to set the **LIBPATH** variable only once; instead of telling **cc** where to look for the libraries every time you compile a program, you can save space on the command line for more important items, such as the names of the files you wish to compile.

The command **setenv** sets environmental variables. Try typing **setenv**. **msh** replies by printing a list of the environmental variables that have already been set. Most are set in the file **profile**, which **msh** reads as it begins; this will be described in detail below.

To see how a program can use an environmental variable, try resetting the environmental variable **HOME**. This variable is used by the *change directory* command **cd** when that command is entered without an argument. To set **HOME** to **B:**, which is the *root directory* on drive B:, type:

```
setenv HOME=b:\
```

Now, type the following commands:

```
cd
pwd
```

The first command changes directories for you; because you did not tell it which directory to go to, it moved you by default to the directory named by the **HOME** environmental variable. **pwd** prints the working directory; as you can see, the current directory is **b:**, which is the directory that **cd** moved you to.

The command **unsetenv** erases environmental variables. For example, you can erase the variable **TIMEZONE** with the following command:

```
unsetenv TIMEZONE
```

Now, type **setenv** again. As you can see, the **TIMEZONE** environmental variable is no longer present.

Directories

You have probably noticed by now that **msh** uses tree-structured directories. This means that its directories branch out from one another; each directory can contain files and sub-directories that themselves can contain files and directories. One directory is called the *root directory*; this is the name of the device. For example, the root directory for drive A is called **a:**. The root directory can have one or more sub-directories; these are also called *child* directories because they all stem from the same *parent* directory. Thus, while a directory can have many child directories, it can have only one parent directory.

Note that two dots ".." stands for the parent directory. The following examples will show how to use this abbreviation.

msh comes with a full set of commands to create and remove directories, and copy, rename, move, and remove files. As you will see, these are quite easy to use, and quite powerful.

To begin, you can *make a directory* with the command **mkdir**. To create a directory called **stuff**, type:

```
mkdir stuff
```

Try it. If you wish, you can specify a full *path name* to create a subdirectory in a directory other than the one you are currently in. For example, to make the subdirectory **temp** in the directory **stuff**, just type:

```
mkdir stuff\temp
```

Try it. Now, tell the *list* command, **ls**, to show you the contents of **stuff**, as follows:

```
ls stuff
```

As you can see, **ls** printed the name of the subdirectory you just created.

The *remove directory* command **rmdir** allows you to erase directories. To remove the directory **temp**, use the following command:

```
rmdir stuff\temp
```

If **temp** had had files and subdirectories in it, **rmdir** would have given you an error message. This is to help prevent you from accidentally erasing valuable files.

Renaming, moving, copying, and removing files

As mentioned above, **msh** has a number of commands to help you handle files.

The *move* command **mv** lets you rename a file. The following example creates a file called **smith**, and then renames it **jones**:

```
echo stuff >smith
mv smith jones
```

Note that if the file **jones** had already existed, it would have been removed and the file **smith** given its name.

You can also use **mv** to *move* a file from one directory to another. For example, the command

```
mv jones stuff
```

will move the file **jones** from the current directory to the directory **stuff**.

As mentioned above, two periods ".." is shorthand for a directory's parent directory. Thus, to move the file **jones** back from the directory **stuff** to the current directory, type the following command:

```
mv stuff\jones ..
```

If you type **ls** without any arguments, it will show the contents of the current directory; it should show that the file **jones** has been returned to the current directory.

The *copy* command **cp** will copy one or more files for you. To copy the file **jones** back into the file **smith**, type:

```
cp jones smith
```

As with the **mv** command, if the file **smith** had already existed, it would have been removed and the new copy of **jones** given its name.

cp can also copy several files at once into another directory. To copy the files **smith** and **jones** into directory **stuff**, type:

```
cp smith jones stuff
```

cp is intelligent enough to know that **stuff** is a directory; it will copy **smith** and **jones** into **stuff** and give the copies the same names as the originals.

The command **rm** *removes* a file. To remove the files **smith** and **jones** from directory **stuff**, type:

```
rm stuff\smith stuff\jones
```

If you type **rm** without an argument, it will print an error message on the screen.

Redirecting input and output

msh allows you to change, or *redirect*, the place from which a program receives input and the place to which it writes output. The technical term for this is *I/O redirection*.

The C language normally defines three channels through which data can be passed: the *standard input*, the *standard output*, and the *standard error*. The standard input and the standard output, respectively, are connected to the keyboard and the screen by default. The *standard error* is the device on which error messages appear; by default, it is the screen. Note that the terminal screen continues to be the standard error, even if the standard output is redirected elsewhere.

A *redirection operator* is a character that tells **msh** to redirect the standard input, standard output, or standard error somewhere other than its default. The following lists the more commonly used of **msh**'s redirection operators:

> *file* Redirect the standard output of a command into *file*. If *file* already exists, replace its contents with the output of the command. For example, typing

```
echo hello >tempfile
```

opens the file **tempfile** and then echoes the argument **hello** into it. If the file **tempfile** already exists, its contents will be replaced with the string **hello**.

>& *file* Redirect both the standard output and the standard error of a command into *file*.

>> *file* Append the standard output of a command onto *file*. If *file* does not exist, create it and fill it with the output of the command. For example, the command

```
echo goodbye >>tempfile
```

appends the word **goodbye** to the end of the file **tempfile**, which you created in the earlier example.

>>& *file* Append both the standard output of a command and the standard error onto *file*. If *file* does not exist, create it and fill it with the output and diagnostic messages generated by the command.

< *file* Use the contents of *file* as the standard input for a command.

For a full list of redirection operators, see the entry for **msh** in the Lexicon.

Redirecting to peripheral devices

As you can see from the examples in the previous section, redirection is most often performed into or out of files on disk. However, as will be described below, C treats peripheral devices as if they were files; therefore, you can use a redirection symbol to send material to, say, the printer or the serial port.

For example, if you have a printer plugged into your Atari ST, turn it on and type the following command:

```
echo hello >prn:
```

This types the word **hello** on your printer.

Logical devices

TOS, the Atari's operating system, has built into it three *logical devices*. **msh** can use these logical devices in exactly the same way that it handles files: it can open them, read data from them, write data to them, and close them again. The logical devices are as follows: **con:**, which is the console's screen; **prn:**, which is the printer port; and **aux:**, which is the auxiliary, or serial, port. These are described in more detail in their respective Lexicon entries.

Redirecting data to the printer port can be quite useful; for example, you can print listings of your programs. Try this exercise. Turn on your printer, and type the following command:

```
pr -n hello.c >prn:
```

As you can see, a listing of your program appears on your printer, with each line numbered for your convenience. The command **pr** formats material for printing, and its **-n** option tells it to insert line numbers. **pr** is, of course, described more fully in the Lexicon.

File-name substitutions

Often, typing in the names of a group of files is tedious. For that reason, **msh** allows you to deal with files in groups, by using *file-name substitutions*.

msh can use the punctuation marks [ ] ? * { and } to substitute for all or part of a file's name. The following describes what each does:

[*list*], [*a–z*]

In the first form, this looks for, or *matches*, any of the characters *l*, *i*, *s*, or *t*; in the second form, it matches all of the characters between *a* and *z*.

Try the following exercise. First, use the **echo** command to create three sample files, as follows:

```
echo stuff1 >filea
echo stuff2 >fileb
echo stuff3 >filec
```

The following command tells the *list* command **ls** to find these files in the current directory:

```
ls file[abc]
```

As you can see, the shell expanded **file[abc]** into **filea fileb filec**, which it then handed to **ls** to find.

The next exercise uses the concatenation command **cat** to display the contents of these three files. Type the following:

```
cat file[a-c]
```

msh expands **file[a-c]** into **file a** through **c**, inclusive, or **filea fileb filec**. As you can see, **cat** opened all three files and displayed their contents for you on the screen.

? Match any character. For example, typing

```
ls file?
```

will list every program in the current directory that is named **file***anyletter*. Note that the '?' is a *wildcard* character; see the entry for **wildcard** in the Lexicon for more information.

* Match any character, any string of characters, or no character. Try typing

```
ls *[a-c]
```

As you can see, **ls** lists all files whose names end with the character 'a' through 'c'. The asterisk is also a wildcard; see the entry on **wildcards** in the Lexicon for more information.

{*l,i,s,t*}
Use the enclosed letters *l,i,s,t* to form a series of words. For example, the command

```
ls file{a,b,c}
```

is equivalent to typing

```
ls filea fileb filec
```

To see how this differs from the '[' ']' characters described above, type the following commands:

```
echo foo[abc]
echo foo{a,b,c}
```

The first command prints

```
foo[abc]
```

whereas the second returns

```
fooa foob fooc
```

Quoted strings

At times, you want to pass a string to a command literally, without its being interpreted or matched by the shell. Passing a string in this manner is called *quoting* it, because you indicate the special character of the string by enclosing it within quotation marks or apostrophes. (An "apostrophe" is also known as a "single quote"; the apostrophe is found on the same key as the quotation mark, directly to the left of the carriage return key.)

Note that if you quote a string with quotation marks instead of apostrophes, **msh** will treat white space as part of the string, but further expand variables within the string. To see how this works, type the following exercise:

```
set A="XYZ"
set B="QRS"
echo $A         $B
echo "$A        $B"
echo '$A        $B'
```

As you can see, in the first case **echo** expanded $A and $B, but threw away the extra spaces between them. In the second case, it expanded $A and $B, and preserved the extra space between them; in the third case, **echo** preserved the extra space between $A and $B, but did not expand them.

Joining and separating commands

msh uses a number of different punctuation marks, or *operators*, to join and separate commands. Each operator performs a specialized task, as follows:

; Commands separated by a semicolon ';' are run one after the other. This allows you to type more than one command on the same line, for convenience.

| Form a *pipe*; that is, pass the standard output of the command on the left into standard input of the command on the right. Try the following example. First, turn on your printer, and then type:

```
ls -l | pr > prn:
```

As you can see, the names of the files in the current directory are being printed on your printer. The command **ls** first read the names of the files in the current directory. (The switch **-l** tells **ls** to write the names in the *long* format, which gives you extra information, such as the size of each file.) Normally, **ls** writes its output onto the screen; the pipe symbol '|', however, told **ls** to pass its output to the pagination command **pr**, which used it as input. Finally, **pr** redirected *its* output to the logical device **prn:**, so that it appeared on your printer.

As you can see, pipes and redirection symbols allow you to construct chains of commands that are quite powerful, yet quite easy to use.

|& Form a pipe that passes to the command on the right both the output and any error messages from the command on the left.

The profile *file*

Whenever you invoke **msh**, it automatically reads a file called **profile** and executes all of the commands it finds there. By altering your **profile**, you can customize **msh** to suit your preferences and the tasks at hand.

The following is a sample **profile**:

```
set drive=a:
setenv PATH=.bin,,$drive\bin,$drive\command
setenv SUFF=,.prg,.tos,.ttp
setenv LIBPATH=$drive\lib,$drive\bin,
setenv TMPDIR=$drive\tmp
setenv INCDIR=$drive\include
setenv TIMEZONE=CST:0:CDT
set prompt='$ '
set history=8
```

The first line,

```
set drive=a:
```

sets the variable **drive** to **a:**; this means that the variable **$drive** will be interpreted as **a:** by **msh**.

The next line,

```
setenv PATH=.bin,,$drive\bin,$drive\command
```

set the **PATH** environmental variable, which tells **msh** where to find executable files. The first directory, **.bin**, stands for **msh** itself; this tells **msh** to check and see if the command you have typed in is built into **msh** itself. The rest of the command

```
,,$drive\bin,$drive\command
```

tells **msh** to look for executable files first in the present directory (as indicated by the two commas with nothing between them), then in the directory **a:\bin**, and finally in the directory **a:\command** (remember that **$drive** is interpreted to mean **a:**). Note that unless this line is set correctly, **msh** will not be able to execute the rest of the commands in **profile**.

The next lines,

```
setenv SUFF=,.prg,.tos,.ttp
setenv LIBPATH=$drive\lib,$drive\bin,
setenv TMPDIR=$drive\tmp
setenv INCDIR=$drive\include\
setenv TIMEZONE=CST:0:CDT
```

set the environmental variables that **msh** exports to various other commands. Each variable is described at length in the Lexicon.

Finally, the lines

```
set prompt='$ '
set history=8
```

set the prompt to a dollar sign '$', and set the **history** buffer to hold the last eight commands entered. This is used with the *history* command; for more information on this command, see the Lexicon entry for **msh**.

You can use the **profile** file to fine-tune **msh** so that it carefully suits your needs and preferences.

After a while, you will grow familiar with **msh** and will want to use its power more fully. See the entry for **msh** in the Lexicon for a complete description of what it can do.

For more information about the commands that come with your copy of Mark Williams C, see the Lexicon entry for **commands**; this entry lists all of the available commands and briefly describes what each one does. Each command has its own entry in the Lexicon, which will give you all the information you need to use it properly.

Compiling with Mark Williams C

This section describes how to compile C programs with Mark Williams C.

In brief, a C compiler transforms files of C source code into machine code. Compilation is complex and involves several steps; however, Mark Williams C simplifies it with the **cc** command, which controls all the actions of the compiler.

Compiling from the GEM desktop

Before you begin, note that Mark Williams C was designed to be run through the micro-shell **msh**; however, you can run **cc** and the compiler from the GEM desktop. To do so, perform the following steps:

1. Rename the file **cc.prg** to **cc.ttp**.
2. Move the following files plus your source code into the same folder:

   cc.ttp
   cc0.prg
   cc1.prg
   cc2.prg
   cc3.prg
   cpp.prg
   crts0.o
   ld.prg
   libc.a
   libm.a

   Also move all of the header files, which have the suffix **.h**.
3. Use the mouse to double-click the icon labelled **CC.TTP**. When the **Open application** box appears, enter the names of the files you wish to compile.

Note that the micro-shell **msh** preserves the case of arguments passed to Mark Williams C. The GEM-DOS desktop, however, translates all arguments to upper case, in some instances changing their meaning.

Compiling through msh

The rest of this section assumes that you are running Mark Williams C under the micro-shell **msh**.

For an example of how **cc** works under the micro-shell **msh**, try compiling the program called **hello.c**, which you typed in earlier to test the MicroEMACS screen editor. Type the following command:

```
cc -V hello.c
```

The switch **-V** tells **cc** to print a brief description of each action it takes. As you can see, **cc** guides the program through all phases of compilation, invokes the linker **ld**, and links in all necessary routines from the libraries to produce a file called **hello.prg** that is ready to execute.

To try out your newly compiled program, type **hello**. The program will print

```
Hello, world
```

on your screen.

The phases of compilation

You probably noticed that when **hello.c** was being compiled, **cc** invoked a number of different programs. This is because Mark Williams C is not just one program, but a number of different programs that work together to produce an executable file for you. Each program performs a *phase* of compilation. The following summarizes each phase:

cpp The C preprocessor. This processes any of the '#' directives, such as **#include** or **#ifdef**, and expands macros.

cc0 The parser. This phase parses programs; it uses recursive descent to translate the program into a parse-tree format, which is independent of both the language of the source code and the microprocessor for which code will be generated.

cc1 The code generator. This phase reads the parse tree generated by **cc0** and translates into machine code. The code generation is table driven, with entries for each operator and addressing mode. Although this phase is followed by an optimizer, it is designed to generate excellent code that needs minimal optimization.

cc2 The optimizer/object generator. This phase optimizes the generated code and writes the object code. It eliminates common code, optimizes span-dependent jumps, and removes jumps to jumps. It also scans the generated code repeatedly to eliminate unnecessary instructions, then writes the object file.

cc3 The compiler also includes a fifth phase, called **cc3**, which is a disassembler. This phase writes an output file in assembly language. This phase is optional, and allows you to examine the code generated by the compiler. If you want Mark Williams C to generate assembly language, use the **-S** option on the **cc** command line.

Unless you specify **-S** on the **cc** command line, the compiler outputs an *object module* that is named after the source file being compiled. This module has the suffix **.o**. An object module is *not* executable; it contains only the code generated by compiling a C source file, plus information needed to relocate parts of the code before routines from the C libraries are linked into the program. See the entry on **object module** in the Lexicon for more information.

As the final step in its execution, **cc** calls the *linker* **ld** to produce an executable program. In the previous example, the linker combines the object module **hello.o** with the function **printf** from the standard C library to produce the executable program called **hello.prg**. The suffix **.prg** indicates that the file is executable.

Edit errors automatically

Often, when you're writing a new program, you face the situation where you try to compile, only to have the compiler produce error messages and abort the compilation. You must then invoke your editor, change the program, close the editor, and try the compilation over again. This cycle of compilation—editing—recompilation can be quite bothersome.

To remove some of the drudgery from compiling, the **cc** command has the *automatic* or MicroEMACS option, **-A**. When you compile with this option, the MicroEMACS screen editor will be invoked automatically if any errors occur. The error or errors generated during compilation will be displayed in one window, and your text in the other, with the cursor set at the number of the line that the compiler indicated had the error.

Try the following example. Use MicroEMACS to enter the following program, which you should call **error.c**:

```
main() {
        printf("Hello, world")
}
```

Note that the semicolon was left off of the **printf** statement. Now, try compiling **error.c** with the following **cc** command:

```
        cc -A error.c
```

You should see no messages from the compiler, because they are all being diverted into a file to be used by MicroEMACS. Then, MicroEMACS will appear automatically. In one window you should see the message:

```
        3: missing ';'
```

and in the other you should see your source code for **error.c**, with the cursor set on line 3.

If you had more than one, typing **<ctrl-X>>** would move you to the next line with an error in it; typing **<ctrl-X><** would return you to the previous error. Note that with some errors, such as those for missing braces or semicolons, the compiler cannot always tell exactly which line the error occurred on, but it will almost always point to a line that is near the source of the error.

Now, correct the error. Close the file by typing **<ctrl-Z>**. **cc** will be invoked again automatically, to produce a normal working executable file. Note that **cc** will continue to invoke the MicroEMACS editor either until the program compiles without error, or until you exit from the editor by typing **<ctrl-U>** followed by **<ctrl-**

X><ctrl-C>.

Compiling multiple source files

Many programs consist of more than one source file. For example, the sample program **factor**, which is provided with Mark Williams C, consists of the files **factor.c** and **atod.c**. To compile a program with multiple source files, just type each file name as a separate argument on the **cc** command line:

```
cc factor.c atod.c -lm
```

This command compiles both source files. The argument **-lm** tells **cc** to include routines from the mathematics library **libm** when linking the object modules to produce an executable file. This option must come *after* the names of all of the source files which reference the library, or it will not work properly.

When the **cc** command line includes several file name arguments, **cc** by default uses the name of the first to form the name of the executable file. In the above example, cc produces the non-executable object modules **factor.o** and **atod.o**, and then links them together to produce the executable file **factor.prg**.

Naming executable files

To give the executable file a name other than the default, use the **-o** (output) option, followed by the desired name. For example, when Mark Williams C compiles the source file **hello.c**, by default it assigned the executable file the name **hello.prg**. Should you wish the executable file to have the name **hello.tos**, use the following command:

```
cc -o hello.tos hello.c
```

Linking without compiling

If you are writing a program that consists of several source files, you probably will want to compile the program, test it, and then change one or more of the source files. Rather than recompile all of the source files, you can save time by recompiling only the modified files and relinking the program.

For example, suppose you modified the **factor** program by changing only the source file **factor.c**. To recompile, you can use the following command:

```
cc factor.c atod.o -lm
```

The option **-lm** tells **cc** that this program needs to have the mathematics library **libm.a** included when it is linked.

In this example, the first two arguments are the C source file **factor.c** and the *object* module **atod.o**, rather than the *source* file **atod.c**. **cc** recognizes that **atod.o** is an object module and simply passes it to the linker without compiling it. You will find this particularly useful when your programs consist of many source files, and you need to compile only a few of them.

To simplify compiling, especially if you are developing systems that use many source modules, you should consider using the **make** command that is included with Mark Williams C. For more information on **make**, see the entry in the Lexicon, or see the tutorial for **make** that appears later in this manual.

Compiling without linking

At times, you may find it useful to compile a source file without linking the resulting object module to the other object modules or the libraries. You would do this, for example, if you wanted to compile a module to insert into a library. Use the **-c** option to tell **cc** not to link the compiled program. Later, you can use another **cc** command to link the program. For example, if you wanted just to compile **factor.c** without linking it, you would type:

```
cc -c factor.c
```

To link the resulting object module with the object module **atod.o** and with the appropriate libraries, type the following command:

```
cc factor.o atod.o -lm
```

Floating point output

A large amount of code is required to print floating point numbers. Because most C programs do not need to print floating point numbers, the conversion routine in the standard C library merely prints the message

```
You must compile with the -f option
to include printf() floating point.
```

To include the routines that print floating point numbers, use the -f option with the cc command.

Assembly language files

C makes most assembly language programming unnecessary; however, you may wish to write small parts of your programs in assembly language for greater speed or to provide access to processor features that C cannot use directly. Mark Williams C includes an assembler, named **as**, which is described in detail in the Lexicon.

To compile a program that consists of the C source file **cprog.c** and the assembly language source file **aprog.s**, simply use the **cc** command in its usual manner:

```
cc cprog.c aprog.s
```

cc recognizes that the suffix **.s** indicates an assembly language source file, and assembles it with the assembler **as**; then it links both object modules to produce an executable file.

The Lexicon entry for the TOS macro **Setexc** includes an example that demonstrates how to compile assembly language files with C-language files.

Generating assembly language output

The **cc** switch **-S** directs the C compiler use the disassembler **cc3** to compile into assembly language rather than an object (**.o**) module. You may wish to examine these assembly-language listings to debug a program, or examine how a particular library routine does its work.

Changing stack size

The size of the stack cannot be altered while a program is running. By default, the linker sets the stack size to two kilobytes; however, a highly recursive function may cause the stack to grow to the point where it invades other data areas. This will cause your program to work incorrectly.

Should your program need more than two kilobytes of stack, include the following global statement anywhere in your program:

```
long _stksize = nL;
```

where *n* is an *even* decimal number of bytes.

The **cc** command is summarized in the Lexicon, under the entry **cc**. Each phase of the compiler has its own entry. The entry for **as** gives full information on how to use this tool, plus listing the set of 68000 machine instructions.

Using the Mark Williams C Libraries

Mark Williams C includes a number of libraries whose routines perform many useful tasks. These include standard input and output (STDIO), memory management, sorting, and searching; mathematics functions; and accessing the GEM AES and VDI routines, as well as the Atari Line A functions.

Mark Williams C also includes the archiver **ar**, which helps you to update the current libraries or create your own libraries.

The following paragraphs will introduce some of the routines included in the Mark Williams C libraries, show you how to include them when you compile a program, and describe briefly how to use the archiver utility.

Strings and string handling

A commonly used data structure is the character *string*. The usual run-time representation for a string is an array of characters delimited by a NUL character.

If you need to move characters, use the library routine **strcpy**. This function takes two arguments: the first points to where the string will be copied to; the second points to where the string will be copied from. **strcpy** then copies all characters, through NUL, and returns the first argument.

You can measure the length of a string by using **strlen**. This function takes one argument, a pointer to a string, and returns the number of characters in the string, excluding the NUL that concludes the string.

strcat concatenates strings (i.e., joins them together). It takes two pointers to strings as arguments, and appends a copy of the second string to the end of the first. The first string is assumed to have enough extra space at the end to hold the new characters. **strcat** returns a pointer to the new result, which is delimited with NUL.

Often, strings must be compared. This must be done, for example, if an array of strings is being sorted. **strcmp** compares strings. It takes two arguments, both pointers to strings, and compares the strings they point to. **strcmp** returns a number less than zero if the first string is less than the second string, using native machine character comparisons; one equal to 0 if the two strings are equal; and one greater than 0 if the first string is longer than the second string.

Applications that deal with fixed-length strings can use the routines **strncat, strncpy,** and **strncmp**. They perform the same functions as their variable-length counterparts; however, all take a third argument that specifies the maximum length of the string.

See the entries in the Lexicon for these routines and for **string**; these will give you examples of how to use them in a C program.

Input and output

The standard library provides routines that perform input and ouput, or *I/O*, at a number of levels. Data can be transferred byte-by-byte, word-by-word, by string, by block, and in a formatted manner. For more information and examples of how to use these routines, see the Lexicon entry for **STDIO**.

The standard I/O header file **stdio.h** includes a type definition, or *typedef*, for the **FILE** type. A **FILE** is a structure that contains all of the information needed by the I/O routines to perform I/O operations on a connection. A pointer to a **FILE** is the external name of an I/O stream, and is passed to the various routines in the I/O library to specify which stream participates in the transfer. Note that a **FILE** can either be a file of data written on a disk or a logical device as defined by the operating system, e.g., the keyboard, the serial port, or the parallel port. C, like **msh**, does not distinguish between logical devices and files on disk—it regards them all merely as sources of data for it to handle.

To open a file and allocate a **FILE** type, use the routine **fopen**. It takes two arguments: the first is a string that contains the name of the file to be opened, and the second is a string that specifies the access mode required. The mode is one of the following: **r**, for plain reading; **w**, for plain writing; **r+w**, read plus write, or update; or **a**, for append. In addition, the mode string can contain the character **b**, for binary, which specifies that this is a binary stream; this ensures that newline characters will not be mapped into a carriage return/line feed sequence.

If the mode is w or a and the named file does not exist, it will be created. If the mode is w and the file does exist, it will be truncated to zero length. If the **FILE** could be opened, fopen returns a pointer to a **FILE** object; if it could not be opened, it returns NULL.

When all processing on a **FILE** is completed, the file must be closed by calling **fclose**. This routine takes one argument, a pointer to a **FILE**. All buffers are flushed and released, and the connection is detached.

The routine **exit** will automatically close all open files and return control of the computer to TOS. Your programs should always call **exit** when they are finished.

See the Lexicon entries for **exit**, **fopen**, **fclose**, and **STDIO** for more information, and for examples of how to use these routines.

Byte-by-byte I/O

The lowest level of I/O is the byte-by-byte level. Here, data are read from or written to a FILE one character at a time. All higher-level I/O routines use these byte-by-byte routines to read and write data.

The most basic read routine is **getc**(*fp*). This function takes one argument, a pointer to a **FILE**, and returns an **int** that contains either the next character from the **FILE** or the end-of-file signal **EOF**. **EOF** is defined in the header file **stdio.h**.

In ASCII mode, **getc** throws away all carriage return characters (0x0D); the line feeds at the end of the lines (0x0A) mark the end of the lines, because the '\n' in C is equal to 0x0A. In binary mode, all characters are passed without interpretation.

The routine **getchar** is equivalent to **getc(stdin)**; it reads characters from the standard input **FILE**, which is normally the keyboard.

The routine **ungetc**(*c*, *fp*) returns *c* to the **FILE** *fp*. This is useful for looking ahead at the next input character and then returning it to the input file. Only one character can be "unread" with **ungetc**.

The most basic write routine is **putc**(*c*, *fp*). This takes two arguments: *c*, which contains the byte to be written; and *fp*, which points to the output **FILE**. **putc** returns the first argument unless write error occurs, in which case it returns **EOF**.

putchar(*c*) is equivalent to **putc**(*c*, **stdout**). It writes characters to the standard output **FILE**, which is normally the video display.

See the Lexicon entries for these routines, which contain more information and examples of how to use them in C programs.

Word-by-word I/O

A program may read the next word (16-bit object) from a **FILE** by using the routine **getw**(*fp*). This routine takes one argument, a pointer to a **FILE**; it returns the word read.

Note that **getw** can return any bit pattern, including control characters. A special character like **EOF** can appear even in the middle of a file. Therefore, to prevent the file from being truncated accidentally, your program must test for end of file by using the macro **feof**(*fp*), from **stdio.h**. This macro looks at the **FILE** pointed to by *fp* and returns true if the last call to **getw** ran into the end of the file.

If a file has an odd size, the last call to **getw** will return the data and an error will be posted to the **FILE**. This error may be detected by using the **ferror**(*fp*) macro. End of file alone is posted if a call to **getw** produces no data.

In a similar manner, **putw**(*w*, *fp*) writes a word to a file. The **ferror** macro must be used to check for I/O errors.

See the Lexicon entries for these routines for examples and more information.

String I/O

A number of routines perform I/O on strings. The most basic one is **fgets(***b, n, fp***)**. It reads a string delimited by a newline character from the **FILE** pointed to by *fp*, and stores it into the array of characters *b*. The newline character is transferred to the buffer. NUL is placed in the buffer immediately after the newline. The integer *n* specifies the length of the buffer; this prevents **fgets** from writing beyond the array if a long line is encountered in the input. **fgets** returns *b* if any characters were read and **NULL** if not.

The routine **gets(***string***)** reads a newline-delimited string from the standard input stream and stores it within the array of characters *string*. Unlike **fgets**, **gets** deletes the newline character from the end of the string and replaces it with NUL.

The most basic string output routine is **fputs(***output*, *fp***)**. This routine writes the string *output* into the **FILE** pointed to by *fp*. **puts(***string***)** writes *string*, followed by a newline, onto the standard output.

For more information and examples of how to use the string I/O routines, see their entries in the Lexicon.

Block I/O

The standard library provides facilities to transfer blocks of memory to and from user programs. These are most often used on binary streams to move raw binary information to and from files; however, they may be used on ASCII streams without altering their data, with the possible exception of altering newline interpretation.

The function **fread(***b, size, n, fp***)** reads *n* objects of *size* bytes into the buffer pointed to by *b* from the **FILE** pointed to by *fp*. It returns the number of items actually read.

Likewise, the routine **fwrite(***b, size, n, fp***)** writes *n* objects, each *size* bytes long, from the buffer *b* to the **FILE** pointed to by *fp*. It returns the number of items written.

See the Lexicon entries for **fread** and **fwrite** for examples and more information. The **feof** and **ferror** macros can be used to check for end of file and transmission errors on block reads and writes.

Formatted I/O

Routines are provided that permit formatted I/O to and from **FILE** streams. Data may be read from and written into a number of formats and bases (decimal, octal, hexadecimal); strings may be truncated or padded; and fields may be justified to the left or to the right.

Although these routines are usually used on ASCII streams, they work perfectly well on binary streams; they are, after all, interfaces to **putc** and **getc**.

The formatted I/O routines **printf** and **scanf** are complex. The details of all their formatting options are described in detail in the Lexicon entries that describe them.

Briefly, all formatted I/O routines work by interpreting one argument as a *format string*. This string consists of *format specifications*, each of which is introduced by a a percent-sign character '%', plus other characters that specify the type of formatting to do. As each format specification is encountered in the format string, an argument is extracted from the list of parameters of the formatted I/O routine and interpreted in a fashion determined by the format specification: the first format specification takes the first argument, the second takes the second argument, and so on. The type of the argument must agree with that expected by the format specification; if this is not the case, such as when a **long** is placed in the argument list where an **int** is expected, the result is undefined. Characters placed between the quotation marks that do not belong to a format string (e.g., commas or spaces) are printed out literally.

For more information on how to use **print** and **scanf**, see their entries in the Lexicon.

Random access

All of the examples seen so far deal with sequential access **FILE** streams; however, the I/O library supports random access transfers as well. Associated with every **FILE** is a *seek pointer*. This pointer starts at the beginning of the file or, when a stream is opened for appending, at the end of a file; as data are read from or written to the **FILE**, it moved forward through the file.

You can obtain the value of this pointer by using **ftell(***fp***)**. It returns the current value of the seek pointer for the **FILE** pointed to by *fp*.

The seek pointer can be moved about in the file with the routine **fseek(***fp*, *where*, *how***)**. This resets the seek pointer in the **FILE** pointed to by *fp* to *where*, also a 32-bit integer. The *how* argument specifies if the seek is relative to the beginning of the file(*how* = 0), to the current-seek position (*how* = 1), or to the end of the file (*how* = 2). **fseek** returns 0 on success or -1 on failure.

Some **FILE** streams cannot perform random access operations; these include the ones attached to the videodisplay, the serial port, or the printer port.

Returning the seek pointer to the start of a file is eased by the routine **rewind(***fp***)**. This routine is equivalent to **fseek(***fp*, **0L, 0)**.

See the Lexicon entries for these routines for more information and examples.

Sorting

Often, data must be sorted. The standard library contains two sort functions. These functions are general, in that they implement only the skeleton of the sort algorithm. The user must provide a comparison function and tell the sort function the size of the objects being sorted.

The **qsort**(*b, n, size, f*) routine implements Hoare's quicksort algorithm. The argument *b* points to the base of the block of data being sorted, and the *n* argument specifies number of elements to be sorted. Each of these objects has *size* bytes; the routine needs the size to be able to move the objects around and to update its internal pointers. *f* points to a function that performs comparisons. The **shellsort**(*b, n, size, f*) routine has exactly the same calling sequence as **qsort**, but uses Shell's sorting method.

qsort can use large amounts of stack, because it is a recursive algorithm. To alter the size of the stack, include the following global statement:

```
long _stacksize = nL;
```

where *n* is an *even* number of bytes.

For more information on these routines and for examples, see the entries for **qsort** and shellsort in the Lexicon.

Dynamic memory allocation

When you build linked data structures or deal with arrays whose size can be determined only at runtime, it is helpful to be able to allocate blocks of memory dynamically. The standard functions **malloc**, **calloc**, and **free** implement a general-purpose memory allocation system used to allocate buffers.

To allocate memory, use **malloc**(*n*). This routine allocates a block of memory of at least *n* bytes and returns a character pointer to it. The block may be larger than requested, if allocating the exact size would create a very small, and probably unusable, block on the list of free memory. The block contains random information; it is not initialized in any way. If no memory is left in the free space pool, a NULL pointer will be returned.

calloc(*n, size*) uses **malloc** to allocate a block of memory large enough to hold *n* objects of size *size*; this memory is then zeroed. If there is insufficient free memory, a NULL pointer is returned.

Blocks of memory that are no longer needed can be returned to the free pool by passing a pointer to the block to **free**(*p*). This routine places the block back in the free list and merges adjacent free areas into single, larger free areas. It is a serious error to pass an incorrect pointer to **free**. No checking is done; a subsequent call to one of the allocation functions will probably return a very strange value.

Mathematics routines

The mathematics library **libm.a** contains a number of transcendental functions. They will calculate the sine, cosine, and tangent of a figure, plus their inverses and hyperbolic forms; calculate both natural and decimal logarithms; compute powers and exponents; and compute Bessel functions.

The Lexicon entry for the **mathematics library** introduces these functions; each has its own entry in the Lexicon, which include fuller descriptions and examples.

Note that to use a mathematics function, you must name the mathematics library on the **cc** command line when you compile your program. For example, if the program **sample.c** contains a mathematics function, use the following form of the **cc** command:

```
cc sample.c -lm
```

The option **-lm** indicates that you want the mathematics library to be included at link time. Note that this option must come at the *end* of the **cc** command, or the program will not link properly.

bios, **xbios**, *and* **gemdos** *functions*

Mark Williams C allows you to call directly the Atari ST's **bios**, **xbios**, and **gemdos** routines. Strictly speaking, these are not library functions; rather, they use routines that are built into the Atari BIOS to perform operating system tasks.

The **bios** routines perform basic I/O, such as managing the peripheral devices and disk drives. **xbios** routines extend the normal **bios** functions, to provide access to system clocks, the sound chip, the intelligent keyboard manager, the mouse, and other devices. **gemdos** routines perform such tasks as I/O with the parallel and serial ports, managing the disk drives, performing file I/O, managing dynamic memory, and managing processes.

These routines can be called directly through C programs. All are defined in the header file **osbind.h**, which is included with your distribution.

The following program demonstrates how the **xbios** function **Setcolor** is used in a C program. It inverts the color setting on a monochrome monitor:

```
#include <osbind.h>

main() {
        int color = Setcolor(0, -1);
        Setcolor(0, ++color%2);
}
```

For more information on these routines, see the entries for **bios**, **gemdos**, and **xbios** in the Lexicon.

UNIX routines

The standard library includes a number of routines that mimic a variety of UNIX system calls, to manipulate files. These work in a somewhat different fashion than similar routines built into TOS, and allow programs written for the UNIX operating system or for MS-DOS to be compiled and run under TOS with minimal alteration.

For more information, see the Lexicon entry on **UNIX routines**.

The AES and VDI libraries

AES and **VDI** are elements in the GEM (graphics environment management) system. Together, they give the user a convenient way to interact with the computer through graphics images.

VDI stands for *virtual device interface*. It consists of a set of basic graphics routines that draw lines, polygons, circles, and ellipses; plus routines that allow your program to receive information from the user; plus a set of text fonts and device drivers.

The VDI allows programmers to create graphics images that can be displayed on any of a number of devices, including the screen, printer, tablet, plotter, video slide camera, and others.

In addition, the VDI allows you to write *metafiles*, which hold the logical description of a graphics image. An image stored in this manner can be manipulated easily by the user, which enhances the interactive powers of the GEM system.

AES stands for *application environment system*. In effect, the AES combines elements of the disk operating system and of the VDI to create a graphics-oriented environment for the user. The AES governs the running of processes, or *applications*, on the Atari ST; it also controls the running of menus and windows on the GEM desktop.

To programmers, AES routines give a way to use windows, menus, graphics objects, dialogues, and the other pre-defined elements of the graphics environment.

The AES and VDI routines themselves live in the Atari ROM BIOS. Calls to them are kept in two libraries: **libaes.a** and **libvdi.a**, respectively. Bindings that declare these routines in the C manner are kept in the header files **aesbind.h** and **vdibind.h**. Three additional header files are also included to help you create programs, as follows: **gemdefs.h** includes a number of mnemonic definitions, and declares some structures used in VDI programs; **obdefs.h** declares structures used to create graphics objects; and **portab.h** defines terms used in the DRI dialect of C.

Compiling programs that use AES and VDI

To compile a program with AES or VDI routines, use the -VGEM option on the cc command line. For example, if the program **sample.c** has AES routines in it, you can compile it with the following command:

```
cc -VGEM sample.c
```

Every AES and VDI routine, header file, and archive is described in the Lexicon. If you are not sure where to begin, first look up the individual entries for **AES** and **VDI**. Each gives more information about how these tools work; each also contains a list and brief summary of each of its library routines. Separate articles also have been written to describe *windows*, *menus*, *graphics objects*, and *metafiles*. These entries describe in detail how to use these specialized graphics forms. The entries for the individual library routines also contain numerous examples. Each example is a complete C program that can be compiled and run, and demonstrates some aspect of the AES/VDI environment. These examples are a good place to begin your study of AES/VDI.

The Line A library

Line A is the interface to the Atari ST's assembly-language-level graphics routines. Its name refers to the fact that its opcodes all begin with the hexadecimal number 'A' (0xA).

Each Line A function consists of few lines of assembly language, which save registers, load parameters, execute one of the unimplemented Line A instructions, restore registers, and return. These perform simple graphics functions, such as drawing lines, displaying characters, or drawing polygons. The GEM VDI routines use Line A to do their work.

Most Line A functions pass their parameters through an external structure, rather than through arguments passed in the usual manner. The exceptions are **linea7**, which uses a specialized structure to copy and move portions of the screen (also called "blitting"); **lineac**, which takes a pointer; and **linead**, which takes two pointers. All functions and structures are declared in the header file **linea.h**.

The entry on Line A in the Lexicon contains more information, and includes two sample programs.

Debugging Programs with Mark Williams C

Mark Williams C comes with several utilities that help you debug your programs. These include **db**, which is a powerful symbolic debugger; **nm**, which prints symbol tables from programs, for analysis; and **od**, which will print an octal dump of a file.

db: *the debugger*

Mark Williams C includes a symbolic debugger to assist you with debugging your programs. **db** can work with a compiled program, and includes a number of commands that allow you to examine just what your program is doing during its execution.

To see what **db** can do, compile the program **hello.c**, which you created earlier in this tutorial, by entering the following command:

```
cc hello.c
```

Now, step through the following script. **db**'s commands are in **boldface** in the left-hand column; the right-hand column gives a brief description of what each command does.

db hello.prg	invoke debugger
printf:b	set tracepoint on **printf**
:p	display all breakpoints
:e	run program
:t	do traceback
:r	look at the registers
printf,20?i	symbolically disassemble 20 instructions
:c	continue execution
:p	display breakpoints; none shown as program is over
:q	quit **db**

As you can see, **db** allows you to set breakpoints, run through the program, and examine what it does in a variety of manners. For a fuller introduction **db**, and instructions on how to use it to debug your programs, see the entry for **db** in the Lexicon.

od: *octal or hexadecimal dump*

od prints out a file in octal machine words. If you type **od** without an argument, it accepts what you type at the keyboard as input; when you type a **<ctrl-Z>** and carriage return, it then returns what you typed in octal. Normally, you give **od** a file name as an argument; to display an octal dump of the file **tempfile**, type:

```
od tempfile
```

od can also display files in hexadecimal or decimal, and in bytes or words, whichever you prefer.

nm: *print symbol tables*

nm prints out the symbol table from an object module or library. It is designed to work with libraries created by with the archiver **ar**, and with object modules compiled with Mark Williams C.

By default, **nm** only prints symbols with a C-style format. To use for the library **libc.a**, simply type

```
nm a:\lib\libc.a
```

For more information on using these debugging tools, see their entries in the Lexicon.

Selected References

The following list names books that you may find helpful in developing your skills with C. This list is by no means exhaustive; however, it should prove helpful to both beginners and experienced programmers.

American National Standards Institute: *Draft Programming Language C (May 1986 Draft).* Washington, D.C.: X3 Secretariat, Computer and Business Equipment Manufacturers Association, 1986.

AT&T Bell Laboratories: *The C Programmer's Handbook.* Englewood Cliffs, N.J.: Prentice-Hall Inc., 1985.

Chirlin, P.M.: *Introduction to C.* Beaverton, Or.: Matrix Publishers Inc., 1984.

Derman, B. (ed.): *Applied C.* New York: Van Nostrand Reinhold Co. Inc., 1986.

Feuer, A.R.: *The C Puzzle Book.* Englewood Cliffs, N.J.: Prentice-Hall Inc., 1982.

Gehani, G.: *Advanced C: Food for the Educated Palate.* Rockville, Md.: Computer Science Press, 1985.

Hancock, L., Krieger, M.: *The C Primer.* New York: McGraw-Hill Book Publishers Inc., 1982.

Hogan, T.: *The C Programmer's Handbook.* Bowie, Md.: Brady Publishing, 1984.

Kernighan, B.W., Ritchie, D.M.: *The C Programming Language.* Englewood Cliffs, N.J.: Prentice-Hall Inc., 1978.

Kernighan, B.W., Plauger, P.J.: *The Elements of Programming Style*, ed. 2. New York: McGraw-Hill Book Co., 1978.

Kochan, S.G.: *Programming in C*. Hasbrouck Heights, N.J.: Hayden Book Co. Inc., 1983.

Knuth, D.E.: *The Art of Computer Programming*, vol. 1: *Basic Algorithms*. Reading, Ma.: Addison-Wesley Publishing Co., 1969.

Knuth, D.E.: *The Art of Computer Programming*, vol. 2: *Seminumerical Algorithms*. Reading, Ma.: Addison-Wesley Publishing Co., 1969.

Knuth, D.E.: *The Art of Computer Programming*, vol. 3: *Sorting and Searching*. Reading, Ma.: Addison-Wesley Publishing Co., 1969.

Plum, T.: *Learning to Program in C*. Cardiff, N.J.: Plum Hall Inc., 1983.

Plum, T.: *C Programming Guidelines*. Cardiff, N.J.: Plum Hall Inc., 1984.

Purdum, J.: *C Programming Guide*. Indianapolis: Que Corp., 1983.

Purdum, J., Leslie, T.C., Stegemoller, A.L.: *C Programmer's Library*. Indianapolis: Que Corp., 1984.

Traister, R.J.: *Programming in C for the Microprocessor User*. Englewood Cliffs, N.J.: Prentice-Hall Inc., 1984.

Traister, R.J.: *Going from BASIC to C*. Englewood Cliffs, N.J.: Prentice-Hall Inc., 1984.

Waite, M., Prata, S., Martin, D.: *C Primer Plus*. Indianapolis: Howard W. Sams Inc., 1984.

Weber Systems, Inc.: *C Language User's Handbook*. New York: Ballantine Books, 1984.

Zahn, C.T.: *C Notes*. New York: Yourdan Press, 1979.

Atari ST information

Balma, P., Fitler, W.: *Programmer's Guide to GEM*. Berkeley, Calif.: SYBEX, Inc., 1986.

Digital Research Institute: *GEM Programmer's Guide*. Pacific Grove, Calif.: Digital Research Institute, Inc., 1984.

General Instrument Corporation: *Programmable Sound Generator Data Manual*. Hicksville, N.Y.: General Instrument Corporation, 1981.

Gerits, K., Englisch, L., Bruckmann, R.: *Atari ST Internals: The Authoritative Insider's Guide*. Grand Rapids, Mich.: ABACUS Software, Inc., 1986.

M68000 16/32-Bit Microprocessor Programmer's Reference Manual, ed. 4. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1984.

Oren, T.: *Professional GEM*. Available through CompuServe, ANTIC-ONLINE, Atari ST forum. *Highly recommended*.

Szczepanowski, N., Gunther, B.: *Atari ST GEM Programmer's Reference*. Grand Rapids, Mich.: ABACUS Software, Inc., 1986.

2. Error messages

This chapter lists all of the error messages that the compiler, the assembler **as**, and the linker **ld** can produce.

The messages are in alphabetical order, and are marked whether they come from the compiler, the assembler, or the linker. Those from the compiler are marked by the compilation phase, and whether it indicates a *fatal*, *error*, *strict*, or *warning* condition. The compilation phases are **cpp**, the preprocessor; **cc0**, the parser; **cc1**, the code generator; **cc2**, the optimizer; and **cc3**, the disassembler.

A fatal message usually indicates a condition that caused the compiler to terminate execution; fatal errors from the later phases of compilation often cannot be fixed by the user, and most likely indicate problems in the compiler itself.

An error message refers to a condition in the source code, such as unbalanced braces, that caused compilation to cease.

Warning messages point out code that is compilable, but may produce trouble when the program is executed. A strict message refers to a passage in the code that is unorthodox and may not be transportable.

Each error message is followed by a brief description and explanation.

. (as, error)
Dot label error. This indicates that a period was used as a label, e.g., ".:".

a (as, error)
Addressing error. This is generated by nearly any kind of operand/instruction mismatch or semantic error in address fields.

address wraparound (**ld**, fatal)
A segment of the program has exceeded the size allowed by the microprocessor's architecture.

ambiguous reference to "*string*" (**cc0**, error)
string is defined as a member of more than one **struct** or **union**, and is referenced via a pointer to one of those **structs** or **unions**, and there is more than one offset that could be assigned.

argument list has incorrect syntax (**cc0**, error)
The argument list of a function declaration contains something other than a comma-separated list of formal parameters.

array bound must be a constant (**cc0**, error)
An array size has been declared with a variable or undefined size.

array bound must be positive (**cc0**, error)
An array was declared to have a negative size.

array bound too large (**cc0**, error)
The array is too large to be compiled with 16-bit index arithmetic. The programmer should devise a way to divide the array into compilable portions, or rethink the approach to handling these data.

array row has 0 length (**cc0**, error)
An array was declared to be zero units long, or some dimension of the array other than the first was made to be flexible.

associative expression too complex (**cc1**, fatal)
An expression involving associative binary operators, such as '+' in **i=i1+i2+i3+...+i30;**, has too many operators. Simplify the expression.

bad argument storage class (**cc0**, error)
An argument was assigned a storage class that the compiler does not recognize. The only valid storage class is **register**.

bad external storage class (**cc0**, error)
An **extern** has been declared with an invalid storage class, e.g., **register** or **auto**.

bad field width (**cc0**, error)
A field width was declared to be negative or greater than will fit into the object holding it.

bad filler field width (**cc0**, error)
A filler field width was declared to be negative or greater than will fit into the object holding it.

bad flexible array declaration (**cc0**, error)
Flexible arrays have missing array bounds, e.g., **char *argv[]**.

baddisk:disk error (**ld**, fatal)
ld either cannot read or cannot write to the mass-storage device. Check the disk you are using to see that it is working correctly.

break not in a loop (**cc0**, error)
A **break** occurs that is not inside a loop or a **switch** statement.

call of non function (**cc0**, error)
What the program attempted to call is not a function.

cannot add pointers (**cc0**, error)
The program attempted to add two pointers. **ints** may be added to or subtracted from pointers, and two pointers to the same type may be subtracted, but no other arithmetic operations are legal on pointers.

cannot apply unary '&' to a bit field (**cc0**, error)
The program attempted to use the address of a bit within a byte, which is illegal; only bytes can be addressed, not the bits within them.

cannot apply unary '&' to a register variable (cc0, error)
Because register variables are stored within registers, they do not have addresses, which means that the unary **&** operator cannot be used with them.

cannot cast double to pointer (cc0, error)
The program attempted to cast a **double** to a pointer.

cannot cast pointer to double (cc0, error)
The program attempted to cast a pointer to a **double**.

cannot cast structure or union (cc0, error)
The program attempted to cast a **struct** or a **union**.

cannot cast to structure or union (cc0, error)
The program atempted to cast a variable to a **union** or **struct**.

string: cannot create (as, error)
The assembler as cannot create the output file it was requested to create. This often is due to a problem with the output device; check and make sure that it is working correctly and not full.

string: cannot create (cpp, fatal)
The preprocessor **cpp** cannot create the output file *string* that it was asked to create. This often is due to a problem with the output device; check and make sure that it is working correctly and is not full.

cannot create *string* (ld, fatal)
The linker **ld** cannot create the output file it was requested to create. This often is due to a problem with the output device; check and make sure that it is working correctly and is not full.

cannot declare array of functions (cc0, error)
extern int (*f[])(); declares **f** to be an array of pointers to functions that return ints. Arrays of functions are illegal.

cannot declare flexible automatic array (cc0, error)
The program does not explicitly declare the number of elements in an automatic array.

cannot initialize fields (cc0, error)
The program attempted to initialize bit fields within a structure. This is not supported.

cannot initialize unions (cc0, error)
The program attempted to initialize a **union** within its declaration. **unions** cannot be initialized.

string: cannot open (cpp, fatal)
The preprocessor **cpp** cannot open the file *string* of source code that it was asked to read. **cpp** may not have been correctly told the directory in which

this file is to be found; check that the file is located correctly.

cannot open include file *string* (**cpp**, error)
The program asked for file *string*, that was not found in the same directory as the source file, nor in the default **include** directory specified by the environmental variable **INCDIR**, nor in any of the directories named in **-I** options given to the **cc** command.

cannot open *string* (seg *number*) (**ld**, fatal)
The linker **ld** cannot open the object module that it was asked to read. Make sure that the storage device is working correctly, and that **ld** has been given the correct names of the file and of the directory in which it is stored.

string: cannot reopen (**cc2**, fatal)
The optimizer **cc2** cannot reopen a file that it has been working with. Make sure that your mass storage device is working correctly and that it is not full.

can't open lib*string*.a (**ld**, fatal)
The linker **ld** cannot open a library that it has been asked to link into your program. Make sure that you named the library correctly and that the environmental parameter **LIBPATH** is set correctly if you used the **-l** option to the **cc** command line.

can't open *string* (**ld**, fatal)
The linker **ld** cannot open a file that it has been asked to work with. Make sure that your mass storage device is working correctly, and that **ld** has been given the correct names of the file and of the directory in which it is stored.

can't open temp file (**ld**, fatal)
The linker **ld** cannot open the temporary file created by the compiler or the assembler. Make sure that your mass storage device is working correctly, and that **ld** has been given the correct names of the file and of the directory in which it is stored.

can't read *string* (**ld**, fatal)
The linker **ld** cannot read the file named. Make sure that your mass storage device is working correctly, and that **ld** has been given the correct names of the file and of the directory in which it is stored.

case not in a switch (**cc0**, error)
The program uses a **case** label outside of a **switch** statement.

class not allowed in structure body (**cc0**, error)
A storage class such as **register** or **auto** was specified within a structure.

compound statement required (**cc0**, error)
A construction that requires a compound statement, e.g., a function definition, array initialization, or **switch** statement, does not have one.

constant expression required (**cc0**, error)
The program uses a variable in a statement that requires a constant expression.

constant "*number*" promoted to long (**cc0**, warning)
The compiler promoted a constant in your program to **long**; although this is not strictly illegal, it may create problems when you attempt to port your code to another system, especially if the constant appears in an argument list.

constant used in truth context (**cc0**, strict)
A conditional expression for an **if**, **while**, or **for** statement has turned out to be always true or always false.

construction not in Kernighan and Ritchie (**cc0**, strict)
This construction is not found in *The C Programming Language*; although it can be compiled by Mark Williams C, it may not be portable to another compiler.

continue not in a loop (**cc0**, error)
The program uses a **continue** statement that is not inside a loop.

#define argument mismatch (**cpp**, warning)
The definition of an argument in a **#define** statement does not match its subsequent use. One or the other should be changed.

declarator syntax (**cc0**, error)
The program used incorrect syntax in a declaration.

default label not in a switch (**cc0**, error)
The program used a **default** label outside a **switch** construct.

disk error (**ld**, fatal)
The linker **ld** encountered a problem with the storage device when it attempted to read or write a file. Check that the disk is working correctly; if **ld** is working with a floppy disk, make sure that the disk is sound and that it is not write-protected.

divide by zero (**cc0**, warning)
The program will divide by zero if the code just parsed is executed. Although the program can be parsed, this statement may create trouble if executed.

duplicated case constant (**cc0**, error)
A case value may appear only once in a **switch** statement.

#else used without #if or #ifdef (**cpp**, error)
The program has an **#else** without a preceding **#if**. Most likely, an extra **#else** was inserted, or an **#if** or **#ifdef** was overlooked.

empty switch (**cc0**, warning)
A **switch** statement has no case labels and no default labels.

#endif used without #if or #ifdef (**cpp**, error)
The program has an **#endif** without a preceding **#if**.

EOF in comment (**cpp**, error)
An **EOF** appears in a comment. The file of source code may have been truncated, or you failed to close a comment; make sure that each open-comment symbol "/*" is balanced with a close-comment symbol "*/".

EOF in macro argument (**cpp**, warning)
An **EOF** appears in a macro argument.

error in #define syntax (**cpp**, error)
The syntax of a **#define** statement is incorrect.

error in enumeration list syntax (**cc0**, error)
The syntax of an enumeration declaration contains an error.

error in expression syntax (**cc0**, error)
The parser expected to see a valid expression, but did not.

error in #include syntax (**cpp**, error)
An **#include** directive must be followed by a string enclosed by either quotation marks (" ") or angle brackets (<>).

expression too complex (**cc1**, fatal)
The code generator cannot generate code for an expression. Simplify your code.

external syntax (**cc0**, error)
This could be one of several errors, most often a missing '{'.

file ends within a comment (**cc0**, error)
The source file ended in the middle of a comment. If the program uses nested comments, it may have mismatched numbers of begin-comment and end-comment markers. If not, the program began a comment and did not end it, perhaps inadvertently when dividing by **something*, e.g., **a=b/*cd;**.

function cannot return a function (**cc0**, error)
The function is declared to return to another function. A function, however, can return a *pointer* to a function, as does **int (*signal(n, a))()**.

function cannot return an array (**cc0**, error)
A function is declared to return an array. A function can return a pointer to a structure or array.

functions cannot be parameters (**cc0**, error)
The program declares a function as a parameter, e.g., **int q(); x(q);**.

identifier "*string*" is being redeclared (**cc0**, error)
The program declares variable *string* to be of two different types. This often may be due to an implicit declaration, the use of a function before a subse-

quent declaration. Check for name conflicts.

identifier "*string*" is not a label (**cc0**, error)
The program attempts to **goto** a nonexistent identifier.

identifier "*string*" is not a parameter (**cc0**, error)
The variable "*string*" did not appear in the parameter list.

identifier "*string*" is not defined (**cc0**, error)
The program uses identifier *string* but does not define it.

identifier "*string*" not usable (**cc0**, error)
string is probably a member of a structure or **union** which appears by itself in an expression.

illegal character constant (**cc0**, error)
Legal character constants consist of a single letter, a backslash '\' followed by one of \, **n**, **t**, **b**, **r**, **f**, **v**, or **a**, or a backslash followed by up to three octal digits.

illegal character (*number* decimal) (**cc0**, error)
A control character was embedded within the source code. *number* is the decimal value of the character.

illegal # construct (**cc0**, error)
The parser recognizes control lines of the form *#line_number* (decimal) or *#file_name*. Anything else is illegal.

illegal control line (**cpp**, error)
A '#' is followed by a word that the compiler does not recognize.

illegal label "*string*" (**cc0**, error)
The program uses the keyword *string* as a **goto** label. Remember that each label must end with a colon.

illegal operation on "void" type (**cc0**, error)
The program tried to manipulate a value returned by a function that had been declared to be of type **void**.

illegal structure assignment (**cc0**, error)
The structures have different sizes.

illegal subtraction of pointers (**cc0**, error)
A pointer can be subtracted from another pointer only if both point to objects of the same size.

illegal use of a pointer (**cc0**, error)
A pointer was used illegally, e.g., multiplied, divided, or anded. You may get the result you want if you cast the pointer to a **long**.

illegal use of a structure or union (**cc0**, error)
  You may take the address of a **struct**, access one of its members, assign it to another structure, pass it as an argument, and return. All else is illegal.

illegal use of floating point (**cc0**, error)
  A **float** was used illegally, e.g., in a bit-field structure.

illegal use of "void" type (**cc0**, error)
  The program used **void** creatively. Strictly, there are only **void** functions; Mark Williams C also supports the cast to **void** of a function call.

illegal use of void type in cast (**cc0**, error)
  The program uses a pointer where it should be using a variable.

string in #if (**cpp**, error)
  A syntax error occurred in a **#if** declaration. *string* describes the error in detail.

inappropriate "alien" modifier (**cc0**, error)
  The **alien** type is used to interface C with non-C functions; your program tried to use **alien** as an internal function rather than as a reference to an external function.

inappropriate "long" (cc0, error)
  Your program used the type **long** inappropriately, e.g., to describe a **char**.

inappropriate "short" (**cc0**, error)
  Your program used the type **short** inappropriately, e.g., to describe a **char**.

inappropriate "unsigned" (cc0, error)
  Your program used the type **unsigned** inappropriately, e.g., to describe a **double**.

indirection through non pointer (**cc0**, error)
  The program attempted to use a scalar as a pointer; you must first cast it to a pointer to something.

initializer too complex (**cc0**, error)
  An initializer was too complex to be calculated at compile time. You should simplify the initializer to correct this problem.

integer pointer comparison (**cc0**, strict)
  The program compares an integer with a pointer without casting one to the type of the other. While this is legal, the comparison may not work on machines with non-integer pointers, e.g., segmented Z8000 or LARGE-model i8086, or on machines with pointers larger than **ints**, e.g., the 68000.

integer pointer pun (**cc0**, strict)
  The program assigns a pointer to an integer, or vice versa, without casting the right-hand side of the assignment to the type of the left-hand side. While this

is permitted, it is often an error if the integer has less precision than the pointer does. Make sure that any functions called in the expression that return pointers are properly declared.

internal compiler error (**cc0**, **cc1**, **cc2**, **cc3**, fatal)
The program produced a state that should not happen during compilation. Forward a copy of the program, preferably on a machine-readable medium, to Mark Williams Company, together with the version number of the compiler, the command line used to compile the program, and the system configuration. For immediate advice during business hours, telephone Mark Williams Company.

Internal error, c=*number* in expr. (**as**, error)
Internal problem; contact Mark Williams Company.

"*string*" is a enum tag (**cc0**, error)

"*string*" is a struct tag (**cc0**, error)

"*string*" is a union tag (**cc0**, error)
string has been previously declared as a tag name for a **struct**, **union**, or **enum**, and is now being declared as another tag. Perhaps the structure declarations have been included twice.

"*string*" is not a tag (**cc0**, error)
A **struct** or **union** with tag *string* is referenced before any such **struct** or **union** is declared. Check your declarations against the reference.

"*string*" is not a typedef name (**cc0**, error)
string was found in a declaration in the position in which the base type of the declaration should have appeared. *string* is not one of the predefined types or a **typedef** name.

"*string*" is not an "enum" tag (**cc0**, error)
An **enum** with tag *string* is referenced before any such **enum** has been declared.

class "*string*" (*number*) is not used (**cc0**, strict)
Space was allocated for the variable *string* of the given *class* by the declaration on line *number*, but it was not used.

label "*string*" undefined (**cc0**, error)
The program does not declare the label *string*, but it is referenced in a **goto** statement.

left side of "*string*" not usable (**cc0**, error)
The left side of *string* should be a pointer, but is not.

lvalue required (**cc0**, error)
The left-hand value of a declaration is missing or incorrect.

m (**as**, error)
Multiple definition. The offending line is involved in the multiple definition of a label.

macro body too long (**cpp**, fatal)
The size of the macro in question exceeds 200 bytes, which is the limit designed into the preprocessor. Try to shorten or split the macro.

macro expansion overflow (**cpp**, fatal)
The program contains a macro extension that exceeds the allowed limit of 200 bytes.

macro *string* redefined (**cpp**, warning)
The program redefined the macro *string*.

macro *string* requires arguments (**cpp**, error)
The macro calls for arguments that the program has not supplied.

macros nested *number* deep, loop likely (**cpp**, error)
Macros call each other *number* times. You would be well advised to simplify the program.

member "*string*" is not addressable (**cc0**, error)
The array *string* has exceeded the machine's addressing capability. Structure members are addressed with 16-bit signed offsets on most machines.

member "*string*" is not defined (**cc0**, error)
The programs references a structure member that has not been declared.

mismatched conditional (**cc0**, error)
In a '?'-':' expression, the colon and all three expressions must be present.

misplaced ":" operator (**cc1**, error)
The program used a colon without a preceding question mark. It may be a misplaced label.

missing "=" (**cc0**, warning)
An equal sign is missing from the initialization of a variable declaration. Note that this is a warning, not an error: this allows Mark Williams C to compile programs with "old style" initializers, such as **int i 1**; however, use of this feature is strongly discouraged.

missing ":" (**cc0**, error)
A colon ':' is missing after a **case** label, a default label, or a '?' in a '?'-':' construction.

missing "," (**cc0**, error)
A comma is missing from an enumeration member list.

missing "{" (**cc0**, error)
: A left brace '{' is missing after a **struct** *tag*, **union** *tag*, or **enum** *tag* in a definition.

missing "(" (**cc0**, error)
: The **if**, **while**, **for**, and **switch** keywords must be followed by parenthesized expressions.

missing "}" (**cc0**, error)
: A right brace '}' is missing from a **struct**, **union**, or definition, from an initialization, or from a compound statement.

missing "]" (**cc0**, error)
: A right bracket ']' is missing from an array declaration, or from an array reference.

missing ")" (**cc0**, error)
: A right parenthesis ')' is missing anywhere after a left parenthesis '('.

missing ";" (**cc0**, error)
: A semicolon ';' does not appear after an external data definition or declaration, after a **struct** or **union** member declaration, after an automatic data declaration or definition, after a statement, or in a **for(;;)** statement.

missing "while" (**cc0**, error)
: A **while** command does not appear after a **do** in a **do-while()** statement.

missing #endif (**cpp**, error)
: An **#if**, **#ifdef**, or **#ifndef** statement was not closed with an **#endif** statement.

missing label name in goto (**cc0**, error)
: A **goto** statement does not have a label.

missing member (**cc0**, error)
: A '.' or "->" is not followed by a member name.

missing output file (**cpp**, fatal)
: The preprocessor **cpp** found a **-o** option that was not followed by a file name for the output file.

missing right brace (**cc0**, error)
: A right brace is missing at end of file. The missing brace probably precedes lines with errors reported earlier.

missing "*string*" (**cc0**, error)
: The parser **cc0** expects to see token *string*, but sees something else.

missing semicolon (**cc0**, error)
: External declarations should continue with ',' or end with ';'.

missing type in structure body (**cc0**, error)
A structure member declaration has no type.

multiple classes (**cc0**, error)
An element has been asigned to more than one storage class, e.g., **extern register**.

multiple #else's (**cpp**, warning)
An **#ifdef**, **#if**, or **#ifndef** statement is followed by more than one **#else** statement.

multiple types (**cc0**, error)
An element has been assigned more than one data type, e.g., **integer float**.

nested comment (**cpp**, warning)
By default, Mark Williams C does not accept nested comments. To tell Mark Williams C to accept nested comments in your program, use the option **-VCNEST** on your **cc** command line.

newline in macro argument (**cpp**, warning)
A macro argument contains a newline character.

no input found (**ld**, fatal)
The **ld** command line names no object or archive files to link.

nonterminated string or character constant (**cc0**, error)
A line that contains single or double quotation marks left off the closing quotation mark. A newline in a string constant may be escaped with '\'.

number has too many digits (**cc0**, error)
A number is too big to fit into its type.

o (as, error)
An unrecognized opcode mnemonic was found. Contrast this with error 'q', where the opcode is recognized but the syntax line is in error.

only one default label allowed (**cc0**, error)
The program uses more than one **default** label in a **switch**.

out of space (**ld**, fatal)
malloc could not allocate adequate space in memory for the linker **ld** to work.

out of space (fBcppfR, **cc0**, **cc1**, **cc2**, **cc3**, fatal)
The compiler ran out of space while attempting to compile the program. To remove this error, examine your source and break up any functions that are extraordinarily large.

out of tree space (**cc0**, fatal)
The compiler allows a program to use up to 350 tree nodes; the program exceeded that allowance.

outdated ranlib (**ld**, warning)
The date stamp on the library file is younger than that in the ranlib header. If the library has been altered, the ranlib can be updated with the archiver **ar**; see the Lexicon entry on **ar** to see how this is done. If the library has not been altered, this message may be due to an installation error; see the Lexicon entry on **ranlib** for more information.

p (**as**, error)
Phase error. The value of a label changed during the assembly. An instruction has a size that differs between the first and second passes.

potentially nonportable structure access (**cc0**, strict)
A program that uses this construction may not be portable to another compiler.

preprocessor assertion failure (**cpp**, warning)
A **#assert** directive that was tested by the preprocessor **cpp** was found to be false.

q (**as**, error)
Questionable syntax. The assembler has no idea how to parse this line, and it has given up.

r (**as**, error)
Relocation error. The program attempted to create or use an expression in a way that the linker cannot resolve.

return type/function type mismatch (**cc0**, error)
What the function was declared to return and what it actually returns do not match, and cannot be made to match.

return(e) illegal in void function (**cc0**, error)
A function that was declared to be type **void** has nevertheless attempted to return a value. Either the declaration or the function should be altered.

risky type in truth context (**cc0**, strict)
The program uses a variable declared to be a pointer, **long**, **unsigned long**, **float**, or **double** as the condition expression in an **if**, **while**, **do**, or '?'-':'. This could be misinterpreted by some C compilers.

s (**as**, error)
Segment error. The program attempted to initialize something in a segment that contains only uninitialized data.

size of struct "string" is not known (**cc0**, error)

size of union "string" is not known (**cc0**, error)
A pointer to a **struct** or **union** is being incremented, decremented, or subjected to array arithmetic, but the **struct** or **union** has not been defined.

size of *string* too large (cc0, error)
: The program declared an array or **struct** that is too big to be addressable, e.g., **long a[20000]**; on a machine that has a 64-kilobyte limit on data size and four-byte **longs**.

sizeof(*string*) set to *number* (cc0, warning)
: The program attempts to set the value of *string* by applying **sizeof** to a function or an **extern**; the compiler in this instance has set *string* to *number*.

storage class not allowed in cast (cc0, error)
: The program **casts** an item as a **register**, **static**, or other storage class.

structure "*string*" does not contain member "*m*" (cc0, error)
: The program attempted to address the variable *string.m*, which is not defined as part of the structure *string*.

structure or union used in truth context (cc0, error)
: The program uses a structure in an **if**, **while**, or **for**, or '?'-':' statement.

switch of non integer (cc0, error)
: The expression in a **switch** statement is not type **int** or **char**. You should cast the **switch** expression to an **int** if the loss of precision is not critical.

switch overflow (cc1, fatal)
: The program has more than ten nested **switches**.

too many adjectives (cc0, error)
: A variable's type was described with too many of **long**, **short**, or **unsigned**.

too many arguments (cc0, fatal)
: No function may have more than 30 arguments.

too many arguments in a macro (cpp, fatal)
: The program uses more than ten arguments with a macro.

too many cases (cc1, fatal)
: The program cannot allocate space to build a **switch** statement.

too many directories in include list (cpp, fatal)
: The program uses more than ten **#include** directories.

too many initializers (cc0, error)
: The program has more initializers than the space allocated can hold.

too many structure initializers (cc0, error)
: The program contains a structure initialization that has more values than members.

trailing "," in initialization list (cc0, warning)
: An initialization statement ends with a comma, which is legal.

type clash (**cc0**, error)
The parser expected to find matching types but did not. For example, the types of **e1** and **e2** in **(x) ? e1 : e2** must either both be pointers or neither be pointers.

type required in cast (**cc0**, error)
The type is missing from a cast declaration.

u (as, error)
A symbol is used but never defined. The symbol's name is displayed.

unexpected end of enumeration list (**cc0**, error)
An end-of-file flag or a right brace occurred in the middle of the list of enumerators.

unexpected EOF (**cc0**, **cc1**, **cc2**, **cc3**, fatal)
EOF occurred in the middle of a statement. The temporary file may have been corrupted or truncated by an earlier phase.

union "*string*" does not contain member *m* (**cc0**, error)
The program attempted to address the variable string *m*, which is not defined as part of the structure *string*.

string: unknown option (**cpp**, fatal)
The preprocessor **cpp** does not recognize the option *string*. Try re-typing the cc command line.

zero modulus (**cc0**, warning)
The program will perform a modulo operation by zero if the code just parsed is executed. Although the program can be parsed, this statement may create trouble if executed.

3. The Lexicon

The rest of this manual consists of the Lexicon. The Lexicon consists of more than 700 articles, each of which describes a function or command, defines a term, or otherwise gives you useful information. The articles are organized in alphabetical order, to ensure that everything is easy to find.

The following page gives a sample article in the Lexicon format. For more information on how to use the Lexicon and how it is organized, see the entry in the Lexicon on **Lexicon**.

example--Sample entry

Give an example of Mark Williams Lexicon format

long example(*foo, bar***) int** *foo*; **long** *bar*;

This is an example of the Mark Williams Lexicon format of software documentation. At this point, each entry has a brief narration that discusses the topic in detail.

The line in **boldface** above gives the usage of the function being described. The imaginary function is called **example**. **long** means that it returns a **long**; if no type is given, then assume that it returns an **int**; and if the function should return nothing, it will be given as type **void**. *foo* and *bar* are **example**'s arguments; *foo* must be declared as an **int**, and *bar* as a **long**.

Example

```
main() {
	printf("Many articles have an example.\n");
}
```

See Also

all other related topics and functions

Notes

If technical terms are used that you do not entirely understand, look them up in the Lexicon. In this way, you will gain a secure understanding of how to use Mark Williams C.

abort—General function (**libc.a/abort**)
End program immediately
abort()

abort terminates a process and prints a message on the screen. It is normally invoked in situations in a program that "should not happen". The termination is carried out by a call to **exit**, with a non-zero exit status.

See Also
exit, _exit

Diagnostics
abort prints the relative address from the beginning of the program, so that you can look the location up in the symbol table. See the entry for **nm** for more information on how to extract the symbol table from an executable program.

abs—General function (**libc.a/abs**)
Return the absolute value of an integer
abs(*n*) int *n*;

abs returns the absolute value of integer *n*.

Example

```
main(){
	extern char *gets();
	extern int atoi();
	char string[64];
	int input;
	for (;;) {
		printf("Enter an integer: ");
		if(gets(string)) {
			input = atoi(string);
			printf("abs(%d) is %d.\n", input, abs(input));
		}
		else break;
	}
}
```

See Also
fabs, floor, int

Notes
On two's complement machines, the **abs** of the most negative integer is itself.

acos—Mathematics function (**libm.a/acos**)
Calculate inverse cosine
#include <math.h>
double acos(*arg*) double *arg*;

acos calculates inverse cosine. *arg* should be in the range of [-1., 1.]; the result will be in the range [0, PI].

Example
This example demonstrates the mathematics functions **acos**, **asin**, **atan**, **atan2**, **cabs**, **cos**, **hypot**, **sin**, and **tan**.

```
#include <math.h>

dodisplay(value, name)
double value; char *name;

{
	if (errno)
		perror(name);
	else
		printf("%10g %s\n", value, name);
	errno = 0;
}

#define display(x) dodisplay((double)(x), "x")

main() {
	extern char *gets();
	double x;
	char string[64];

	for(;;) {
		printf("Enter number: ");
		if(gets(string) == 0)
			break;

		x = atof(string);
		display(x);
		display(cos(x));
		display(sin(x));
		display(tan(x));
		display(acos(cos(x)));

		display(asin(sin(x)));
		display(atan(tan(x)));
		display(atan2(sin(x),cos(x)));
		display(hypot(sin(x),cos(x)));
		display(cabs(sin(x),cos(x)));
	}
}
```

See Also
errno, errno.h, mathematics library, perror

Diagnostics
Out-of-range arguments set **errno** to **EDOM** and return 0.

address—Definition

An **address** is the location where an item of data is stored in memory. An address on the 68000 is simply a 24-bit integer that is stored as a 32-bit integer. On the 68000, the upper eight bits are ignored; this is not true with more advanced microprocessors in this family, such as the 68020. On machines with memory-mapped I/O, such as the 68000, some addresses may be used to control or communicate with peripheral devices. Thus, using an incorrect address as an argument to **poke** may accidentally disable a peripheral device.

See Also
peekb, peekl, peekw, pokeb, pokel, pokew

AES—Definition

AES stands for *application environment services*. It draws and manipulates predefined graphics elements, such as icons, pull-down menus, and windows. It is the highest level of GEM, and the one that a programmer will deal with most often.

AES consists of the following elements: a kernel, a screen manager, buffers, and a set of "libraries". Each is briefly described below.

The **kernel** performs rudimentary I/O and provides limited multi-tasking capability. It manipulates concurrently executing routines, or "processes", in the following manner. When a process has executed to the point where it makes a request from the kernel, it's placed on a "not ready" list, where it sleeps. When a "event" occurs that the program is awaiting (that is, when the user manipulates the mouse or types on the keyboard, when the system's timer signals that a certain amount of time has elapsed, or when a message is received from another process), the kernel moves the process from the not-ready list to the end of the "ready" list, and returns a description of the event to the process.

Note that each "event generator" (i.e., mouse, keyboard, and timer) has its own **buffer**, which ensures that no event is "dropped on the floor", or lost, while another is being processed.

The **screen manager** tracks the mouse pointer on the screen, and manages windows and menus. It signals when a mouse button is pressed with the mouse pointer fixed on a significant area of the screen (e.g., the work area in a window), returns a message when the user manipulates a window, and drops the appropriate menu when the pointer crosses into the menu bar at the top of the screen.

Finally, AES contains a number of sets, or "libraries", of functions that create and manipulate screen elements. These functions are accessed through the library **libaes.a**, and their bindings are carried in the file **aesbind.h**.

The following names each AES routine and briefly describes what it does.

appl_exit	tell the AES that the program is exiting
appl_find	get another application's handle
appl_init	initialize a new application
appl_read	read a message from another process
appl_tplay	replay recorded AES events
appl_trecord	record AES events
appl_write	send a message to another process
evnt_button	await a mouse button event
evnt_dclick	set/get double-clicking speed
evnt_keybd	await a keyboard event
evnt_mesag	await a message
evnt_mouse	wait for mouse to enter a rectangle
evnt_multi	await more than one event
evnt_timer	wait a given amount of time
form_alert	perform an alert dialogue
form_center	center dialogue box on screen
form_dial	reserve/release dialogue box
form_do	use dialogue box
form_error	display preset error box
fsel_input	display/run file selector box
graf_growbox	draw expanding box outline
graf_handle	return VDI handle
graf_mbox	draw moving box
graf_mkstate	return current mouse states
graf_mouse	change mouse pointer's shape
graf_rubbox	draw box that expands with mouse pointer
graf_shrinkbox	draw a shrinking outline
graf_slidebox	find center of box's "slider"
graf_watchbox	check if mouse pointer is within box
menu_bar	display/erase menu bar
menu_icheck	display/remove checks by menu items
menu_ienable	enable/disable menu items
menu_register	name desk accessory on desk menu
menu_text	change text of menu item
menu_tnormal	show menu title in normal/reverse video
objc_add	add an object to object tree
objc_change	change an object's state

objc_delete	delete object from object tree
objc_draw	draw an object
objc_edit	edit text within an object
objc_find	find if mouse is over an object
objc_order	change order of object within its tree
objc_set	compute object's location
rc_copy	copy a rectangle
rc_equal	compare two rectangles
rc_intersect	calculate overlap of rectangles
rc_union	combine rectangles
rsrc_free	free memory allocated to resource
rsrc_gaddr	get address of data structure
rsrc_load	load resource file into RAM
rsrc_obfix	convert character coordinates
rsrc_saddr	store index to data structure
scrp_read	find name of scrap directory
scrp_write	set name of scrap directory
shel_envrn	search for environmental variable
shel_find	find a file name
shel_read	return name of parent program
shel_write	invoke another program or exit from GEM
wind_calc	calculate window size
wind_close	close a window
wind_create	create a window
wind_delete	delete window
wind_find	find a window under mouse pointer
wind_get	get information about a window
wind_open	open a window
wind_set	set values for window
wind_update	inhibit/allow updates to windows

Each routine has its own entry within the Lexicon; its bindings are given, with a fuller description and, often, an example.

Programming the AES
Some graphics-based systems have been designed to automate as much work as possible. The AES is not such a system. In programming the AES, you must specifically guide each function each step of the way. This means that you must do more work, but it also means that you have fuller, and finer, control of the operation of the program. For example, program flow under an automated system often appears to function as follows:

```
action_occurs;
if (action was desired)
        good;
else
        tough_luck;
```

Programming under the AES more often resembles the following model:

```
information received   /* e.g., mouse moved, keyboard pressed */
parse information
if (information meets test) {
        pass_to_routine(information);
        perform_action();
        cleanup_debris();
        return;
} else if (information meets second test) {
        pass_to_otherroutine(information);
        take_alternative_action();
        cleanup_debris();
        return;
} else ignore();
```

The second model of programming obviously is harder to work with than the first. However, it has the advantage of protecting you from random, system-generated errors—or at least gives you the tools with which to work around such errors should they occur.

In programming for AES, note that each process must be declared to the AES through the function **appl_init**. This gives the process a handle, so it can be recognized and manipulated by the kernel, and notifies the AES that this program is a GEM application. When a process is finished, it is good practice to close it with the function **appl_exit**. This frees up AES structures allocated to the process, and ensures that the process terminates gracefully.

Note that not all C programs use the AES specifically. Programs that use only UNIX routines or STDIO need never worry about the AES. All programs that use the graphics interface, however, must run under the AES; this means that all programs that use the VDI must begin with **appl_init** and close with **appl_exit**.

The AES provides sophisticated routines to help draw windows and menus, and create graphics objects. See the entries for **window**, **menu**, and **object** for more details.

For information about compiling AES programs, see the entry for **TOS**.

See Also
aesbind.h, **gemdefs.h**, **libaes.a**, **libvdi.a**, **menu**, **object**, **TOS**, **window**

Notes
The AES binding library uses the object file **crystal.o** to access the AES services. A program should *never* call this function directly; it is automatically linked with **libaes.a**. You should never name a function or a global variable **crystal** if your program uses the AES.

Note that both the AES and the VDI use trap 2 to access the services.

aesbind.h—Header file
aesbind.h is the header file that declares the the GEM AES routines contained in the library **libaes.a**, and shows a sample call for each. It also defines the following structures:

Rect Describe a rectangle by its x, y coordinates and its width and height. This structure is called **GRECT** in the header file **obdefs.h**, and is described as follows:

```
typedef struct { int x, y, w, h; } Rect;
```

Mouse Pass pointers to the x, y coordinates, mouse button state, and keyboard state:

```
typedef struct { int *x, *y, *b, *k; } Mouse;
```

Prect Pass pointers to the x, y coordinates, width, and height of a rectangle:

```
typedef struct { int *x, *y, *w, *h; } Prect;
```

See Also
AES, header file, TOS

alignment—Definition
Alignment refers to the fact that the address of a data entity must be *aligned* on a certain numeric boundary in memory, such that *address* modulo *number* equals zero. For example, on 68000 and the PDP-11, an integer must be aligned along an even address, i.e., *address*%2==0. Generally speaking, alignment is a problem only if you write programs in assembly language; for C programs, Mark Williams C will ensure that data types are aligned properly under most foreseeable conditions.

Alignment may be a problem when porting programs to the VAX. On this computer, certain data types have quad-word boundaries, and exceeding these boundaries can mean a significant penalty in the speed with which programs execute.

Processors react differently to alignment problems; an alignment problem on the VAX or the 8086 causes programs to run more slowly, whereas on the 68000 they cause bus errors.

See Also
data types, declarations

appl_exit—AES function (**libaes.a/appl_exit**)
Exit from an application
#include <aesbind.h>
int appl_exit()

appl_exit is an AES routine that notified the AES that the program no longer requires its services. It frees up the AES structures and the handle associated with the process. It does not terminate program execution.

appl_exit returns zero if an error occurred, and a number greater than zero if one did not.

Example
For examples of how to use this routine, see the entries for **evnt_multi** and **window**.

See Also
AES, appl_init, TOS

appl_find—AES function (**libaes.a/appl_find**)
Get the ID of another application
#include <aesbind.h>
int appl_find(*filename*) char *_filename_;

appl_find is an AES routine that fetches the handle of another application. *filename* is the name of the file in which the application is stored; it cannot be longer than eight characters. **appl_find** returns the handle if it is found, and -1 if an error occurred.

See Also
AES, TOS

appl_init—AES function (**libaes.a/appl_init**)
Initiate an application
#include <aesbind.h>
int appl_init()

appl_init is an AES routine that declares an application. It registers the application with AES, and initializes all resources used by the the application. It returns the application's handle if all went well, and -1 if an error occurred.

Example
For an example of this routine, see the entries for **evnt_multi**, **menu**, **object**, and **window**.

See Also
AES, appl_exit, TOS

appl_read—AES function (**libaes.a/appl_read**)
Read a message from another application
#include <aesbind.h>
int appl_read(*handle, length, buffer***) int** *handle, length*; **char** **buffer*;

appl_read is an AES routine that reads a message from another application. *handle* is the AES handle of the application that owns the pipe to be read, and *length* is the number of bytes to read from the pipe. *buffer* is the place into which the message is written. It returns zero if an error occurred, and a number greater than zero if one did not.

Note that this routine is used only in specialized programming situations, to receive messages sent with the routine **appl_write**. Normally, an application receives messages by using the routines or **evnt_multi**.

See Also
AES, appl_write, evnt_mesag, TOS

appl_tplay—AES function (**libaes.a/appl_tplay**)
Replay AES activity
#include <aesbind.h>
int appl_tplay(*buffer, number, speed***) char** **buffer*; **int** *number, speed*;

appl_tplay is an AES routine that replays a set of AES events. These events must be recorded with the function **appl_trecord**. *buffer* is the name of the buffer in which the actions are stored. *number* is the number of actions that you wish to replay, and *speed* is a number from one to 10,000 that indicates how fast the actions should be replayed. **appl_tplay** always returns one.

See Also
AES, appl_trecord, TOS

appl_trecord—AES function (**libaes.a/appl_trecord**)
Record user actions
#include <aesbind.h>
int appl_trecord(*buffer, capacity***) char** **buffer*; **int** *capacity*;

appl_trecord is an AES routine that records a user's AES actions. Each recorded action requires an **int** and a **long**'s worth of storage. The **int** indicates the type of event being recorded, as follows:

0	timer event
1	mouse button event
2	mouse event
3	keyboard event

The **long** can hold a variety of information, depending on the type of event being recorded, as follows:

timer	milliseconds elapsed
button	low word: state (0=up, 1=down)
	high word: number of clicks
mouse	low word: X coordinate
	high word: Y coordinate
keyboard	low word: character typed
	high word: keyboard state

buffer is the buffer into which the user's actions are recorded. *capacity* is the number of events that can be stored. This should equal the amount of storage available to *buffer*, divided by six (the number of bytes used by each event).

appl_trecord returns the number of events actually recorded. These events can be replayed with the function **appl_tplay**.

See Also
AES, **appl_tplay**, TOS

appl_write—AES function (**libaes.a/appl_write**)
Send a message to another application
#include <aesbind.h>
int appl_write(*handle, length, buffer***) int** *handle, length*; **char** **buffer*;

appl_write is an AES routine that sends a message to another application. *handle* is the handle of the application to which the message is being sent, and *length* is the length of the message, in bytes. *buffer* gives the address where you write your message. **appl_write** returns zero if an error occurred, and a number greater than zero if one did not.

Note that this routine is used only in specialized programming situations. The target application must use the routine **appl_read** to receive messages sent via **appl_write**.

See Also
AES, **appl_read**, TOS

ar—Command

The librarian/archiver

ar *option* [*modifier*][*position*] *archive* [*member* ...]

The librarian **ar** edits and examines libraries. It combines several files into a file called an *archive* or *library*. Archives reduce the size of directories and allow many files to be handled as a single unit. The principal use of archives is for libraries of object files. The linker **ld** understands the archive format, and can search libraries of object files to resolve undefined references in a program.

The mandatory *option* argument consists of one of the following command keys:

d Delete each given *member* from *archive*. The ranlib header is updated if present.

m Move each given *member* within *archive*. If no *modifier* is given, move each *member* to the end. The ranlib header is modified if present.

p Print each *member*. This is useful only with archives of text files.

q Quick append: append each *member* to the end of *archive* unconditionally. The ranlib header is *not* updated.

r Replace each *member* of *archive*. The optional *modifier* specifies how to perform the replacement, as described below. The ranlib header is modified if present.

t Print a table of contents that lists each *member* specified. If none is given, list all in *archive*. The modifier **v** tells **ar** to give you additional information.

x Extract each given *member* and place it into the current directory. If none is specified, extract all members. *archive* is not changed.

The *modifier* may be one of the following. The modifiers **a**, **b**, **i**, and **u** may be used only with the **m** and **r** options.

a If *member* does not exist in *archive*, insert it after the member named by the given *position*.

b If *member* does not exist in *archive*, insert it before the member named by the given *position*.

c Suppress the message normally printed when **ar** creates an archive.

i If *member* does not exist in *archive*, insert it before the member named by the given *position*. This is the same as the **b** modifier, described above.

k Preserve the modify time of a file. This modifier is useful only with the **r**, **q**, and **x** options.

s Modify an archive's ranlib header, or create it if it does not exist. This is used only with the **r**, **m**, and **d** options.

u Update *archive* only if *member* is newer than the version in the *archive*.

v Generate verbose messages.

All archives are written into a specialized file format. Each archive starts with a "magic number" called **ARMAG**, which identifies the file as an archive. The members of the archive follow the magic number; each is preceded by an **ar_hdr** structure, as follows:

```
#define DIRSIZ 14
#define ARMAG 0177535                /* magic number */
struct ar_hdr {
        char ar_name[DIRSIZ];        /* member name */
        time_t ar_date;              /* time inserted */
        short ar_gid;                /* group owner */
        short ar_uid;                /* user owner */
        short ar_mode;               /* file mode */
        size_t ar_size;              /* file size */
};
```

The structure at the head of each member is followed the data of the file, which occupy the number of bytes specified by the variable **ar_size**.

See Also
commands, ld, nm, ranlib

Notes
It is recommended that each object-file library you create with **ar** have a name that begins with the string **lib**. This will allow you to call that library with the -l option to the **cc** command.

Note that **ar** now adjusts the time file in the **ranlib** header so that out-of-date ranlib headers are now dated in 1970, and up-to-date **ranlib** headers are dated a decade into the future. This should eliminate improper **outdated ranlib** error messages from the linker.

arena—Definition

Mark Williams C uses an arena, rather than a heap, for allocation of dynamic memory. An **arena** is the area of memory that is available for a program to allocate dynamically at run time. It consists of an area of memory that is divided into *allocated* and *unallocated* blocks. The unallocated blocks together form the "free memory pool".

Portions of the arena can be allocated using the functions **malloc**, **calloc**, or **realloc**; returned to the free memory pool with **free**; or checked to see if they are allocated or not with **notmem**.

See Also
calloc, free, malloc, notmem, realloc

argc—Definition
Argument passed to **main**
int argc;

argc is an abbreviation for **argument count**. It is the traditional name for the first argument to a C program's **main** routine. By convention, it holds the number of arguments that are passed to **main** in the argument vector **argv**. Note that because **argv[0]** is always the name of the command, the value of *argc* is always one greater than the number of command-line arguments that the user enters.

Example
For an example of how to use **argc**, see the entry for **argv**.

See Also
argv, main
The C Programming Language, page 110

argv—Definition
Argument passed to **main**
char *argv[];

argv is an abbreviation for **argument vector**. It is the traditional name for a pointer to an array of string pointers passed to a C program's **main** function, and is by convention the second argument passed to **main**. Note that by convention, **argv[0]** always points to the name of the command itself.

Under the draft ANSI standard for the C language, the default arguments to a C program are **int argc** and **char *argv[]**. Mark Williams C passes these arguments and looks for them in two different ways: in the command tail of the basepage sructure, and in the environment.

Why a different convention?
TOS allows programs to be run in a number of different ways: under a shell, from the desktop with arguments (**.ttp**), or from the desktop without arguments (**.tos**, **.prg**). The Mark Williams conventions for passing argument are designed to increase run-time flexibility; programs compiled under Mark Williams C should run transparently from the shell, or from the desktop, using every possible run-time environment.

Using the environment to pass parameters also has the advantage of lifting the limit on the number or size of arguments that can be passed; it also has the advantage of not mapping all of the arguments to upper case.

TOS conventions
The current TOS convention for passing arguments is to pass up to 127 characters in the command tail of the **Pexec** command. If the tail is parsed by the desktop, it will be limited to 40 upper-case characters.

Mark Williams convention
The Mark Williams convention is first to parse the argument into words, then pass the words within the **Pexec** environment. Within the **Pexec** environment, the arguments begin immediately after the environmental variable **ARGV** and continue to the end of the environment. The arguments may contain any ASCII character except NUL, which is used to terminate both individual arguments and the **Pexec** environment as a whole.

The Mark Williams library function **execve** executes a given *command* with a specifed *argv* and *envp*. It copies *envp* into an allocated buffer, appends the string ARGV=*iovector* environment, and then appends the strings to the array to which *argv* points. This concatentation of strings, which is terminated by an empty string, becomes the *environment* passed to **Pexec**. In another part of the allocated buffer, **execve** concatenates up to 127 characters, starting with *argv[1]* and continuing through *argv[ ]*, separating the arguments with spaces. This concatenation of strings, which is prefixed by a count, becomes the *command tail* passed to **Pexec**. When **execve** now calls **Pexec(0,** *command*, *command_tail*, *environment***)**, TOS copies *environment* into a newly allocated buffer, copies *command tail* into the newly allocated basepage, loads *command*, and executes it.

Summary
Mark Williams C puts the arguments into the environment so that programs that use the the Mark Williams run-time start-up routine, **crts0.o**, will find them there. It puts them into the command tail, so that programs that use the .ttp-style run-time start-up (**crtsg.o**) will find them there.

The Mark Williams run-time start-up module, **crts0.o**, looks for arguments in the environment. If it finds them there, it uses them. If no arguments were found in the environment, **crts0.o** assumes that it was started from the desktop or a TOS convention command line interpreter, so it looks in the command tail and parses the contents into arguments that are delimited by space and tab characters.

Mark Williams C looks for arguments in the environment because a command may need more arguments than can be fit into the 40-character command tail available when a program is run with the **.ttp** feature. In addition, a command (e.g., **egrep**) can take arguments that contain literal spaces or tabs; these would be interpreted to be word separators if arguments were passed simply through the command tail.

As a last resort, Mark Williams C also looks for arguments in the command tail, because 40 characters mapped to upper case are better than nothing.

The **execve** function passes arguments in both the environment and the command tail, and the run-time start-up routine **crts0.o** takes arguments from the command tail if the environment has none. Mark Williams C uses both conventions in both places to allow as many programs to work in as many environments as possible.

Example
This example demonstrates both **argc** and **argv[]**, to create the command **echo**. For another example of **argc**, see the entry for **basepage**.

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

	for (i = 1; i < argc; ) {
		printf("%s", argv[i]);
		if (++i < argc)
			putchar(' ');
	}

	putchar('\n');
	return 0;
}
```

See Also
argc, crts0.o, crtsd.o, crtsg.o, main, Pexec
The C Programming Language, page 110

array—Definition

An **array** is a collection of data elements of the same type or structure, which are stored in consecutive memory and which share the same name but are differentiated by a subscript. For example, the array **foo[3]** has three elements: **foo[0]**, **foo[1]**, and **foo[2]**.

Note that the numbering of elements within an array always begins with '0'.

Arrays, like other data elements, may be automatic (**auto**), **static**, or external (**extern**).

Arrays can be multi-dimensional; that is to say, each element in an array can itself be an array. To declare a multi-dimensional array, use more than one set of square brackets. For example, the multi-dimensional array **foo[3][10]** is a two-dimensional array that has three elements, each of which is an array of ten elements. Note that the second sub-script is always necessary in a multi-dimensional array, whereas the first is not. For example, **foo[][10]** is acceptable, whereas **foo[10][]** is not; the first form is an indefinite number of ten-element arrays, which is correct C, whereas the second form is ten copies of an indefinite number of elements, which is illegal.

The C Programming Language, page 83, forbids the initialization of automatic arrays. Mark Williams C lifts this restriction. It allows you to initialize automatic arrays and structures, provided that you know the size of the array, or of any array contained within a structure. The initialization has the same form as that of the external aggregate, but is performed on entry to the routine instead of at compile time. Note, however, that because this feature is not defined as part of the language, its use will limit the portability of your program.

See Also

declarations, flexible array, struct

The C Programming Language, pages 25, 83, 210

as—Command

Mark Williams assembler

as [-glx] [-o *outfile*] *file ...*

as is the Mark Williams assembler. It consists of one program, called **as**, which turns files of assembly language into relocatable object modules, similar to those produced by the C compiler. Relocatable object modules produced by the assembler and the compiler are of the same format.

as is a multipass assembler for writing small subroutines in assembly language. Because it is not intended to be used for full-scale assembly-language programming, it lacks many of the more elaborate facilities of full-fledged assemblers, such as conditional compilation or user-defined macros. However, **as** does optimize span-dependent instructions.

Usage

Normally, the assembler **as** is invoked automatically by **cc** to assemble programs with a suffix of **.s**. However, you can invoke **as** directly from the shell **msh**, by using the following command:

as [-glx] [-o *outfile*] *file ...*

The following describes the available options:

-o Write the assembled executable into *outfile*. The default is **l.out**.

-g Give all symbols that are undefined at the end of the first pass the type undefined external, as though they had been declared with a **.globl** directive.

-l Generate a listing on the standard output.

-x Strip all non-global symbols that begin with the character 'L' from the symbol table of the object module. This speeds the linking of files by removing compiler-generated labels from the symbol table.

Lexical conventions
Assembler tokens consist of identifiers (also known as "symbols" or "names"), constants, and operators.

An *identifier* is a string of alphanumeric characters, including the period '.' and the underscore '_'. The first character must not be numeric. Only the first 16 characters of the name are significant; the rest are thrown away. Upper case and lower case are different. The machine instructions, assembly directives, and symbols that are used frequently are in lower case.

Numeric constants are defined by the assembler by using the same syntax as the C compiler: a sequence of digits that begins with a zero '0' is an octal constant; a sequence of digits with a leading '0x' is a hexadecimal constant ('A' through 'F' have the decimal values 10 through 15); and any strings of digits that do not begin with '0' are interpreted as decimal constants.

A character constant consists of an apostrophe followed by an ASCII character. The constant's value is the ASCII code for the character, right-justified in the machine word.

A blank space can be represented either as **0x20** (its ASCII value in hexadecimal), or as an apostrophe followed by a space ('), which on paper looks like just an apostrophe alone.

The following gives the multi-character escape sequences that can be used in a character constant to represent special characters:

\b	Backspace	(0010)
\f	Formfeed	(0014)
\n	Newline	(0012)
\r	Carriage return	(0015)
\t	Tab	(0011)
\v	Vertical tab	(0013)
nnn	Octal value	(0*nnn*)

Spaces and tab characters can be used freely between tokens, but not within identifiers. A space or a tab character must separate adjacent tokens not otherwise separated, e.g., an instruction opcode and its first operand.

Masks
as accepts a register mask syntax for the **movem** instruction. The syntax is as follows:

```
movem       $<rmask>,-(<an>)
movem       $<fmask>,<adr>
movem       <adr>,$<fmask>
movem       (<an>)+,$<fmask>
```

The abbreviations between angle brackets '<' '>' mean the following:

<an> The registers a0 through a7.

<adr> The effective address (not register direct), i.e., the location of the address.

<rmask> (reverse mask) This can be either a word whose bits show which registers to save, with bit 0 indicating register a7 to bit 15 indicating register d0; or a list of the registers to save, enclosed in braces '{' '}'.

<fmask> (forward mask) This, too, is either a word whose bits show which registers to save or restore, with bit 0 indicating register d0 through bit 15 indicating register a7; or a list of these registers enclosed in braces.

Note that if the {*list*} variety of mask is used, the assembler automatically produces a consistent value for all addressing modes (bits backward for destination, minus the contents of register a*N*). If a word value is used, the bits are not modified. Thus:

```
movem.l     ${d2-d7,a2-a5},-(sp)
movem.l     (sp)+,${d2-d7,a2-a5}
```

produces the same code as:

```
movem.l     $0x3F3C,-(sp)
movem.l     (sp)+,$0x3CFC
```

Note, too, that ranges that include both register sets are allowed; thus

```
movem.l     ${d0-a5},4(a5)
```

will save d0 through a5. The instruction

```
movem.l     ${a5-d0},4(a5)
```

does the same thing. Likewise,

```
movem.l     ${d2,d3-d5,a3,a5-a7},-(sp)
```

results in code that saves d2, d3 through d5, a3, and a5 through a7. The instruction

```
movem.l     ${d0},-(sp)
```

saves d0.

Comments
Comments are introduced by a slash ('/') and continue to the end of the line. The assembler ignores all comments.

Program sections
The assembler permits the division of programs into a number of sections, each corresponding (roughly) to a functional area of the address space. Each program section has its own location counter during assembly. The eight

program sections are subdivided into three groups that contain code and data, as follows:

shared:	**shri**	shared instruction
	shrd	shared data
private:	**prvi**	private instruction
	prvd	private data
uninitialized:	**bssi**	uninitialized instruction
	bssd	uninitialized data
	strn	strings

All Mark Williams assemblers use the same set of sections; this increases the portability of programs among operating systems. In most instances, the programmer need not worry about what all of the program sections are, and can simply write code under the keywords **.prvi** or **.shri**, and write data under the keywords **.prvd** or **.shrd**. At the end of assembly, the sections of a program are concatenated so that within the assembly listing the program looks like a contiguous block of code and data.

The current location
The special symbol '.' (period) is a counter that represents the current location. The current location can be changed by an assignment; for example:

```
. = .+START
```

The assignment must not cause the value to decrease and it must not change the program section, i.e., the right-hand operand must be defined in the same section as is the current section.

Expressions
An expression is a sequence of symbols that represent a value and a program section. Expressions are made up of identifiers, constants, operators, and brackets. All binary operators have equal precedence and are executed in a strict left-to-right order, unless altered by brackets. Note that square brackets, '[' and ']', are used to group the elements of expression, because parentheses are used for addressing indexed registers.

Types
Every expression has a *type*, which is determined by that expression's *operands*. The simplest operands are *symbols*, which yield the following types:

undefined A symbol is defined if it is a *constant* or a *label*, or when it is assigned a defined value; otherwise, it is undefined. A symbol may become undefined if it is assigned the value of an undefined expression. It is an error to assemble an undefined expression in pass 2. With option fB-gfR, pass 1 allows assembly of undefined expressions, but phase errors may be produced if undefined expressions are used in certain contexts, such as in a **.blkw** or **.blkb**.

absolute	An absolute symbol is one defined ultimately from a constant or from the difference of two relocatable values of the same type.
register	The machine registers.
Relocatable	All other user symbols are either defined labels (in a program section) or externals. These are relocated at link time. Every user program section and external symbol defines a unique type class.

Each keyword in the assembler has a secret type that identifies it internally; however, all secret types are converted to an absolute constant in expressions. Thus, any keyword can be used in an expression to obtain the basic value of the keyword.

Note that the type of an expression does not include such attributes as length, so the assembler will not remember whether a particular variable was defined as a word or a byte. Addresses and constants have different types, but the assembler does not treat a constant as an immediate value unless it is preceded by a dollar sign '$'. If a constant is used where an address is expected, the constant will be treated like an address (and vice versa). The programmer must distinguish between variables and addresses or immediate values.

Operators
The following table shows various characters interpreted as operators in expressions.

+	Addition
-	Subtraction
*	Multiplication
-	Unary negation
~	Unary complement
^	Type transfer (cast)
|	Segment construction

Type propagation
When operands are combined within expressions, the resulting type is a function of both the operator and the types of the operands. The '*', '~', and unary '-' operators can manipulate only absolute operands and always yield an absolute result.

The '+' operator signifies the addition of two absolute operands to yield an absolute result, and the addition of an absolute to a relocatable operand to yield a result with the same type as the relocatable operand.

The binary '-' operator allows two operands of the same type, including relocatable, to be subtracted to yield an absolute result; it also allows an absolute to be subtracted from a relocatable, to yield a result with the same type as the relocatable operand.

The binary '^' operator yields a result with the value of its left operand and the type of its right operand. It may be used to create expressions (usually intended to be used in an assignment statement) with any desired type.

Statements
A program consists of a sequence of statements separated by newlines or by semicolons. There are four kinds of statements: null statements, assignment statements, keyword statements, and machine instructions.

Any statement may be preceded by any number of labels. There are two kinds of labels: *name* and *temporary*.

A name label consists of an identifier followed by a colon (':'). The program section and value of the label are set to that of the current location counter. It is an error for the value of a label to change during an assembly. This most often happens when an undefined symbol is used to control a location counter adjustment.

A temporary label consists of a digit ('0' through '9') followed by a colon (':'). Such a label defines temporary symbols of the form *xf* and *xb*, where *x* is the digit of the label. References of the form *xf* refer to the first temporary label *x*: forward from the reference; those of the form *xb* refer to the first temporary label *x*: back from the reference. Such labels conserve symbol table space in the assembler.

A null statement is an empty line, or a line that contain only labels or a comment. Null statements can occur anywhere. They are ignored by the assembler, except that any labels are given the current value of the location counter.

Note that the programmer is responsible for proper alignment of data. See the entry on **alignment** for more information.

Assignment statements
An assignment statement consists of an identifier that is followed by an equal sign '=' and an expression. The value and type of the identifier are set to those of the expression. Any symbol that is defined by an assignment statement may be redefined, either by another assignment statement or by a label. An assignment statement is equivalent to the **equ** keyword statement found in many assemblers.

Assembler directives
Assembler directives give instructions to the assembler. Each directive keyword begins with a period, and some are followed by operands.

Changing the current program section
These directives change the current program section to the named section.

.bssd **.shrd**
.bssi **.shri**
.prvd **.strn**
.prvi

The current location counter is set to the highest previous value of the location counter for the selected section.

.ascii *string*
In this directive, the first non-whitespace character, typically a quotation mark, after the keyword is taken as a delimiter. Successive characters from the string are assembled into successive bytes until this delimiter is again encountered. To include a quotation in a string, use some other character for the delimiter.

It is an error for a newline to be encountered before reaching the final delimiter. The multi-character escape sequences that are described above in the subsection *Constants* may be used in the string to represent newlines and other special characters.

.blkb *expression*
This directive assembles blocks that are filled with zeros. The size of the block is *expression* bytes.

.blkl *expression*
This directive assembles blocks that are filled with zeros. The size of the block is *expression* longs.

.blkw *expression*
This directive assembles blocks that are filled with zeros. The size of the block is *expression* words.

.byte *expression* [, *expression*]
Here, the *expressions* in the list are truncated to byte size and assembled into successive bytes. Expressions in the list are separated by commas.

.even The directives **.even** and **.odd** force alignment by inserting NUL, if necessary, to set the location counter to the next even or odd location, respectively.

.globl *identifier* [, *identifier*]
Here, the identifiers separated by commas are marked as global. If they are defined in the current assembly, they may be referenced by other object modules; if they are undefined, they must be resolved by the linker before execution.

.long *expression* [, *expression*]
In this directive, the *expressions* in the list are truncated to **long** and the resulting data are assembled into successive **longs**. Expressions in

the list are separated by commas.

.page This causes the assembly listing to skip to the top of a new page by inserting a form-feed character into the file. The title is printed at the top of the page.

.title *string*
Here, *string* appears on the top of every page in the assembly listing. This directive also causes the listing to skip to a new page.

.odd The directives **.even** and **.odd** force alignment by inserting NUL, if necessary, to set the location counter to the next even or odd location, respectively.

.globl *identifier* [, *identifier*]

.word *expression* [, *expression*]
The *expressions* in this list are truncated to word size and the resulting data are assembled into successive words. Expressions in the list are separated by commas.

Conventions
C compiler conventions, naming conventions, function calling conventions, the management of arguments, and return values are all described in detail in the Lexicon entry for **calling conventions.**

68000 register names
The assembler for the Motorola 68000 microprocessor uses a subset of the machine opcodes and register names provided by the manufacturer's assembler. All unsupported names are longer synonyms for names that are supported. Assembler directives, statement syntax, and expression syntax are different.

The following register names are predefined. In general, length of operation is specified by opcode. The **-l** suffixes are used only in indexed addressing to differentiate 16-bit and 32-bit indices.

16-bit	*32-bit*
usp	sp
ccr	pc
sr	d0.l
d0	d1.l
d1	d2.l
d2	d3.l
d3	d4.l
d4	d5.l
d5	d6.l
d6	d7.l
d7	a0.l
a0	a1.l
a1	a2.l

a2	a3.l
a3	a4.l
a4	a5.l
a5	a6.l
a6	a7.l
a7	sp.l

Address descriptors
The following syntax is used for general source and destination address descriptors. The syntax is a subset of that used by Motorola assemblers, except that the character '$' is used to specify immediate data, and that the suffix :s appended to an absolute address forces absolute short addressing. Note that short address modes are *not* supported by the TOS system executable format.

In the examples, the symbols **a**, **d**, and **r** refer to address, data, and any register, respectively, and the symbol 'e' refers to any expression.

dn	Data register direct
an	Address register direct
(a)	Address register indirect
(a)+	Address register postincrement
-(a)	Address register predecrement
e(a)	Address register displacement
e(a,r)	Address register short index
e(a,r.l)	Address register long index
e:s	Absolute short address
e	Absolute long address
e(pc)	Program counter displacement
e(pc,r)	Program counter short index
e(pc,r.l)	Program counter long index
$e	Immediate data
l	Label

ea represents the effective address of any data address. **an** indicates any register from a0 to a7; **dn**, any register from d0 to d7.

The addressing modes are classified into four categories that are used in the instruction listings to distinguish allowed addresses:

* Data addresses are all addresses except address registers.

* Memory addresses are all addresses except data and address registers.

* Control addresses are all memory addresses, except address register predecrement and address register postincrement.

* Alterable addresses are all addresses except program counter displacement, program counter index, and immediate.

Failure to observe category restrictions will generate address errors.

Machine instructions
The following machine instructions are defined. For the most part, they form a subset of the instructions provided by Motorola assemblers that eliminates long synonyms such as **bsr.l** or **add.w**. The conditions **hs** (higher or same) and **lo** (lower) are provided as synonyms for **cc** (carry clear) and **cs** (carry set).

In the examples **an**, **dn**, and **rn** refer to address, data, and registers, **ea** refers to general effective addresses, **l** refers to direct addresses, **e** refers to a general expression, and **n** refers to an absolute expression.

Many syntactically correct instructions may prove to have semantic errors because of restrictions of effective addresses to data, alterable, memory, or control categories. Contrary to appearances, no 68000 instruction operates on all addressing modes; some modes are always forbidden. These restrictions are noted at the end of each instruction description in the 68000 user's manual. In the following listing, instructions have been classified according to their allowed addressing modes. Each classification is named by the lexicographically first instruction in the class.

ABCD Type: These instructions accept only two kinds of operands: data register direct and address register predecrement. The BCD instructions operate on byte size operands only.

```
abcd    dn,dn
abcd    -(an),-(an)

abcd    C100
addx    D140
addx.b  D100
addx.l  D180
sbcd    8100
subx    9140
subx.b  9100
subx.l  9180
```

ADD Type: These instructions take a data-register source to a memory-alterable destination or any source to a data-register destination. If the operation size is byte, then address-register direct sources are forbidden.

```
add     dn,ea
add     ea,dn

add     D040
add.b   D000
add.l   D080
sub     9040
sub.b   9000
sub.l   9080
```

ADDA Type: These instructions accept any source effective address. The **cmp**

instruction cannot combine byte operations with address-register sources.

adda	ea,an	D0C0
adda.l	ea,an	D1C0
cmp	ea,dn	B040
cmp.b	ea,dn	B000
cmp.l	ea,dn	B080
cmpa	ea,an	B0C0
cmpa.l	ea,an	B1C0
movea	ea,an	3040
movea.l	ea,an	2040
suba	ea,an	90C0
suba.l	ea,an	91C0

ADDI Type: These instructions require a data-alterable destination-effective address. The **nbcd** instruction, set according to condition, and the **tas** instructions are implicitly byte sized.

addi	$n,ea	0640
addi.b	$n,ea	0600
addi.l	$n,ea	0680
clr	ea	4240
clr.b	ea	4200
clr.l	ea	4280
cmpi	$n,ea	0C40
cmpi.b	$n,ea	0C00
cmpi.l	$n,ea	0C80
eor	dn,ea	B140
eor.b	dn,ea	B100
eor.l	dn,ea	B180
nbcd	ea	4800
neg	ea	4440
neg.b	ea	4400
neg.l	ea	4480
negx	ea	4040
negx.b	ea	4000
negx.l	ea	4080
not	ea	4640
not.b	ea	4600
not.l	ea	4680
scc	ea	54C0
scs	ea	55C0
seq	ea	57C0
sf	ea	51C0
sge	ea	5CC0
sgt	ea	5EC0
shi	ea	52C0
shs	ea	54C0

sle	ea	5FC0
slo	ea	55C0
sls	ea	53C0
slt	ea	5DC0
smi	ea	5BC0
sne	ea	56C0
spl	ea	5AC0
st	ea	50C0
subi	$n,ea	0440
subi.b	$n,ea	0400
subi.l	$n,ea	0480
svc	ea	58C0
svs	ea	59C0
tas	ea	4AC0
tst	ea	4A40
tst.b	ea	4A00
tst.l	ea	4A80

ADDQ Type: These instructions take an immediate-source operand in the range 1 to 8 and an alterable effective-address destination operand. If the operation size is byte, then address-register direct destinations are forbidden.

addq	$n,ea	5040
addq.b	$n,ea	5000
addq.l	$n,ea	5080
subq	$n,ea	5140
subq.b	$n,ea	5100
subq.l	$n,ea	5180

AND Type: These instructions take two forms: data register direct source to memory-alterable destinations, and data source effective address to a data register direct destination.

and	dn,ea
and	ea,dn

and	C040
and.b	C000
and.l	C080
or	8040
or.b	8000
or.l	8080

ANDI Type: These instructions combine an immediate source operand with either a data-alterable effective address destination operand or the status register. The whole status register or only the low byte is selected, depending on whether the operation size is word or byte.

andi	$n,ea
andi	$n,sr

andi	0240
andi.b	0200
andi.l	0280
eori	0A40
eori.b	0A00
eori.l	0A80
ori	0040
ori.b	0000
ori.l	0080

ASL Type: The shift instructions come in three flavors: immediate shift count of data register, data register shift count of data register, and shift by one of a word at a memory-alterable effective address. The memory shift opcode is formed from the opcodes given by setting bits 6-7, and by moving bits 3-4 to positions 9-10.

asl	$n,dn
asl	dn,dn
asl	ea

asl	E140
asl.b	E100
asl.l	E180
asr	E040
asr.b	E000
asr.l	E080
lsl	E148
lsl.b	E108
lsl.l	E188
lsr	E048
lsr.b	E008
lsr.l	E088
rol	E158
rol.b	E118
rol.l	E198
ror	E058
ror.b	E018
ror.l	E098
roxl	E150
roxl.b	E110
roxl.l	E190
roxr	E050
roxr.b	E010
roxr.l	E090

BCHG Type: The bit instructions take an immediate or data register source operand and a data-alterable destination effective address. The operation size is implicitly **long** for data register destinations and implicitly byte for other destinations.

bchg	$n,ea	
bchg	dn,ea	
bchg	0140	
bclr	0180	
bset	01C0	
btst	0100	

CHK Type: These instructions take a data-source effective address and a data-register destination. Source and destination are implicitly word-sized for **chk**, **muls**, and **mulu**. Source is word sized, and destination is **long** for **divs** and **divu**.

chk	ea,dn	4180
divs	ea,dn	81C0
divu	ea,dn	80C0
muls	ea,dn	C1C0
mulu	ea,dn	C0C0

JMP Type: These instructions require control-effective addresses.

jmp	ea	4EC0
jsr	ea	4E80
lea	ea,an	41C0
pea	ea	4840

MOVE Type: Move instructions take any source effective address to data-alterable destination effective addresses, but byte moves from address registers are forbidden. When the destination is the condition-code or status register, the source must be a data effective address and the instruction size is implicitly byte or word respectively. When the status register is the source the destination must be a data-alterable effective address. When the user stack pointer is an operand, the other operand is an address register and the instruction size is implicitly **long**.

move	ea,ea	3000
move.b	ea,ea	1000
move.l	ea,ea	2000
move	ea,ccr	44C0
move	ea,sr	46C0
move	sr,ea	40C0
move	an,usp	4E60
move	usp,an	4E68

MOVEM Type: These instructions take two forms: an immediate-register mask source with a control or predecrement destination, or a control or postincrement source with an immediate-register mask destination. The bit ordering in register masks is the programmer's responsibility.

movem	$n,ea	4880
movem	ea,$n	4C80
movem.l	$n,ea	48C0
movem.l	ea,$n	4CC0

MOVEP Type: The move-peripheral instruction uses data register and address register indirect with displacement operands.

movep	e(an),dn	0108
movep	dn,e(an)	0188
movep.l	e(an),dn	0148
movep.l	dn,e(an)	01C8

Miscellaneous Instructions: the remaining instructions have operand syntax which is self explanatory. Mnemonics with ".s" are short displacements, within +127 or -128 bytes (*not* words).

bcc	l	6400
bcc.s	l	6400
bcs	l	6500
bcs.s	l	6500
beq	l	6700
beq.s	l	6700
bge	l	6C00
bge.s	l	6C00
bgt	l	6E00
bgt.s	l	6E00
bhi	l	6200
bhi.s	l	6200
bhs	l	6400
bhs.s	l	6400
ble	l	6F00
ble.s	l	6F00

blo	l	6500
blo.s	l	6500
bls	l	6300
bls.s	l	6300
blt	l	6D00
blt.s	l	6D00
bmi	l	6B00
bmi.s	l	6B00
bne	l	6600
bne.s	l	6600
bpl	l	6A00
bpl.s	l	6A00
bra	l	6000
bra.s	l	6000
bsr	l	6100
bsr.s	l	6100
bvc	l	6800
bvc.s	l	6800
bvs	l	6900
bvs.s	l	6900
cmpm	(an)+,(an)+	B148
cmpm.b	(an)+,(an)+	B108
cmpm.l	(an)+,(an)+	B188
dbcc	dn,l	54C8
dbcs	dn,l	55C8
dbeq	dn,l	57C8
dbf	dn,l	51C8
dbge	dn,l	5CC8
dbgt	dn,l	5EC8
dbhi	dn,l	52C8
dbhs	dn,l	54C8
dble	dn,l	5FC8
dblo	dn,l	55C8
dbls	dn,l	53C8
dblt	dn,l	5DC8
dbmi	dn,l	5BC8
dbne	dn,l	56C8
dbpl	dn,l	5AC8
dbra	dn,l	50C8
dbt	dn,l	50C8
dbvc	dn,l	58C8
dbvs	dn,l	59C8
exg	rn,rn	C100
ext	dn	4880
ext.l	dn	48C0
link	an,$n	4E50

moveq	$n,dn	7000
nop		4E71
reset		4E70
rte		4E73
rtr		4E77
rts		4E75
stop	$n	4E72
swap	dn	4840
trap	$n	4E40
trapv		4E76
unlk	an	4E58

See Also
as68toas, cc, cpp, commands, drtomw, ld

Diagnostics
as reports errors on the standard error device. It gives a one-letter error code, the line number, the input file (if more than one specified), and a symbol where appropriate. See the section on **Errors**, presented earlier in this manual, for interpretation of error codes.

as68toas—Command
Convert DRI assembler to Mark Williams assembler
as68toas <*oldfile*.**asm** >*newfile*.**s**

as68toas converts files of 68000 assembly language from the DRI dialect into the Mark Williams dialect. It accepts DRI-style instructions from the standard input device (normally the keyboard), and produces Mark Williams-style instructions on the standard output (normally the screen). If it cannot handle a given instruction, it will notify you via the standard error (normally the screen).

As shown above, files can be converted automatically under the microshell by using the redirection operators '<' and '>'. Thus, to convert the file **foo.asm**, which is written in DRI-style assembly language, into a file of Mark Williams-style assembly language, called **foo.s**, simply type:

```
as68toas <foo.asm >foo.s
```

Note that files of Mark Williams-style assembly language *must* have the suffix .s; otherwise, they will not be accepted by the assembler **as**.

See Also
as, commands, drtomw, TOS

ASCII—Definition
ASCII is an acronym for the American Standard Code for Information Interchange. It is a table of seven-bit binary numbers that encode the letters of the alphabet, numerals, punctuation, and the most commonly used control sequen-

ces for printers and terminals. ASCII codes are used on all microcomputers sold in the United States.

The following table gives the ASCII characters in octal, decimal, and hexadecimal numbers, their definitions, and expands abbreviations where necessary.

000	0	0x00	NUL	**<ctrl-@>**	NUL character
001	1	0x01	SOH	**<ctrl-A>**	Start of header
002	2	0x02	STX	**<ctrl-B>**	Start of text
003	3	0x03	ETX	**<ctrl-C>**	End of text
004	4	0x04	EOT	**<ctrl-D>**	End of transmission
005	5	0x05	ENQ	**<ctrl-E>**	Enquiry
006	6	0x06	ACK	**<ctrl-F>**	Positive acknowledgement
007	7	0x07	BEL	**<ctrl-G>**	Bell
010	8	0x08	BS	**<ctrl-H>**	Backspace
011	9	0x09	HT	**<ctrl-I>**	Horizontal tab
012	10	0x0A	LF	**<ctrl-J>**	Line feed
013	11	0x0B	VT	**<ctrl-K>**	Vertical tab
014	12	0x0C	FF	**<ctrl-L>**	Form feed
015	13	0x0D	CR	**<ctrl-M>**	Carriage return
016	14	0x0E	SO	**<ctrl-N>**	Shift out
017	15	0x0F	SI	**<ctrl-O>**	Shift in
020	16	0x10	DLE	**<ctrl-P>**	Data link escape
021	17	0x11	DC1	**<ctrl-Q>**	Device control 1 (XON)
022	18	0x12	DC2	**<ctrl-R>**	Device control 2 (tape on)
023	19	0x13	DC3	**<ctrl-S>**	Device control 3 (XOFF)
024	20	0x14	DC4	**<ctrl-T>**	Device control 4 (tape off)
025	21	0x15	NAK	**<ctrl-U>**	Negative acknowledgement
026	22	0x16	SYN	**<ctrl-V>**	Synchronize
027	23	0x17	ETB	**<ctrl-W>**	End of transmission block
030	24	0x18	CAN	**<ctrl-X>**	Cancel
031	25	0x19	EM	**<ctrl-Y>**	End of medium
032	26	0x1A	SUB	**<ctrl-Z>**	Substitute
033	27	0x1B	ESC	**<ctrl-[>**	Escape
034	28	0x1C	FS	**<ctrl-\>**	Form separator
035	29	0x1D	GS	**<ctrl-]>**	Group separator
036	30	0x1E	RS	**<ctrl-^>**	Record separator
037	31	0x1F	US	**<ctrl-_>**	Unit separator
040	32	0x20	SP	Space	
041	33	0x21	!	Exclamation point	
042	34	0x22	"	Quotation mark	
043	35	0x23	#	Pound sign	
044	36	0x24	$	Dollar sign	
045	37	0x25	%	Percent sign	
046	38	0x26	&	Ampersand	
047	39	0x27	'	Apostrophe	

050	40	0x28	(	Left parenthesis
051	41	0x29	)	Right parenthesis
052	42	0x2A	*	Asterisk
053	43	0x2B	+	Plus sign
054	44	0x2C	,	Comma
055	45	0x2D	-	Hyphen (minus sign)
056	46	0x2E	.	Period
057	47	0x2F	/	Virgule (slash)
060	48	0x30	0	
061	49	0x31	1	
062	50	0x32	2	
063	51	0x33	3	
064	52	0x34	4	
065	53	0x35	5	
066	54	0x36	6	
067	55	0x37	7	
070	56	0x38	8	
071	57	0x39	9	
072	58	0x3A	:	Colon
073	59	0x3B	;	Semicolon
074	60	0x3C	<	Less-than symbol (left angle bracket)
075	61	0x3D	=	Equal sign
076	62	0x3E	>	Greater-than symbol (right angle bracket)
077	63	0x3F	?	Question mark
0100	64	0x40	@	At sign
0101	65	0x41	A	
0102	66	0x42	B	
0103	67	0x43	C	
0104	68	0x44	D	
0105	69	0x45	E	
0106	70	0x46	F	
0107	71	0x47	G	
0110	72	0x48	H	
0111	73	0x49	I	
0112	74	0x4A	J	
0113	75	0x4B	K	
0114	76	0x4C	L	
0115	77	0x4D	M	
0116	78	0x4E	N	
0117	79	0x4F	O	
0120	80	0x50	P	
0121	81	0x51	Q	
0122	82	0x52	R	
0123	83	0x53	S	
0124	84	0x54	T	
0125	85	0x55	U	

0126	86	0x56	V	
0127	87	0x57	W	
0130	88	0x58	X	
0131	89	0x59	Y	
0132	90	0x5A	Z	
0133	91	0x5B	[	Left bracket (left square bracket)
0134	92	0x5C	\	Backslash
0135	93	0x5D	]	Right bracket (right square bracket)
0136	94	0x5E	^	Circumflex
0137	95	0x5F	_	Underscore
0140	96	0x60	`	Grave
0141	97	0x61	a	
0142	98	0x62	b	
0143	99	0x63	c	
0144	100	0x64	d	
0145	101	0x65	e	
0146	102	0x66	f	
0147	103	0x67	g	
0150	104	0x68	h	
0151	105	0x69	i	
0152	106	0x6A	j	
0153	107	0x6B	k	
0154	108	0x6C	l	
0155	109	0x6D	m	
0156	110	0x6E	n	
0157	111	0x6F	o	
0160	112	0x70	p	
0161	113	0x71	q	
0162	114	0x72	r	
0163	115	0x73	s	
0164	116	0x74	t	
0165	117	0x75	u	
0166	118	0x76	v	
0167	119	0x77	w	
0170	120	0x78	x	
0171	121	0x79	y	
0172	122	0x7A	z	
0173	123	0x7B	{	Left brace (left curly bracket)
0174	124	0x7C	|	Vertical bar
0175	125	0x7D	}	Right brace (right curly bracket)
0176	126	0x7E	~	Tilde
0177	127	0x7F	DEL	Delete

See Also
string

asctime—Time function (**libc.a/ctime**)

Convert time structure to ASCII string

#include <time.h>

char ***asctime(***tmp***) tm_t ****tmp*;

asctime takes the data found in *tmp*, and turns it into an ASCII string that can be read by humans. *tmp* is declared to be of the type **tm_t**, which is a structure defined in the header file **time.h**. This structure must first be initialized by either **gmtime** or **localtime** before it can be used by **asctime**. For a further discussion of **tm_t**, see the entry for **time**.

Example

The following example demonstrates the functions **asctime**, **ctime**, **gmtime**, **localtime**, and **time**, and shows the effect of the environmental variable **TIMEZONE**. For a discussion of the variable **time_t**, see the entry for **time**.

```
#include <time.h>
main() {
        time_t timenumber;
        tm_t *timestruct;

        time(&timenumber);
        printf("%s", ctime(&timenumber));

        timestruct = gmtime(&timenumber);
        printf("%s", asctime(timestruct));

        timestruct = localtime(&timenumber);
        printf("%s", asctime(timestruct));
}
```

The following gives an "optimized" form of the above program. It shows more clearly how return values can be passed as arguments, and how nesting can increase the work done by each line of code.

```
#include <time.h>
main() {
        time_t t;
        time(&t);
        printf("%s", ctime(&t));
        printf("%s", asctime(gmtime(&t)));
        printf("%s", asctime(localtime(&t)));
}
```

See Also

time

Notes
asctime returns a pointer to a statically allocated data area that is overwritten by successive calls.

asin—Mathematics function (**libm.a/asin**)
Calculate inverse sine
#include <math.h>
double asin(*arg*) double *arg*;

asin calculates the inverse sin of *arg*, which must be in the range [-1., 1.]. The result will be in the range [-**PI**/2, **PI**/2].

Example
For an example of this function, see the entry for **acos**.

See Also
mathematics library

Diagnostics
Out-of-range arguments set **errno** to **EDOM** and return 0.

assert—Debugging macro
Check assertion at run time
#include <assert.h>
assert(*condition*)

assert checks the value of the given *condition*. If the *condition* is false (0), **assert** prints an error message and exits. **assert** should be used to detect situations that are expected never to happen. Note that the **-DNDEBUG** argument to **cc** disables all checking of assertions.

Example

```
#include <assert.h>
main() {
        int a = 1;
        int b = 2;

        assert( a>b );
}
```

See Also
#assert, assert.h, cc

Diagnostics
assert prints **assert(*condition*) failed** when *condition* is not true. Because assert is a macro that uses **printf**, it expands into an illegal C statement if *condition* includes quotation marks (""). It also cannot be used in an expression.

assert.h—Header file

Text of **assert** message

#include <assert.h>

assert.h is the header file that contains the **assert** macro definition.

See Also

assert, header file

#assert—Definition

Check assertion at compile time

#assert *expression*

The Mark Williams C preprocessor **cpp**, in addition to the directives mentioned in *The C Programming Language*, recognizes the **#assert** directive. It has the form:

#assert *constant_expression*

The preprocessor evaluates the constant expression. If it is false (zero), **cpp** prints a diagnostic message. The condition being tested must be an expression that involves constants of the form acceptable to the preprocessor's **#if** function. This tool should be used to ensure that variables in complex preprocessor code are correct throughout the program.

Example

If the line

```
#assert SIZE < 80
```

is included in a program, the assertion will succeed if **SIZE** is less than 80, and fail if it is 80 or more.

See Also

cpp

The C Programming Language, page 86

Diagnostics

The failure of an **#assert** causes the message

```
Preprocessor assertion failure
```

to appear on the standard error device; however, failure of an **#assert** directive does not terminate compilation.

atan—Mathematics function (**libm.a/atan**)

Calculate inverse tangent

#include <math.h>

double atan(*arg*) double *arg*;

atan calculates the inverse tangent. *arg* may be any real number. The result will be in the range [-**PI**/2, **PI**/2].

Example
For an example of this function, see the entry for **acos**.

See Also
errno, mathematics library

atan2—Mathematics function (**libm.a/atan2**)
Calculate inverse tangent
double atan2(*num*, *den*) double *num*, *den*;

atan2 calculates the inverse tangent of the quotient of its arguments *num/den*. *num* and *den* may be any real numbers. The result will be in the range [-**PI**, **PI**]. The sine of the result will have the same sign as *num*, and the cosine will have the same sign as *den*.

Example
For an example of this function, see the entry for **acos**.

See Also
errno, mathematics library

atof—General function (**libc.a/atof**)
Convert ASCII strings to floating point
double atof(*string*) char * *string*;

atof converts the argument *string* to a binary representation of a double-precision floating point number. The argument *string* must be the ASCII representation of a floating-point number. It can contain a leading sign, any number of decimal digits, and one decimal point. It can be terminated with an exponent, which consists of an 'e' or 'E' and followed by an optional leading sign and any number of decimal digits. **atof** ignores leading blanks and tabs; it stops scanning when it encounters any unrecognized character.

Example
For example of this function, see the entry for **acos**.

See Also
atoi, atol, float, long, printf, scanf

Notes
No overflow checks are performed. **atof** returns 0 if it receives a string it cannot interpret.

atoi—General function (**libc.a/atoi**)
Convert ASCII strings to integers
int atoi(*string*) char * *string*;

atoi converts the argument *string* to the binary representation of an integer. *string* may contain a leading sign and any number of decimal digits. **atoi** ignores leading blanks and tabs; it stops scanning when it encounters any non-numeral other than the leading sign, and returns the resulting **int**.

Example
The following demonstrates **atoi**. It takes a string typed at the terminal, turns it into an integer, then prints that integer on the screen. To exit, type **<ctrl-C>**.

```
main() {
	extern char *gets();
	extern int atoi();
	char string[64];
	for(;;) {
		printf("Enter numeric string: ");
		if(gets(string))
			printf("%d\n", atoi(string));
		else
			break;
	}
}
```

See Also
atof, atol, int, printf, scanf

Notes
No overflow checks are performed. **atoi** returns 0 if it receives a string it cannot interpret.

atol—General function (**libc.a/atol**)
Convert ASCII strings to long integers
long atol(*string*) char *string*;

atol converts the argument *string* to a binary representation of a **long**. *string* may contain a leading sign (but no trailing sign) and any number of decimal digits. **atol** ignores leading blanks and tabs; it stops scanning when it encounters any non-numeral other than the leading sign, and returns the resulting **long**.

Example

```
main() {
        extern char *gets();
        extern long atol();
        char string[64];
        for(;;) {
                printf("Enter numeric string: ");
                if(gets(string))
                        printf("%ld\n", atol(string));
                else
                        break;
        }
}
```

See Also
atof, atoi, float, long, printf, scanf

Notes
No overflow checks are performed. **atol** returns 0 if it receives a string it cannot interpret.

auto—Definition

auto is an abbreviation for an *automatic variable*. This is a variable that applies only to the function that invokes it, and vanishes when the functions exits. The word **auto** is a C keyword, and may not be used to name any function, macro, or variable.

See Also
extern, keywords, stack, static, storage class
The C Programming Language, page 28

\auto—Definition

\auto is a directory that is scanned by TOS when it boots. TOS looks for this directory on the disk in drive A:. If it is present, TOS executes all of the files stored there that have the suffix **.prg**, in the order in which they appear. This is useful for automatically setting up such tools as RAM disks.

Note that when TOS executes the programs in **\auto**, the AES and VDI have not yet been initialized, so no GEM applications can be run. The current directory of the programs run from **\auto** is the root of the boot disk. If Line A functions are used, they must provide their own **contrl**, **intin**, and **intout** arrays. You can place **msh.prg** into **\auto** and enter it automatically when you boot your system; however, subsequent attempts to run any GEM application through **msh** generates effects that are unpredictable and usually unwelcome.

Example
The following example shows a few things that you can do in a program that is placed in **\auto**. It demonstrates the functions **Cursconf**, **Iorec**, **Kbrate**, **linea0**, **Ptermres**, **Rsconf**, **Setprt**, **stime**, and **time**, the global variable **_stksize**, and the header files **basepage.h** and **xbios.h**.

```
#include <linea.h>
#include <osbind.h>
#include <time.h>
#include <basepage.h>
#include <xbios.h>
long _stksize = 256;                          /* We need very little stack for this */

main() {
/*
 * Init: linea0(): initialize la_data for graphics
 * Initializing these pointers allows linea graphics in \auto\*.prg
 */
        {
                static int intin[128], intout[128], ptsin[128], ptsout[128];
                static int *contrl[4];
                linea0();
                INTIN = intin;

                INTOUT = intout;
                PTSIN = ptsin;
                PTSOUT = ptsout;
                CONTRL = contrl;
        }

/*
 * Init: stime(): set initial system time from the keyboard clock
 * time() reads the keyboard clock, stime() will set the GEM-DOS time
 */
        {
                time_t t;
                time(&t);
                stime(&t);
        }

/*
 * Init: Iorec(): resize the input/output buffers
 * Increasing the buffer sizes may or may not be necessary
 * It depends on how fast the buffers are filled and emptied
 */
        {
                register struct iorec *ip;
                static char auxin[1024], auxout[1024], midi[1024], kbd[1024];
                static struct iorec tmp = { 0, 1024, 0, 0, 256, 768 };
```

```
                ip = Iorec(IO_AUX); tmp.io_buff = auxin; *ip = tmp;
                ip += 1; tmp.io_buff = auxout; *ip = tmp;
                ip = Iorec(IO_MID); tmp.io_buff = midi; *ip = tmp;
                ip = Iorec(IO_KBD); tmp.io_buff = kbd; *ip = tmp;
        }

/*
 * Init: Rsconf(): configure rs232 port
 * Set the default baud rate and control protocol for the serial port
 */
        Rsconf(RS_B9600, RS_XONXOFF, -1, -1, -1, -1);

/* Init: Setprt(): set printer configuration */
        Setprt(PR_SERIAL|PR_EPSON|PR_MONO|PR_MATRIX);

/*
 * Init: Cursconf(): set cursor configuration
 * This slows the blink down to half the normal speed
 */
        Cursconf(CC_SET, (int)Cursconf(CC_GET, 0)*2);

/*
 * Init: Kbrate(): set keyboard repeat configuration
 * Again, simply slow it down a bit
 */
        {
                register int start, delay;
                start = Kbrate(-1, -1);
                delay = start & 0xff;
                start >>= 8;

                start &= 0xff;
                start *= 2;
                delay *= 4;
                Kbrate(start, delay);
        }

/*
 * Init: terminate and stay resident, so the buffers we assigned do not
 * get clobbered by the next program that runs
 */
        Ptermres(BP->p_hitpa-BP->p_lowtpa, 0);
}
```

See Also

TOS

aux:—TOS device

TOS logical device for serial port auxiliary device

TOS gives names to its logical devices. Mark Williams C uses these names, to allow the **STDIO** library routines to access these devices via TOS. **aux:** is the logical device for the the serial port auxiliary device.

Example

```
#include <stdio.h>
main(){
	FILE *fp, *fopen();
	if ((fp = fopen("aux:","w")) != NULL)
		fprintf(fp,"aux: enabled.\n");
	else printf("aux: cannot open.\n");
}
```

See Also
con:, **prn:**, **Rsconf**

Notes
aux: may be spelled **aux:** or **AUX:**.

backspace—Definition

Mark Williams C recognizes the literal character '\b' for the ASCII space character BS (octal 010). This character may be used as a character constant or in a string constant, like the other character constants: '\a', which rings the audible bell on the terminal; '\f', to pass a formfeed command to the printer; '\r', for a carriage return; '\t', for a horizontal tab character; and '\v', the vertical tab character.

See Also

ASCII, character constant

basepage.h—Header file

TOS header file

#include <basepage.h>

basepage.h is a header file that defines the GEM base page structure. Its text is as follows:

```
#ifndef BASEPAGE_H
#define BASEPAGE_H
typedef struct {
        long      p_lowtpa;        /* Low transient program area */
        long      p_hitpa;         /* High transient program area */
        long      p_tbase;         /* Text segment base */
        long      p_tlen;          /* Text segment length */
        long      p_dbase;         /* Data length base */
        long      p_dlen;          /* Data length length */

        long      p_bbase;         /* Bss segment base */
        long      p_blen;          /* Bss segment length */
        long      p_fxx0[3];       /* Fill area one */
        long      p_env;           /* Environment string pointer */
        long      p_fxx1[20];      /* Fill block two */
        char      p_cmdlin[128];   /* Command line */
} BASEPAGE;
extern BASEPAGE _start[];
#define BP (&_start[-1])
#endif
```

See Also

header file, TOS

Bconin—bios function 2 (osbind.h)

Receive a character

#include <osbind.h>
#include <bios.h>

long **Bconin**(*handle*) **int** *handle*;

Bconin receives a character from a peripheral device. *handle* is an integer that indicates which device is being read, as follows:

0	**prn:** (the line printer)
1	**aux:** (the auxiliary serial port)
2	**con:** (the console)
3	the MIDI port
4	the intelligent keyboard (output only)
5	the raw screen (output only)

When **Bconin** reads from **con:**, it returns the key's raw scan code in the low byte of the high word and either an ASCII character or zero in the low byte of the low word, depending upon whether the key typed generates an ASCII character or not; when it is reading from **aux:**, it returns the character in the low byte of the low word.

For a table of keyboard scan codes, see the entry for **keyboard.** Note, too, that this function is unaffected by redirection of either **con:** or **aux:**.

Example
This example emulates a simple dumb terminal. It demonstrates the functions **Bconin, Bconout, Bconstat, Bcostat,** and **Pterm0**.

```
#include <osbind.h>
#include <bios.h>

main()
{
	register long c;

	for (;;) {
		if (Bconstat(BC_CON)) {
			c = Bconin(BC_CON);

			if ((int)c == 0) {
				c >>= 16;
				if (c == KC_UNDO)
					break;
				else
					Bconout(BC_CON, '\a');

			} else {
				while (Bcostat(BC_AUX) == 0)
					;
				Bconout(BC_AUX, (int)c);
			}
		}
```

```
		if (Bconstat(BC_AUX)) {
			c = Bconin(BC_AUX);
			Bconout(BC_CON, (int)c);
		}
	}
	Pterm0();
}
```

See Also
aux:, Bconout, Bconstat, Bcostat, bios, con:, keyboard, TOS

Bconout—bios function 3 (**osbind.h**)

Send a character to a peripheral device

#include <osbind.h>
#include <bios.h>
void Bconout(*handle, character***) int** *handle, character*;

Bconout sends characters to an output device. *handle* is an integer that indicates which device to send characters, as follows:

0	**prn:** (the line printer)
1	**aux:** (the auxiliary serial port)
2	**con:** (the console)
3	the MIDI port
4	the intelligent keyboard (output only)
5	the raw screen (output only)

character is the character being output, which is encoded in the lower eight bits of the integer. **Bconout** returns nothing. This function is unaffected by redirection of the logical devices **con:** or **aux:**.

If *handle* is set to five, characters are displayed on the screen as with device number 2, but control characters are not interpreted. This allows the display of graphics characters from the Atari character set, in the range of one through 31.

Example
For an example of this function, see the entry for **Bconin**.

See Also
Bconin, Bconstat, Bcostat, bios, TOS

Bconstat—bios function 1 (**osbind.h**)

Return the input status of a peripheral device

#include <osbind.h>
#include <bios.h>
long Bconstat(*device***) int** *device*;

Bconstat reads the input status of the specified peripheral device. *device* is an integer that encodes the Bcostat of the desired device, as follows:

0	**prn:** (the line printer)
1	**aux:** (the auxiliary serial port)
2	**con:** (the console)
3	the MIDI port
4	the intelligent keyboard (output only)
5	the raw screen (output only)

Bconstat returns -1 if at least one character is ready to be handled, and 0 if no characters are ready. This function is unaffected by redirection.

Example
For an example of this function, see the entry for **Bconin.**

See Also
Bconin, Bconout, Bcostat, bios, TOS

Bcostat—bios function 8 (**osbind.h**)
Read the output status of a peripheral device
#include <osbind.h>
#include <bios.h>
long **Bcostat(***handle***) int** *handle*;

Bcostat reads the output status of a peripheral device. *handle* is a number that indicates the device to be checked, as follows:

0	**prn:** (the line printer)
1	**aux:** (the auxiliary serial port)
2	**con:** (the console)
3	the MIDI port
4	the intelligent keyboard (output only)
5	the raw screen (output only)

Bcostat returns -1 if the device is ready, 0 if it is not. This function is unaffected by redirection.

Example
For an example of this function, see the entry for **Bconin.**

See Also
Bconin, Bconout, Bconstat, bios, TOS

bios.h—Header file
#include <bios.h>

bios.h is a header file that includes all constants and structures used by the GEM-DOS **bios** functions. For a list of these functions, see the entry for **bios**.

See Also
bios, header file, TOS, xbios.h

bios—TOS function
Call an input/output routine in the TOS BIOS
#include <osbind.h>
extern long bios(*n, f1, f2 ... fn*);

bios allows you to call an input/output function directly in the Atari BIOS. It works by building a stack frame and executing trap no. 13. Unless the **-VNOTRAP** option is used when compiling a program, the instruction **jsr bios_** is replaced by a trap no. 13 instruction.

n is the number of the function, and *f1* through *fn* are the parameters to be used with the routine. In most circumstances, it is unnecessary to call **bios**, for the header file **osbind.h** defines a number of functions that use it directly. All structures and constants used by these functions are contained in the header file **bios.h**.

The following functions call **bios** to deal with the peripheral devices:

Bconin	receive a character
Bconout	output a character
Bconstat	return input status of device
Bcostat	return output status of device
Drvmap	return map of logical drives
Getbpb	return pointer to BIOS parameter block
Getmpb	copy memory parameter block
Getshift	get/set status for shift/alt/control keys
Mediach	check if medium has been changed
Rwabs	read/write a disk drive
Setexc	set an exception vector
Tickcal	return system timer's calibration

See Also
osbind.h, TOS

Notes
No **bios** function checks for incorrect device numbers. Passing a bogus device number to a routine will crash the system.

Note that the Atari BIOS will support up to three recursive calls at any one time. Using more than three will cause the system to crash.

Note that all **bios** functions are unbuffered. Combining them with buffered routines, such as those in the STDIO library, will lead at best to unpredictable results.

BIOS—Definition

BIOS is an acronym for *basic input/output system*. In most machines, the BIOS consists of a routine carried in the read-only memory (ROM).

See Also
bios, STDIO

Bioskeys—xbios function 24 (osbind.h)

Reset the keyboard to its default
#include <osbind.h>
#include <xbios.h>
void Bioskeys()

Bioskeys resets the keyboard to its default settings, and returns nothing. It undoes whatever changes were made with the function **Keytbl**.

Example

```
#include <osbind.h>
main() {
        Bioskeys();
}
```

See Also
Keytbl, TOS, xbios

bit—Definition

bit is an abbreviation for **binary digit**. It is the basic unit of data processing, the computer analogue of Democrites' atoms. A bit can have a value of either zero or one, and can be concatenated into strings. A bit can be used either as a placeholder to construct a number with an absolute value, or as a flag whose value as a particular meaning under specially defined circumstances. In the former use, a string of bits builds an integer. In the latter use, a string of bits forms a **map**, in which each bit has a meaning beyond its numeric value.

See Also
bit map, byte, integer, nybble

bit map—Definition

A **bit map** is a string of bits in which each bit has a symbolic, rather than numeric, value. For example, the **Drvmap** function returns a 16-bit map of the active drives on the Atari ST. The bits indicate which of 16 possible disk drives is available, with bit 0 (i.e., 1<<0) corresponding to drive A, bit 1 to drive B, etc.

See Also
bit, byte
The C Programming Language, page 136

Notes
C permits the manipulation of bits within a byte through the use of bit field routines. However, programs that use bit fields often run more slowly than those that use masking and shifting.

bombs—Definition
When a program goes seriously wrong on the Atari ST, TOS takes the following default actions:

1. It stores a description of the program's state in a buffer in low memory.
2. It displays one or more "cherry bombs" on the screen; persons with older versions of the operating system may see little "mushroom clouds" instead. The number of bombs seen is equal to the number of the processor exception.
3. TOS attempts to terminate the program and continue processing.

You use the debugger **db** to display the program state saved in low memory by TOS. Use the following commands:

db -k	enter **db**
:r	display contents of registers
:f	print type of fault
:q	quit

This prints the processor registers at the time of the fault and identifies the fault. The exceptions that occur on the 68000 processor are listed in the header file **signal.h**.

See Also
db, signal.h, TOS

boot—Definition
Boot is an abbreviation for *bootstrapping procedure*; this refers to the procedure by which a computer loads certain elementary routines to organize and test memory, and initialize peripheral devices. The term *warm boot* is used with some operating systems to refer to the second-stage bootstrapping procedure, which is done simply to restore portions of the operating system that may have been overlaid by user code during the operation of a program, or that reinitializes the system state without going through the entire boot procedure.

See Also
exit

Notes
TOS does not warm boot on program termination.

buffer—Definition

A **buffer** is a portion of memory reserved for a particular purpose. In the context of C, a buffer most often is an area set aside to hold data for a peripheral device; often, although not always, this involves setting aside a portion of the arena with **malloc** or its related functions.

Many operating systems automatically place data from a peripheral device into a buffer. Buffers normally can be cleared with **fflush**, by pressing the carriage return key on routines that perform input, or by sending a newline character on routines that perform output. The function **close**, which closes a file, will flush all buffers; **exit** calls **close** by default.

Note that combining in one program unbuffered and and buffered I/O functions on the same file or device may produce results that are, at best, unpredictable.

On the Atari ST, all STDIO routines use buffering by default. **stdin** and **stdout** stderr is not. Buffering can be turned off with the function **setbuf**. All Atari functions that perform I/O are not buffered.

See Also
arena, array, Cconrs, Cconws, fflush, malloc, setbuf, STDIO
The C Programming Language, page 173

byte—Definition

A **byte** is a group of eight bits, which often is used to encode a character. Note that "byte" is not a legitimate term of data organization in C. Data types are defined as multiples of the data type **char**; what a **char** is defined to be depends on the hardware. Although a **char** is often defined as being eight bits long, the same as a byte, this definition is not universal.

See Also
bit, char, data formats, nybble

byte ordering—Definition

Byte ordering is the order in which a given machine stores successive bytes of a multibyte data item. The following example displays a few simple examples of byte ordering:

```
main() {
	union {
		char b[4];
		int i[2];
		long l;
	} u;
	u.l = 0x12345678L;
	printf("%x %x %x %x\n", u.b[0], u.b[1], u.b[2], u.b[3]);
	printf("%x %x\n", u.i[0], u.i[1]);
	printf("%lx\n", u.l);
}
```

When run on a PDP-11 under the COHERENT operating system, it gives the following results:

```
34 12 78 56
1234 5678
12345678
```

When run on the 68000 under TOS or on the Z8000 under COHERENT, it gives the following results:

```
12 34 56 78
1234 5678
12345678
```

When run on the i8086 under UDI or COHERENT, it gives the following results:

```
78 56 34 12
5678 1234
12345678
```

As can be seen, the order of the bytes differs between the machines.

See Also
canon.h, data formats

C language—Overview

The following summarizes how Mark Williams C implements the C language.

Identifiers:

Characters allowed: **A-Z, a-z, _, 0-9**
Compiler and linker are case sensitive.
Number of significant characters in a variable name:
at compile time: **128**
at link time: **16**
C identifier tag appended by compiler: _ at end of identifier

Reserved identifiers (keywords):

alien	**entry**	**return**
auto	**extern**	**short**
break	**float**	**sizeof**
case	**for**	**static**
char	**goto**	**struct**
continue	**if**	**switch**
default	**int**	**typedef**
do	**long**	**union**
double	**readonly**	**unsigned**
else	**register**	**while**

The keyword **entry** is not implemented. The proposed ANSI standard for C adds **const**, **signed**, and **volatile** to the above set, and deletes **entry** and **readonly**. Mark Williams C reserves the keywords **readonly** and **alien**, but these are not implemented on the 68000.

Data formats (in bits)

char:	8
double:	64
float:	32
int:	16
long:	32
pointer:	32
short:	16
unsigned char:	8
unsigned short:	16
unsigned int:	16
unsigned long:	32

float *format:*

DECVAX floating point format:

- 1 sign bit
- 8-bit exponent
- 24-bit normalized mantissa with hidden bit

DECVAX double format:

- Same as **float**, but with 56 bits of mantissa

Reserved values:

- +- infinity, -0

All floating-point operations are done as **doubles**

Limits:

Maximum bitfield size: 16 bits
Maximum number of **cases in a switch**: no formal limit
Maximum block nesting depth: no formal limit
Maximum parentheses nesting depth: no formal limit
Maximum structure size: no formal limit
Maximum array size: 64 kilobytes

Preprocessor instructions:

#assert	**#ifdef**
#define	**#ifndef**
#else	**#include**
#endif	**#line**
#file	**#undef**
#if	

Structure name-spaces:

Supports both Berkeley and Kernighan and Ritchie conventions for structure in union.

Register variables:

Five available for **ints** or **longs**
Three available for pointers

Function linkage:

Return values for **chars**, **ints**, **longs**, or pointers in d0
Return values for **doubles** in d0 and d1
Pointers to returned structures in d0, copied to destination by caller
Parameters pushed on stack in reverse order, **chars** and **shorts** pushed as words, **longs** and pointers pushed as **longs**, structures copied onto stack
Caller is responsible for clearing parameters off stack
Stack frame linkage is done through a6

Register usage:

d0, d1: Scratch data and function return values
d2: Scratch data
d3, d4, d5, d6, d7: Register variables for **longs** and **ints**
a0, a1, a2: Scratch addresses
a3, a4, a5: Register pointers for any type or structure
a6: Call frame linkage pointer
a7: Stack pointer

Special features and optimizations:

* By default, the compiler makes the following substitutions:

```
jsr gemdos_   trap $1
jsr micrortx  trap $5
jsr bios_     trap $13
jsr xbios_    trap $14
```

This reduces the overhead for system calls and makes the code reentrant (although the system itself may not be). Turn off this feature with the option **-VNOTRAP**.

* Branch optimization is performed: this uses the smallest branch instruction for the required range.

* Unreached code is eliminated.

* Duplicate instruction sequences are removed.

* Jumps to jumps are eliminated.

* Cross-jumps are eliminated. This changes code like this:

```
              move a, b
              bra  LABEL1
      LABEL0: move c, b
              bra  LABEL2
      LABEL1:move b, d
              bra  LABEL3
      LABEL2:move f, d
              bra  LABEL3
```

to:

```
              move a, b
              move b, d
              bra  LABEL3
      LABEL0:move c, b
              move f, d
              bra  LABEL3
```

See Also
byte ordering, data formats, data types, declarations, keywords, Lexicon, memory allocation

cabs—Mathematics function (**libm.a/cabs**)
Complex absolute value function
#include <math.h>
double cabs(*z*) struct { double *r*, *i*; } *z*;

cabs computes the absolute value, or modulus, of its complex argument *z*. The absolute value of a complex number is the length of the hypotenuse of a right triangle whose sides are given by the real part *r* and the imaginary part *i*. The result is the square root of the sum of the squares of the parts.

Example
For an example of this function, see the entry for **acos**.

See Also
hypot, mathematics library

calling conventions—Definition
This entry discusses the Mark Williams C function calling conventions. This information is helpful to users who wish to interface C programs with assembly language routines or with object code generated by other language processors. Programs that depend upon specific details of these calling conventions may not be portable to other processors or other C compilers.

In general, Mark Williams C pushes arguments from right to left. Mark Williams C pushes function arguments as follows:

char	as a word
short	as a word
int	as a word
long	as a long word
float	as a pair of long words
double	as a pair of long words
pointer	as a long word

"Word" in this instance means a 68000 (16-bit) word.

An underbar '_' is appended to the name of the function. This makes assembly language programmers append '_' to the names of their C callable functions. This is also used by the utility **nm** to choose the symbols printed with the **-a** option.

An **add**, **lea**, or **addq** instruction after the call removes the arguments from the stack.

The C prologue executes a **link** to allocate space for automatics and saved registers. Because C functions may use registers a3 through a5 and d3 through d7 for register variables, the C prologue saves used registers, and the C epilogue restores them. The C epilogue executes an **unlk** before returning.

Parameters and local variables in the called function are referenced as offsets from the (frame pointer) register. The stack-pointer register points below the local variable with the lowest address.

Functions return values as follows:

char	in d0.W
int	in d0.W
long	in d0.L
float	in d0 and d1
double	in d0 and d1
pointer	in d0.L

Functions that return **struct** or **union** actually return a pointer to the **struct** or **union**. The code generated for the function call will move the result to its destination.

C does not require that the number of arguments passed to a function be the same as the number of arguments specified in the function's declaration. Routines with a variable number of arguments are not uncommon. The two formatted I/O routines in the standard library (**printf** and **scanf**) are, in fact, routines that take a variable number of arguments.

Consider the following program as an example:

```
long f(a, b, c)
char a;
int b;
long c;
{
    return ((a * b) + c);
}
main() {
        char a = 1;
        int b = 2;
        long c = 3;

        f(a,b,c);
}
```

When compiled with the **-S** option, it produces the following code:

```
        .shri

        .globl f_
f_:
        link        a6, $0
        move        10(a6), d0
        muls        8(a6), d0
        ext.l       d0
        add.l       12(a6), d0
        unlk        a6
        rts
        .globl main_

main_:
        link        a6, $-8
        moveq       $1, d0
        move.b      d0, -2(a6)
        moveq       $2, d0
        move        d0, -4(a6)

        moveq       $3, d0
        move.l      d0, -8(a6)
        move.l      -8(a6), -(a7)
        move        -4(a6), -(a7)
        move.b      -2(a6), d0
        ext         d0
        move        d0, -(a7)

        jsr         f_
        addq        $8, a7
        unlk        a6
        rts
```

The symbols **main** and **f** have become **main_** and f_. The automatic variables in **main** are addressed at negative offsets from a6: **char a** is located at -2(a6), **int b** at -4(a6), and **long c** at -8(a6). A byte of unused storage follows **a** so that **b** occurs on an even address. **main** pushes **c**, then **b**, then sign extends **a** and pushes the resulting word. The arguments in f are addressed at positive offsets from a6: **char a** is located at 8(a6), **int b** at 10(a6), and **long c** at 12(a6). **char c** is treated as an **int**. The result expression is computed into d0.L. When **f** returns, **main** pops the arguments with an **addq** instruction.

In **f** after execution of the **link**, the stack appears as follows:

```
 --------------------
 |     high A6      | <--A6 = frame pointer for f
 --------------------
 |      low A6      |
 --------------------
 |   high return    |
 --------------------
 |    low return    |
 --------------------
 |        a         | parameters
 --------------------
 |        b         |
 --------------------
 |      high c      |
 --------------------
 |      low c       |
 --------------------
```

The following function returns a structure:

```
struct date {
        int month, day, year;
} today;

struct date
mkda(m, d, y)
{
        struct date tmp;

        tmp.month=m;
        tmp.day  =d;
        tmp.year =y;
        return(tmp);
}

main()
{
        today = mkda(3, 20, 85);
}
```

When this program translated into assembly language by compiling it with the -S option, the result is as follows:

```
        .bssd

        .globl today_
today_:
        .blkb       0x6
```

```
        .shri

        .globl mkda_
mkda_:

        .bssd

L10001:
        .blkb  0x6

        .shri

        link   a6, $-6
        move   8(a6), -6(a6)
        move   10(a6), -4(a6)
        move   12(a6), -2(a6)
        lea    -6(a6), a1
        movea.l       $L10001, a0
        move   $6, d0
        bra.s  L10002

L10003:
        move.b      (a1), (a0)
        addq        $1, a0
        addq        $1, a1

L10002:
        dbf         d0,      L10003
        move.l      $L10001, d0
        unlk        a6
        rts
        .globl main_

main_:
        link        a6, $0
        moveq       $85, d0
        move        d0, -(a7)
        moveq       $20, d0
        move        d0, -(a7)
        moveq       $3, d0
        move        d0, -(a7)

        jsr         mkda_
        addq        $6, a7
        movea.l     d0, a1
        movea.l     $today_, a0
        move        $6, d0
        bra.s       L10004

L10005:
        move.b      (a1), (a0)
        addq        $1, a0
        addq        $1, a1
```

```
L10004:
        dbf     d0,     L10005
        unlk    a6
        rts
```

See Also
memory allocation

calloc—General function (**libc.a/calloc**)
Allocate dynamic memory
char ***calloc**(*count*, *size*) **unsigned** *count*, *size*;

calloc is one of a set of routines that helps manage a program's arena. **calloc** calls **malloc** to obtain a block large enough to contain *count* items of *size* bytes each; it then initializes the block to zeroes and returns a pointer to it. Dynamic memory that is no longer needed can be returned to the free memory pool with the function **free**.

See Also
arena, free, lcalloc, lmalloc, lrealloc, malloc, notmem, realloc

Diagnostics
calloc returns **NULL** if insufficient memory is available.

Notes
The related function **lcalloc** takes unsigned long arguments, and therefore can allocate memory blocks that are larger than 64 kilobytes.

canon.h—Header file
Canonical conversion for the 68000
#include <canon.h>

canon.h defines canonical conversion routines used for the 68000, to ensure that byte ordering is correct.

See Also
byte ordering

carriage return—Definition
Mark Williams C recognizes the literal character '\r' for the ASCII carriage return character CR (octal 015). This character "throws the carriage", i.e,. returns the cursor to the beginning of the line. The newline character '\n' drops the cursor down to the next line. With the UNIX library routines, \n is a synonym for \n plus \r. TOS routines, such as **Cconws**, need both characters explicitly.

\r may be used as a character constant or in a string constant, like the other character constants: '\a', which rings the audible bell on the terminal; '\b', to backspace; '\f', to pass a formfeed command to the printer; '\t', for a horizontal tab; and '\v', for a vertical tab.

See Also
ASCII, character constant

cat—Command
Concatenate files
cat [-u] [*file* ...]

cat copies each *file* to the standard output. A *file* specified by '-' indicates the standard input. If no *file* is specified, **cat** reads the standard input.

The **-u** option makes the output unbuffered. Otherwise, **cat** buffers the output in units of the machine's disk block size.

Note that **<ctrl-S>** pauses the outputting of text, and **<ctrl-Q>** resumes outputting.

See Also
commands, msh

Notes
Redirecting the output of **cat** into one of its input files is an error, as the command will never terminate. For example:

```
cat * >out
```

will cause the system to loop, with the file **out** being read and written into until the file system runs out of space.

Cauxin—**gemdos** function 3 (**osbind.h**)
Read a character from the serial port
#include <osbind.h>
long Cauxin()

Cauxin reads a character from the serial port **aux:**, and returns the character read. It is affected by redirection.

Example
The following example creates a dumb terminal emulator that operates through the serial port. It demonstrates the macros **Cauxin, Cauxis, Cauxos, Cauxout, Cconis, Cconout,** and **Crawcin**. You can exit from the program by typing **<ctrl-Z>**. Run the example either from the GEM desktop, or with the **tos** command.

```
#include <osbind.h>

main() {
	char c;

	for (;;) {
		if (Cauxis())
			Cconout(c = Cauxin());
		if (Cconis()) {
			if ((c = Crawcin()) == 26) {
				break;
			} else {
				if (Cauxos())		/* If ready */
					Cauxout(c);	/* send char */
				else			/* Otherwise */
					Cconout('\07');	/* ring bell */
			}
		}
	}
}
```

See Also
crtsg.0, gemdos, tos, TOS

Notes
TOS defines handle 2 as being **aux:**, the serial port. The microshell **msh** normally redirects handle 2 to another device; because **Cauxin** and its related functions can be redirected, any program that uses **Cauxin, Cauxis, Cauxos**, or **Cauxout** must be run directly from the GEM desktop, or run under the shell with the **tos** command, which re-redirects handle 2 to the **aux:** device.

An alternative is to use **Bconin** and its relatives instead of the **Cauxin** family when writing programs to be run under **msh**.

Cauxis—gemdos function 18 (**osbind.h**)
Check if characters are waiting at serial port
#include <osbind.h>
long Cauxis()

Cauxis checks to see if characters are waiting to be read at the serial port. It returns -1 if there are characters waiting, and 0 if there are not.

Example
For an example of how to use this macro, see the entry for **Cauxin**.

See Also
gemdos, tos, TOS

Notes
This function must be compiled with the **-VGEM** option, and run either from the GEM desktop or with the **tos** command.

Cauxos—gemdos function 19 (**osbind.h**)
Check if serial port is ready to receive characters
#include <osbind.h>
long Cauxos()

Cauxos checks the output status of the serial port. **Cauxos** returns -1 if the serial port is ready to send a character, and 0 if it is not.

Example
For an example of how to use this macro, see the entry for **Cauxin**.

See Also
gemdos, tos, TOS

Notes
This function must be compiled with the -VGEM option, and run either from the GEM desktop or with the **tos** command.

Cauxout—gemdos function 4 (**osbind.h**)
Write a char to the serial port
#include <osbind.h>
void Cauxout(*c*) int *c*;

Cauxout writes the character *c* to the serial port, and returns nothing.

Example
For an example of how to use this macro, see the entry for **Cauxin**.

See Also
gemdos, tos, TOS

Notes
This function must be compiled with the -VGEM option, and run either from the GEM desktop or with the **tos** command.

cc—Command
Compiler driver
cc [*options*] *file* ...

cc is the program that controls the compilation process. It guides files of source and object code through each phase of compilation and linking. **cc** has many options to assist in the compilation of C programs; in essence, however, all you need to do to produce an executable file from your C program is type **cc**

followed by the name of file (or files) that holds your program. It checks whether the file names you give it are reasonable, selects the right phase for each file, and performs other tasks that ease the compilation of your programs.

cc assumes that each *file* name that ends in **.c** or **.h** is a C program and processes it with the C compiler. It assumes that each *file* argument that ends in **.s** is an assembly-language program and processes it with the assembler **as**. It passes all files with the suffixes **.o** or **.a** unchanged to the linker **ld**.

The normal operation of the **cc** command is as follows. First, it compiles and assembles the source files, naming the resulting object files by replacing the **.c** or **.s** suffixes with the **.o** suffix. Then, it links the object files with the C run-time startoff routine and the standard C library, and leaves the result in file *file*.**prg**. If only one object file is created during compilation, it is deleted after linking; however, if more than one object file is created, or if an object file of the same name had been written before the present compilation, the object files are not deleted.

cc looks for the compiler and its other tools in directories that the user names. The names of these directories together compose cc's *environment*, and each name comprises an *environmental variable*. An environmental variable is set through the micro-shell **msh**, by using the command **setenv**. The user must set the following environmental variables for **cc** to work correctly:

LIBPATH This names the directories that hold the phases of the compiler, the libraries, and the C run-time start-up routines. Note that if you have more than one version of a file, cc will use the first one that it finds along the **LIBPATH**.

INCDIR This names the "default" directory within which the C preprocessor **cpp.prg** will look for files that are called with a **#include** statement. This default directory is searched along with the directory of the source file and the directories specified with **-I** options.

TMPDIR This names the directory into which temporary files should be written. The default if this variable is not set is the directory in which the source files are kept. Note that this variable need be set only if space is a problem on any of your storage devices.

These environmental variables should be set in your **profile** file. See the entry for **msh** for more information about **profile**.

cc's behavior may be altered by means of command line options. These are described below. **cc** passes all other options through to the linker **ld** unchanged, and correctly interprets to **ld** the **-e**, **-o**, and **-u** options, which are described below.

The C compiler itself consists of several *phases*. The preprocessor **cpp** expands **#define** and **#include** directives, among others, in the source program. The parser **cc0** parses the preprocessor output. The code generator **cc1** generates code for the program. The optimizer **cc2** optimizes the generated code and writes the object file. If an assembly-language listing is requested, the disassembler **cc3** writes it.

Note that a number of the options are esoteric and are not used typically when compiling a C program. In general, the most commonly used options are **-A** (to invoke the editor automatically when errors occur), **-f** (to include floating-point routines), **-l***name* (to pass the name of a library to the linker), **-o** *name* (rename the executable file), **-V** (run in verbose mode), and a number of the **-V***string* variant options.

-A MicroEMACS option. If an error occurs during compilation, **cc** automatically invokes the MicroEMACS screen editor. The error or errors are displayed in one window and the source code file in the other, with the cursor set to the line number indicated by the first error message. Typing **<ctrl-X>>** moves to the next error, **<ctrl-X><** moves to the previous error. To recompile, close the edited file with **<ctrl-Z>**. Compilation will continue either until the program compiles without error, or until you exit from the editor by typing **<ctrl-U>** followed by **<ctrl-X><ctrl-C>**.

-B[*string*]
Backup option. Use alternate versions of the compiler for **cc0**, **cc1**, **cc2**, and **cc3**. If *string* is supplied, **cc** prepends it to the names of the phases of the compiler to form the pathnames where these are found. Otherwise, **cc** prepends the name of the current directory. If a **-t** option was previously given, only the parts of the compiler specified by it are affected. Any number of **-B** and **-t** options may be used, with each **-t** option specifying the passes affected by the subsequent **-B** option. For example, the command

```
cc -tp2 -Bnew hello.c
```

will compile **hello.c** using **newcc2** in place of the ordinarily used **\lib\cc2**, and using **newcpp** in place of the ordinarily used **\lib\cpp**.

-c Compile option. Suppress linking and the removal of the object files.

-D*name*[=*value*]
Define *name* to the preprocessor, as if set by a **#define** directive. If *value* is present, it is used to initialize the definition.

-E Expand option. Run the C preprocessor and write its output onto the standard output.

-f Floating point option. Add object files with floating point output to the linker command line. Because the floating point conversion routines require approximately five kilobytes, the standard C library does not include them; the **-f** option tells the compiler to include them. If a program is compiled without the **-f** option but attempts to print a floating point number during execution by using the **e**, **f**, or **g** format specifications to **printf**, the message

```
You must compile -f
```

will be printed and the program will exit.

-I*directory*
Include option. Specify the directory the preprocessor should search for files given in **#include** directives, using the following criteria: If the **#include** statement reads

```
#include "file.h"
```

cc searches for **file.h** first in the source directory, then in the directory named in the **-I***directory* option, and finally in the system's default directories. If the **#include** statement reads

```
#include <file.h>
```

cc searches for **file.h** first in the directory named in the **-I***directory* option, and then in the system's default directories. Multiple -I*directory* options are searched for in their order of appearance.

-K Keep option. Save the intermediate files generated during the compilation in the current directory, using file names generated by replacing **.c** with a descriptive suffix.

-l *name*
library option. Pass the name of a library to the linker. **cc** expands **-l***name* into **lib***name***.a** and searches **LIBPATH**.

-M *string*
Machine option. Use an alternate version of **cc0**, **cc1**, **cc1a**, **cc1b**, **cc2**, **cc3**, **as**, **lib*.a**, and **crts0.o**, named by fixing *string* between the directory name and the pass and file names.

-N[p0123sdlrt]*string*
Name option. Rename a specified pass to *string*. The letters **p0123sdlrt** refer, respectively, to **cpp**, **cc0**, **cc1**, **cc2**, **cc3**, the assembler, the linker, the libraries, the run-time start-up, and the temporary files. For example, the **-VGEM** option described below implicitly executes the option **-Nrcrtsg.o** to change the name of the run-time start-up module.

-o *name*
Output option. Rename the executable file from the default *file***.prg** to *name*.

-q[p0123s]
quit option. Terminate compilation after running the specified pass. The letters **p0123s** refer, respectively, to **cpp**, **cc0**, **cc1**, **cc2**, **cc3**, and the as-

sembler. For example, to terminate compilation after running the parser **cc0**, type **-q0**.

-Q Quiet option. Suppress all messages.

-S Suppress the object-writing and link phases, and invoke the disassembler **cc3**. This option produces an assembly-language version of a C program for examination, for example if a compiler problem is suspected. The assembly-language output file name replaces the **.c** suffix with **.s**. This is equivalent to the **-VASM** option.

-U *name*
Undefine symbol *name*. Use this option to undefine symbols that the preprocessor defines implicitly, such as the name of the native system or machine.

-V Verbose option. **cc** prints onto the standard output a blow-by-blow description of each action it takes. This option is normally used for to check the compiler if you suspect something is wrong with it; it can also be used on heavily loaded system to reassure the user that compilation is in fact proceeding.

V*string*
Variant option. Toggle (i.e., turn on or off) the variant *string* during the compilation. Options marked **Strict:** generate messages that warn of the conditions in question. **cc** recognizes the following variants:

-VASM Output assembly-language code; identical to **-S** option, above. Default is **off**.

-VCNEST Allow nested comments. Default is **off**.

-VCOMM Permit **.comm**-style data items. Default is **on**.

-VFLOAT Include floating point **printf** routines. Same as **-f** option, above.

-VGEM Use routines designed for GEM environment. This uses runtime startup routine **crtsg.o** and links in the libraries **libaes.a** and **libvdi.a**. Default is **off**.

-VGEMACC Use routines designed for a GEM desk accessory. This uses runtime startup routine **crtsd.o** and links in the libraries **libaes.a** and **libvdi.a**. Default is **off**.

-VGEMAPP Use routines designed for a GEM application. This is a synonym for **-VGEM**. Default is **off**.

-VNOOPT Turn off optimization. Default is **off**.

-VNOTRAPS
Turn off trap substitution. By default, all **gemdos**, **bios**, **xbios**, and **micro_rtx** calls are traps. By setting this option,

subroutine calls will be generated instead of traps. A trap is a single-word instruction, analogous to an interrupt; it is faster and takes up less space than an ordinary subroutine call. This option allows the user to test routines that have any of the aforementioned names. Default is **off**.

-VPSTR Put strings into the shared segment, if possible. Used to generate ROMable code. Default is **off**.

-VQUIET Suppress all messages; identical to **-Q** option. Default is **off**.

-VSBOOK Strict: note deviations from *The C Programming Language*. Default is **off**.

-VSCCON Strict: note constant conditional. Default is **off**.

-VSINU Implement struct-in-union rules instead of Berkeley-member resolution rules. Default is **off**, i.e., Berkeley rules are the default.

-VSLCON Strict: **int** constant promoted to **long** because value is too big. Default is **on**.

-VSMEMB Strict: check use of structure/union members for adherence to standard rules of C. Default is **on**.

-VSNREG Strict: register declaration reduced to auto. Default is **off**.

-VSPVAL Strict: pointer value truncated. Default is **off**.

-VSRTVC Strict: risky types in truth contexts. Default is **off**.

-VSTAT Report commands run and statistics; same as **-V** option.

-VSUREG Strict: note unused registers. Default is **off**.

-VSUVAR Strict: note unused variables. Default is **on**.

-Z Pause between passes and prompt for disk change. Used with the compiler using single-sided disks.

See Also

as, cc0, cc1, cc2, cc3, commands, cpp, ld

cc0—Definition

cc0 is the Mark Williams C's *parser*. It parses C programs using the method of recursive descent, to translate the program into a tree format.

See Also

cc, cc1, cc2, cc3, cpp

cc1—Definition

cc1 is the Mark Williams C code generator. This phase generates code from the trees created by the parser, **cc0**. The code generation is table driven, with entries for each operator and addressing mode.

See Also

cc, cc0, cc2, cc3, cpp

cc2—Definition

cc2 is the optimizer/object generator phase of Mark Williams C. It optimizes the code generated by **cc1**, and writes the object code. The Mark Williams Company compiler uses multiple optimization algorithms. One optimizes jump sequences; it eliminates common code, optimizes span-dependent jumps, and removes jumps to jumps. The other function scans the generated code repeatedly to eliminate unnecessary instructions.

See Also

cc, cc0, cc1, cc3, cpp

cc3—Definition

cc3 is the disassembler phase of Mark Williams C. It writes its output in assembly language rather than in object code. This phase is optional, and allows the user to examine the code generated by the compiler. To produce an assembly-language output of a C program, use the **-S** option on the **cc** command line; for example,

```
cc -S foo.c
```

tells **cc** to produce a file of assembly language instead of an object module.

See Also

cc, cc0, cc1, cc2, cpp

Cconin—gemdos function 1 (**osbind.h**)

Read a character from the standard input

#include <osbind.h>

long Cconin()

Cconin reads a character from the standard input and echoes it to the standard output. It returns the character read.

Example
This example gets characters from the keyboard and displays them on the screen until a **<ctrl-Z>** is typed.

```
#include <osbind.h>

main() {
	int c = 0;

	while (c != 0x1A)
		Cconout((int)(c = Cconin()));
}
```

See Also
gemdos, TOS

Cconis—**gemdos** function 11 (**osbind.h**)
Find if a character is waiting at standard input
#include <osbind.h>
int Cconis()

Cconis checks to see if characters are waiting at from the standard input. It returns -1 if a character is waiting, and zero if no character is waiting.

Example
This example displays a moving asterisk until any non-shift key is typed. **Cconis** is also demonstrated in the example for **Cauxin**.

```
#include <osbind.h>

main() {
	int x=0;
	int dir=0;

	Cconws("\033H\033f");		/* Home, cursor disabled */

	while (Cconis() == 0) {		/* Until a key is typed */
		if(dir == 0) {		/* if left to right */
			Cconws("\010 *");
			if(++x > 78)
				dir++;
		} else {		/* if right to left */
		Cconws("\010\010\033K*");	/* Back up, clear to end */
		if (--x <= 0)
			dir=0;
		}
	}
	x = Cconin();			/* Eat the character */
	Cconws("\033e");		/* Turn cursor on. */
}
```

See Also
gemdos, screen control, TOS

Cconos—**gemdos** function 16 (**osbind.h**)
Check if console is ready to receive characters
#include <osbind.h>
long Cconos()

Cconos checks to see if the console is ready to receive characters. It returns -1 if the console is ready, and 0 if it is not.

Example
This program exits with a status of 1 if the console cannot be written to; otherwise, it displays a message and exits with a status of 0.

```
#include <osbind.h>
main() {
        if (Cconos() == 0) {
                exit(1);
        }
        Cconws("The console is ready...\n\r");
        exit(0);
}
```

See Also
gemdos, screen control, TOS

Notes
As of this writing, **Cconos** always returns -1, and does no checking.

Cconout—**gemdos** function 2 (**osbind.h**)
Write a character onto standard output
#include <osbind.h>
void Cconout(*c*) int*c*;

Cconout writes character *c* onto the standard output. It returns nothing.

For information on the screen handling escape sequences used by this routine, see the entry for **screen control**.

Example
For an example of this function, see the entry for **Cauxin**.

See Also
gemdos, screen control, TOS

Cconrs—gemdos function 10 (osbind.h)
Read and edit a string from the standard input
#include <osbind.h>
**void Cconrs(*string*) char ****string*;

Cconrs reads and edits *string*, which it receives from the standard input. The first byte of *string* holds the length of the data portion of the buffer; the second byte holds the actual number of characters read; and the remainder holds the characters read, with a NUL character appended to the end.

Example
This example reads an edited string from **stdin** and writes it and its length to **stdout**. **buff[0]** is the the size of the data portion of the buffer, and **buff[1]** is the length read.

```
#include <osbind.h>
main() {
	unsigned char buff[130];

	buff[0] = 128;
	Cconrs(buff);
	printf("String '%s' is %d bytes long\n", &buff[2], buff[1]);
}
```

See Also
gemdos, TOS

Notes
<ctrl-C> aborts a program if typed in response to a **Cconrs**.

Cconws—gemdos function 9 (osbind.h)
Write a string onto standard output
#include <osbind.h>
**void Cconws(*string*) char ****string*;

Cconws writes *string* onto the standard output. It stops writing when it reads the NUL. **Cconws** returns nothing.

Example
This example writes a NUL-terminated string to **stdout**. Note the \r used with the \n.

```
#include <osbind.h>
main() {
	Cconws("This is a NUL-terminated string.\r\n");
}
```

See Also
gemdos, screen control, TOS

Notes
Note that **<ctrl-S>**, **<ctrl-Q>**, and **<ctrl-C>** act, respectively as **XON**, **XOFF**, and **abort** while **Cconws** is acting.

cd—Command
Change directory
cd *directory*

The micro-shell **msh** keeps track of the directory in which the user is currently working. If a command is not specified by a complete path name beginning with the name of the storage device on which it is kept, **msh** prefixes it with the name of the current working directory. **cd** changes the current working directory to *directory*. If no *directory* is specified, the directory named in the **$HOME** environmental variable becomes the current working directory.

For example, consider a disk on drive B:\ that has two directories: **foo** and **bar**. By definition, the *root* directory is **B:**, and **foo** and **bar** each are sub-directories of **B:**. To change to the sub-directory **foo**, you would type:

```
cd foo
```

To move from **foo** to **bar**, type the full path name of **bar**:

```
cd b:\bar
```

Note that the symbol '..' stands for a directory's *parent* directory; in this example, both **foo** and **bar** have **B:** as their parent directory. By definition, a root directory has no parent. So, to move back from **bar** to **foo**, you could type:

```
cd ..\foo
```

This first moves you from **bar** to **bar**'s parent directory, **B:**; then from the parent directory into **foo**.

See Also
commands, msh, pwd

ceil—General function (**libm.a/ceil**)
Numeric ceiling function
#include <math.h>
double ceil(*z***) double** *z***;**

ceil returns a double-precision floating point number whose value is the smallest integer greater than or equal to *z*.

Example
The following example demonstrates how to use **ceil**:

```
#include <math.h>
dodisplay(value, name)
double value; char *bname;
{
	if (errno)
		perror(name);
	else
		printf("%10g %s\n", value, name);
	errno = 0;
}

#define display(x) dodisplay((double)(x), "x")
main() {
	extern char *gets();
	double x;
	char string[64];

	for(;;) {
		printf("Enter number: ");
		if(gets(string) == 0)
			break;
		x = atof(string);

		display(x);
		display(ceil(x));
		display(floor(x));
		display(fabs(x));
		display(sqrt(x));
	}
}
```

See Also
abs, fabs, floor, frexp

char—Definition

char is a C data type. It is the smallest addressable unit of data, and it usually consists of eight bits (one byte) of storage. By definition, **sizeof char** equals one, with all other data types defined as multiples thereof. All Mark Williams compilers sign-extend **char** when it is cast to a larger data type.

See Also
byte, data formats, declarations, unsigned

character constant—Definition

A **character constant** is a constant of the form '*X*', where *X* is any printable character, and is enclosed between two apostrophes. The value of the constant is the machine value of the character it represents, whatever it might happen to be on your system.

Selected non-printable characters can also be represented as character constants by using the following escape sequences:

\0 NUL
\0*NN* octal number
\a bell
\b backspace
\f formfeed
\n newline
\r carriage return
\t tab character
\v vertical tab
\x*NN* hexadecimal number

An octal value can also be directly output by preceding the three-digit octal number with a backslash '\'; for example

```
'\065'
```

will print the letter 'A' on any machine that uses the ASCII table.

See Also
ASCII, backspace, carriage return, horizontal tab, newline, vertical tab
The C Programming Language, page 35

clearerr—STDIO macro
Present stream status
#include <stdio.h>
clearerr(*fp***) FILE ****fp*;

clearerr resets the error flag of the argument *fp*. If an error condition is detected by the related macro **ferror, clearerr** can be called to clear it.

See Also
ferror, STDIO

CLK_TCK—Manifest constant
CLK_TCK is a manifest constant that is set in the header file **time.h**. It is defined as being equivalent to the rate at which the system clock ticks. On the Atari ST, the system clock ticks 200 times per second.

See Also
time, time.h

clock—Time function (**libc.a/clock**)

Get number of clock ticks since system boot

#include <time.h>
clock_t clock()

clock returns the number of times the clock has ticked since the system was last turned on. The number of ticks per second is defined by the manifest constant **CLK_TCK**, which is declared in the header file **time.h**. Note that this value varies from computer to computer. On the Atari ST, the clock ticks every five milliseconds.

clock returns a value of the type **clock_t**; this type is defined in **time.h** as being equivalent to an **unsigned long**. Note that this value will overflow **clock_t** and be reset to zero approximately 148 days after the machine is turned on.

Example
For an example of this function, see the entry for **Pexec**.

See Also
CLK_TCK, time, time.h

close—UNIX system call (**libc.a/close**)

Close a file

close(*fd*) int *fd*;

close closes a file that is identified by the file descriptor *fd*, which was returned by **creat**, **dup**, or **open**. **close** frees the associated file descriptor. Because each program can have a limited number of open files, programs that process many files should **close** files whenever possible. Mark Williams C closes all open files automatically when a program exits.

Example
For an example of this function, see the entry for **open**.

See Also
creat, open, STDIO, UNIX routines

Diagnostics
close returns -1 if an error occurs, such as its being handed a bad file descriptor; otherwise, it returns zero.

cmp—Command

Compare bytes of two files

cmp [-ls] *file1 file2* [*skip1 skip2*]

cmp compares *file1* and *file2* character-by-character for equality. If *file1* is '-', **cmp** reads the standard input.

Normally, **cmp** notes the first difference and prints the line and character position, relative to any skips. If it encounters an end-of-file flag on one file but not on the other, it prints the message "EOF on file*n*". The following are the options that can be used with **cmp**:

-l Note each differing byte by printing the positions and octal values of the bytes of each file.

-s Print nothing, but return the exit status.

If the skip counts are present, **cmp** reads *skip1* bytes on *file1* and *skip2* bytes on *file2* before it begins to compare the two files.

See Also
commands, diff, msh

Diagnostics
The exit status is zero for identical files, one for non-identical files, and two for errors, e.g., bad command usage or inaccessible file.

Cnecin—gemdos function 8 (**osbind.h**)
Perform modified raw input from standard input
#include <osbind.h>
long Cnecin()

Cnecin reads a character from the standard input and returns it. The character is not echoed to the standard output.

Example
This example reads characters from the standard input device, changes their case, and writes them out to the standard output device until a **<ctrl-D>** character is typed.

```
#include <osbind.h>
#include <ctype.h>

main() {
	unsigned char c;

	while((c=Cnecin()) != 0x04) {
		if(isupper(c))			/* Toggle case of char */
			c = tolower(c);
		else
			c = toupper(c);
		Crawio(c);
		if(c == 0x0D)			/* If a <RETURN> */
			Crawio(0x0A);		/* Append a line feed */
	}
}
```

See Also
gemdos, screen control, TOS

Notes
This routine has been documented elsewhere as recognizing the special meanings of the characters **<ctrl-C>**, **<ctrl-S>**, and **<ctrl-Q>**; this information, however, appears to be incorrect.

commands—Overview

Mark Williams C includes a number of commands. They are listed below, with the command given on the left and a description on the right.

ar	the archiver/librarian
as	the assembler
as68toas	convert DRI to Mark Williams assembler
cat	concatenate files
cc	the compiler driver
cd	change directory
cmp	compare two files
cp	copy a file
cpp	the C preprocessor
cshconv	run under the Beckemeyer C shell
cursconf	change cursor style and position
date	print/set the system date and time
db	symbolic debugger
df	measure free space on disk
diff	compare two files
drtomw	convert from DRI to Mark Williams
echo	repeat/expand an argument
egrep	find embedded strings
exit	leave **msh**

file	determine file type
gem	run a GEM-DOS program
getcol	get a color palette entry
getpal	get color palette
getphys	get base of physical screen memory
getrez	get screen resolution
help	print help files on screen
hidemouse	hide mouse pointer
htom	redraw screen, moving from high to medium resolution
kbrate	get/set the keyboard's repeat rate
kick	force TOS to reread the floppy disk cache
ld	the linker
ls	list directory contents
ltom	redraw screen, moving from low to medium resolution
make	programming discipline
me	MicroEMACS screen editor
mf	measure free space in RAM
mkdir	create a directory
msh	the Mark Williams micro-shell
msleep	suspend processing for *n* milliseconds
mtoh	redraw screen, moving from medium to high resolution
mtol	redraw screen, moving from medium to low resolution
mv	rename a file
nm	print symbol tables
od	print an octal dump of a file
pr	format ASCII files for printing
pwd	print the current directory
rdy	create, save, and load a rebootable RAM disk
rm	remove a file
rmdir	remove a directory
rsconf	set attributes of serial (auxiliary) port
set	set a shell variable
setcol	set a palette color
setenv	set an environmental variable
setpal	set the color palette
setphys	set the physical base of the screen's memory
setprt	set attributes of parallel port
setrez	set screen resolution
show	display saved screen image
showmouse	show the mouse pointer
size	print size of a file
sleep	suspend processing for *n* seconds
snap	take a "snapshot" of the current screen image
sort	sort ASCII files
strip	strip symbol tables from objects
tail	print the end of a file

tos	run unredirected GEM-DOS program
touch	change a file's date
uniq	list/destroy duplicate lines
unset	discard a shell variable
unsetenv	discard an environmental variable
version	print/assign version number
wc	count words/lines in ASCII files

Note that many of the commands are built into **msh** itself, whereas the others are executable programs in their own right. For a list of the commands that are built into **msh**, type the command

```
set in .bin
```

Note that commands not built into **msh** must be stored in one of the directories named in the environmental variable **PATH**, so that they can be found automatically by **msh**. Note, too, that commands not built into **msh** can be run independently from the GEM desktop; in most instances, this will require that the suffix be changed from **.prg** to **.ttp**, so the command in question can receive arguments.

For more information on any of these commands, see its entry within the Lexicon.

See Also
Lexicon, msh

compound number—Definition

A **compound number** is a number that consists of two numbers of different types. In the context of C, this applies usually to floating point numbers, which are constructed of a sign bit; an exponent; and a *mantissa*, or base upon which the exponent operates.

See Also
data formats, double, float

con:—TOS device

TOS logical device for the console

TOS gives names to its logical devices. Mark Williams C uses these names, to allow the **STDIO** library routines to access these devices via TOS. **con:** is the logical device that describes the console.

Example
The following example demonstrates how to open the console device.

```
#include <stdio.h>
main(){
	FILE *fp, *fopen();
	if ((fp = fopen("con:","w")) != NULL)
		fprintf(fp,"con: enabled.\n");
	else printf("con: cannot open.\n");
}
```

See Also
aux:, prn:

Notes
con: may be spelled **con:** or **CON:**.

cos—Mathematics function (**libm.a/cos**)
Calculate cosine
double cos(*radian***) double** *radian*;

cos calculates the cosine of its argument *radian*, which must be in radian measure.

Example
For an example of this function, see the entry for **acos**.

See Also
mathematics library

cosh—Mathematics function (**libm.a/cosh**)
Calculate hyperbolic cosine
#include <math.h>
double cosh(*radian***) double** *radian*;

cosh calculates the hyperbolic cosine of *radian*, which is in radian measure.

Example
The following example demonstrates how to use **cosh**:

```
#include <math.h>
dodisplay(value, name)
double value; char *name;
{
	if (errno) perror(name);
	else printf("%10g %s\n", value, name);
	errno = 0;
}
```

```
#define display(x) dodisplay((double)(x), "x")
main() {
        extern char *gets();
        double x;
        char string[64];

        for(;;) {
                printf("Enter number: ");
                if(gets(string) == 0)
                        break;
                x = atof(string);

                display(x);
                display(cosh(x));
                display(sinh(x));
                display(tanh(x));
        }
}
```

See Also
mathematics library

Diagnostics
When overflow occurs, **cosh** returns a huge value that has the same sign as the actual result.

cp—Command

Copy a file
cp *oldfile newfile*
cp *oldfile1 ... oldfileN directory*

cp copies files. In its first form, **cp** copies the contents of *oldfile* to *newfile*, which is created if necessary. If *newfile* is a directory, **cp** copies *oldfile* to a file of the same name in directory *newfile*.

In its second form, **cp** copies each *file*, from *oldfile1* through *oldfileN*, into *directory*.

If a file is copied to itself, the result is undefined, but probably undesirable.

See Also
commands, msh, mv, wildcards

cpp—Command

C preprocessor
cpp [*option*...] [*file*...]

cpp is the C preprocessor. It performs the operations described in appendix A of *The C Programming Language* such as file inclusion, conditional code selection, constant definition, and macro definition. The **cc** command runs **cpp** as the first step in compiling a C program; and **cpp** can be run by itself.

cpp reads each input *file*, or the standard input if no file is specified, processes directives accordingly, and writes its product on the standard output. The product is a C program that is identical to the concatenated input files with preprocessor directives completed.

The following summarizes **cpp**'s options:

-D*VARIABLE*
Define *VARIABLE* for the preprocessor at compilation time. For example, the command

```
cc -DLIMIT=20 foo.c
```

tells the preprocessor to define the variable **LIMIT** to be 20. The compiled program acts as though the directive **#define LIMIT 20** were included before its first line.

-E Strip all comments and line numbers from the source code. This option is used for preprocessing assembly language or other sources, and should not be used with the other compiler phases.

-I *directory*
C allows two types of **#include** directives in a C program, i.e., **#include "file.h"** and **#include <file.h>**. The **-I** option adds directories that the preprocessor searches for files named in these directives. By default, **cpp** looks for these files in the directory named by the **INCDIR** environmental variable and the directory of the source file. For information on how to set this variable, see the Lexicon's entries for it and for **setenv**.

-o *file*
Write output into *file*. If this option is missing, **cpp** writes its output onto the standard output device, which may be redirected.

-U*VARIABLE*
Undefine *VARIABLE*, as if an **#undef** directive were included in the source program. This is used to undefine the variables that **cpp** defines by default, i.e., **GEMDOS** and **M68000**.

In addition to the directives described in *The C Programming Language* **cpp** processes the **#assert** directive. The form of this directive is **#assert** *constant-expression*. **cpp** evaluates the *constant-expression*; if its value is false (zero), **cpp** prints a diagnostic message on the console. Assertion failures are nonfatal; they do not stop compilation.

Note that **cpp**, like all other phases of the compiler, can be run on its own. Thus, **cpp** can be used to modify files that do not contain C programs. For example, assembly sources can be preprocessed with **cpp** to provide file inclusion, conditional assembly, and macro expansion. All of **cpp**'s directives (e.g., **#ifdef**) can be used. To assemble a file that contains preprocessor directives, use the following commands:

```
\lib\cpp -E file.s -o file.p
as -gxo file.o file.p
```

As noted above, the option **-E** tells **cpp** to omit the source file line number directives that it usually provides for the C compiler.

See Also
#assert, cc, cc0, cc1, cc2, cc3, #include, ld
The C Programming Language, page 86

Cprnos—**gemdos** function 17 (**osbind.h**)
Check if printer is ready to receive characters
#include <osbind.h>
long Cprnos()

Cprnos attempts to execute a "handshake" routine to see if the printer is ready to receive characters. It returns -1 if the printer is ready, and 0 if it is not.

Example
The following example demonstrates **Cprnos**.

```
#include <osbind.h>

main() {
	if(Cprnos() != 0)
		Cconws("Printer Ready.\n\r");
	else
		Cconws("Printer not ready.\n\r");
}
```

See Also
gemdos, TOS

Cprnout—**gemdos** function 5 (**osbind.h**)
Send a character to the printer port
#include <osbind.h>
void Cprnout(*c*) int *c*;

Cprnout sends the character *c* to the printer port, and returns nothing.

Example
This example writes a line to the printer.

```
#include <osbind.h>
main() {
	unsigned char *c="This is printed on the printer.\r\n";
	while (*c != '\0')
		Cprnout(*c++);
}
```

See Also
gemdos, TOS

Crawcin—gemdos function 7 (**osbind.h**)

Read a raw character from standard input
#include <osbind.h>
long Crawcin()

Crawcin reads a raw character from the standard input, and returns it to the calling program. The character is not echoed to the standard output, and the special meanings of the characters **<ctrl-C>**, **<ctrl-S>**, and **<ctrl-Q>** are ignored.

Example
This example reads characters from the standard input device, and writes characters out to the standard output device until a **<ctrl-Z>** is typed. **Crawcin** is also demonstrated in the example for **Cauxin**.

```
#include <osbind.h>
main() {
	unsigned char c;

	while((c = Crawcin()) != 0x1A) {
		Crawio(c);
		if(c == 0x0D)
			Crawio(0x0A);
	}
}
```

See Also
gemdos, TOS

Crawio—gemdos function 6 (**osbind.h**)

Perform raw I/O with the standard input
#include <osbind.h>
long Crawio(*c*) int *c*;

Crawio performs raw I/O with the standard input. If the argument *c* equals 0xFF, then a character is read from the standard input and returned. If *c* does not equal 0xFF, then it is written onto the standard output.

Example
This example reads characters from the standard input device, and writes them on the standard output device until a **<ctrl-Z>** is typed.

```
#include <osbind.h>
main() {
	unsigned char c;

	while ((c = Crawio(0xFF)) != 0x1A) {
		Crawio(c);
		if (c == 0x0D)
			Crawio(0x0A);
	}
}
```

See Also
gemdos, TOS

creat—UNIX function (**libc.a/creat**)
Create/truncate a file
int creat(*file*, *mode*) char ****file*; **int *mode*;

creat creates a new *file* or truncates an existing *file*. It returns a file descriptor that identifies *file* for subsequent system calls. If *file* already exists, its contents are erased. **creat** ignores its *mode* argument. This argument exists for compatibility with implementations of **creat** under the UNIX system and other operating systems.

Example
For an example of how to use this routine, see the entry for **open**.

See Also
STDIO, UNIX routines

Diagnostics
If the call is successful, **creat** returns a file descriptor. It returns **-1** if it could not create the file, typically because of insufficient system resources, or nonexistent path.

crts0.o—C runtime startup
C runtime startup

crts0.o is the runtime startup routine for C programs compiled into TOS object format.

crts0 provides an efficient, portable environment for C programs. When used with the micro-shell **msh**, it can provide arbitrarily long argument lists, easily configured environmental parameters, and redirection of up to six input/output channels.

The runtime startup module, **crts0.o**, is the first code executed when your program is run. As its first action, it parses the environment string list passed by TOS into a vector of string pointers. This vector is saved in the the variable **external char **environ**, for the use of the library routine **getenv()**, and passed as the parameter **char *envp[]**, for the information of the function **main()**.

If the environment vector contains a parameter named **ARGV**, then the run time start-up assumes that the program was executed by **msh** (or by some other program that agrees that programs should have arguments), and that the remainder of the environment vector is an argument vector that should be passed as the parameter **char *argv[]** to the function **main()**.

If the parameter **ARGV** has a value, such as **ARGV=CCAP??**, then the value should consist of characters from the set **[CAPF?]**. The characters describe the origin of the system file handles as Console, Auxiliary port, Printer port, File, or unknown. The runtime startup stores the value of ARGV, if it exists, into the external variable **char *_iovector** for the use of the routines that emulate the functions of the COHERENT operating system.

If no **ARGV** parameter is found in the environment, then the run time start-up program assumes that the program was executed by a simple **GEMDOS Pexec()**. The buffer **cmdtail** is parsed to form the argument vector for **main()**. **ARGV[0]** is supplied by the external variable **char _cmdname[]**, which should be supplied by your program, or it will be set to ? by the library. The value of the variable **_iovector** will be set to the default **CCAP?????????????????????????;**.

See Also
argv, runtime startup, system

crtsd.o—C runtime startup
C runtime startup, GEM environment

crtsd.o is the runtime startup routine for C programs that designed to be used as a GEM desktop accessory.

crtsd.o can be invoked on the **cc** command line in one of two ways. First, the **-VGEMACC** option will include it, well as the libraries **libaes.a** and **libvdi.a**. Second, **crtsd.o** can be used independently of the libraries by using the **name** option **Nrcrtsd.o**.

See Also
argv, cc, crts0.o, crtsg.o, runtime startup

crtsg.o—C runtime startup
C runtime startup, GEM environment

crtsg.o is the runtime startup routine for C programs that use the GEM VDI and AES routines.

crtsg.o is a simple but fast runtime startup routine. Note the following differences from the default runtime startup **crts0.o**:

1. **ARGV**, **ARGC**, and **ENVP** are all set to zero.
2. **getenv** is not enabled; this means programs that use **crtsg.o** will cannot read environmental parameters.
3. **stderr** will send error messages to the auxiliary port rather than to the console.

crtsg.o can be invoked on the **cc** command line in one of two ways. First, the **-VGEM** option will include it, well as the libraries **libaes.a** and **libvdi.a**. Second, **crtsg.o** can be used independently of the libraries by using the **name** option Nrcrtsg.o.

See Also
argv, cc, crts0.o, crtsd.o, runtime startup

cshconv—Command

Run a Mark Williams C program under the Beckemeyer C shell
cshconv *filename* [*argument* ...]

cshconv allows users of the Beckemeyer C shell to call the Mark Williams compiler and utilities, and programs that are compiled with Mark Williams C.

The user must compile the source file **cshtomwc.c** into one of two executable forms, depending on the version of the C shell being used. Users with environmental C shells (i.e., those in which the **setenv** command exists) should use the default version by compiling

```
cc -o cshconv cshtomwc.c
```

In this version, the environment is reparsed from scratch because the Mark Williams run-time start-up truncates the environmental parameter list when it finds a statement of the form **ARGV=**. This environment is then loaded into **argv[]**, and the number of arguments is loaded into **argc**. Next, the I/O vectors are set to the corresponding input, output, and error files and, finally, **cshconv** calls for the execution of the specified Mark Williams C program.

Users with pre-environmental C shells (i.e., those in which the **setenv** command does not exist) must edit the source file **cshtomwc.c** to set the default environmental variables to values that match the ones on their system. In particular, the user must alter the environmental strings of the following block of code in cshtomwc.c:

```
char *defenv[] = {
        "PATH=.bin,,a:\\bin,b:\\bin",
        "SUFF=,.prg,.tos,.ttp",
        "TMPDIR=a:\\tmp",
        "INCDIR=a:\\include",
        "LIBPATH=a:\\lib,b:\\lib",
        0
};
```

Note that double backslashes are necessary to prevent the C compiler from interpreting the backslash as a C escape character.

When these changes are made, compile the source code under the microshell **msh** with the following command line:

```
cc -DOLDCSHELL -o cshconv cshtomwc.c
```

In this version, the environmental pointer points to the user-defined environment **defenv[]**, the I/O vectors are set to the corresponding input, output, and error file handles, and **cshconv** calls for the execution of the specified Mark Williams program.

Caveats
Note that the following restrictions are placed on programs run under **cshconv**:

1. You must give a complete file name for the program you wish to run.
2. You must set the environment strings that Mark Williams C requires to run properly, i.e., **INCDIR**, **LIBPATH**, **PATH**, **SUFF**, and **TMPDIR**.
3. The environments must be in Mark Williams C format, with the exception of **PATH**, which can use the C shell's list separator.
4. If you redirect **stdin** to a file or a pipe, the program you call will not find EOF on **stdin**.
5. If you redirect **stdout** into a file, all material written to **stderr** will end up there as well.

See Also
argv, **commands**

ctime—Time function (**libc.a/ctime**)
Convert system time to an ASCII string
char *ctime(*timep*) time_t *timep*;

ctime converts the system's internal time to a form that can be read by humans. It takes a pointer to the internal time type **time_t**, as defined in the header file **time.h**, and returns a fixed-length string in the form:

```
Thu Mar 14 11:12:14 1987\n
```

Note that **time_t** is defined as being equivalent to a **long**. Mark Williams C defines the internal system time as being equivalent to the number of seconds that have passed since January 1, 1970 00h00m00s GMT.

ctime is implemented as a call to **localtime** followed by a call to **asctime**.

Example
For an example of this function, see the entry for **asctime**.

See Also
time

Notes
ctime returns a pointer to a statically allocated data area that is overwritten by successive calls.

ctype—Overview

The **ctype** macros and functions test a character's *type*, and can transform some types of characters into other types. They are:

isalnum	test if a number
isalpha	test if alphabetic
isascii	test if ASCII
iscntrl	test if a control character
isdigit	test if a numeric digit
islower	test if lower case
isprint	test if printable
ispunct	test if punctuation mark
isspace	test if a tab, space, or return
isupper	test if upper case
toascii	change to ASCII character
tolower	change to lower case
_tolower	change to lower case
toupper	change to upper case
_toupper	change to upper case

These are defined in the header file **ctype.h**, and each is described further in its own Lexicon entry.

Example
The following example demonstrates the macros **isalnum**, **isalpha**, **isascii**, **iscntrl**, **isdigit**, **islower**, **isprint**, **ispunct**, **isspace**, and **toupper**. It changes a text file to all upper-case characters, and prints some information about the type of characters it contains.

```
#include <ctype.h>
#include <stdio.h>
main(){
	FILE *fp;
	int filename[20];
	int ch;
	int alnum = 0;
	int alpha = 0;
	int control = 0;
	int printable = 0;
	int punctuation = 0;
	int space = 0;

	printf("Enter name of text file to examine: ");
	gets(filename);
	if ((fp = fopen(filename,"r")) != NULL) {
		while ((ch = fgetc(fp)) != EOF) {
			if(isascii(ch)) {
				if(isalnum(ch)) alnum++;
				if(isalpha(ch)) alpha++;
				if(iscntrl(ch)) control++;
				if(isprint(ch)) printable++;
				if(ispunct(ch)) punctuation++;
				if(isspace(ch)) space++;
				putchar(islower(ch) ? toupper(ch) : ch);

			} else {
				printf("%s is not ASCII.\n", filename);
				exit(1);
			}
		}
		printf("%s has the following:\n", filename);
		printf("%d alphanumeric characters\n", alnum);
		printf("%d alphabetic characters\n", alpha);
		printf("%d control characters\n", control);

		printf("%d printable characters\n", printable);
		printf("%d punctuation marks\n", punctuation);
		printf("%d white space characters\n", space);
		exit(0);
	} else printf("Cannot open '%s'.\n", filename);
}
```

See Also
ctype.h, Lexicon

ctype.h—Header file
Header file for data tests
#include <ctype.h>

ctype.h is a header file that holds the texts of the macros described in the overview entry **ctype**.

See Also
ctype, header file

cursconf—Command
Set the cursor's configuration
cursconf *task* [*rate*]

cursconf is a command that uses the **xbios** function **Cursconf** to alter the cursor's configuration. It can take one or two arguments. *task* indicates what to do, as follows:

0	hide the cursor
1	show the cursor
2	set the cursor to blink
3	set the cursor not to blink
4	set the cursor to blink at *rate*
5	return the current blink rate

If *task* is set to **4**, then you should give **cursconf** the argument *rate*, which sets the rate at which the cursor blinks. *rate* should be set to proportions of the normal rate parameter, which is one half of the normal cycle time (60 Hz for the color mmonitor, 70 Hz for the monochrome monitor, and 50 Hz for monitors set in PAL mode). For example, setting *rate* to 35 will cause the cursor to blink twice a second on a monochrome monitor.

See Also
commands, TOS

Cursconf—**xbios** function 21 (**osbind.h**)
Get or set the cursor's configuration
#include <osbind.h>
#include <xbios.h>
int Cursconf(*function*, *rate***) int** *function*, *rate*;

Cursconf gets or sets the cursor's configuration. *function* is an integer that tells TOS to do one of the following:

0	Hide the cursor
1	Show the cursor
2	Set the cursor to blink
3	Set the cursor not to blink
4	Set the cursor to blink at *rate*
5	Return the current blink rate

rate, as noted above, sets the rate at which the cursor blinks. It is used to set the rate only if *function* is set to 4; otherwise it is ignored. *rate* should be set to proportions of the normal rate parameter, which is one-half the normal cycle time (60 Hz for the color monitor, 70 Hz for the monochrome monitor, and 50 Hz for monitors set in PAL mode). For example, setting *rate* to 35 will cause the cursor to blink twice a second on a monochrome monitor.

Note that **Cursconf** returns the current cursor blink rate when *function* is set to 5; otherwise, it returns a meaningless value.

Example
This example creates a utility for the micro-shell **msh** that can turn off or turn on the cursor's blink mode. Because this example uses **argv**, do *not* compile it with the **-VGEM** option. For an example of using **Cursconf** in a GEM program, see the entry for **\auto**.

```
#include <osbind.h>
#define JUNK 50                    /* Place-holding value that has no meaning */

main(argc, argv)
int argc;
char *argv[];
{
	if ((argc-1) == 0) {
		Cursconf(3, JUNK);
		exit(0);
	}

	else if (((argc-1) == 1) && (strcmp(argv[1], "blink") == 0)) {
		Cursconf(2, JUNK);
		exit(0);
	}

	else {
		printf("Usage: cursor [blink]\n");
		exit(1);
	}
}
```

See Also
screen control, TOS, xbios

daemon—Definition

A **daemon**, in the context of C programming, is a process that is designed to perform a particular task or control a particular device without requiring the intervention of a human operator. Under the COHERENT system, for example, the line printer is controlled by the line printer daemon **lpd**. The daemon periodically checks when a file has been queued for printing; when it detects one, it starts up the printer and passes the file to it without needing human intervention.

See Also
process

data formats—Definition

Mark Williams Company has written C compilers for a number of different computers; these computers have different architectures and define data formats in different ways. Note that these formats may not be compatible with code produced by other processors or other C compilers.

The following table gives the sizes, in **chars**, of the data types as they are defined by various microprocessors.

Type	*8086 SMALL*	*8086 LARGE*	*Z8001*	*Z8002*	*68000*	*PDP11*	*VAX*
char	1	1	1	1	1	1	1
double	8	8	8	8	8	8	8
float	4	4	4	4	4	4	4
int	2	2	2	2	2	2	4
long	4	4	4	4	4	4	4
pointer	2	4	4	2	4	2	4
short	2	2	2	2	2	2	2

Mark Williams C places some alignment restrictions on data. Byte ordering is set by the microprocessor; see **byte ordering** for more information.

See Also
byte ordering, C language, data types, declarations, double, float, memory allocation

data types—Definition

The following describes the data types recognized by Mark Williams C. The left-hand column below gives compound type specifiers mentioned in *The C Programming Language*; the right-hand column gives additional specifiers recognized by Mark Williams C.

short int	**unsigned short int**
long int	**unsigned short**
unsigned int	**unsigned long int**
long float	**unsigned long**
	unsigned char

The first pair of additional **unsigned** terms have the same meaning, as do the second pair. The type **unsigned char** is an addition to the language. If used in arithmetic expressions, it is automatically cast to **unsigned int**.

See Also
C language, char, data formats, double, float, int, long, pointer, short, unsigned

date—Command

Print/set the date and time
date [**-i**] [[*yymmdd*]*hhmm*[*.ss*]]

date prints the time of day and the current date, including the time zone. If an argument is given, the system's current time and date is changed, as follows:

yy	year (00-99)
mm	month (01-12)
dd	day (01-31)
hh	hour (00-23)
mm	minute (00-59)
ss	seconds (00-59)

For example, typing

```
date 860512141233
```

sets the date to May 12, 1986, and the time to 2:12:33 P.M. Note that at least *hh* and *mm* must be specified—the rest are optional. The command

```
date -i
```

displays the current date and time in the form acceptable to **date** as input.

The library time conversion routines used by **date** look for the environmental variable **TIMEZONE**, which specifies local time zone and daylight saving time information in the format described in **ctime**.

See Also
commands, ctime, msh, time, TIMEZONE

dayspermonth—Time function (**libc.a/dayspermonth**)

Return number of days in a given month
#include <time.h>
int dayspermonth(*month, year*) int *month, year*;

dayspermonth returns the number of days in a given month of a given year A.D. *month* is the number of the month in question, from one to 12. *year* is the year A.D. in which *month* appears. Note that there is no year 0.

See Also
isleapyear, time, time.h

db—Command

Assembler-level symbolic debugger
db [-fkor] [*mapfile*] [*datafile*]

db is an assembler-level debugger. It allows you to run object files and executable programs under trace control, run programs with embedded breakpoints, and dump and patch files in a variety of forms.

What is **db**?
db is a symbolic debugger, which means that it works with the symbol tables that the compiler builds into the object files it generates. For that reason, it will not work with programs that have had their symbol tables stripped out. Likewise, because **db** is designed to work on the level of assembly language, the user needs a working knowledge of 68000 assembly language and microprocessor architecture.

Invoking **db**
To invoke **db**, type its name, plus the options you want (if any) and the name of the files with which you will be working. *mapfile* is an object file that supplies a symbol table. *datafile* is the executable program to be debugged. If possible, **db** accesses *datafile* with write permission.

The following options to the **db** command specify the format of *program*:

-f Map *program* as a straight array of bytes.

-k The *kernel* option. This allows a user to debug all of the Atari ST's memory. The default *symbolfile* in **os.sym** defines the documented locations in low memory. The *symbolfile* is used to provide symbolically interpreted output. All of the ST's memory, from address 0 in RAM to the end of the ROM, is available for display or patching. Note that this option allows the user to perform a post-mortem on programs that crash: use the command **:r** to display the registers and the command **:f** to display the fault identifier in the process dump area. These commands are described in detail below.

-o *program* is an object file. If *mapfile* is given, it is another object file that provides the symbol table.

-r Read file only, even though you can write into it. This is used to give a file additional protection.

Commands and addresses
db executes commands that you give it from the standard input. A command usually consists of an *address*, which tells **db** where in the program to execute the command; and then the command name and its options, if any.

An address is represented by an *expression*, which can be built out of one or more of the following elements:

* The '.', which represents the current address. When an address is entered, the current address is set to that location. The current address can be advanced by typing **<RETURN>**.

* The name of a register. **db** recognizes the register names d0 through d7, a0 through a7, pc, and sp. Typing the name of a register displays its contents.

* The names of global symbols and symbolic addresses can be used in place of the addresses where they occur. This is useful when setting a breakpoint at the beginning of a subroutine.

* An integer constant, which can be used in the same manner as a global symbol. The default is decimal; a leading **0** indicates octal and 0x indicates hexadecimal.

* The following binary operators can be used:

+	addition
-	subtraction
*	multiplication
/	integer division

  All arithmetic is done in **longs**.

* The following unary operators can be used:

~	complementation
-	negation
*	indirection

  All operators are supported with their normal level of precedence. Parentheses '()' can be used for binding.

Display commands
The following commands merely display information about *program*. The symbol '.' represents the *address*, which defaults to the current display address if omitted. *count* defaults to one.

address[,count]?[format]

Display the *format count* times, starting at *address*. The *format* string consists of one or more of the following characters:

^	reset display address to '.'
+	increment display address
-	decrement display address
b	byte
c	**char**; control and non-**chars** escaped
C	like 'c' except '\0' not displayed
d	decimal
f	**float**
F	**double**
i	machine instruction, disassembled
l	**long**
n	output '\n'
o	octal
p	symbolic address
s	string terminated by '\0', with escapes
S	string terminated by '\0', no escapes
u	**unsigned**
w	word
x	hexadecimal
Y	time

The format characters **d**, **o**, **u**, and **x**, which specify a numeric base, can be followed by **b**, **l**, or **w**, which specify a datum size, to describe a single datum for display. A format item may also be preceded by a count that specifies how many times the item is to be applied. Note that *format* defaults to the previously set format for the segment (initially **i** for instructions). Except where otherwise noted, **db** increments the display address by the size of the datum displayed after each format item.

Execution commands

In the following commands, *address* defaults to the address where execution stopped, unless otherwise specified; *count* and *expr* default to 1. *commands* is an arbitrary string of **db** commands, terminated by a newline. A newline may be included by preceding it with a backslash '\'.

[address]=

Print *address* in current display base. *address* defaults to '.'. The command = assigns values to locations in the traced process. The size of the assigned value is determined from the last display format used. You can and set display the registers of the traced process, just like any other address in the traced process. Thus,

```
d0?l
d0=0
```

displays the value of register **d0** as a **long**, and then sets **(long) d0** to zero. To display the character in the low byte of d0, use:

```
d0+3?c
```

To set the low byte of d0 to ASCII **<esc>**, use

```
d0+3=033
```

[address[,count]]=value[,value[,value]...]
Patch the contents starting at *address* to the given *value*. *address* defaults to '.'. Up to ten *values* can be listed.

? Print verbose version of last error message.

[address] **:a**
Print *address* symbolically. *address* defaults to '.'.

*[address]***:b***[commands]*
Set breakpoint at *address*; save *commands* to be executed when breakpoint is encountered. *commands* defaults to **.:a\ni+.?i\n:x**.

:br *[commands]*
Set breakpoint at return from current routine. The defaults are the same as for **:b**, above.

[address] **:c**
Continue execution from *address*.

[address] **:d[r][s]**
Delete breakpoint at *address*. If optional **r** or **s** is specified, delete return or single-step breakpoint. *address* defaults to '.'.

*[address]***:e***[commandline]*
Begin traced execution of the object file at *address* (default, entry point). The *commandline* is parsed and passed to the traced process. **argv[0]** must be typed directly after **:e** if supplied. For example, **:e3 foo bar baz** sets **argv[0]** to **3**, **argv[1]** to **foo**, **argv[2]** to **bar**, and **argv[3]** to **baz**. Quotation marks, apostrophes, and redirection are parsed as by **msh**, but special characters '?*[]' and shell punctuation '(){}|;' are not.

:f Print type of fault which stopped the traced process.

*[expr]***:l***[filename]*
The log option. If *expr* is non-zero, open *filename* as a log file; if *expr* is zero, close the currently open log file. **db** echoes all its responses into the open log file.

[expr] **:n**
Set default numeric display base to *expr*: **8**, **10**, and **16** indicate, respectively, octal, decimal, and hexadecimal.

:p Display breakpoints.

[expr] :q
If *expr* is nonzero, quit the current level of command input (see :x). *expr* defaults to 1. End of file is equivalent to **:q**.

:r Display registers.

[address],*[count]*:**s[c]***[commands]*
Single-step execution starting at *address*, for *count* steps, executing *commands* at each step. *commands* defaults to .?i.

After a single-step command, **<RETURN>** is equivalent to **.,1:s[c]**. If the optional **c** is present, **db** turns off single-stepping at a subroutine call and turns it back on upon return.

[depth] :t
Print a call traceback to *depth* levels. If *depth* is 0 (default), unwind the whole stack.

[expr] :x
If *expr* is nonzero, read and execute commands from the standard input up to end of file or **:q**. *expr* defaults to 1.

Example of the commands
The following example shows how each **db** command can be used to examine an executable file. It uses the following C program, called **count.c**, which counts the number of ASCII characters in a file:

```
#include <ctype.h>
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
	FILE *fp;
	int result, ch;

	if ((fp = fopen(argv[1], "r")) != NULL) {
		while ((ch = fgetc(fp)) != EOF) {
			if(isascii(ch)) result++;
			else fatal(argv[1], "Not ASCII");
		}
		printf("%s: %d characters\n", argv[1], result);
	}
	else fatal(argv[1], "Cannot open");
}
```

```
fatal(filename, message)
char *filename, *message;
{
        printf("%s: %s\n", filename, message);
}
```

For purposes of this example, **count.prg** will be used to count the characters in a text file called **tester**. Its contents are as follows:

```
Sonnet 30

When to the sessions of sweet silent thought
I summon up remembrance of things past,
I sigh the lack of many a thing I sought,
And with the old woes new wail my dear time's waste:
Then can I drown an eye, unused to flow,
For precious friends hid in death's dateless night,
And weep afresh love's long since canceled woe,
And moan the expense of many a vanished sight:
Then can I grieve at grievances foregone,
And heavily from woe to woe tell o'er
The sad account of fore-bemoaned moan,
Which I new pay as if not paid before.
But if the while I think on thee, dear friend,
All losses are restored, and sorrows end.
```

To begin, compile **count.c** by typing the following command:

```
cc -V count.c
```

When the program has been compiled, invoke **db** with the following command:

```
db count.prg
```

Addressing commands

As noted above, **db** offers several different ways to set the *address*, or the position within the program that you are examining. One way is by entering a variable name. Type **printf**. **db** replies:

```
printf_         link     a6, $0x0
```

Another way to set the address is by entering an absolute address. Type **0600**. **db** replies:

```
main_+0x70      jsr      printf_.l
```

The symbol '.' (dot) echoes the current address. Type a dot; **db** will reply:

```
main_+0x70      jsr      printf_.l
```

which is, as expected, identical to the previous reply.

The equal sign '=' displays the absolute address of any variable that precedes it. To see how this works, type **printf=**. **db** replies:

```
0x1C6
```

which is the address of **printf**.

Instructions can be shown, beginning at a named address. The *format* must be introduced with a question mark '?'. For example, **.,?i** shows the current line in the instruction space, as indicated by the format string "?i". When this command is typed, **db** replies:

```
main_+0x70      jsr      printf_.l
```

Now, show the next five instructions from the current point by typing **.,5?i**. **db** replies:

```
main_+0x70      jsr      printf_.l
main_+0x76      lea.l    0xA(a7), a7
main_+0x7A      bra      main_+0x92
main_+0x7C      move.l   $0x24FB, -(a7)
main_+0x82      movea.l  0xA(a6), a0
```

Once a format is set, it remains the default until the format is reset with another format string. For example, the command **printf,20** prints 20 instructions, beginning with **printf**; the format ?i remains in effect. Type this command. **db** replies:

```
printf_         link     a6, $0x0
printf_+0x4     pea.l    0x8(a6)
printf_+0x8     move.l   $_stdout_, -(a7)
printf_+0xE     jsr      sprintf_+0x3C.l
printf_+0x14    addq.w   $0x8, a7
printf_+0x16    unlk     a6
printf_+0x18    rts
fprintf_        link     a6, $0x0
fprintf_+0x4    pea.l    0xC(a6)
fprintf_+0x8    move.l   0x8(a6), -(a7)
fprintf_+0xC    jsr      sprintf_+0x3C.l
fprintf_+0x12   addq.w   $0x8, a7
fprintf_+0x14   unlk     a6
fprintf_+0x16   rts
sprintf_        link     a6, $0xFFE6
sprintf_+0x4    pea.l    0xFFE6(a6)
sprintf_+0x8    move.w   $0x8000, -(a7)
sprintf_+0xC    move.l   0x8(a6), -(a7)
sprintf_+0x10   jsr      _stropen_.l
sprintf_+0x16   lea.l    0xA(a7), a7
```

Typing **,20** prints the next 20 instructions, beginning from where the previous command left off. When you type this, **db** replies:

```
sprintf_+0x1A   pea.l     0xC(a6)
sprintf_+0x1E   pea.l     0xFFE6(a6)
sprintf_+0x22   jsr       sprintf_+0x3C.l
sprintf_+0x28   addq.w    $0x8, a7
sprintf_+0x2A   pea.l     0xFFE6(a6)
sprintf_+0x2E   clr.w     -(a7)
sprintf_+0x30   jsr       fputc_.l
sprintf_+0x36   addq.w    $0x6, a7
sprintf_+0x38   unlk      a6
sprintf_+0x3A   rts
sprintf_+0x3C   link      a6, $0xFF96
sprintf_+0x40   movem.l   d7/a4/a5, (a7)
sprintf_+0x44   move.l    0xC(a6), 0xFFFC(a6)
sprintf_+0x4A   movea.l   0xFFFC(a6), a0
sprintf_+0x4E   move.l    (a0), d0
sprintf_+0x50   movea.l   d0, a4
sprintf_+0x52   addq.l    $0x4, 0xFFFC(a6)
sprintf_+0x56   move.b    (a4)+, d0
sprintf_+0x58   ext.w     d0
sprintf_+0x5A   move.w    d0, d7
```

Finally, the command **:a** displays an address symbolically. The default is the current address. Type this command; **db** replies:

```
sprintf_+0x5A
```

which is the same address as that of the last instruction in the previous example; in other words, the address advanced as the command was processed.

To reset and display the address at the point where the instruction **fatal** is, type **fatal:a**. **db** replies:

```
fatal_
```

Execution commands
db allows you to execute portions of your program; this is done by setting *breakpoints*, or points where execution stops. Breakpoints are set with the command **:b**. Set breakpoints at **main**, **printf**, and **fatal** as follows:

```
main:b
printf:b
fatal:b
```

The command **:p** displays the current breakpoints:

```
00000110 (main_) i+.?i\n:x\n
000001C6 (printf_) i+.?i\n:x\n
000001A6 (fatal_) i+.?i\n:x\n
```

Now, begin execution with the command **:e**. As noted above, **:e** can take arguments; the arguments correspond to the elements in the array **argv**; in this example, use the following command to pass as an argument the name of the text file **tester**, whose text is given above:

```
:e tester
```

db replies:

```
main_           link      a6, $0xFFF8
```

The program has executed up to the first breakpoint, set on **main**. The command *n*:t performs a call traceback on the stack to *n* levels; the default is zero, which means to unwind the whole stack. Type:

```
:t
```

db replies:

```
0x035E10  main_(0x0002, 0x0003, 0x561A, 0x0003, 0x55F6)
```

Note that the address of **main_** has changed because the program is now loaded into memory.

The command **:c** continues execution of the program to the next breakpoint. When you type it, **db** will reply:

```
printf_         link      a6, $0x0
```

Perform another stack traceback by typing **:t**. **db** replies:

```
0x035DF6  printf_(0x0003, 0x52C8, 0x0003, 0x2D61, 0x0272)
0x035E10  main_(0x0002, 0x0003, 0x561A, 0x0003, 0x55F6)
```

Type **:c** to continue execution to the next breakpoint. **db** replies:

```
tester: 626 characters
Child process terminated (0)
```

The first line shows the output of of the program; in this case, a message that the file **tester** has 626 characters. The message about the child process indicates that the program has finished execution and exited; the number in parentheses is the value that **exit** returned to the calling program (in this case, **db**).

Now, type **:p** to print a list of the breakpoints. **db** makes no reply because no breakpoints remain set; all have been erased as the program executed.

Finally, quit the debugging session by typing **:q**.

Example of debugging
This example shows how to use **db** to track down a simple bug. It uses the following program, called **bug.c**:

```
#include <stdio.h>

main() {
        output(NULL, stdout);   /* send number to stdout */
}
```

```
output(number, fp)
int number;
FILE *fp;
{
	fprintf(fp, "The number is %d.\n", number);
}
```

This program passes a number to the routine **output**, which writes it into the named file or device. The program illustrates a common error in C programming.

To begin, compile **bug.c** by using the following command:

```
cc -V bug.c
```

You should see no error messages during compilation. When compilation is finished, try running the program. Instead of writing its message on the standard output device, the program should generate a bus error (as indicated by the appearance of two "bombs" on the screen).

Now, invoke **db** with the following command:

```
db bug.prg
```

One way to approach this problem is to set a breakpoint on **main** and step through the program. The following sets the breakpoint:

```
main:b
```

The **:e** commands performs traced execution at the program's entry point. When you type **:e**, **db** replies as follows:

```
main_           link     a6, $0x0
```

The **:s** commands performs single-step execution. The following commands follows the program through five steps:

```
5:s
```

db replies as follows:

```
main_+0x4       move.l   $_stdout_, -(a7)
main_+0xA       clr.l    -(a7)
main_+0xC       jsr      output_.l
output_         link     a6, $0x0
output_+0x4     move.w   0x8(a6), -(a7)
```

The command **:t** allows you to perform a stack traceback. **db** replies as follows:

```
0x0343F6  output_+0x4(0x0000, 0x0000, 0x0003, 0x3AC6)
0x034406  main_+0x12(0x0001, 0x0003, 0x3C14, 0x0003, 0x38F0)
```

The number in parentheses indicate what is being passed on the stack to the routine. Each four-digit number represents a machine word (two bytes). The first line indicates the source of the trouble: the routine **output** is being passed *four* words, when it is defined as receiving three: an **int** and a pointer. The problem, of course, is that **main** passed **output** two pointers, NULL and **stdout**; on the 68000, unlike on some other processors, NULL and zero are *not* identical. (For more information on this topic, see the Lexicon entries for **pointer**, NULL, and **data formats**.)

Another, simpler approach to this problem is to enter **db** and then immediately set a breakpoint with **:b**, perform a traced execution with **:e** followed by a stack traceback with the **:t** command. **db** replies as follows:

```
0x03435C  fputc_+0x32(0x0054, 0x0000, 0x0003)
0x0343D4  sprintf_+0x74(0x0000, 0x0003, 0x0003, 0x43F0)
0x0343E4  fprintf_+0x12(0x0000, 0x0003, 0x0003, 0x3A4A, 0x0000)
0x0343F6  output_+0x18(0x0000, 0x0000, 0x0003, 0x3AC6)
0x034406  main_+0x12(0x0001, 0x0003, 0x3C14, 0x0003, 0x3BF0)
```

Again, the display shows how **output** was passed an improper argument, which made it pass an improper argument to **fprintf**.

See Also
commands, od

Dcreate—gemdos function 57 (osbind.h)

Create a directory
#include <osbind.h>
**long Dcreate(*path*) char ****path*;

Dcreate creates a directory; it returns zero if the directory was created successfully, one if it was not. *path* points to the subdirectory's path name, which should be a NUL-terminated string. **Dcreate** returns a negative value when an error occurs.

Example
The following example uses **Dcreate** to create a directory.

```
#include <osbind.h>
extern int errno;

main(argc, argv) int argc; char **argv; {
	int status;

	if (argc < 2) {
		Cconws("Usage: Dcreate pathname\r\n");
		Pterm(1);
	}
```

```
        if ((status = Dcreate(argv[1])) != 0) {
                errno = -status;
                perror("Dcreate failure");
                Pterm(1);
        }
        Cconws("Directory ");
        Cconws(argv[1]);
        Cconws(" created.\r\n");
        Pterm0();
}
```

See Also
gemdos, TOS

Ddelete—gemdos function 58 **(osbind.h)**
Delete a directory
#include <osbind.h>
**long Ddelete(*path*) char ****path*;

Ddelete deletes a directory; it returns zero if the deletion was successful, non-zero if the deletion failed. *path* points to the subdirectory's path name, which must be a NUL-terminated string.

Example
The following example deletes a directory

```
#include <stdio.h>
#include <osbind.h>
#define EACCESS (-36)       /* Access violation error code    */

extern int errno;

main(argc, argv) int argc; char **argv; {
        int status;

        if (argc < 2) {
                Cconws("Usage: Ddelete pathname\r\n");
                Pterm(1);
        }
```

```
        if ((status = Ddelete(argv[1])) != 0) {
                if (status == EACCESS) {
                        fprintf(stderr, "\nDirectory %s contains files\n",
                                argv[1]);
                } else {
                        errno = -status;
                        perror("Ddelete failure");
                }
                Pterm(1);
        }
        printf("Directory %s deleted.\n", argv[1]);
        Pterm0();
}
```

See Also
gemdos, TOS

declarations—Overview

Mark Williams C recognizes the following as legal declarations for data types:

char
double
enum
float
int
long
long float
long int
short
short int
struct
union
unsigned char
unsigned int
unsigned long
unsigned long int
unsigned short
unsigned short int
void

The following pairs of terms are synonymous; the more commonly used term is given on the *right*:

long float	**double**
long int	**long**
short int	**short**
unsigned long int	**unsigned long**
unsigned short int	**unsigned short**

See Also
C language, data formats, data types, Lexicon

#define—Definition

#define tells the C preprocessor **cpp** to define a variable as a manifest constant. For example, the instruction

```
#define MAXARGS 9
```

tells **cpp** to replace every instance of the string **MAXARGS** with the numeral **9** throughout the program. (Note that numerals are manifest constants by definition.)

#define instructions are very useful because their judicious use allows a programmer to write code that more easily understood, maintained, and enhanced. With them, a programmer can modify a major parameter throughout his program just by changing a single line of code. They also allow a programmer to use a variable name that suggests the function of the parameter it represents; for example, the name **MAXARGS** within a program obviously refers to the maximum number of arguments, whereas the numeral **9** could refer to nearly anything.

See Also
cpp, manifest constant

desk accessory—Definition

A **desk accessory** is a program that is loaded by TOS into the GEM desktop when it is booted. The desktop gives each accessory its own icon, keeps it resident in memory, and gives you direct access to it. When you build a menu, the routine **menu_bar** will automatically include the name of the accessory when it builds the list displayed under the **desk** entry.

To compile a desk accessory with Mark Williams C, use the option **-VGEMACC**. This will automatically link in the special run-time start-up routine **crtsd.o**, and otherwise perform all that is needed to create a desk accessory. Note that all desk accessories must have the suffix **.acc**. Therefore, to compile the program **foo.c** into a desk accessory, use the following form of the **cc** command:

```
cc -VGEMACC -o foo.acc foo.c
```

To install a desk accessory, move the compiled program into your system's root directory. If you have a hard disk, it should be in directory **c:**; otherwise, it should be in the root directory of the disk with which you boot TOS. Do *not* place it into the directory **\auto**; this will cause all manner of unpleasant things to happen. The program will be loaded into the desktop automatically when you reboot your system.

Because of their specialized nature, desk accessories restrict the number and variety of programming tools you can use with them. Note the following:

* Do not use any **stdio** routines
* Do not use the **malloc** routines found in **libc.a**
* Do not use **exit**, **Pterm**, **Pterm0**, or **Ptermres**
* Do not return from **main**

Also, you should keep the following in mind as you write your accessory:

* If you use **rsc_load**, remember to use **rsc_unload** before you give up control, if possible.

* Do not use **evnt_timer** calls: use **evnt_multi** instead.

Example
The following example is the digital clock desk accessory. It is a public domain program written by Jan Gray in 1986. It displays a digital clock on the GEM screen.

```
#include <gemdefs.h>
#include <osbind.h>

/* Macros to extract times from TOS time format */
#define MINS(t) ((t >> 5) & 0x3f)
#define HRS(t) (t >> 11)
#define DIGIT(d) ((d) + '0')

/* Some manifest constants */
#define NO_WINDOW -1                    /* no window opened */
#define NO_POSITION -1                  /* window has no position yet */
#define TEMPLATE "hh:mm AM"
#define TEMP_LEN 8

/* A window descriptor, used in this example */
typedef struct window {
        int id;                         /* GEM window ID from wind_create() */
        int x;                          /* X origin on the screen for the window */
        int y;                          /* Y origin on the screen for the window */
        int w;                          /* width of the window */
        int h;                          /* height of the window */
} Window;

/*
 * Main program: Initialize the desk accessory and call the
 * routine that maintains the clock.
 */
main() {
        int menuID;                     /* Where this is on the desk menu */
        extern int gl_apid;             /* The application ID for this DA */
```

```
        appl_init();
        menuID = menu_register(gl_apid, "  Digital Clock");
                                                /* Register as a desk accessory */
        events(menuID);                         /* Call event loop routine */
                                                /* Never returns! */
}

/* Loop processing events; wake up every 30 seconds to update time. */
events(menuID)
int menuID;                                     /* Where this accessory is in the desk menu */
{
        Window wind;                            /* Place to keep track of the window */
        int event;                              /* Which event from evnt_multi */
        int msgbuf[8];                          /* Message buffer */
        int ret;                                /* Dummy return buffer */

/* Initialize the clock window, which doesn't exist yet. */
        wind.id = NO_WINDOW;                    /* No window yet */
        wind.x = NO_POSITION;                   /* No position for non-existent window */

        for (;;) {                              /* Until reboot */
                event = evnt_multi(MU_MESAG | MU_TIMER,/* Wait for either */
                        0, 0, 0,                /* a message or a */
                        0, 0, 0, 0, 0,          /* timer event */
                        0, 0, 0, 0, 0,
                        msgbuf, 30000, 0,       /* 30 seconds */
                        &ret, &ret, &ret, &ret, &ret, &ret);

/* Event has been received, now what is it? */
                if (event & MU_MESAG) switch (msgbuf[0]) {
                case AC_OPEN:
                        if (msgbuf[4] == menuID)
                                if (wind.id == NO_WINDOW)
                                        openWindow(&wind);
                                else
                                        wind_set(wind.id, WF_TOP, 0, 0, 0, 0);
                        break;

                case AC_CLOSE:
                        if (msgbuf[3] == menuID)
                                wind.id = NO_WINDOW;
                        break;

                case WM_CLOSED:
                        if (msgbuf[3] == wind.id)
                                closeWindow(&wind);
                        break;
```

```
		case WM_MOVED:
			wind_set(wind.id, WF_CURRXYWH, msgbuf[4], msgbuf[5],
				msgbuf[6], msgbuf[7]);
			wind.x = msgbuf[4]; wind.y = msgbuf[5];
			wind.w = msgbuf[6]; wind.h = msgbuf[7];
			break;

		case WM_NEWTOP:
		case WM_TOPPED:
			if (msgbuf[3] == wind.id)
				wind_set(wind.id, WF_TOP, 0, 0, 0, 0);
			break;
		}

		if (event & MU_TIMER && wind.id != NO_WINDOW)
			update(&wind);
	}
}

/*
 * Update the title on the clock window to reflect current GEMDOS
 * time.
 */
update(wp)
Window *wp;				/* The clock window descriptor */
{
	static char time[] = TEMPLATE;	/* Time string buffer */
	unsigned t = Tgettime();	/* the current DOS time */
	unsigned hrs = HRS(t);		/* extract hours */
	unsigned hrs12 = (hrs % 12 == 0) ? 12 : hrs % 12;
	unsigned mins = MINS(t);	/* extract minutes */

/*
 * Create time string for window title...
 * Do things the hard way: sprintf() would spend too much memory.
 */
	time[0] = (hrs12 >= 10) ? DIGIT(1) : ' ';
	time[1] = DIGIT(hrs12 % 10);
	time[3] = DIGIT(mins / 10);
	time[4] = DIGIT(mins % 10);
	time[6] = (hrs < 12) ? 'A' : 'P';
	wind_set(wp->id, WF_NAME, time, 0, 0);/* Set window title to time */
}
```

```
/*
 * Create and open a window just big enough to hold the time on its title bar
 * and the CLOSER box.
 */
openWindow(wp)
Window *wp;                                     /* Window descriptor */
{
        int workW;                              /* work area width */
        int workH;                              /* work area height */
        int ret;                                /* Dummy return buffer */

        if (wp->id == NO_WINDOW) {              /* If there is no window */
                if (wp->x == NO_POSITION) {     /* If there is no position */

/*
 * Position the clock in the center of the screen.  This is a hack to
 * determine the size and position of the window.
 */
                        graf_handle(&wp->w, &ret, &ret, &wp->h);
                        wp->w *= TEMP_LEN + 3;
                        wind_get(0, WF_WORKXYWH, &wp->x, &wp->y,
                                &workW, &workH);
                        wp->x += (workW - wp->w) / 2;
                        wp->y += (workH - wp->h) / 2;
                }

/* Create a window with name, closer and moveable */
                wp->id = wind_create(NAME|CLOSER|MOVER,
                        wp->x, wp->y, wp->w, wp->h);
                wind_open(wp->id, wp->x, wp->y, wp->w, wp->h);
                update(wp);
        }
}

/* Remove the time window from the screen */
closeWindow(wp)
Window *wp;                                     /* Window descriptor */
{
        if (wp->id != NO_WINDOW) {              /* Only if there is a window */
                wind_close(wp->id);             /* close that window */
                wind_delete(wp->id);            /* delete that window */
                wp->id = NO_WINDOW;             /* remember it's gone */
        }
}
```

See Also

crtsd.o, TOS

df—Command

Measure free space on disk

df [-a] *device*

df measures the amount of free space left on a floppy disk, on a logical device on a hard disk, or on a RAM disk. *device* is the name of the device you wish to check; for example, to check the amount of space left on the disk in drive A:, type:

```
df a:
```

The default device is the one you are logged into.

The option **-a** prints the amount of space left on all devices.

See Also
commands, mf, msh

Dfree—**gemdos** function 54 (**osbind.h**)
Get the location of free space on a drive
#include <osbind.h>
void Dfree(*fs, drive***) long** *fs***[4]; int** *drive***;**

Dfree retrieves information about free space on a disk drive, and writes it into the arguments *fs* and *drive*, which it keeps. *fs* points to an array of four **unsigned longs** that hold, respectively, the amount of free space on a drive, the number of clusters on the drive, the sector size in bytes, and the cluster size in sectors. *drive* is the number of the disk drive itself, with zero indicating the default drive, one indicating drive A, etc.

Example
This example displays disk statistics for the default drive.

```
#include <osbind.h>

struct disk_info {
	unsigned long di_free;		/* free allocation units */
	unsigned long di_many;		/* how many AUs on disk */
	unsigned long di_ssize;		/* sector size */
	unsigned long di_spau;		/* sectors per AU */
};

main() {
	long fs;
	long fb;
	int dd;
	long ts;
	long tb;

	struct disk_info disk;
```

```
        dd = Dgetdrv();
        Dfree(&disk, dd+1);
        fs = disk.di_free*disk.di_spau;
        ts = disk.di_spau*disk.di_many;
        fb = fs * disk.di_ssize;
        tb = ts * disk.di_ssize;

        printf("Disk %c: has %d bytes free in %d sectors\n",
              dd+'A', fb, fs);
        printf("from total of %d bytes in %d sectors (cluster size %d)\n",
              tb, ts, disk.di_spau*disk.di_ssize);
}
```

See Also
gemdos, TOS

Dgetdrv—gemdos function 25 (**osbind.h**)
Find which disk drive is the current drive
#include <osbind.h>
int Dgetdrv()

Dgetdrv returns an integer that indicates the current drive: 0 corresponds to drive A, and so on through 15 corresponding to drive P.

Example
This example prints the default drive.

```
#include <osbind.h>
main() {
        printf("'%c:' is the current default drive.\n",
              (char) Dgetdrv() + 'A');
}
```

See Also
Dsetdrv, gemdos, TOS

Dgetpath—gemdos function 71 (**osbind.h**)
Get the current directory name
#include <osbind.h>
long Dgetpath(*buffer, drive*) char *buffer; int drive;

Dgetpath gets the name of the current directory. *buffer* points to the area where the buffer name is to be stored. *drive* holds a number that indicates the disk drive to be examined, as follows: 0, the default drive; 1, drive A; etc.

Example
This example prints the current path name and device string.

```
#include <osbind.h>

main() {
	int drv;
	char pathbuf[66];                        /* Path buffer */
	char *buf;

	buf = pathbuf;
	*buf++ = (drv=Dgetdrv())+'a';            /* Get drive */
	*buf++ = ':';                            /* d: */
	Dgetpath(buf, drv);                      /* Rest of path */
	printf("Current path is %s\n", pathbuf );  /* Display it */
}
```

See Also
Dsetpath, gemdos, TOS

diff—Command

Summarize differences between two files

diff [-b] [-c *symbol***]** *file1 file2*

diff compares *file1* with *file2*, and summarizes the changes needed to turn *file1* into *file2*.

Two options involve input file specification. First, the standard input may be specified in place of a file by entering a hyphen '-' in place of *file1* or *file2*. Second, if *file1* is a directory, **diff** looks within that directory for a file that has the same name as *file2*, then compares *file2* with the file of the same name in directory *file1*.

The default output script has lines in the following format:

```
1,2 c 3,4
```

The numbers *1,2* refer to line ranges in *file1*, and *3,4* to ranges in *file2*. The range is abbreviated to a single number if the first number is the same as the second. The letter 'c' indicates that lines 1,2 of *file1* should be *changed* to lines 3,4 of *file2*. **diff** then prints the text from each of the two files. Text associated with *file1* is preceded by '< ', whereas text associated with *file2* is preceded by '> '.

The following summarizes **diff**'s options.

-b Ignore trailing blanks and treat more than one blank in an input line as a single blank. Spaces and tabs are considered to be blanks for this comparison.

-c *symbol*
Produce output suitable for the C preprocessor **cpp**; the output contains **#ifdef**, **#ifndef**, **#else**, and **#endif** lines. *symbol* is the string used to build the **#ifdef** statements. If you define *symbol* to the C preprocessor

cpp, it will produce *file2* as its output; otherwise, it will produce *file1*. Note that this option does *not* work for files that already contain **#ifdef**, **#ifndef**, **#else**, and **#endif** statements.

See Also
commands, egrep

Diagnostics
diff's exit status is 0 when the files are identical, 1 when they are different, and 2 if a problem was encountered (e.g., could not open a file).

difftime—Time function (**libc.a/difftime**)
Return difference between two times
#include <time.h>
double difftime(*time1, time2***) time_t** *time1, time2***;**

difftime calculates the difference, in seconds, between *time1* and *time2*.

Both arguments are of type **time_t**, which is the current system time, and which is defined in the header file **time.h**. Note that the function **time** returns the current time in this format.

Mark Williams C defines the current system time as being the number of seconds since January 1, 1970, 0h00m00s GMT.

See Also
time, time.h

directory—Definition
A **directory** is a function that maps names to files; in other words, it associates the names of a file with their locations on the mass storage device. Under some operating systems, directories are also files, and can be handled like a file.

Directories allow files to be organized on a mass storage device in a rational manner, by function or owner. Note that the documentation for TOS uses the term "folder" as a synonym for "directory".

See Also
file, msh

Dosound—xbios function 32 (**osbind.h**)
Start up the sound daemon
#include <osbind.h>
#include <xbios.h>
void Dosound(*buffer***) char** **buffer***;**

Dosound starts up a daemon to control the sound generator. *buffer* points to buffer that holds the commands and arguments to be passed to the daemon.

Each command consists of an eight-bit hexadecimal number followed by one or more characters; the commands are as follows:

0x00-0x0F
: Each of these commands is followed by a one-character argument; each writes its argument into the appropriate register in the GI sound generator, with 0x00 corresponding to register 0, 0x01 to register 1, and so on. For a fuller explanation of what each register governs in the sound register, see the entry for **Giaccess**.

0x80 This takes a one-character argument and writes it into the temporary register.

0x81 This command takes three one-character arguments. It takes the character that had been loaded into a temporary register with the **0x80** command, loads it into a sound generator register, and controls its execution. The first argument is the number of the register into which the previously stored character is to be loaded. The second argument is a two's-complement number that is added to the contents of the temporary register. The third argument is an end-point value. The instruction that was loaded is executed continually, once each update, and the contents of the temporary register are incremented; this process ends when the value stored in the temporary register equals that of the end-point value.

0x82-0xFF
: Each of these commands takes a one-byte argument. If the argument is zero, sound processing is halted. If the argument is greater than zero, it is taken to indicate the number of timer ticks (each tick being 20 milliseconds long) that must pass until the next sound process is performed. In effect, these commands can set how long a tone is sustained.

Example
This example generates an interesting series of sounds. Type a key *after* the bell sounds.

```
#include <osbind.h>

char noise[]={
	0xFF, 0x50,		/* Delay a while... */
	0x00, 0xF6,		/* Load reg 0 (Channel A freq, fine) */
	0x01, 0x02,		/* Load reg 1 (Channel A freq, coarse) */
	0x02, 0xDE,		/* Load reg 2 (Channel B freq, fine) */
	0x03, 0x01,		/* Load reg 3 (Channel B freq, coarse) */
	0x04, 0x3F,		/* Load reg 4 (Channel C freq, fine) */
	0x05, 0x01,		/* Load reg 5 (Channel C freq, coarse) */
	0x06, 0x00,		/* Load reg 6 (Noise period) */
```

```
        0x07, 0xF8,         /* Load reg 7 (Voice enable) */
        0x08, 0x10,         /* Load reg 8 (Channel A volume) */
        0x09, 0x10,         /* Load reg 9 (Channel B volume) */
        0x0A, 0x10,         /* Load reg A (Channel C volume) */
        0x0B, 0x00,         /* Load reg B (Env period fine tune E) */
        0x0C, 0x30,         /* Load reg C (Env period coarse tune E) */
        0x0D, 0x09,         /* Load reg D (Env shape/cycle) */
        0xFF, 0x30,         /* Delay */
        0x00, 0x00,         /* Load reg 0 (Channel A freq, fine) */
        0x01, 0x01,         /* Load reg 1 (Channel A freq, coarse) */
        0x07, 0x3E,         /* Load reg 7 (Voice enable) */
        0x08, 0x0B,         /* Load reg 8 (Channel A vol) */

        0x09, 0x00,         /* Load reg 9 (Channel B vol) */
        0x0A, 0x00,         /* Load reg A (Channel C vol) */
        0x80, 0x01,         /* Init temp register */
        0x81, 0x00, 0x01, 0xFF,
                            /* Loop defined... */
        0x01, 0x02,         /* Next step down */
        0x80, 0x01,         /* Init temp register again */
        0x81, 0x00, 0x01, 0xFF,
                            /* Loop again */
        0x07, 0x3F,         /* Disable voices... */
        0xFF, 0x40,         /* Delay 40 ticks... */
        0x00, 0x34,         /* Load reg 0 (Channel A freq, fine) */
        0x01, 0x00,         /* Load reg 1 (Channel A freq, coarse) */

        0x02, 0x00,         /* Load reg 2 (Channel B freq, fine) */
        0x03, 0x00,         /* Load reg 3 (Channel B freq, coarse) */
        0x04, 0x00,         /* Load reg 4 (Channel C freq, fine) */
        0x05, 0x00,         /* Load reg 5 (Channel C freq coarse) */
        0x06, 0x00,         /* Load reg 6 (Noise period) */
        0x07, 0xFE,         /* Load reg 7 (Voice enable) */
        0x08, 0x10,         /* Load reg 8 (Channel A vol) */
        0x09, 0x00,         /* Load reg 9 (Channel B vol) */
        0x0A, 0x00,         /* Load reg A (Channel C vol) */
        0x0B, 0x00,         /* Load reg B (Env period fine tune E) */
        0x0C, 0x10,         /* Load reg C (Env period coarse tune E) */
        0x0D, 0x09,         /* Load reg D (Env shape/cycle) */
        0xFF, 0x00          /* Terminate delay timer */
};

main() {
        Dosound( noise );               /* Make some noise... */
        while ( Cconis() == 0 )         /* Loop until user types a key */
                Cconws("Listen... ");
        Cconin();                       /* Get the key. */
        Dosound( noise );               /* Make some noise again */
}
```

See Also
daemon, Giaccess, TOS, xbios

double—Definition

A **double** is the data type that encodes a double-precision floating-point number. On most machines, **sizeof(double)** is defined as four machine words, or eight **chars**. Programmers who wish to write portable code should *not* use routines that depend on a **double** being 64 bits long. Different formats are used to encode **doubles** on various machines. These formats include IEEE, DECVAX, and BCD (binary coded decimal) as mentioned above; they are described in the entry for **float**.

See Also
data formats, declarations, float, portability

drtomw—Command

Convert from DRI to Mark Williams format
drtomw [-f] *file ...*

drtomw converts an object, an executable object, or an archive from DRI to Mark Williams format. It writes the converted file into a temporary file, which it then writes over the original file; this will fail if the disk with the input files is write-protected or if the input file is set as read-only. The option **-f** forces conversion despite a possible error condition, as described below.

drtomw generates messages to indicate to the user the type of file given as input, whether object file or archive. Normally, the format of a file cannot be distinguished easily by its contents; therefore, **drtomw** distinguishes file format by the suffix to the file name: relocatable objects should the suffix **.o**, whereas executable objects should have any other extension or no extension at all.

When working with a DRI archive, **drtomw** first converts the archive into a Mark Williams object archive, and then converts all of the object files within it to Mark Williams object files. The archive will still need a ranlib header, which may be added by using the command:

ar rs *archname*.a ranlib.sym

drtomw converts DRI executeable files to Mark Williams format. This involves appending a Mark Williams format header to the end of the file. If characters are present beyond the end of the relocation bytes of the executeable file, **drtomw** reports this and aborts the conversion unless you use the -f (force) flag.

See Also
as, as68toas, commands

Drvmap—bios function 10 (**osbind.h**)

Get a map of the logical disk drives

#include <osbind.h>
#include <bios.h>
long Drvmap();

Drvmap returns a bit map of the system's logical configuration of disk drives. In this map, bit 0 corresponds to drive A, bit 1 to drive B, etc.

Example

```
#include <osbind.h>
main() {
	long drivemap;
	int drv;
	long drvmsk=1;
	drivemap = Drvmap();
	puts("Drives on system:\n");

	for(drv = 0 ; drv < 16 ; drv++) {
		if(drvmsk & drivemap)
			printf("\tdrive %c:\n", (drv+'A'));
		drvmsk <<= 1;
	}
}
```

See Also

bios, bit map, TOS

Dsetdrv—gemdos function 14 (**osbind.h**)

Make a drive the current drive

#include <osbind.h>
long Dsetdrv(*drive*) int *drive*;

Dsetdrv makes *drive* the current disk drive. *drive* can be any integer between 0 and 15, with 0 indicating drive A, 1 indicating drive B, and so on through 15 indicating drive P. **Dsetdrv** returns a bit map of the drive configuration, with bits 0 through 15 indicating drives A through P, respectively; setting a bit to 1 indicates that the respective disk drive is present on the system.

Example

This example sets the default drive to **B:**. Upon exiting, the default drive is reset to A:.

```
#include <osbind.h>
#define DRIVE_A 0
#define DRIVE_B 1
#define DRIVE_C 2
#define E_DRIVE (-46L) /* Invalid Drive Specified */
main() {
	long drivemap;

	if((drivemap=Dsetdrv(DRIVE_B)) < 0) {
		if(drivemap == E_DRIVE)
			printf("Invalid drive (%c:) specified.\n",
				(DRIVE_B + 'A'));
		else
			printf("GEMDOS error %ld\n", drivemap);
	} else {
		int drv;
		long drvmsk=1;

		printf("Current drive is '%c:'. Others are:\n",
			(DRIVE_B + 'A'));
		for(drv = 0 ; drv < 16 ; drv++) {
			if(drvmsk & drivemap)
				printf("\tdrive %c:\n", (drv+'A'));
			drvmsk <<= 1;
		}
	}
}
```

See Also
Dgetdrv, Drvmap, gemdos, TOS

Notes
The **msh** built-in function **pwd** and **cd** maintain their own idea of the current drive. Programs, like the example, which reset the current drive render the shell's data invalid. A **cd** to a completely specified path will fix this.

Dsetpath—gemdos function 59 (osbind.h)

Set the current directory
#include <osbind.h>
long Dsetpath(*path***) char ****path***;**

Dsetpath sets the current directory; it returns 0 if the directory could be set, and non-zero if it could not. *path* points to the directory's path name, which must be a NUL-terminated string.

Example
This example allows the user to set and display the default path, or get the current path string for device specified. If **drv** equals -1, it uses the default drive and returns a pointer to the path buffer.

```
#include <osbind.h>
char *getpath(pathbuf,drv)
char *pathbuf;
int drv; {
	char *buf;

	buf = pathbuf;			/* Target buffer */
	if (drv < 0)			/* If drive is default */
		drv=Dgetdrv();		/* get default drive no. */
	*buf++ = drv+'A';		/* Put drive letter in string */
	*buf++ = ':';			/* d: */
	Dgetpath(buf, drv+1);		/* get the rest of the path */
	return(pathbuf);		/* Return the buffer address */
}

/*
 * Allow default directory to be changed.
 */

main(argc, argv) int argc; char **argv; {
	char path[80];
	char *dst;
	char *src;

	if(argc < 2) {			/* No new path? display old */
		Cconws("Current path is ");
		Cconws(getpath(path,-1));
		Cconws("\r\n");
		Pterm0();		/* Then exit. */
	}
	Cconws("Old path was ");
	Cconws(getpath(path,-1));
	Cconws("\r\n");

	dst = src = argv[1];		/* Get new path */

	while ( *src != '\0' ) {	/* Scan for device */
		if ( *src++ == ':' ) {	/* If found, set device */
			int drv;	/* Move pointer past ":" */

			drv = src[-2];
			if(drv > '`')
				drv -= 'a';
			    else
				drv -= 'A';
			if(drv >= 0 && drv <= 15)
				Dsetdrv(drv);
			dst = src;
			break;
		}
	}
```

```
        if (*dst != '\0') {
                if ( Dsetpath(dst) != 0 ) {
                        Cconws("Setpath failed, Path is ");
                        Cconws(getpath(path,-1));
                        Cconws("\r\n");
                        Pterm(1);
                }
        }
        Cconws("Path now set to ");
        Cconws(getpath(path,-1));
        Cconws("\r\n");
        Pterm0();
}
```

See Also
Dgetpath, Dsetdrv, Dgetdrv, gemdos, TOS

Notes
The **msh** functions **pwd** and **cd** maintain their own idea of the current path. Programs, like the example, which reset the current drive tender the shell's data invalid. A **cd** to a completely specified path will fix this.

dup—UNIX system call (libc.a/dup)

Duplicate a file descriptor
dup(*fd*) int *fd*;

dup duplicates the existing file descriptor *fd*, and returns the new descriptor. The returned value is the smallest file descriptor that is not already in use by the calling process. *fd* must be less than six under TOS.

Example
The following example duplicates a file descriptor.

```
main() {
        int fd, result;
        fd = 2;
        if ((result = dup(fd)) != -1)
                printf("file descriptor duplicated successfully \n");
        else printf("duplication unsuccessful \n");
}
```

See Also
STDIO, UNIX routines

Diagnostics
dup returns a number less than zero when an error occurs, such as a bad file descriptor or no file descriptor available.

dup2—UNIX system call (**libc.a/dup2**)
Duplicate a file descriptor
dup2(*fd*, *newfd***) int** *fd*, *newfd***;**

dup2 duplicates a file descriptor. Unlike its cousin **dup**, **dup2** allows the requesting process to specify a new file descriptor *newfd*, rather than having the system select one. If *newfd* is already open, the system closes it before assigning it to the new file. **dup2** returns the duplicate descriptor. Under TOS, *fd* must be greater than five, and *newfd* greater than six.

See Also
STDIO, UNIX routines

Diagnostics
dup2 returns a number less than zero when an error occurs, such as a bad file descriptor or no file descriptor available.

echo—Command

Repeat/expand an argument

echo [-n] [*argument* ...]

echo prints each *argument* on the standard output, placing a space between each *argument*. It appends a newline to the end of the output unless the **-n** flag is present.

If *argument* is a **msh** variable, **echo** will expand it before printing it. For example, if you type

```
set esc=<esc>
set cls=${esc}E ; echo $cls
```

where **<esc>** indicates the escape character, **echo** will send the characters **<esc>E** to your terminal, which will clear the screen and home the cursor.

See Also

commands, msh

ecvt—General function (**libc.a/ecvt**)

Convert floating point numbers to strings

char *ecvt(*d, w, dp, signp***) double** *d*; **int** *w*, **dp*, **signp*;

ecvt converts *d* into a NUL-terminated ASCII string of numerals that is *w* characters wide; it rounds the last digit and returns a pointer to the result. On return, **ecvt** sets *dp* to point to an integer that indicates the location of the decimal point relative to the beginning of the string, to the right if positive, to the left if negative; and it sets *signp* to point to an integer that indicates the sign of *d*, zero if positive and nonzero if negative. **ecvt** performs conversions within static string buffers that are overwritten by each execution.

See Also

fcvt, frexp, gcvt, ldexp, modf, printf

edata—Linker-defined symbol

extern int edata[];

edata is the location after the shared and private data segments. It is defined by the linker when the linker binds the program together for execution. The value of **edata** is merely an address. The location to which this address points contains no known value, and may be an illegal memory location for the program. The value of **edata** does not change while the program is running.

Example
For an example of this function, see the entry for **memory allocation**.

See Also
end, etext

egrep—Command
Extended pattern search
egrep [*option* ...] [*pattern*] [*file* ...]

egrep searches each *file* for occurrences of *pattern* (also called a regular expression). If no *file* is specified, it searches the standard input. Normally, it prints each line matching the *pattern*.

The simplest *patterns* accepted by **egrep** are ordinary alphanumeric strings. **egrep** can also process *patterns* that include the following wildcard characters:

^ Match beginning of line, unless it appears immediately after '[' (see below).

$ Match end of line.

* Match zero or more repetitions of preceding character.

. Match any character except newline.

[*chars*]
Match any one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

[^*chars*]
Match any character *except* one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

c Disregard special meaning of character *c*.

| Match the preceding pattern *or* the following pattern. For example, the pattern **cat|dog** matches either **cat** or **dog**. A newline within the *pattern* has the same meaning as '|'.

+ Match one or more occurrences of the immediately preceding pattern element; it works like '*', except it matches at least one occurrence instead of zero or more occurrences.

? Match zero or one occurrence of the preceding element of the pattern.

(...) Parentheses may be used to group patterns. For example, **(Ivan)+** matches a sequence of one or more occurrences of the four letters 'I' 'v' 'a' or 'n'.

Because the metacharacters '*', '?', '$', '(', ')', '[', ']', and '|' are also special to the micro-shell **msh**, patterns that contain those characters must be quoted by enclosing *pattern* within double quotation marks.

The following lists the available options:

-b With each output line, print the block number in which the line started (used to search file systems).

-c Print how many lines match, rather than the lines themselves.

-e The next argument is *pattern* (useful if the pattern starts with '-').

-f The next argument is a file that contains a list of patterns separated by newlines; there is no *pattern* argument.

-h When more than one *file* is specified, output lines are normally accompanied by the file name; **-h** suppresses this.

-l Print the name of each file that contains the string, rather than the lines themselves.

-n The line number in the file accompanies each line printed.

-s Suppress all output, just return status.

-v Print a line only if the pattern is *not* found in the line.

-y Lower-case letters in the pattern match lower-case *and* upper-case letters on the input lines. A letter escaped with '/' in the pattern must be matched in exactly that case.

See Also
commands

Diagnostics
egrep returns an exit status of 0 for success, 1 for no matches, and 2 for error.

Notes
egrep uses a deterministic finite automaton (DFA) for the search. It builds the DFA dynamically, so it begins doing useful work immediately. This means that **egrep** is considerably faster than other, earlier pattern-searching commands, on almost any length of file.

end—Linker-defined symbol

extern int end[];

end is the location after the uninitialized data segment; it is defined by the linker when the linker binds the program together for execution. The value of **end** is merely an address. The location to which it points contains no known value, and may be illegal memory locations for the program. The value of end does not change while the program is running.

Example
For an example of this function, see the entry for **memory allocation**.

See Also
edata, etext

enum—Definition

An **enum** declaration is a data type whose syntax resembles those of the **struct** and **union** declarations. **enum** declares a type and a set of identifiers that can be used as values for objects of the declared type. For example,

```
enum opinion {yes, maybe, no} guess;
```

declares an enumerated type **opinion** with three values: **yes**, **no**, and **maybe**. It also declares a variable of type **opinion** enum **guess**. **guess** may only have a value of either **yes**, **no**, or **maybe**. As with **a struct** or **union** declaration, the tag (**opinion** in this example) is optional; if present, it may be used in subsequent declarations. After the above declaration, the statement

```
register enum opinion *op;
```

declares a register pointer to an object of type **opinion**.

All identifiers in an enumeration declaration must be distinct from other identifiers in the program. The identifiers act as constants and may appear wherever constants are appropriate. Mark Williams C assigns values to the identifiers from left to right, normally beginning with 0 and increasing by 1. The values often are **ints**, although if the range of values is small enough, the **enum** will be an **unsigned char**. If an identifier in the declaration is followed by an equal sign and a constant, the identifier is assigned the given value, and subsequent values increase by 1 from that value.

To add **enum** to the formal definition of C, amend the list of type-specifiers in Appendix A of *The C Programming Language* to include **enum**-*specifier*, and add the following syntax:

```
      enum-specifier:
        enum { enum-list }
        enum identifier { enum-list }
        enum identifier
  enum-list:
        enumerator
        enum-list , enumerator
  enumerator:
        identifier
        identifier = constant-expression
```

See Also
declarations

environ—Definition
extern char **environ;

environ is a pointer set by the run-time start-up routine. It points to the environment vector, which is equal to the third argument passed to **main**, **char *envp[]**; this, in turn, is the handle that the function **getenv** uses to find the environment.

Example
For an example of how this element is used in a C program, see the entry for **memory allocation**.

See Also
envp

envp—Definition
Variable passed to **main**
char *envp[];

envp is an abbreviation for **env**ironmental **p**arameter. It is the traditional name for a pointer to an array of string pointers passed to a C program's **main** function, and is by convention the third argument passed to **main**.

Example
For an example of this function, see the entry for **memory allocation**.

See Also
argc, **argv**, **main**

EOF—Manifest constant
EOF is an acronym for "end of file"; it is the manifest constant defined in **stdio.h** that is used to signal that the end of a file has been reached.

To signal EOF to a program reading from the console keyboard under TOS, you should type **<ctrl-Z>** followed by **<RETURN>** on a line by itself. **<ctrl-Z>** as an EOF signal is implemented by the **read** routine. Programs that use TOS calls to read the console must implement an EOF signal themselves.

Example

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

See Also
manifest constant, stdio.h

errno—UNIX data (**crts0.o**)
External integer for return of error status
extern int errno;

errno is an external integer that is set to the negative value of any error status returned by TOS to the UNIX system call emulation routines. The routine **perror()** or the array of string **sys_errlist** may be used to provide a textual translation of **errno.**

Mathematical functions also use **errno** to indicate classifications of errors on return. It is defined within the header file **errno.h**. Because not every function uses **errno**, it should be polled only in connection with those functions that document its use and the meaning of the various status values.

The error codes returned by TOS are listed in the entry for **error codes**, below.

See Also
errno.h, error codes, mathematics library, perror, UNIX routines

errno.h—Header file
Error numbers used by **errno** function
#include <errno.h>

errno.h is a header that defines and describes the error numbers returned by **errno**.

See Also
errno, header file, TOS

error codes—Definition
The following lists the error codes returned by TOS:

BIOS-level errors:

AE_OK	0L	OK, no error
AERROR	-1L	basic, fundamental error
AEDRVNR	-2L	drive not ready

AEUNCMD	-3L	unknown command
AE_CRC	-4L	CRC error
AEBADRQ	-5L	bad request
AE_SEEK	-6L	seek error
AEMEDIA	-7L	unknown media
AESECNF	-8L	sector not found
AEPAPER	-9L	no paper
AEWRITF	-10L	write fault
AEREADF	-11L	read fault
AEGENRL	-12L	general error
AEWRPRO	-13L	write protect
AE_CHNG	-14L	media change
AEUNDEV	-15L	unknown device
AEBADSF	-16L	bad sectors on format
AEOTHER	-17L	insert other disk

GEMDOS-level errors:

AEINVFN	-32L	invalid function number
AEFILNF	-33L	file not found
AEPTHNF	-34L	path not found
AENHNDL	-35L	too many open files no handles left
AEACCDN	-36L	access denied
AEIHNDL	-37L	invalid handle
AENSMEM	-39L	insufficient memory
AEIMBA	-40L	invalid memory block address
AEDRIVE	-46L	invalid drive was specified
AEXDEV	-48L	cross device rename not documented
AENMFIL	-49L	no more files

Miscellaneous error codes:

AERANGE	-64L	range error
AEINTRN	-65L	internal error
AEPLFMT	-66L	invalid program load format
AEGSBF	-67L	setblock failure due to growth restrictions

See Also
errno, errno.h, perror

etext—Linker-defined symbol
extern int etext[];

etext is the location after the shared and private text (code) segments; it is defined by the linker when it binds the program together for execution. The value of **etext** is merely an address. The location to which it points contains no known value, and may be illegal memory locations for the program. The value of **etext** does not change while the program is running.

Example
For an example of this function, see the entry for **memory allocation**.

See Also
edata, end, malloc

evnt_button—AES function (**libaes.a/evnt_button**)
Await a specific mouse button event
#include <aesbind.h>
int evnt_button(*clicks, button, state, record***)**
int *clicks, button, state*; **Mouse** *record*;

evnt_button is an AES routine that waits for a specified button event. *clicks* is the number of clicks to await. *button* is the number of the button to await, counting from the left, as follows: 0x1, leftmost button; 0x2, second from left; 0x4, third from left; etc.

state is the button state to await: zero indicates up and one indicates down. **evnt_button** returns zero if an error occurred, and a number greater than zero if one did not.

record points to where **evnt_button** writes the result of a button event. It is declared to be of type **Mouse**, which is a structure of four pointers to integers that is declared in the header file **aesbind.h**, as follows:

x	X coordinate of mouse pointer
y	Y coordinate of mouse pointer
b	button state when event occurred
k	state of control, alt, and shift keys, OR'd together: 0x0: all keys up 0x1: right shift key down 0x2: left shift key down 0x4: control key down 0x8: alt key down

evnt_button returns the number of times the button entered the desired state.

Example
For an example of this routine, see the entry for **v_circle**.

See Also
AES, TOS

Notes
Note that this routine can be told only to wait for one specified button event, e.g., for button 1 alone. If you attempt to tell it to wait for button 1 *or* button 2, it will react as if you told it to wait for button 1 *and* button 2, i.e., for both buttons to be pressed simultaneously.

evnt_dclick—AES function (**libaes.a/evnt_dclick**)

Get/set double-click interval

#include <aesbind.h>

int evnt_dclick(*speed, getset*) int *speed, getset*;

evnt_dclick is an AES routine that gets or sets the mouse's double-click speed. *speed* is the double-click speed, from zero through four, with zero being the slowest and four the fastest. It is ignored if *getset* is set to zero. *getset* is a flag: zero tells AES to return the current speed, and one tells it to set the new speed. **evnt_dclick** returns the old click speed (if *getset* is set to zero) or the new click speed (if it is set to one).

See Also

AES, TOS

evnt_keybd—AES function (**libaes.a/evnt_keybd**)

Await a keyboard event

#include <aesbind.h>

int evnt_keybd()

evnt_keybd is an AES routine that awaits a keyboard event; in other words, it waits for the user to press a key on the keyboard. **evnt_keybd** returns the code of the key pressed.

Example

The following example prints out the scan code for each key pressed. Pressing the **<return>** key exits.

```
#include <aesbind.h>
#include <gemdefs.h>
#define RETURN 0x1C0D

main() {
	unsigned key;

	appl_init();
	for(;;) {
		key = evnt_keybd();

		switch(key) {
			case RETURN:
				appl_exit();
				exit(0);
```

```
                        default:
                                printf("The scan code is:  %x\n", key);
                                break;
                }
        }
}
```

See Also

AES, keyboard, TOS

evnt_mesag—AES function (libaes.a/evnt_mesag)

Await a message

#include <aesbind.h>

**int evnt_mesag(*buffer*) char ****buffer*;

evnt_mesag is an AES routine that awaits a message. *buffer* points to where the message is to be written.

GEM uses 12 predefined messages to pass information among its applications. Each message is eight **ints** long, and has the following structure:

0	Type of message
1	Handle of application
2	Number of extra bytes in message; i.e., number of bytes beyond 16
3-7	Contents of message

The following lists the predefined messages by the value of word 0, as defined in the header file **gemdefs.h**:

MN_SELECTED (menu selected) Word 3 gives the number within its object tree of the title of the selected menu, and word 4 gives the number of the selected item.

WM_REDRAW (redraw a window) Word 3 gives the window's handle; words 4 through 7 give, respectively, the X coordinate, the Y coordinate, the width, and the height of the window to be drawn.

WM_TOPPED (make a window the topmost window) Word 3 gives the window handle.

WM_CLOSED (close-window box clicked) Word 3 gives the window's handle.

WM_FULLED (full-window box clicked) Word 3 gives the window's handle.

WM_ARROWED (arrow or scroll bar clicked) Word 3 gives the window's handle. Word 4 gives the action requested, as follows:

0	Page up
1	Page down
2	Row up
3	Row down
4	Page left
5	Page right
6	Column left
7	Column right

WM_HSLID (horizontal slider moved) Word 3 gives the window's handle. Word 4 gives the slider's position: zero indicates the leftmost position, and 1,000 the rightmost.

WM_VSLID (vertical slider moved) Word 3 gives the window's handle. Word 4 gives the slider's position: zero indicates the leftmost position, and 1,000 the rightmost.

WM_SIZED (window size altered) Word 3 gives the window's handle. Words 4 through 7 give, respectively, the X coordinate, the Y coordinate, the new width, and the new height.

WM_MOVED (window position altered) Word 3 gives the window's handle. Words 4 through 7 give, respectively, the new X coordinate, the new Y coordinate, the width, and the height.

AC_OPEN (desk accessory opened) Word 3 gives the desk accessory's menu item identifier, as set by the function **menu_register**.

AC_CLOSE (desk accessory closed) Word 3 gives the desk accessory's menu item identifier, as set by the function **menu_register**.

evnt_mesag always returns one.

Example
For an example of this routine, see the entry for **window**.

See Also
AES, TOS, **window**

evnt_mouse—AES function (**libaes.a/evnt_mouse**)
Wait for mouse to enter specified rectangle
#include <aesbind.h>
int evnt_mouse(*inout, rectangle, record***)**
int *inout*; **Rect** *rectangle*; **Mouse** *record*;

evnt_mouse is an AES routine that waits for the mouse pointer to enter or leave a specified rectangle on the screen. *inout* tells AES whether to wait for the pointer to enter (zero) or leave (one) the rectangle. Note that the screen manager constantly checks the location of the mouse; it is more accurate to say that **evnt_mouse** waits for the mouse pointer to be found inside or outside the rectangle.

rectangle is of the type **Rect**, which is defined in the header file **aesbind.h**. **Rect** consists of four elements:

x	X coordinate of rectangle
y	Y coordinate of rectangle
w	width of rectangle
h	height of rectangle

record points to where **evnt_mouse** writes the result of a mouse button event. It is declared to be of type **Mouse**, which is a structure of four pointers to integers. **Mouse** is declared in the header file **aesbind.h**, as follows:

x	X coordinate of mouse pointer
y	Y coordinate of mouse pointer
b	button state when event occurred
k	state of control, alt, and shift keys: 0x0: all keys up 0x1: right shift key down 0x2: left shift key down 0x4: control key down 0x8: alt key down

evnt_mouse always returns one.

See Also
AES, TOS

evnt_multi—AES function (**libaes.a/evnt_multi**)
Await one or more specified events
#include <aesbind.h>
int evnt_multi(*events, clicks, button, state, m1inout, rectangle1, m2inout, rectangle2, &buffer, lowtime, hightime, record, key, times***)**
int *events, clicks, button, state, m1inout, m2inout, lowtime, hightime*;
Rect *rectangle1, rectangle2*; **Mouse** *record*; **long** *buffer*; **int** **key, *times*;

evnt_multi is an AES routine that awaits one or more of a set of events. It is one of the most complex AES functions, and the one most commonly used.

events is a flag that indicates the events for which the process is waiting, as follows:

0x01	keyboard event
0x02	mouse button event
0x04	first defined mouse event
0x08	second defined mouse event
0x10	message from another process
0x20	timer event

clicks is the number of mouse button clicks the process is awaiting. *button* is a mask of the number of the mouse button that the processing is awaiting, from one to 16 (as counted from the left): **0x1** indicates the leftmost button; **0x2**, the button second from the left; **0x4**, the button third from the left, etc. Note that as of this writing no mouse has more than three buttons. *state* is the button state being awaited: zero indicates up, and one indicates down.

evnt_multi can await either or both of two mouse events. *m1inout* indicates that the process is waiting for the mouse pointer to enter (zero) or exit (one) the first mouse rectangle. Note that the screen manager is constantly polling the screen to check the location of the mouse; it is more accurate to say that **evnt_multi** waits for the mouse pointer to be found inside or outside the rectangle. *rectangle1* defines the area on the screen to be watched. It is declared to be of type **Rect**, which is declared in the header file **aesbind.h**; **Rect** consists of four elements, as follows:

x	X coordinate of rectangle
y	Y coordinate of rectangle
w	width of rectangle
h	height of rectangle

m2inout and *rectangle2* define the second mouse event being awaited; they are defined in exactly the same manner as *m1inout* and *rectangle1*.

buffer is the space into which AES writes any message from another process.

lowtime and *hightime* are, respectively, the low word and the high word of the time interval that the process will wait before it "times out", in milliseconds.

record points to where **evnt_multi** writes the result of a mouse button event. It is declared to be of type **Mouse**, which is a structure of four pointers to integers that is declared in the header file **aesbind.h**, as follows:

x	X coordinate of mouse pointer
y	Y coordinate of mouse pointer
b	button state when event occurred
k	state of control, alt, and shift keys: 0=up, 1=down 0x0: all keys up 0x1: right shift key down 0x2: left shift key down 0x4: control key down 0x8: alt key down

If a keyboard event occurs, *key* points to the code of the key pressed. See the entry **keyboard** for a table of the key codes.

Finally, *times* points to where to number of times the mouse button entered the desired state.

evnt_multi returns a number that indicates which event occurred, encoded in the same manner as the variable *events*, above.

Example
This example demonstrates how to use **evnt_multi**. It displays a window; the mouse pointer changes from an arrow to a bumblebee when it moves from inside to outside the window. The program exits when a key is typed.

```
#include <aesbind.h>
#include <gemdefs.h>
int nowhere = 0;                        /* place for unused pointers to point at */

main() {
/* declarations for window */
        int handle;
        char *title = " TITLE ";

/* declarations for evnt_multi() */
        int selection;                  /* code for event that occurred */
        unsigned int which = (MU_KEYBD | MU_M1 | MU_M2);
        int clicks = 1;                 /* number of clicks expected on mouse button */
        int button = 1;                 /* which button; 1 = leftmost */
        int buttonstate = 0;            /* button state expected; 0 = down */
        int into = 0;                   /* 1st mouse event: 0 = into rectangle */
        Rect bigrect;                   /* rectangle for both mouse events */
        Rect norect;                    /* someplace for rectangle to come from */
        int outof = 1;                  /* 2nd mouse event: 1 = out of rectangle */
        int *buffer = &nowhere;         /* buffer for messages; not used here */
        int lowtime = nowhere;          /* low word for timer event; not used here */
        int hightime = nowhere;         /* high word for timer event; not used here */
        Mouse place;                    /* where mouse event occurred; not used here */
        int key = 0;                    /* which key was pressed; not used here */
        int times = 0;                  /* no. of times mouse button entered state */
```

```
/* initialize rectangles used, in rasters */
        bigrect.x = 210;
        bigrect.y = 100;
        bigrect.w = 220;
        bigrect.h = 200;

        norect.x = 0;
        norect.h = 0;
        norect.w = 0;
        norect.h = 0;

/* initialize place, although not used here */
        place.x = place.y = place.b = place.k = &nowhere;

        appl_init();
        handle = wind_create(NAME, bigrect);
        wind_set(handle, WF_NAME, title, 0, 0);
        graf_growbox(norect, bigrect);
        wind_open(handle, bigrect);
        for(;;) {
                selection = evnt_multi(which, clicks, button, buttonstate,
                        into, bigrect, outof, bigrect, buffer, lowtime,
                        hightime, place, &key, &times);

                switch(selection) {
                case MU_KEYBD:
                        wind_close(handle);
                        graf_shrinkbox(norect, bigrect);
                        appl_exit();
                        exit(0);
                case MU_M1:
                        graf_mouse(ARROW, &nowhere);
                        which = (MU_KEYBD | MU_M2);
                        break;
                case MU_M2:
                        graf_mouse(BUSY_BEE, &nowhere);
                        which = (MU_KEYBD | MU_M1);
                        break;
                default:
                        break;
                }
        }
}
```

See Also
AES, keyboard, TOS

Notes
Note that, with regard to button events, you can tell **evnt_multi** to wait only for one specified event, e.g., for button 1 to be pressed. If you tell it to wait for button 1 *or* button 2 to be pressed, it will act as if you told it to wait for button 1 *and* button 2 to be pressed, which the hardware cannot handle.

evnt_timer—AES function (**libaes.a/evnt_timer**)

Wait for a specified length of time

#include <aesbind.h>

int evnt_timer(*lowtime, hightime***) int** *lowtime, hightime*;

evnt_timer is an AES routine that awaits a timer event, i.e., that waits for a given length of time to pass. The time interval to wait before "timing out" is given in milliseconds. *lowtime* is the low word of the time interval, and *hightime* is the high word. **evnt_timer** always returns one.

See Also

AES, TOS

Notes

As of this writing, using **evnt_timer** within a desk accessory will cause the system to crash *if* the desk accessory performs any calls to a BDOS routine. For more information on BDOS, see the entries for **VDI** and **metafile**.

executable file—Definition

An **executable file** is one that can be loaded directly by the operating system and executed. Normally, an executable file is one that has gone through both the *compilation* phase, where it has been rendered into machine language, and the *link* phase, in which the compiled program has all operating system-specific information added and all library functions are copied into the program.

See Also

file

execve—UNIX system function

int execve—Execute a command

execve(*file, argv, env***)**

char **file*, **argv*[], **env*[]

execve permits you to tell to TOS execute a specific command. This is done through the GEMDOS call **Pexec**. The calling program is suspended while the command is being executed; the calling program returns when the command has finished executing. *file* is the complete path name of the file to be executed. *argv* points to a list of arguments to be passed to the command. *env* points to a list of status environmental parameters. If the **Pexec** status is negative, then **errno** is set to the absolute value of the status.

See Also
environment, Pexec, system

exit—Command
Exit from **msh**
exit [*status*]

exit terminates the shell **msh**. **msh** executes **exit** directly. The optional argument *status* is an integer which is returned as the exit status.

See Also
commands, msh

exit—General function (**libc.a/exit**)
Terminate a program directly
int exit(*status***) int** *status*;

exit terminates a program gracefully. It flushes all buffers, closes each open file, and then returns the given *status*. Some systems, such as the Series III under ISIS, throw away the **exit**. On TOS, it is returned to the parent program as the result of **Pexec**.

See Also
_exit, runtime startup, system, UNIX routines
The C Programming Language, page 154

_exit—UNIX system call (**libc.a/_exit**)
Terminate a program directly
_exit(*status***) int** *status*;

_exit terminates a program. It returns *status* to the calling program, and never returns.

See Also
exit, Pterm, runtime startup, system, UNIX routines

Notes
Programs should normally terminate via **exit**, which flushes buffered I/O and closes associated files. Note that on the Atari ST, **_exit** is implemented via the function **Pterm**.

exp—Mathematics function (**libm.a/exp**)
Compute exponent
#include <math.h>
double exp(*z***) double** *z*;

exp returns the exponential of z, or e^z.

Example
The following example demonstrates **exp**:

```
#include <math.h>

dodisplay(value, name)
double value; char *name;
{
	if (errno)
		perror(name);
	else
		printf("%10g %s\n", value, name);
	errno = 0;
}

#define display(x) dodisplay((double)(x), "x")

main() {
	extern char *gets();
	double x;
	char string[64];

	for(;;) {
		printf("Enter number: ");
		if(gets(string) == 0)
			break;
		x = atof(string);

		display(x);
		display(exp(x));
		display(pow(10.0,x));
		display(log(exp(x)));
		display(log10(pow(10.0,x)));
	}
}
```

See Also
errno, mathematics library

Diagnostics
exp indicates overflow by an **errno** of **ERANGE** and a huge returned value.

extern—Definition

extern indicates that a C element belongs to the *external* storage class. Both variables and functions may be declared to be **extern**. **extern** symbols are "visible" outside of the source file of definition. All functions and all data defined outside of functions are implicitly **extern** unless declared **static**.

When a source file references data that are defined in another file, it must declare the data to be **extern**, or the linker will return an error message of the form:

```
undefined symbol name
```

For example, following declares the array **tzname**:

```
extern char tzname[2][32];
```

When a function calls a function defined in another source file or a library, it should make an **extern** declaration of the function. In the absence of a declaration, **extern** functions are assumed to return **ints**, which may cause serious problems if the function actually returns a 32-bit pointer, a long **int**, or a **double**.

See Also
auto, pun, register, static, storage class
The C Programming Language, pages 28, 72, 204

fabs—Mathematics function (**libm.a/fabs**)
Compute absolute value
#include <math.h>
double fabs(*z*) double *z*;

fabs implements the absolute value function. It returns *z* if *z* is zero or positive, or -*z* if *z* is negative.

Example
For an example of this function, see the entry for **ceil**.

See Also
abs, ceil, floor, frexp, mathematics library

Fattrib—**gemdos** function 67 (**osbind.h**)
Get and set file attributes
#include <osbind.h>
long Fattrib(*name, readset, setatrib*) char *_name_;
int *readset, setatrib*;

Fattrib gets and sets file attributes. *name* points to the file's name, which must be a NUL-terminated string. *readset* contains a 0 if you wish to read the file's attributes, or a 1 if you wish to set them. *setatrib* contains an integer than encodes the file's attributes, as follows: 0x01, read only; 0x02, hidden from directory search; 0x04 set to system, hidden from directory search; 0x08, contains volume label in first 11 bytes; 0x10, file is a subdirectory; and 0x20, file has been written to and closed. **Fattrib** returns the file's attributes if they have been read successfully; otherwise, it cannot be relied on to return meaningful information.

Example

```
#include <osbind.h>
extern int errno;

char *atrtable[] = {
	"read only",
	"hidden",
	"system file",
	"volume label",
	"subdirectory",
	"written to and closed"
};
```

```
main(argc, argv) int argc; char **argv; {
	unsigned attribs;
	unsigned point;
	int i;

	if (argc < 2) {
		printf("Usage: Fattrib file\n");
		Pterm(1);
	}
	if ((attribs = Fattrib(argv[1], 0, 0)) < 0) {
		printf("Can't Fattrib file %s --\n", argv[1]);
		errno = -attribs;
		perror("Fattrib failure");
		Pterm(1);
	}

	printf("File %s:", argv[1]);
	if (attribs == 0) {
		printf(" normal file\n");
		Pterm0();
	}
	point = 1;
	for (i=0 ; i<6 ; i++) {
		if (point & attribs)
			printf(" (%s)", atrtable[i]);
		point <<= 1;
	}
	printf("\n");
}
```

See Also
gemdos, TOS

fclose—STDIO Function (**libc.a/fclose**)
Close stream
#include <stdio.h>
**int fclose(*fp*) FILE ****fp*;

fclose closes the stream **fp**. It calls **fflush** on the given *fp*, closes the associated file, and releases any allocated buffer. The library function **exit** calls **fclose** for open streams.

Example
For examples of how to use this function, see the entries for **fopen** and **fseek**.

See Also
STDIO
The C Programming Language, page 153

Diagnostics
fclose returns **EOF** on error.

Fclose—**gemdos** function 62 (**osbind.h**)
Close a file
#include <osbind.h>
long Fclose(*handle*) int *handle*;

Fclose closes a file. *handle* is the file handle that was returned by **Fopen()**, **Fcreate()**, **Fdup()**, or inherited by the process. **Fclose** returns 0 if the file could be closed, and non-zero if it could not.

Example
For example of how to use this macro, see the entries for **Fseek** and **Fcreate**.

See Also
gemdos, TOS

Fcreate—**gemdos** function 60 (**osbind.h**)
Create a file
#include <osbind.h>
long Fcreate(*name, type*) char *name; int *type*;

Fcreate creates a file. *name* points to the file's path name, which must be a NUL-terminated string. *type* contains a number that encodes the file's attributes, as follows: 0x01, read-only; 0x02, hidden from directory search; 0x04, set to system, hidden from directory search; and 0x08, contains volume label in first 11 bytes. **Fcreate** returns a file handle, which is understood by TOS.

Example
The following example, when compiled, takes two arguments, *file1* and *file2*; it then copies *file1* into *file2*. If *file2* does not exist, it is created.

```
#include <osbind.h>
#include <stdio.h>
#include <stat.h>
extern int errno;

main(argc, argv) int argc; char **argv; {
        int status;
        int inhand;
        int outhand;
        struct DMABUFFER *mydta;
        char *buffer;
        long copysize;

        if (argc < 3) {
                Cconws("Usage: Fcreate source target\r\n");
                Pterm(1);
        }

        if ((inhand = Fopen(argv[1], 0)) < 0) {
                fprintf(stderr,"\nCan't open input file %s",argv[1]);
                errno = -inhand;
                perror("Fopen failure");
                Pterm(1);
        }

        Fsetdta(mydta=(struct DMABUFFER *)malloc(sizeof(struct DMABUFFER)));

        if ((status=Fsfirst(argv[1], 0xF7)) != 0) {
                Fclose(inhand);
                fprintf(stderr,"\nError getting stats on input file %s",
                                argv[1]);
                errno = -status;
                perror("Fsfirst failure");
                Pterm(1);
        }

        status = mydta->d_fattr & 7;

        if((outhand = Fcreate(argv[2], status)) < 0) {
                Fclose(inhand);
                fprintf(stderr,"\nCan't open output file %s",argv[2]);
                errno = -outhand;
                perror("Fcreate failed");
                Pterm(1);
        }
```

```
buffer = (char *)malloc(4096);
copysize = mydta->d_fsize;
while (copysize>4096) {
        if ((status=Fread(inhand, 4096L, buffer)) < 0) {
                Fclose(inhand);
                Fclose(outhand);
                Fdelete(argv[2]);
                fprintf(stderr,"\nRead error on %s", argv[1]);
                errno = -status;
                perror("Read failure");
                Pterm(1);
        }
        if ((status=Fwrite(outhand, 4096L, buffer)) < 0) {
                Fclose(inhand);
                Fclose(outhand);
                Fdelete(argv[2]);
                fprintf(stderr,"\nWrite error on file %s", argv[2]);
                errno = -status;
                perror("Write failure");
                Pterm(1);
        }
        copysize -= 4096;
}
if (copysize > 0) {
        if ((status=Fread(inhand, copysize, buffer)) < 0) {
                Fclose(inhand);
                Fclose(outhand);
                Fdelete(argv[2]);
                fprintf(stderr,"\nRead error on %s", argv[1]);
                errno = -status;
                perror("Read failure");
                Pterm(1);
        }
        if ((status=Fwrite(outhand, copysize, buffer)) < 0) {
                Fclose(inhand);
                Fclose(outhand);
                Fdelete(argv[2]);
                fprintf(stderr,"\nWrite error on %s", argv[2]);
                errno = -status;
                perror("Write failure");
                Pterm(1);
        }
}
```

```
        Fclose(inhand);
        Fclose(outhand);
        printf("File %s copied to file %s.\n", argv[1], argv[2]);
        free(mydta);
        Fsetdta(NULL);
        Pterm0();
}
```

See Also
gemdos, TOS

fcvt—General function (**libc.a/fcvt**)
Convert floating point numbers to ASCII strings
char *fcvt(*d, w, dp, signp*) double *d*; int *w*, **dp*, **signp*;

fcvt converts floating point numbers to ASCII strings. **fcvt** converts *d* into a NUL-terminated string of decimal digits that is *w* characters wide. It rounds the last digit and returns a pointer to the result. On return, **fcvt** sets *dp* to point to an integer that indicates the location of the decimal point relative to the beginning of the string: to the right if positive, and to the left if negative. Finally, it sets *signp* points to an integer that indicates the sign of *d*: zero if positive, and nonzero if negative. **fcvt** rounds the result to FORTRAN F-format.

See Also
ecvt, frexp, gcvt, ldexp, modf, printf

Notes
fcvt performs conversions within static string buffers that are overwritten by each execution.

Fdatime—**gemdos** function 87 (**osbind.h**)
Get or set a file's date/time stamp
#include <osbind.h>
void Fdatime(*info, handle, getset*)
int *handle, getset, info*[2];

Fdatime retrieves or sets a file's time/date stamp. *handle* is the file's handle that was set when the file was first opened. *getset* indicates whether the stamp is to be reset or retrieved: 0 indicates get, and 1 indicates set. *info* points to a buffer that holds two integers; this buffer either have the time/date stamp written into it, or will hold the new time/date stamp that is to replace the previous stamp, depending on whether the stamp is to be retrieved or reset. In either case, the first integer of *info* encodes the time and the second integer encodes the date, as follows:

info[1]	0-4	no. of two-second increments (0-29)
	5-10	no. of minutes (0-59)
	11-15	no. of the hour (0-23)
info[2]	0-4	day of the month (1-31)
	5-8	no. of the month (1-12)
	9-15	no. of the year (0-119, 1980 = 0).

Fdatime returns nothing.

Example
The following example demonstrates **Fdatime**.

```
#include <osbind.h>
#include <errno.h>
#include <time.h>

main(argc, argv)
int argc; char *argv[]; {
	int fd;
	rtetd_t rtd;		/* Backwards time, date */
	utetd_t utd;		/* Forwards date, time */
	time_t t;		/* COHERENT time */
	tm_t *tp;		/* Time fields */

	if (argc < 2) {
		printf("Usage: Fdatime <filename>\n");
		exit(1);
	}

	if ((fd = Fopen(argv[1], 0)) < 0) {
		errno = -fd;
		perror(argv[1]);
		exit(1);
	}

	Fdatime(&rtd, fd, 0);
	utd.g_date = rtd.g_rdate;
	utd.g_time = rtd.g_rtime;
	tp = tetd_to_tm(utd);
	t = jday_to_time(tm_to_jday(tp));

	printf("%s", asctime(tp));
	printf("%s", ctime(&t));
	return 0;
}
```

See Also
gemdos, TIMEZONE, TOS

Notes
msh updates the time it returns by one hour if the daylight savings time flag is set in the TIMEZONE environmental parameter. Therefore, during the summer months, the time returned by this routine may be one hour behind the time returned by the **date** command.

Fdelete—**gemdos** function 65 (**osbind.h**)
Delete a file
#include <osbind.h>
long Fdelete(*name***) char ****name*;

Fdelete deletes a file. *name* points to the file's name, which must be a NUL-terminated string. **Fdelete** returns 0 if the file could be deleted, and non-zero if it could not.

Example
For examples of how to use this macro, see the entries for **Fseek** and **Fcreate**.

See Also
gemdos, TOS

Fdup—**gemdos** function 69 (**osbind.h**)
Generate a substitute file handle
#include <osbind.h>
long Fdup(*handle***) int** *handle*;

Fdup generates a substitute file handle for a standard file handle: between one and five, inclusive. It returns the new, non-standard file handle if successful, or the error code **EINHNDL** (invalid handle) or **ENHNDL** (no handles left, i.e., too many files open) if not.

See Also
gemdos, TOS

Notes
Fdup returns with no error indication if the argument it is passed is a file handle that has been processed by **Fclose**; however, the system will generate an address error when the process terminates.

feof—STDIO macro (**stdio.h**)
Discover stream status
#include <stdio.h>
feof(*fp***) FILE ****fp*;

feof is a macro that tests the status of the argument stream *fp*. It returns a number other than zero when *fp* has reached the end of file, and 0 otherwise. One use of **feof** is to distinguish a value of -1 returned by **getw** from an **EOF**.

Example
For an example of how to use this function, see the entry for **fopen**.

See Also
STDIO

ferror—STDIO macro (**stdio.h**)
Discover stream status
#include <stdio.h>
ferror(*fp***) FILE ****fp*;

ferror is a macro that tests the status of the argument stream *fp*. It returns a number other than zero if an error has occurred on *fp*. Any error condition that is discovered will persist either until the stream is closed or until **clearerr** is used to clear it. For write routines that employ buffers, **fflush** should be called before **ferror**, in case an error occurs on the last block written.

See Also
STDIO

fflush—STDIO function (**libc.a/fflush**)
Flush stream output buffer
#include <stdio.h>
fflush(*fp***) FILE ****fp*;

fflush writes any buffered output data associated with the stream *fp*. The file stays open after **fflush** is called. **fclose** calls **fflush**; there is no need for the user program to call it directly under ordinary conditions.

Example
For an example of this routine, see the entry for **v_gtext**.

See Also
STDIO

Diagnostics
fflush returns **EOF** if it cannot flush the contents of the buffers.

Fforce—**gemdos** function 70 (**osbind.h**)
Force a file handle
#include <osbind.h>
long Fforce(*shandle, nshandle***) int** *shandle, nshandle*;

Fforce forces the standard file handle, i.e., zero through five, to point to the same file as the non-standard file handle, i.e., six and up. **Fforce** returns

E_OK (no error) if successful, or **EIHNDL** (invalid handle) if not.

See Also
Fdup, gemdos, TOS

fgetc—STDIO function (**libc.a/fgetc**)
Read character from stream
#include <stdio.h>
int fgetc(*fp*) FILE *fp;

fgetc reads characters from the input stream *fp*. It is a function whose body is the macro **getc**. In general, it behaves the same as **getc**; it runs more slowly than **getc**, but yields a smaller object module when compiled.

Example
This example counts the number of lines and "sentences" in a file.

```
#include <stdio.h>
main(){
	FILE *fp;
	int ch, nlines, nsents;
	int filename[20];
	nlines = nsents = 0;
	printf("Enter file to test: ");
	gets(filename);
	if ((fp = fopen(filename,"r")) != NULL) {
		while ((ch = fgetc(fp)) != EOF) {
			if (ch == '\n') ++nlines;
			else if (ch == '.' || ch == '!' || ch == '?') {

				if ((ch = fgetc(fp)) != '.') {
					++nsents;
					ungetc(ch, fp);
				}
				else for(ch='.'; (ch=fgetc(fp))=='.';)
					;
			}
		}
		printf("%d line(s), %d sentence(s).\n", nlines, nsents);
	} else
		printf("Cannot open %s.\n", filename);
}
```

See Also
getc, STDIO

Diagnostics
fgetc returns **EOF** at end of file or on read error.

Fgetdta—gemdos function 47 (**osbind.h**)

Get a disk transfer address

#include <osbind.h>
#include <stat.h>
DMABUFFER ***Fgetdta()**

Fgetdta gets and returns the disk transfer address that had been set by **Fsetdta**, and will be used by **Fsfirst** and **Fsnext**.

Example

The following example creates a version of the **find** utility for TOS. It generates a full path name and description for every file in your file system; its output can be piped to if you wish to find where you stored a particular file, as follows:

find | egrep *filename*

This example demonstrates the TOS functions **Fgetdta**, **Fsetdta**, **Fsfirst**, and **Fsnext**. It also demonstrates the use of **isascii**, **isupper**, **free**, **malloc**, **strcat**, **strcpy**, **strlen**, and **tolower**.

This example also demonstrates how to use the global variable **_stksize** to check for stack overflow.

```
#include <osbind.h>
#include <stat.h>
#include <ctype.h>
extern long _stksize;

/* Translate string to lower case */
char *lowercase(name)
char *name;
{
	register char *p = name; register int c;
	while (c = *p) *p++ = isascii(c) && isupper(c) ? tolower(c) : c;
	return name;
}

/* Concatentate path suffix to path prefix */
char *dircat(pfx, sfx)
register char *pfx, *sfx;
{
	extern char *malloc(), *strcat();
	register char *p; register int nb, npfx;
	nb = (npfx = strlen(pfx)) + 1 + strlen(sfx) + 1;
```

```
	if ((p = malloc(nb)) == 0) exit(1);
	strcpy(p, pfx);
	if (npfx != 0 && pfx[npfx-1] != '\\') strcat(p, "\\");
	return strcat(p, sfx);
}

/* Search the directory specified by dname */
find(name)
char *name;
{
	register char *globname, *newname; DMABUFFER dumb, *saved;

	if ((long)&saved <= _stksize+128) {
		printf("Stack near overflow in find()\n\r"); return;
	}

	globname = dircat(name, "*.*");
	saved = (DMABUFFER *)Fgetdta();
	Fsetdta(&dumb);

	if (Fsfirst(globname, 0xFF) == 0) {
		do {
			if (dumb.d_fname[0] != '.') {
				newname = dircat(name, dumb.d_fname);
				printf("%s\n", lowercase(newname));
				find(newname);
				free(newname);
			}

		} while (Fsnext() == 0);
	}
	free(globname);
	Fsetdta(saved);
}

main()
{
	find(""); return 0;
}
```

See Also

Fsetdta, Fsfirst, Fsnext, gemdos, TOS

fgets—STDIO function (libc.a/fgets)

Read line from stream

#include <stdio.h>

char *fgets(*s*, *n*, *fp*) char *s; int *n*; FILE *fp;

fgets reads characters from the stream *fp* into string *s* until *n*-1 characters have been read or until a newline or EOF is encountered. It retains the newline, if any, and appends a NUL character at the end of of the string. **fgets** returns the argument *s* if any characters were read, and **NULL** if none were read.

Example
This example looks for the pattern pattern given by **argv[1]** in standard input or in file **argv[2]**. It demonstrates the functions **pnmatch**, **fgets**, and **freopen**.

```
#include <stdio.h>
#define MAXLINE 128
char buf[MAXLINE];

main(argc, argv) int argc; char *argv[]; {
	if (argc != 2 && argc != 3)
		fatal("Usage: pnmatch pattern [ file ]");
	if (argc == 3 && freopen(argv[2], "r", stdin) == NULL)
		fatal("cannot open input file");
	while (fgets(buf, MAXLINE, stdin) != NULL) {
		if (pnmatch(buf, argv[1], 1))
			printf("%s", buf);
	}

	if (!feof(stdin))
		fatal("read error");
	exit(0);
}

fatal(s) char *s; {
	fprintf(stderr, "pnmatch: %s\n", s);
	exit(1);
}
```

See Also
STDIO
The C Programming Language, page 155

Diagnostics
fgets returns NULL if an error occurs, or if EOF is seen before any characters are read.

fgetw—STDIO function (libc.a/getw)

Read integer from stream
#include <stdio.h>
**int fgetw(*fp*) FILE ****fp*;

fgetw is a function that reads an integer from the stream *fp*.

See Also
STDIO

Notes
fgetw returns the value EOF on errors. A call to **feof** or **ferror** may be necessary to distinguish this value from a valid word.

field—Definition

A **field** is an area that is set apart from whatever surrounds it, and that is defined as containing a particular type of data. In the context of C programming, a field is either an element of a structure, or a set of adjacent bits within an **int**.

See Also
bit map, data formats, structure
The C Programming Language, page 136

file—Command

Name a file's type
file *file* ...

file names the type of each *file* named. It examines files to make an educated guess about their format.

file recognizes the following classes of text files: files of commands to the shell; files containing the source for a C program; files containing assembly language source; files containing unformatted documents that can be passed to **nroff**; and plain text files that fit into none of the above categories.

file recognizes the following classes of non-text or binary data files: the various forms of archives, object files, and link modules for various machines, and miscellaneous binary data files.

See Also
commands, ls, msh, size

Notes
Because **file** only reads a set amount of data to determine the class of a text file, mistakes can happen.

file—Definition

A **file** is a mass of bits that has been given a name and is stored on a nonvolatile medium. These bits may be ASCII characters or machine-readable material of some sort. Under the UNIX system, the COHERENT system, and related operating systems, external devices can mimic files, in that they can be opened, closed, read, and written to in a manner identical to that of files.

See Also
close, fopen, fclose, FILE, open

FILE—Definition
Descriptor for a file stream
#include <stdio.h>

FILE describes a *stream* or a peripheral device through which data flow. It is defined in the header file **stdio.h**. A pointer to **FILE** is returned by **fopen**, **freopen**, and related functions.

See Also
fopen, freopen, stdio.h, stream

file descriptor—Definition
A **file descriptor** is an integer that appears as an entry in a table of files. It is used by routines like **open**, **close**, and **lseek** to work with files. Note that a file descriptor is *not* the same as a file pointer, which is used by routines like **fopen**, **fclose**, or **fread**. Note, too, that TOS routines use the term **handle** as a synonym for "file descriptor".

fileno—STDIO function (**libc.a/fileno**)
Get file descriptor
#include <stdio.h>
fileno(*fp***)** FILE ***fp*;

fileno returns the file descriptor, a small, non-negative integer, associated with the stream *fp*. This file descriptor, called the *handle*, is the integer returned by the **open** or **creat** call, which a routine such as **fopen** used to create the stream.

See Also
STDIO

flexible arrays—Definition
Flexible arrays are arrays whose length is not declared explicitly. Each has exactly one empty '[]' array bound declaration, and if the array is multidimensional, then the flexible dimension of the array must be the first array bound in the declaration.

Flexible arrays occur in only a few contexts; for example, as parameters:

```
char *argv[];
char p[][8];
```

as **extern** declarations:

```
extern int end[];
```

as **extern** or **static** initialized definitions:

```
static char digit[]="01234567";
```

or as a member of a structure, usually, though not necessarily, the last:

```
struct nlist {
        struct nlist *next;
        char name[];
};
```

See Also
array, data types

float—Definition

Floating point numbers are a subset of the real numbers. Each has a built-in radix point that shifts, or "floats", as the value of the number changes. It consists of one sign bit, which indicates whether the number is positive or negative; bits that encode the number's *exponent*; and bits that encode the number's *fraction*, or the number upon which the exponent works. Note that elsewhere, the fraction is often called the *mantissa*. In general, the magnitude of the number encoded depends upon the number of bits in the exponent, whereas its precision depends upon the number of bits in the fraction.

The exponent often uses a *bias*. This is a value that is subtracted from the exponent to yield the power of two by which the fraction will be increased.

Floating point numbers come in two levels of precision: single precision, called **floats**; and double precision, called **doubles**. With most microprocessors, **sizeof(float)** returns four, which indicates that it is four **chars** (bytes) long; and **sizeof(double)** returns eight.

Several formats are used to encode **floats**, including IEEE, DECVAX, and BCD (binary coded decimal). Mark Williams C uses DECVAX format. Each format is described below.

DECVAX Format
The 32 bits in a **float** consist of one sign bit, an eight-bit exponent, and a 23-bit fraction, as follows:

```
Sign  Exponent    Fraction
|s   eeeeeee|e   fffffff|ffffffff|ffffffff|
    Byte 4      Byte 3     Byte 2   Byte 1
```

Note that the exponent has a bias of 129.

If the sign bit is set to one, the number is negative; if it is set to zero, then the number is positive. If the number is all zeroes, then it equals zero; an exponent and fraction of zero plus a sign of one ("negative zero") is by definition not a number. All other forms are numeric values.

The format for **doubles** simply adds another 32 fraction bits to the end of the **float** representation, as follows:

```
Sign  Exponent    Mantissa
|s    eeeeeee|e  fffffff|ffffffff|ffffffff|ffffffff|ffffffff|ffffffff|ffffffff|
   Byte 8       Byte 7     Byte 6   Byte 5   Byte 4   Byte 3   Byte 2   Byte 1
```

For this reason, a **double** under Mark Williams C has double the precision of a **float**, but the same magnitude.

IEEE Format
The IEEE encoding of a **float** is the same as that in the DECVAX format. Note, however, that the exponent has a bias of 127, rather than 129.

Unlike the DECVAX format, IEEE format assigns special values to a number of floating point numbers. Note that in the following description, a *tiny* exponent is one that is all zeroes, and a *huge* exponent is one that is all ones:

- A tiny exponent with a fraction of zero equals zero, regardless of the setting of the sign bit.
- A huge exponent with a fraction of zero equals infinity, regardless of the setting of the sign bit.
- A tiny exponent with a fraction greater than zero is a denormalized number, i.e., a number that is less than the least normalized number.
- A huge exponent with a fraction greater than zero is, by definition, not a number. These values can be used to handle special conditions.

The 64 bits in a **double** unlike the IEEE format, does not increase the number of exponent bits, but consist of a sign bit, an 11-bit exponent, and a 52-bit fraction, as follows:

```
Sign     Exponent           Mantissa
|s      eeeeeee|eeee  ffff|ffffffff|ffffffff|ffffffff|ffffffff|ffffffff|ffffffff|
   Byte 8          Byte 7    Byte 6   Byte 5   Byte 4   Byte 3   Byte 2   Byte 1
```

Note that the exponent has a bias of 1,023. The rules of encoding are the same as for **floats**.

BCD Format
The BCD ("binary coded decimal") format is used in accounting, to eliminate rounding errors that alter the worth of an account by a fraction of a cent. For that reason, BCD format consists of a sign (**s**), an exponent (**e**), and a chain of four-bit numbers, each of which is defined to hold the digits zero through nine (**d**).

A BCD **float** has a sign bit, seven bits of exponent, and six four-bit decimal numbers, as follows:

```
 Sign Exponent    Mantissa
|s    eeeeeee|   dddd  dddd|dddd dddd|dddd dddd|
   Byte 4         Byte 3      Byte 2    Byte 1
```

A BCD **double** has a sign bit, 11 bits of exponent, and 13 four-bit decimal numbers, as follows:

```
 Sign Exponent        Mantissa
|s    eeeeeee|eeee  dddd|dddd dddd|dddd dddd|dddd dddd|dddd dddd|dddd dddd|dddd dddd|
   Byte 8       Byte 7      Byte 6     Byte 5     Byte 4     Byte 3     Byte 2     Byte 1
```

Note that passing the hexadecimal numbers A through F in a decimal digit yields unpredictable results.

Note the following rules in handling BCD numbers:

* A tiny exponent with a fraction of zero equals zero.
* A tiny exponent with a fraction of non-zero indicates a denormalized number.
* A huge exponent with a fraction of zero indicates infinity.
* A huge exponent with a fraction of non-zero is, by definition, not a number; these non-numbers are used to indicate errors.

See Also
data formats, declarations, double
The Art of Computer Programming, vol. 2, page 180*ff*

floor—Mathematics function (**libm.a/floor**)
Set a numeric floor
#include <math.h>
double floor(*z*) double *z*;

floor sets a numeric floor. It returns a double-precision floating point number whose value is the largest integer less than or equal to *z*.

Example
For an example of this function, see the entry for **ceil**.

See Also
abs, ceil, fabs, frexp, mathematics library

Flopfmt—xbios function 10 (**osbind.h**)
Format tracks on a floppy disk
#include <osbind.h>
#include <xbios.h>
int Flopfmt(*buffer, filler, device,sectors, track, side, interleave, magic, new*)

char **buffer*, **filler*, **magic*; **int** *device, sectors, track, side, interleave, new*;

Flopfmt formats a track on a floppy disk. The Atari SF314 and SF354 floppy disk drives each support 80 tracks per disk, and zero to ten sectors per track.

buffer points to a buffer large enough to hold the image of an entire track. *filler* is unused, and can be set to anything. *device* is the number of the floppy disk drive, i.e., zero or one. *sectors* is the number of sectors to format per track; the usual number is nine. *track* is the number of the track that you wish to format, from zero to 79. *side* is the side of the floppy disk on which you wish to write, i.e., zero or one. *interleave* is the number that governs the interleaving of sectors; it is usually set to one. *magic* is a magic number that must be set to 0x87654321.

Finally, *new* is the word-fill value that is used for new sectors; a good setting is 0xE5E5.

Flopfmt returns zero if the information was written correctly, and returns a numeric error code if it was not. If bad sectors are discovered, their numbers are written into *buffer* in the form of a NUL-terminated string. The user then has the choice of attempting to re-format the sector, or recording this string to map out bad sectors in any further attempts to write to that track.

Example
This example formats a single-sided floppy disk and initializes the first two tracks. It demonstrates the macros **Flopfmt**, **Flopwr**, and **Protobt**.

```
#include <stdio.h>
#include <osbind.h>

#define BLANK (0xE5E5)              /* Standard sector format value */
#define MAGIC (0x87654321L)         /* Manditory magic number value */
#define BUFSIZE (9*1024)            /* Buffer size for 9 sectors */

extern int errno;                   /* Error number for perror() */

main() {
        int track;                  /* Track counter */
        int side;                   /* Side counter */
        int status;                 /* Status word... */
        short *bf;                  /* Buffer ptr. */
        char reply;                 /* Reply... */
        short *middle;              /* Pointer for bad block dump */

        side = 0;     /* Only format side 0 */
        printf("Really format disk in drive B? ");
        fflush(stdout);
        if ((reply = Crawcin()) != 'y' && reply != 'Y') {
                printf("No. Floppy in drive B not formatted.\n");
                Pterm0();
        }
```

```
printf("Yes\n");
printf("Press any key when ready...");
fflush(stdout);
Crawcin();
printf("\n");
bf = (short *) malloc(BUFSIZE);

/* First -- Format the floppy */
for (track=0; track<80; track++) {
	printf("now formatting track %d:", track);
	fflush(stdout);
	status = Flopfmt(bf, 0L, 1, 9, track, side, 1, MAGIC, BLANK);

	if (status) {
		middle = bf;
		printf("\t%d\n", status);
		while (*middle) {
			printf("\tBad sector %d\n", *middle++);
		}
	} else {
		printf("\tokay\n");
	}
}
printf("Format of all tracks completed\n");
printf("Any key to continue...");
fflush(stdout);
Crawcin();
printf("Initializing directory structure\n");

/*
 * Now, clear out the first two tracks (all zeros...
 * First, zero out the buffer...
 */
for (track = 0; track < (BUFSIZE>>1); bf[track++] = 0);

/* Now, write it to all sectors of the first two tracks */
for (track=0;track<2;) {
	printf("Zeroing track %d.\n", track);
	if (status = Flopwr(bf, 0L, 1, 1, track++, 0, 9)) {
		errno = -status;
		perror("Flopwr failure");
	}
}

/* Now, we will prototype the boot block... */
Protobt(bf, (long)Random(), 2, 0);
```

```
	/* Finally, write this out to the boot sector... */
	status = Flopwr(bf, 0L, 1, 1, 0, 0, 1);
	if (status) {
		errno = -status;
		perror("Write of boot-block failed.");
	}

	/* Verify the write... */
	status = Flopver(bf, 0L, 1, 1, 0, 0, 1);
	if (status) {
		errno = -status;
		perror("Verify of boot-block failed.");
	}

	printf("Program done. Disk in drive B is formatted.\n");
	free(bf);
	Pterm0();
}
```

See Also
TOS, xbios

Floprd—xbios function 8 (**osbind.h**)
Read sectors on a floppy disk
#include <osbind.h>
#include <xbios.h>
int Floprd(*buffer, filler, device, sector, track, side, count***)**
char ****buffer*, ******filler*; **int *device, sector, track, side, count*;

Floprd reads one or more sectors on a floppy disk. *filler* is not used, but must be passed properly for this function to work. *buffer* must point to a buffer that is large enough to hold the number of sectors read. *device* is the number of the device, i.e., zero or one. *sector* is the sector at which to begin reading, i.e., one through nine. *track* is the track number to seek to, i.e., zero through 79. *side* is the side of the floppy to read, zero or one. Finally, *count* is the number of sectors to read; this can be no greater than the number of sectors on the track.

Floprd returns zero if the read succeeded, and returns an error code number if it did not.

Example

```
#include <osbind.h>
#include <bios.h>
#define uword(x)	((unsigned)(x))
#define ulong(x)	((unsigned long)(x))
#define can2(x,y)	(uword(x)|(uword(y)<<8))
#define can3(x,y,z)	(can2(x,y)|(ulong(z)<<16))
```

```
struct bbpb bb;
main() {
        Floprd(&bb, 0L, 1, 1, 0, 0, 1);     /* read the boot block */
        printf("serial number: %lu\n",
                can3(bb.bp_serial[0],bb.bp_serial[1],bb.bp_serial[2]));

        printf("bytes per sector: %u\n",
                can2(bb.bp_bps[0],bb.bp_bps[1]));
        printf("sectors per cluster: %u\n",
                uword(bb.bp_spc));

        printf("reserved sectors: %u\n",
                can2(bb.bp_res[0],bb.bp_res[1]));
        printf("number of fats: %u\n",
                uword(bb.bp_nfats));
        printf("root directory entries: %u\n",
                can2(bb.bp_ndirs[0],bb.bp_ndirs[1]));
        printf("sectors on media: %u\n",
                can2(bb.bp_nsects[0],bb.bp_nsects[1]));

        printf("media descriptor: %u\n",
                uword(bb.bp_media));
        printf("sectors per fat: %u\n",
                can2(bb.bp_spf[0],bb.bp_spf[1]));
        printf("sectors per track: %u\n",
                can2(bb.bp_spt[0],bb.bp_spt[1]));

        printf("heads per device: %u\n",
                can2(bb.bp_nsides[0],bb.bp_nsides[1]));
        printf("hidden sectors: %u\n",
                can2(bb.bp_nhid[0],bb.bp_nhid[1]));
        printf("check sum: %x\n", can2(bb.bp_chk[0],bb.bp_chk[1]));
        return 0;
}
```

See Also
Flopwr, TOS, xbios

Flopver—xbios function 19 (osbind.h)

Verify a floppy disk
#include <osbind.h>
#include <xbios.h>
int Flopver(*buffer, filler, device, sector, track, side, count***)**
char ****buffer*, ******filler*; **int *device, sector, track, side, count*;

Flopver reads a sector from a floppy disk, to verify that it can in fact be read. *buffer* points to a buffer of 1,024 bytes into which a list of bad sectors (if any) will be written. *filler* is not used, and can be initialized to anything. *device* is the number of the floppy disk, and can be set to zero or one. *sector* is the number of the sector to read, one through nine. *track* is the track on which to seek the sector in question, zero through 79. *side* is the side of the disk to read, zero or one. Finally, *count* is the number of sectors to read, and can be no greater than the number of sectors available on a track.

Flopver returns zero if it could read the sector, and returns an error code if it could not. If it found bad sectors, it writes a NUL-terminated string of the numbers of those sectors into *buffer*; otherwise, it writes zero into *buffer*.

Example
For an example of how to use this macro, see the entry for **Flopfmt**.

See Also
Flopfmt, Floprd, Flopwr, TOS, xbios

Flopwr—xbios function 9 (**osbind.h**)
Write sectors on a floppy disk
#include <osbind.h>
#include <xbios.h>
int Flopwr(*buffer, filler, device, sector, track, side, count***)**
char ***buffer*, ******filler*; **int *device, sector, track, side, count*;

Flopwr writes one or more sectors on a floppy disk. *filler* is not used, but must be passed properly for this function to work. *buffer* points to a buffer that holds the information to written onto the disk. *device* is the number of the device, i.e., zero or one. *sector* is the sector at which to begin writing, i.e., one through nine. *track* is the track number to seek to, i.e., zero through 79. *side* is the side of the floppy on which to write, zero or one. Finally, *count* is the number of sectors to write; this can be no greater than the number of sectors on the track.

Flopwr returns zero if it succeeded in writing the information, and returns an error code number if it did not. Note that writing over the boot sector on the disk (sector 1, side 0, track 0) is not recommended.

Example
For an example of how to use this macro, see the entry for **Flopfmt**.

See Also
Floprd, TOS, xbios

fopen—STDIO function (**libc.a/fopen**)
Open a stream for standard I/O
#include <stdio.h>
FILE *fopen (*name, type***) char ****name*, ******type*;

fopen allocates and initializes a **FILE** structure, or *stream*; opens or creates the file *name*; and returns a pointer to the structure for use by other **STDIO** routines. *name* may refer either to a real file or to one of the devices **aux:**, **con:**, or **prn:**. *type* is a string that consists of one or more of the characters "rwab", to indicate the mode of the string, as follows:

r	read ASCII; error if file not found
rb	read binary data
w	write ASCII; truncate if found, create if not found
wb	write binary data
a	append ASCII; no truncation, create if not found
ab	append binary data
rw	read and write ASCII; no truncation, error if not found
rwb	read and write binary data
wr	write and read ASCII; truncate if found, create if not found
wrb	write and read binary data
ar	append and read ASCII; no truncation, create if not found
arb	append and read binary data

r+, **w+**, and **a+** are synonyms for **rw**, **wr**, and **ar**, respectively. The modes that contain **a** set the seek pointer to point at the end of the file, so that data may be appended; all other modes set it to point at the beginning of the file.

Example
This example copies **argv[1]** to **argv[2]** using stdio routines. It demonstrates the functions **fopen**, **fread**, **fwrite**, **fclose**, and **feof**.

```
#include <stdio.h>
char buf[BUFSIZ];

main(argc, argv) int argc; char *argv[]; {
	register FILE *ifp, *ofp;
	register unsigned int n;

	if (argc != 3)
		fatal("Usage: copy source destination");
	if ((ifp = fopen(argv[1], "rb")) == NULL)
		fatal("cannot open input file");
	if ((ofp = fopen(argv[2], "wb")) == NULL)
		fatal("cannot open output file");

	while ((n = fread(buf, 1, BUFSIZ, ifp)) != 0) {
		if (fwrite(buf, 1, n, ofp) != n)
			fatal("write error");
	}
```

```
		if (!feof(ifp))
			fatal("read error");
		if (fclose(ifp) == EOF || fclose(ofp) == EOF)
			fatal("cannot close");
		exit(0);
}

fatal(s) char *s; {
		fprintf(stderr, "copy: %s\n", s);
		exit(1);
}
```

See Also
FILE, freopen, STDIO
The C Programming Language, page 151, 167

Diagnostics
fopen returns **NULL** or if it cannot allocate a **FILE** structure, if the *type* string is nonsense, or if the call to **open** or **creat** fails. Currently, only 20 **FILE** structures can be allocated per program, including **stdin**, **stdout**, and **stderr**.

Fopen—**gemdos** function 61 (**osbind.h**)
Open a file
#include <osbind.h>
long Fopen(*name, mode***) char ***name***; int** *mode*;

Fopen opens a file. *name* points to the file's path name, which must be a NUL-terminated string. *mode* is an integer than encodes the mode in which the file is opened, as follows: zero, read only; one, write only; and two, read or write. **Fopen** returns a file handle, which is understood by TOS.

Example
For examples of how to use this macro, see the entries for **Fseek** and **Fcreate**.

See Also
gemdos, TOS

form_alert—AES function (**libaes.a/form_alert**)
Display an alert box
#include <aesbind.h>
int form_alert(*button, string***) int** *button*; **char ***string*;

form_alert is an AES routine that displays an alert dialogue box on the screen. An alert dialogue box consists of three elements: an icon, which is selected from a predefined set of three; text, which describes the alert; and one or more "exit buttons", or little boxes that the user clicks to indicate what he wants to do.

button defines which exit button is the default; the default button is drawn with a heavier outline and it is the one selected if the user presses the return key instead of using the mouse. The default is set as follows: zero, no default button; one, first exit button; two, second exit button; and three, third exit button.

string points to the string used with the alert box. The string has the following format:

> [*n*] [*text*] [*exit*]

The square brackets are entered literally. *n* refers to the number of the icon you wish to display, as follows:

0	no icon
1	NOTE icon (exclamation point)
2	WAIT icon (question mark)
3	STOP icon (stop sign)

text is the text displayed within the alert box. Note that an alert box can hold no more than five lines of text, each no longer than 40 characters. A vertical bar '|' indicates a line break. *exit* describes the exit buttons. It can have no more than 20 characters. If you want more than one exit button, separate their texts with a vertical bar. For example,

```
[3][Cannot find file|Do you wish to try again?][Quit|Try again]
```

indicates that you want the STOP icon (icon no. 3), that the box is to have two lines of text ("Cannot find file/Do you wish to try again?"), and that you want two exit buttons, one marked "Quit" and the other marked "Try again".

form_alert returns the number of the exit button selected.

Example
This example shows a program called **alert.c**; it opens a text file and displays its contents, and keeps the text on the screen until a key is pressed. The program uses **fsel_input** to accept information from the user, and **form_alert** to handle various error conditions. Note that the line

```
char DIRPATH[80] = "b:\\examples\\*.*";
```

points to the directory examined; you should insert the name of the directory you wish to work with. The default is **a:**.

```
#include <ctype.h>
#include <aesbind.h>

#define CANTOPEN 0
#define NOTASCII 1
#define FOULUP 2
#define UNDEFINED 3
```

```
char DIRPATH[80] = "b:\\examples\\*.*";
static char *STRING[] = {
	"[2][Cannot Open File][Quit|Try Again]",
	"[3][File Is Not ASCII][OK]",
	"[3][Foul-up in fsel_input][OK]",
	"[3][Undefined Error][OK]"
};

FILE *newopen() {
	FILE *tmp;
	char name[80];
	int button;

	name[0] = '\0';
	if (fsel_input(DIRPATH, name, &button) == 0)
		alert(FOULUP);
	else
		return(fopen(name, "r"));
}

main() {
	FILE *fp;
	int ch;

	appl_init();
	while ((fp = newopen()) == NULL) {
		alert(CANTOPEN);
	}

	while ((ch = fgetc(fp)) != EOF) {
		if (isascii(ch))
			putchar(ch);
		else
			alert(NOTASCII);
	}
	evnt_keybd();		/* stop processing until keyboard hit */
	appl_exit();
	exit(0);
}

alert(flag) int flag; {
	int button = 1;
	if(flag > UNDEFINED)
		flag = UNDEFINED;
	if(form_alert(button, STRING[flag]) == 2)
		return;
	appl_exit();
	exit(1);
}
```

See Also
AES, cc, gem, TOS

form_center—AES function (**libaes.a/form_center**)
Center an object on the screen
#include <aesbind.h>
#include <obdefs.h>
int form_center(*picture, location***) OBJECT ****picture*; **Prect** *location*;

form_center is an AES routine that centers an object on the screen.

picture points to the object being manipulated. The type **OBJECT** is defined in the header file **obdefs.h**.

location points to where the object is centered on the screen. It is declared to be of type **Prect**, which is defined in the header file **aesbind.h. Prect** is a structure that consists of four pointers to integers, as follows:

x	X value of centered coordinate
y	Y value of centered coordinate
w	width of centered object
h	height of centered object

form_center always returns one.

Example
For an example of this routine, see the entry for **object**.

See Also
AES, obdefs.h, object, TOS

form_dial—AES function (**libaes.a/form_dial**)
Reserve/free screen space for dialogue
#include <aesbind.h>
int form_dial(*flag, smallbox, bigbox***) int** *flag*; **Prect** *smallbox, bigbox*;

form_dial is an AES routine that either reserves space for a dialogue box, or frees space previously reserved. *flag* indicates whether the space is to be reserved or freed; zero indicates reserve and three indicates free. The space being reserved was originally designed to be a box that grows from *smallbox* to *bigbox*, as shown by the bindings.

Both *smallbox* and *bigbox* are of type **Prect**, which is declared in the header file **aesbind.h. Prect** is a structure that consists of four pointers to integers, as follows:

- **x** X value of centered coordinate
- **y** Y value of centered coordinate
- **w** width of centered object
- **h** height of centered object

form_dial returns zero if an error occurred, and a number greater than zero if one did not.

Example
For an example of this routine, see the entry for **object**.

See Also
AES, form_do, object, TOS

form_do—AES function (**libaes.a/form_do**)
Handle user input in form dialogue
#include <aesbind.h>
int form_do(*tree, object***)**
long *tree*; **int** *object*;

form_do is an AES routine that handles text the user may need to input into an object. *tree* points to the object tree that will accept the text. *object* indicates the object within the tree that has an editable text field; zero indicates that the tree contains no editable text field. **form_do** returns the index of the object that closes the dialogue.

Example
For an example of this routine, see the entry for **object**.

See Also
AES, form_dial, object, TOS

form_error—AES function (**libaes.a/form_error**)
Display a DOS error alert
#include <aesbind.h>
int form_error(*error***)**
int *error*;

form_error is an AES routine that displays a preset DOS error alert. *error* is an integer that indicates which error message you wish to display, as follows:

0	Undefined
1	Undefined
2	Cannot find file or folder
3	Same as **2**
4	No room to open another document
5	Item with this name already exists
6	Undefined
7	Undefined
8	Not enough RAM to run application
9	Undefined
10	Same as **8**
11	Same as **8**
12	Undefined
13	Undefined
14	Undefined
15	Specified drive does not exist
16	Cannot delete current folder
17	Undefined
18	Same as **2**

Note that the above numbers correspond to error codes under MS-DOS. All codes greater than 18 are associated with no specific error message. **form_error** returns the number of the exit button that the user clicked, from one through three. At present, all error alerts have only one exit button.

Example
This example displays the preset error forms.

```
#include <aesbind.h>

main() {
	int counter;

	appl_init();
	for (counter = 0; counter <= 20; counter++)
		form_error(counter);
	appl_exit();
}
```

See Also
AES, TOS

fprintf—STDIO function (**libc.a/printf**)
Format output
#include <stdio.h>
fprintf(*fp*, *format* [, *arg*] ...**)**

FILE **fp*; **char** **format*;

fprintf uses the *format* string to specify an output format for each *arg*, which it then writes into the file *fp*. See **printf** for a description of **fprintf**'s formatting codes.

See Also
printf, sprintf, STDIO
The C Programming Language, page 152

Notes
Because C does not perform type checking, it is essential that an argument match its specification. For example, if the argument is a **long** and the specification is for an **int**, **fprintf** will peel off the first word of that **long** and present it as an **int**.

fputc—STDIO function (**libc.a/fputc**)
Write character to stream
#include <stdio.h>
fputc(*c*, *fp*) **char** *c*; **FILE** **fp*;

fputc writes the character *c* onto file stream *fp*, and returns *c* upon success.

Example
The following example demonstrates **fputc**.

```
#include <stdio.h>
main(){
	FILE *fp, *fout;
	int ch;
	int infile[20];
	int outfile[20];
	printf("Enter name to copy: ");
	gets(infile);
	printf("Enter name of new file: ");
	gets(outfile);

	if ((fp = fopen(infile,"r")) != NULL) {
		if ((fout = fopen(outfile,"w")) != NULL)
			while ((ch = fgetc(fp)) != EOF)
				fputc(ch, fout);
		else printf("Cannot write %s.\n", outfile);
	}

	else printf("Cannot read %s.\n", infile);
	fclose(fp);
	fclose(fout);
}
```

See Also
STDIO

Diagnostics
EOF is returned when a write error occurs, e.g., when a disk runs out of space.

fputs—STDIO function (**libc.a/fputs**)
Write string to stream
#include <stdio.h>
fputs(*string*, *fp***) char ***string*; **FILE ***fp*;

fputs writes *string* onto the stream *fp*. Unlike its cousin **puts**, it does not append a newline character to the end of *string*.

See Also
STDIO
The C Programming Language, page 155

fputw—STDIO function (**libc.a/fputw**)
Write an integer to a stream
#include <stdio.h>
fputw(*word*, *fp***) int** *word*; **FILE ***fp*;

fputw writes *word* to the stream *fp*, and returns the value written.

See Also
STDIO

Diagnostics
EOF is returned when an error occurs. A call to **ferror** may be needed to distinguish this value from a valid data item.

fread—STDIO function (**libc.a/fread**)
Read data from stream
#include <stdio.h>
int fread(*buffer*, *size*, *n*, *fp***) char *buffer;**
unsigned *size*, *n*; **FILE ***fp*;

fread reads *n* items of *size* bytes long each, from stream *fp* into memory location *buffer*, and returns the number of items read.

Example
For an example of how to use this function, see the entry for **fopen**.

See Also
fwrite, STDIO

Diagnostics
fread returns 0 on end of file or error, and the number of items read otherwise.

Fread—**gemdos** function 63 (**osbind.h**)
Read a file
#include <osbind.h>
long Fread(*handle, n, buffer***)**
int *handle*; **long** *n*; **char** **buffer*;

Fread reads *n* bytes from a file opened by **Fopen** or **Fcreate**.

handle is the file handle generated when the file was opened; *buffer* points to the location where the material being read is stored. **Fread** returns *n* if the file was read successfully, and an error code if it was not.

Example
For examples of how to use this macro, see the entries for **Fseek** and **Fcreate**.

See Also
gemdos, TOS

free—General function (**libc.a/malloc**)
Return dynamic memory to free memory pool
free(*ptr***) char** **ptr*;

free helps to manage a program's arena. It returns to the free memory pool memory that had previously been allocated by **malloc** or **calloc**. **free** marks the block indicated by *ptr* as unused and coalesces it with contiguous free blocks. *ptr* must have been obtained from **malloc**, **calloc**, or **realloc**.

Example
For an example of how to use this routine, see the entry for **malloc**. For an example of this function in a TOS application, see the entry for **Fgetdta**.

See Also
arena, calloc, malloc, notmem, realloc, setbuf

Diagnostics
free prints a message and calls **abort** if it discovers that the arena has been corrupted, which most often occurs by storing past the bounds of an allocated block.

Frename—**gemdos** function 86 (**osbind.h**)
Rename a file
#include <osbind.h>
long Frename(*n, oldpath, newpath***) int** *n*;
char **oldpath, newpath*;

Frename renames a file. *oldpath* points to the file's old path name, and *newpath* to its new path name; both names must be NUL-terminated strings. *newpath* must not be the name of an existing file. *n* is reserved for TOS, and must be zero. **Frename** can move a file to another subdirectory, but only on the same disk drive. It returns zero if the file could be renamed, non-zero if it could not.

Example
This example renames a file.

```
#include <stdio.h>
#include <osbind.h>

extern int errno;                       /* global for last error... */

main(argc, argv) int argc; char **argv; {
        int status;

        if (argc < 3) {
                printf("Usage: Frename oldname newname\n");
                Pterm(1);
        }
        if ((status=Frename(0, argv[1], argv[2])) != 0) {
                errno = -status;
                perror("Rename failed");
                Pterm(1);
        }
        printf("File %s renamed to %s\n", argv[1], argv[2]);
        Pterm0();
}
```

See Also
gemdos, TOS

freopen—STDIO function (**libc.a/freopen**)
Open a stream for standard I/O
#include <stdio.h>
FILE *freopen (*name, type, fp***)**
char **name*, **type*; **FILE** **fp*;

freopen reinitializes *fp*, closes the file currently associated with it, opens or creates file *name*, and returns a pointer to the structure for use by other STDIO routines. *name* may refer either to a real file or to one of the devices **aux:**, **con:**, or **prn:**.

type is a string that consists of one or more of the characters "**rwab**" (for read, write, append, binary) to indicate the mode of the stream. For additional discussion of the *type* variable, see **fopen**. **freopen** differs from **fopen** only in that *fp* specifies the stream to be used. Any stream previously associated with *fp* is closed by **fclose**. **freopen** is usually used to change the meaning of **stdin**, **stdout**, or **stderr**.

Example
For an example of how to use **freopen**, see the entry for fgets.

See Also
fopen, STDIO

Diagnostics
freopen returns **NULL** if the *type* string is nonsense or if the file cannot be opened. Currently, only 20 **FILE** structures can be allocated per program, including **stdin**, **stdout**, and **stderr**.

frexp—General function (**libc.a/frexp**)
Separate mantissa and exponent
double frexp(*real*, *ep***) double** *real*; **int** ***ep**;

frexp breaks double-precision floating point numbers into mantissa and exponent. It returns the mantissa *m* of its *real* argument, such that 1/2 <= *m* < 1 or *m*=0, and stores the binary exponent *e* in location *ep*. These numbers satisfy the equation *real* = *m* * 2^*e*.

See Also
atof, ceil, fabs, floor, ldexp, modf

fscanf—STDIO function (**libc.a/scanf**)
Format input from a file
#include <stdio.h>
fscanf(*fp*, *format* [, *arg*] ...)
FILE **fp*; **char** **format*;

fscanf reads the file *fp*, and uses the string *format* to specify a format for each *arg*, which must be a pointer. For more information on **fscanf**'s conversion codes, see **scanf**.

See Also
STDIO
The C Programming Language, page 152

Notes
Because C does not perform type checking, it is essential that an argument match its specification. For that reason, **fscanf** is best used only to process data that you are certain are in the correct data format, such as data previously written out with **fprintf**.

fseek—STDIO function (libc.a/fseek)
Seek on stream
#include <stdio.h>
int fseek(*fp, where, how***)**
FILE ****fp*; **long *where*; **int** *how*;

fseek changes the location where the next read or write operation will occur in stream *fp*. It handles any effects the seek routine might have had on the internal buffering strategies of the system. The arguments *where* and *how* specify the desired seek position. *where* indicates the new seek position in the file; it is measured from the start of the file if *how* equals zero, from the current seek position if *how* equals one, and from the end of the file if *how* equals two.

Example
This example opens file **argv[1]** and prints its last **argv[2]** characters (default, 100). It demonstrates the functions **fseek, ftell,** and **fclose**.

```
#include <stdio.h>
extern long atol();
main(argc, argv) int argc; char *argv[]; {
	register FILE *ifp;
	register int c;
	long nchars, size;

	if (argc < 2 || argc > 3)
		fatal("Usage: tail file [ nchars ]");
	nchars = (argc == 3) ? atol(argv[2]) : 100L;
	if ((ifp = fopen(argv[1], "r")) == NULL)
		fatal("cannot open input file");
	if (fseek(ifp, 0L, 2) == -1)		/* Seek to end */
		fatal("seek error");
	size = ftell(ifp);			/* Find current size */
	size = (size < nchars) ? 0L : size - nchars;
```

```
        if (fseek(ifp, size, 0) == -1)          /* Seek to point */
                fatal("seek error");
        while ((c = getc(ifp)) != EOF)
                putchar(c);                     /* Copy rest to stdout */
        if (fclose(ifp) == EOF)
                fatal("cannot close");
        exit(0);
}

fatal(s) char *s; {
        fprintf(stderr, "tail: %s\n", s);
        exit(1);
}
```

See Also
ftell, STDIO

Diagnostics
For any diagnostic error, **fseek** returns -1; otherwise, it returns zero. Note that if **fseek** goes beyond the end of the file, it will not return an error message until the corresponding read or write is performed.

Fseek—**gemdos** function 66 (**osbind.h**)
Move a file pointer
#include <osbind.h>
long Fseek(*n, handle, mode***) long** *n*; **int** *handle, mode*;

Fseek moves a file pointer. *handle* is the file's handle, which was generated when the file was opened; *n* is a signed long integer that indicates the number of bytes the pointer is to be moved. *mode* contains an integer that encodes the manner in which the pointer is to be moved, as follows: zero, move *n* bytes from beginning of file; one, move *n* bytes from current location; and two, move *n* bytes from the end of the file. **Fseek** returns the number of bytes that the file pointer is now located from the beginning of the file.

Example
This example demonstrates **Fseek**. It copies one file into another.

```
#include <osbind.h>
#include <stat.h>
#include <errno.h>
char buffer[8192];                      /* 8K buffer */

void reverse(buffer,len)
char *buffer; int len;
{
        register char place, *forward, *backward;
        forward = &buffer[0];
        backward = &buffer[len];
```

```
	while (forward < backward) {
		place = *--backward;
		*backward = *forward;
		*forward++ = place;
	}
}

fatal(error, msg)
int error; char *msg;
{
	errno = -error;
	perror(msg);
	exit(1);
}

main(argc, argv)
int argc; char *argv[];
{
	int status, infd, outfd, size;
	DMABUFFER dma;

	if (argc < 3) {
		printf("Usage: Fseek source target\n");
		exit(1);
	}

	if ((infd = Fopen(argv[1], 0)) < 0)
		fatal(infd, argv[1]);
	Fsetdta(&dma);

	if ((status=Fsfirst(argv[1], 0xF7)) != 0)
		fatal(status, argv[1]);
	status = dma.d_fattr & 7;

	if ((outfd = Fcreate(argv[2], status)) < 0)
		fatal(outfd, argv[2]);
	while (dma.d_fsize > 0) {
		if (dma.d_fsize > sizeof(buffer))
			size = sizeof(buffer);
		else
			size = dma.d_fsize;

		Fseek(dma.d_fsize-size, infd, 0);
		if ((status=Fread(infd, (long)size, buffer)) < 0)
			Fdelete(argv[2]), fatal(status, argv[1]);
		reverse(buffer, size);

		if ((status=Fwrite(outfd, (long)size, buffer)) < 0)
			Fdelete(argv[2]), fatal(status, argv[2]);
		dma.d_fsize -= size;
	}
```

```
        Fclose(infd);
        Fclose(outfd);
        printf("File %s copied to file %s.\n", argv[1], argv[2]);
        return 0;
}
```

See Also
Fsnext, gemdos, TOS

Diagnostics
For any diagnostic error, **Fseek** returns -1; otherwise, it returns zero. Note that if **Fseek** goes beyond the end of the file, it will not return an error message until the corresponding read or write is performed.

fsel_input—AES function (libaes.a/fsel_input)

Select a file
#include <aesbind.h>
int fsel_input(*directory, file, button***) char ****directory, file*; **int ****button*;

fsel_input is an AES routine that allows the user to select a file in the current directory, or create a new file. It displays a box on the screen; within the box is a window that shows the contents of *directory*.

The user can use the mouse to scroll through the contents of *directory* and select one; she can also move up or down within the directory tree, or specify a new directory. The box also contains two "exit buttons", one marked "Cancel" and the other marked "OK".

directory, as noted above, points to a buffer that holds the name of the directory being read. Note that *directory* must be large enough to hold the full path name for any file selected, including those selected from subdirectories within the directory first displayed.

To avoid accidentally creating a C-language escape character, be sure to use two backslashes '\\' to separate elements of the path name. The default directory is named **a:**\\. The path name must end with a string that indicates which files you wish to examine in the directory; for example, "*.*" displays all the files in a directory, whereas "*.c" displays only the C programs.

If the user clicks a directory while using this function, the name in the buffer that *directory* points to is altered to reflect this change.

file is the name of the first file in *directory*. It is initialized by AES. If the user selects a file other than the first one in the directory, what *file* points to is also altered to reflect this change.

button points to a integer that indicates which exit button the user selected: zero indicates that she selected the Cancel button, and one indicates that the OK button was selected.

fsel_input returns zero if an error occurred, and a number greater than zero if one did not.

Example
For an example of this function, see the entry for **form_alert**.

See Also
AES, TOS

Fsetdta—**gemdos** function 26 (**osbind.h**)
Set disk transfer address
#include <osbind.h>
#include <stat.h>
**void Fsetdta(*c*) DMABUFFER ****c*;

Fsetdta sets the pointer *c* to the address of a DMA buffer, a 44-byte buffer that can be subsequently used by the macro **Fsfirst**. It returns nothing.

Example
For an example of of this function, see the entry for **Fgetdta**.

See Also
Fgetdta, Fsfirst, gemdos, TOS

Fsfirst—**gemdos** function 78 (**osbind.h**)
Search for first occurrence of a file
#include <osbind.h>
#include <stat.h>
int Fsfirst(*name*, *attrib*) char ****name*; **int *attrib*;

Fsfirst searches for the first occurrence of a file name. *name* points to the file's name, which must be a NUL-terminated string. *attrib* is an integer that encodes the search's attributes, as follows:

0x00	normal files only; no hidden files, subdirectories, system files, or volume labels will match
0x01	include read-only files
0x02	include files hidden from directory search
0x04	include system files
0x08	include volume-label files
0x10	include subdirectory files
0x20	include files that have been written to and closed

Note that if you specify volume label, no other type of file can be searched for. The order in which file matches are found depends on the order in which the files are arranged in the directory, and is not governed by alphabetical order or creation date.

If the search is successful, **Fsfirst** takes the 44-byte DMA buffer that had been created with **Fsetdta**, and fills it as follows: bytes zero through 20, reserved for TOS; byte 21, file attributes; bytes 22-23, the file's time stamp; bytes 24-25, the file's date stamp; bytes 26-29, the file's size; and bytes 30-43, the file's name. The DMA buffer is declared in the header file **stat.h**.

Fsfirst returns **E_OK** (success) if the search succeeded, and **EFILNF** (file not found) if it did not.

Example
For an example of this function, see the entry for **Fgetdta**.

See Also
Fsetdta, Fsnext, gemdos, stat.h, TOS

Fsnext—**gemdos** function 79 (**osbind.h**)
Search for next occurrence of file name
#include <osbind.h>
#include <stat.h>
int Fsnext()

Fsnext continues the search for a file, by using the information that had been written into the 44-byte file name buffer by **Fsfirst** or by a previous call to **Fsnext**. If **Fsnext** finds another file with the given name, it updates the DMA buffer to accommodate the name and attributes of the newly found file. The DMA buffer is declared in the header file **stat.h**.

Fsnext returns **E_OK** (success) if the search was successful, and **ENMFIL** (no more files) if it was not.

Example
For an example of this function see the entry for **Fgetdta**.

See Also
Fsfirst, gemdos, stat.h, TOS

fstat—General function (**libc.a/stat**)
Find file attributes
#include <stat.h>
**fstat(*descriptor, statptr*) FILE ****descriptor*; **struct stat ****statptr*;

fstat returns a structure that contains the attributes of a file. *descriptor* points to the file descriptor, as returned by the library function **fopen**, and *statptr* points to a structure of the type **stat**, which is defined in the header file **stat.h**.

The following summarizes the structure **stat** and defines the permission and file type bits.

```
struct stat {
        dev_t st_dev;
        int_t st_ino;
        unsigned short st_mode;
        short st_nlink;
        short st_uid;
        short st_gid;
        dev_t st_rdev;
        size_t st_size;
        time_t st_atime;
        time_t st_mtime;
        time_t st_ctime;
};

#define S_IJRON 0x01      /* Read-only */
#define S_IJHID 0x02      /* Hidden from search */
#define S_IJSYS 0x04      /* System, hidden from search */
#define S_IJVOL 0x08      /* Volume label in first 11 bytes */
#define S_IJDIR 0x10      /* Directory */
#define S_IJWAC 0x20      /* Written to and closed */
```

The majority of entries in the structure **stat** are there to preserve compatibility with the COHERENT operating system. Most return meaningless values when used on the Atari ST, with the following exceptions: **st_atime**, **st_mtime**, and **st_ctime** all return the time that the file or directory was last modified.

See Also
ls, msh, open, stat, stat.h

Diagnostics
fstat returns -1 if the file is not found or if *statptr* is invalid.

ftell—STDIO function (**libc.a/ftell**)
Return current position of file pointer
#include <stdio.h>
long ftell(*fp***) FILE ***fp*;

ftell returns the current position of the seek pointer. Like its cousin **fseek**, **ftell** takes into account any buffering that is associated with the stream *fp*.

Example
For an example of how to use this function, see the entry for **fseek**.

See Also
fseek, STDIO

function—Definition

A **function** is the C term for a portion of code that is named, can be invoked by name, and that performs a task. Many functions can accept data in the form of arguments, modify the data, and return a value to the statement that invoked it.

Although functions most often are described as though they were nouns, programmers would do well to think of them verbs, for a function's name is the predicate of almost every C statement.

See Also
data types, portability

fwrite—STDIO function (**libc.a/fwrite**)

Write to stream
#include <stdio.h>
int fwrite(*buffer, size, n, fp***)**
char ****buffer*; **unsigned** *size, n*; **FILE ****fp*;

fwrite writes *n* items of *size* bytes each from *buffer* to stream *fp*, and returns the number of items written.

Example
For an example of how to use this function, see the entry for **fopen**.

See Also
fread, STDIO

Diagnostics
fwrite normally returns the number of items written; if an error occurs, the returned value will not be the same as *n*.

Fwrite—**gemdos** function 64 (**osbind.h**)

Write into a file
#include <osbind.h>
long Fwrite(*handle, n, buffer***) int** *handle*; **long** *n*; **char ****buffer*;

Fwrite writes *n* bytes into a file. *handle* is the file handle that was generated when the file was opened by **Fopen** or **Fcreate**. *buffer* points to the material to be written. **Fwrite** returns *n* if the material was written successfully, and an error code if it was not.

Example
For examples of how to use this macro, see the entries for **Fseek** and **Fcreate**.

See Also
gemdos, TOS

gcvt—General function (**libc.a/gcvt**)

Convert floating point numbers to ASCII strings

char *gcvt(*d*, *w*, *buffer*)
double *d*; int *w*; char **buffer*;

gcvt converts floating point numbers to ASCII strings. It converts its argument *d* into a string of numerals that is *w* characters wide and terminated with NUL. Unlike its cousins **ecvt** and **fcvt**, **gcvt** uses a *buffer* that is defined by the caller. *buffer* must be large enough to hold the result. When generating its output, **gcvt** will mimic **fcvt** if possible; otherwise, it mimics **ecvt**. **gcvt** returns *buffer*.

See Also
ecvt, fcvt, frexp, ldexp, modf, printf

gem—Command

Run a GEM-DOS program

gem *command args*

gem allows you to run a GEM-DOS *command* under the micro-shell **msh**. It resets file handle 2 to the **aux:** device. Unlike its cousin, the **tos** command, **gem** enables the mouse cursor.

Note that **gem** does *not* read the environment; you must specify exactly where the program is located, and give its full name.

Because some GEM programs read resource files and expect to find them in the current directory, you should use **gem** with a **cd** command. For example,

```
set game='cd c:\games; gem game.prg; cd'
```

allows you to run the GEM application **game.prg** by typing **$game**. When you exit from **game**, you will be returned to your **HOME** directory.

When you are finished, just exit from the GEM-DOS program in the normal way, and **gem** will return you to **msh**.

See Also
commands, msh

Notes
Some Atari GEM programs appear to depend on the GEM desktop to perform unspecified clean-up after they run, and thus cannot be run through the **gem** command. These programs include Atari Logo and Atari BASIC. Running these programs under **msh** may damage memory-resident programs, such as RAM disks.

gemdefs.h—Header file

GEM structures and definitions

#include <gemdefs.h>

gemdefs.h is a header file that declares structures and definitions useful for programming in the GEM environment. Many of the mnemonics used through GEM programs are also defined in this file.

See Also

AES, header file, TOS, VDI

gemdos—TOS function

Call a routine from GEM-DOS

#include <osbind.h>

extern long gemdos(*n, arg1...argn***);**

gemdos allows you to call a GEM-DOS routine directly from your program. *n* is the number of the routine, and *arg1* through *argn* are the argument numbers to be used with the routine. In most circumstances, it is unnecessary to use **gemdos** directly, for a library of functions that use it are defined in the header file **osbind.h**.

The following functions use **gemdos**:

Cauxin	Read character from serial port
Cauxis	Return serial port input status
Cauxos	Return serial port output status
Cauxout	Write character to serial port
Cconin	Read character from console
Cconis	Return console input status
Cconout	Write character to console
Cconos	Return console output status
Cconrs	Read and edit string from console
Cconws	Write a string to the console
Cnecin	Read character from console, no echo
Cprnos	Check parallel port output status
Cprnout	Write character to parallel port
Crawin	Read raw character from console
Crawio	Perform raw I/O with console
Dcreate	Create a subdirectory
Ddelete	Remove a subdirectory
Dfree	Find free space on disk
Dgetdrv	Return current disk drive
Dgetpath	Return current directory
Dsetdrv	Set the default drive
Dsetpath	Set the current directory
Fattrib	Get/set file attributes
Fclose	Close a file

Fcreate	Create a file
Fdatime	Get/set file's date stamp
Fdelete	Delete a file
Fdup	Duplicate a file's handle
Fforce	Force a file handle
Fgetdta	Get a disk transfer address
Fopen	Open a file
Fread	Read a file
Frename	Rename a file
Fseek	Move a file pointer
Fsetdta	Set disk transfer address
Fsfirst	Search for first occurrence of file
Fsnext	Search for next occurrence of file
Fwrite	Write into a file
Malloc	Allocate dynamic memory
Mfree	Free dynamic memory
Mshrink	Shrink amount of allocated memory
Pexec	Load or execute a process
Pterm	Terminate a process
Pterm0	Terminate a TOS process
Ptermres	Terminate a process but keep in memory
Tgetdate	Get date
Tgettime	Get time
Tsetdate	Set date
Tsettime	Set time
Sversion	Get TOS version number

See Also
osbind.h, TOS

Notes
No **gemdos** function will support a recursive call. Attempting to use a recursive call with a **gemdos** function will crash the system.

Note that all **gemdos** functions are unbuffered. Combining them with buffered I/O routines, such as those in the STDIO library, will lead at best to unpredictable results.

gemout.h—Header file

GEM-DOS file formats and magic numbers

#include <gemout.h>

gemout.h is a header file that declares formats for the GEM-DOS executable files and archives. It also includes a number of "magic numbers" used in handling these formats.

See Also
header file, TOS

Getbpb—bios function 7 (**osbind.h**)

Get pointer to BIOS parameter block for a disk drive

#include <osbind.h>
#include <bios.h>
char *Getbpb(*device***);**
int *device***;**

Getbpb returns a pointer to the BIOS parameter block for a given disk drive. *device* is an integer that indicates which drive you wish to examine: zero, drive A; one, drive B; etc. If the BIOS parameter block cannot be determined for whatever reason, **Getbpb** returns zero.

Example

The following example dumps the BIOS parameter block for the disk in drive B:.

```
#include <osbind.h>
#include <bios.h>

main() {
	struct bpb *bp;
	bp = (struct bpb *) Getbpb(1);
	printf("Disk in drive B:\n");
	printf("\tSector Size:\t%5d bytes\n", bp->bp_recsiz);
	printf("\tCluster Size:\t%5d bytes (%d sectors)\n",
		bp->bp_clsizb, bp->bp_clsiz);

	printf("\tDirectory:\t%5d sectors\n", bp->bp_rdlen);
	printf("\tFAT:\t\t%5d sectors\n", bp->bp_fsiz);
	printf("\tData Clusters:\t%5d\n", bp->bp_numcl);
	printf("\tFlags:\t\t %4x\n", bp->bp_flags);
}
```

See Also
bios, TOS

getc—STDIO macro (**stdio.h**)

Read character from stream

#include <stdio.h>
int getc(*fp***) FILE ***fp***;**

getc is a macro that reads a character from the stream *fp*, and returns an **int**.

See Also
fgetc, getchar, STDIO
The C Programming Language, page 152

Diagnostics
getc returns **EOF** at end of file or on read error.

Notes
Because **getc** is a macro, arguments with side effects probably will not work as expected. Also, **getc** is a complex macro, and its use in expressions of too great a complexity may cause unforeseen difficulties. Use of the function **fgetc** may solve such a problem.

getchar—STDIO macro (**stdio.h**)
Read character from stream
#include <stdio.h>
int getchar()

getchar is a macro that reads a character from the standard input. It is equivalent to **getc(stdin)**.

See Also
getc, STDIO
The C Programming Language, page 144, 152

Diagnostics
getchar returns **EOF** at end of file or on read error.

getcol—Command
Get a color value
getcol *position*

getcol is a command that uses the **xbios** function **Setcolor** to read the color for a position on the current color palette. *position* is the palette position in question, from zero through 15.

See Also
commands, setcolor, Setcolor, TOS

getenv—General function (**libc.a/getenv**)
Get environmental variable
char *getenv(*variable***) char ***variable***;**

A program may read certain variables in its *environment*. This allows the program to accept information that is specific to you. The environment consists of an array of strings, each having the form *VARIABLE=VALUE*. When called with the string *VARIABLE*, **getenv** returns a pointer to the string *VALUE*.

See Also
cc, environment, msh

Diagnostics
When *VARIABLE* is not found or has no value, **getenv** returns **NULL**.

Getmpb—**bios** function 0 (**osbind.h**)
Copy memory parameter block
#include <osbind.h>
#include <bios.h>
void Getmpb(*pointer***);**
char ****pointer*;**

Getmpb tells TOS to copy its memory parameter block into the 24-byte space pointed to by *pointer*. The useful portions of the memory parameter block are described in the example; as of this writing, the memory parameter block does not appear to be utilized by TOS. Note, too, that the lists returned are in system-protected memory; unless the user is in supervisor mode, accessing these lists will generate a bus error.

Example
The following example demonstrates **Getmpb**. It prints out the amount of memory free and memory used.

```
#include <osbind.h>
#include <bios.h>

long chase(cp, mp)
char *cp; register struct mdb *mp;
{
	register long save, total;
	struct mdb mdb;
	printf("%s:\n", cp);
	total = 0;

	while (mp != (struct mdb *)0L) {
		save = Super(0L); mdb = *mp; Super(save);
		total += mdb.md_size;
		printf("\t%06lx: %ld bytes owned by %lx\n",
			mdb.md_base, mdb.md_size, mdb.md_proc);
		mp = mdb.md_next;
	}

	printf("%ld bytes total.\n", total);
}
```

```
main() {
        struct mpb mpb;
        Getmpb(&mpb);
        chase("Free Memory", mpb.mp_free);
        chase("Used Memory", mpb.mp_used);
        return 0;
}
```

See Also
bios, TOS

getpal—Command

Get the color palette settings

getpal

getpal uses the **xbios** function **Setpallete** (*sic*) to read and return the current settings of the color palette.

See Also
commands, setpal, Setpallete, TOS

getphys—Command

Get the base of the physical screen's display

getphys

getphys is a command that uses the **xbios** function **Physbase** to obtain the base of the screen display's physical memory. The address of the base is returned to the standard output.

See Also
commands, Physbase, setphys, TOS

getrez—Command

Get screen's current resolution

getrez

getrez is a command that uses the **xbios** function **Getrez** to read the screen's current resolution. It returns to the standard output a code that indicates the current resolution, as follows: zero indicates high resolution; one, medium resolution; and two, low resolution.

See Also
commands, Getrez, setrez, TOS

Getrez—xbios function 4 (**osbind.h**)
Read the current screen resolution
#include <osbind.h>
#include <xbios.h>
int Getrez()

Getrez reads the current screen resolution, and returns the following:

0	low resolution
1	medium resolution
2	high resolution

Example
This program prints out the current resolution of the video display. For another example, see the entry for **Prtblk**.

```
#include <osbind.h>
#include <xbios.h>

struct reztab { int r_rez; char *r_name; } reztab[] = {
	GR_LOW, "low",
	GR_MED, "medium",
	GR_HIGH, "high",
	-1, "unknown"
};

main() {
	register struct reztab *rp;
	register int rez;
	rez = Getrez();

	for (rp = reztab; rp->r_rez != rez && rp->r_rez != -1; rp += 1)
		;
	printf("Your ST is in %s resolution mode.\n", rp->r_name);
}
```

See Also
TOS, xbios

gets—STDIO function (**libc.a/gets**)
Read line from stream
#include <stdio.h>
char *gets(*s*) char *s;

gets reads characters from the standard input into the string *s*, up to the next newline or EOF. It discards the newline, if any, appends a trailing NUL character, and returns *s*.

See Also
STDIO

Diagnostics
gets returns NULL if an error occurs or if EOF is seen before any characters are read.

Notes
Unlike its cousin **fgets**, **gets** deletes newlines.

Getshift—bios function 11 (**osbind.h**)
Get or set the status flag for shift/alt/control keys
#include <osbind.h>
#include <bios.h>
long Getshift(*flag*) int *flag*;

Getshift gets or sets the status flag for the shift, alt, and control keys. If *flag* is -1, then the status flags of the keys are read and a map returned; if *flag* is any number other than -1, then the flags are set to *flag*, and a map of their previous settings returned. The map is laid out as follows: bit 0, right shift key; bit 1, left shift key; bit 2, control key; bit 3, alt key; and bit 4, caps lock key. If a bit is set to zero, the key is not depressed; if it is set to one, the key is depressed.

Example
This example displays characters, scan codes, and shift states until you type **<ctrl-D>**.

```
#include <osbind.h>
#include <bios.h>
#include <ctype.h>

struct shift { int s_bit; char *s_name; } shift[] = {
	GS_LSH, "left shift",
	GS_RSH, "right shift",
	GS_CTRL, "control",

	GS_ALT,		"alternate",
	GS_CAPS, "caps lock",
	GS_RMB,		"right mouse",
	GS_LMB,		"left mouse",
	0
};

main() {
	register int c, s;
	register long cc;
	register struct shift *sp;
```

```
        do {
                cc = Bconin(BC_CON);
                s = Getshift(-1);
                c = cc;                 /* get low word */
                cc >>= 16;      /* get scan code */
                Bconout(BC_RAW, c);

                if (isascii(c) && ! isprint(c))
                        printf(": ^%c: ", c+'@');
                else
                        printf(":  %c: ", c);
                printf("%02lx:%02x:%02x", cc, c, s);

                for (sp = shift; sp->s_bit > 0; sp += 1)
                        if (s & sp->s_bit)
                                printf("[%s]", sp->s_name);
                printf("\n");
        } while (c != ('D' & (' '-1)));
}
```

See Also
bios, TOS

Gettime—**xbios** function 23 (**osbind.h**)
Read the current time
#include <osbind.h>
#include <xbios.h>
long Gettime()

Gettime reads and returns the intelligent keyboard's setting of the current time. It returns a 32-bit mask whose bits indicate the following:

0-4	no. of two-second increments (0-29)
5-8	no. of minutes (0-59)
9-15	no. of hours (0-23)
16-20	day of the month (1-31)
21-26	month (1-12)
27-31	year (0-119, 0 indicates 1980)

Example
This example gets the keyboard time. Note that if you have not set the keyboard time since you booted your computer, the time returned by this example will not be correct.

```
#include <osbind.h>

main() {
	register unsigned long time;
	int seconds;
	int minutes;
	int hours;
	int day;
	int month;
	int year;

	time = Gettime();			/* Get system time */
	seconds = (time & 0x001F) << 1;		/* Bits  0:4 */
	minutes = (time >> 5)  & 0x3F;		/* Bits  5:10 */
	hours = (time >> 11) & 0x1F;		/* Bits 11:15 */

	day = (time >> 16) & 0x1F; /* Bits 16:20 */
	month = (time >> 21) & 0x0F;		/* Bits 21:24 */
	year = ((time >> 25) & 0x7F)+1980;		/* Bits 25:31 */

	printf("The ATARI ST thinks it is %d sec past %d min\n",
		seconds, minutes);
	printf("past the hour of %d", hours);
	printf(" on %d/%d/%d\n", month, day, year);
}
```

For another example of this function, see the entry for **time**.

See Also
Kgettime, Settime, time, TOS, xbios

Notes
The time data in the bit map returned by **Gettime** is in exactly the reverse order of the data returned by the **gemdos** functions.

getw—STDIO function (**libc.a/getw**)
Read integer from stream
#include <stdio.h>
int getw(*fp*) FILE *fp;

getw reads a word (an **int**) from the stream *fp*.

See Also
getc, STDIO

Notes
getw returns EOF on errors. A call to **feof** or **ferror** may be necessary to distinguish this value from a valid data word. The bytes of the word it receives are assumed to have been written by **putw**, which writes them in the natural byte ordering of the machine.

Giaccess—xbios function 28 (**osbind.h**)
Access a register on the GI sound chip
#include <osbind.h>
#include <xbios.h>
char Giaccess(*data, register***) char** *data*; **int** *register*;

Giaccess accesses a register on the GI sound chip. *register* is the name of the register being accessed, zero through 15. Bit 7 of this variable indicates whether this register is to be read or written to: zero indicates read, one indicates write. *data* is the eight-bit value being passed to the register when this macro is in write mode; if **Giaccess** is in read mode, this value is ignored.

Giaccess returns the value read if in read mode, and a meaningless value if in write mode.

The Atari ST's sound generator is controlled by 16 eight-bit registers. The sound generator itself has three channels, named A, B, and C. Each can be programmed independently. Note that the contents of the address register remain unaltered until reprogrammed, which allows you to use the same data repeatedly without having to resend them. What each register does is listed in the following:

0,1 Set pitch and period length for channel A. The eight bits of register 0 set the pitch, and the first four bits of register 1 control the period length; the lower the number formed by the 12 significant bits of these registers, the higher the pitch of the tone generated.

2,3 Set the pitch and period length for channel B.

4,5 Set the pitch and period length for channel C.

6 The low five bits of this register control the generation of "white noise"; the smaller the value to which these bits are set, the higher the pitch of the noise generated.

7 This register holds an eight-bit map whose bits toggle various aspects of sound generation; for each bit, zero indicates on and one indicates off. The bits control the following functions:

0	Channel A tone
1	Channel B tone
2	Channel C tone
3	Channel A white noise
4	Channel B white noise
5	Channel C white noise
6	Port A; 0=input, 1=output
7	Port B; 0=input, 1=output

8 Bits 0 through 3 set the signal volume for channel A; the settings can be zero through 15, with zero being the softest setting and 15 the loudest. Setting bit 4 indicates that the "envelope" generator, register 13, should be used; in this case, the contents of bits 0 through 3 are ignored.

9 Same as register 8, only for channel B.

10 Same as register 8, only for channel C.

11,12 Control tone generation. A tone is constructed of four aspects: attack, decay, sustain, and release. *Attack* defines how long a tone takes to reach is loudest point; *decay* defines how long that loudest point is held before it softens to the volume that is sustained; *sustain* defines how long the sustained level is held; and *release* defines how long it takes a tone to decay into silence. These registers govern the four aspects of tone generation; register 11 holds the low byte, register 12 the high byte.

13 Bits 0 through 3 set envelope generator's waveform. A tone's "envelope" is the "shape" of the tone generated, which is best studied by experimental listening.

14,15 Control the Atari ST's I/O ports. Register 14 controls port A, and register 15 port B. If set to output by register 7, the contents of these registers can be exported. Note that these ports have nothing to do with sound generation, and are used on the Atari ST to control the floppy disk drives.

Example
This example uses **Giaccess** to set the select lines for the floppy disk drives. It is not recommended that this be done from user programs in general.

```
#include <osbind.h>

prompt(strng)           /* Write prompt; wait for key to be typed */
char *strng;
{
        Cconws(strng);      /* Write the string */
        Crawcin();          /* Wait for a key */
        Cconws("\r\n");     /* CR-LF to console */
}
```

```
main() {
	prompt("Let drives stop; then press any key to continue");
	Giaccess((Giaccess(0,14) & 0xF8),14|0x80);
	prompt("Both lights on... Hit any key");
	Giaccess((Giaccess(0,14) & 0xF8)|2,14|0x80);
	prompt("Drive B selected... Hit any key");
	Giaccess((Giaccess(0,14) & 0xF8)|4,14|0x80);
	prompt("Drive A selected... Hit any key");
	Giaccess((Giaccess(0,14) & 0xF8)|6,14|0x80);
	prompt("Neither drive selected... Hit any key");
	Pterm0();
}
```

See Also
Offgibit, Ongibit, TOS, xbios
Programmable Sound Generator Data Manual

GMT—Definition

GMT is an abbreviation of Greenwich Mean Time, the time recorded at the Greenwich Observatory in England, where by international convention the Earth's 0 meridian is fixed.

See Also
gmtime, localtime, time.h, TIMEZONE

gmtime—Time function (libc.a/ctime)

Convert system time to system calendar structure
#include <time.h>
tm_t *gmtime(*timep*) time_t *timep*;

gmtime converts the internal time from seconds since midnight January 1, 1970 GMT, into fields that give integer years since 1900, the month, day of the month, the hour, the minute, the second, the day of the week, and yearday. It returns a pointer to the structure **tm_t**, which defines these fields, and which is itself defined in the header file **time.h**. Unlike its cousin, **localtime**, **gmtime** returns Greenwich Mean Time (GMT).

Example
For an example of how to use this function, see the entry for **asctime**.

See Also
localtime, time

Notes
gmtime returns a pointer to a statically allocated data area that is overwritten by successive calls.

graf_dragbox—AES function (**libaes.a/graf_dragbox**)

Draw a dragable box

#include <aesbind.h>

int graf_dragbox(*width, height, startx, starty, boundary, &finishx, &finishy***)**

int *width, height, startx, starty, finishx, finishy*; **Rect** *boundary*;

graf_dragbox is an AES routine that allows the user to drag a box within a specified boundary rectangle. The boundary rectangle puts limits on how far the box can be dragged; it can be set to the entire screen, to a window, or to some other boundary.

width and *height* give, respectively the width and height of the box, in rasters. Note that the number of raster on the screen varies with the degree of screen resolution; the following gives the dimensions of the screen in rasters, by resolution:

Resolution	*Width*	*Height*
High	640	400
Medium	640	200
Low	320	200

startx and *starty* give, respectively, the starting X and Y coordinates for the box. *finishx* and *finishy* hold the coordinates after the box has been dragged; these values are set by the function.

boundary is the outline of the boundary rectangle. It is declared to be of type **Rect**, which is defined in the header file **aesbind.h**. **Rect** consists of four elements:

x	X coordinate of rectangle
y	Y coordinate of rectangle
w	width of rectangle
h	height of rectangle

graf_dragbox returns zero if an error occurred, and a number greater than zero if one did not.

Example

For an example of this function, see the entry for **vro_cpyfm**.

See Also

AES, TOS

Notes

graf_dragbox returns when the mouse button is released. If it is called while the mouse button is up, it returns immediately.

graf_growbox—AES function (**libaes.a/graf_growbox**)
Draw a growing box
#include <aesbind.h>
int graf_growbox(*small, big*) Rect *small, big*;

graf_growbox is an AES routine that draws a growing box on the screen. The box drawn by **graf_growbox** does not stay on the screen; this routine is designed merely to add a "star wars"-style flourish to GEM programs. The arguments *small* and *big* are both defined as being of type **Rect**, which is defined in the header file **aesbind.h**. **Rect** consists of four elements:

x	X coordinate of rectangle
y	Y coordinate of rectangle
w	width of rectangle
h	height of rectangle

The box grows from the dimensions described in *small* to those described in *large*. The unit of measure is the number of rasters for the screen; the number of rasters held by the screen varies with the degree of resolution, as follows:

Resolution	*Width*	*Height*
High	640	400
Medium	640	200
Low	320	200

graf_growbox returns zero if an error occurred, and a number greater than zero if one did not.

Example
For an example of this routine, see the entry for **window**.

See Also
AES, gem, graf_shrinkbox, TOS, window

graf_handle—AES function (**libaes.a/graf_handle**)
Get VDI handle
#include <aesbind.h>
int graf_handle(*chwidth, chheight, bwidth, bheight*)
int *chwidth, *chheight, *bwidth, *bheight;

graf_handle is an AES routine that returns the VDI handle for the "virtual workstation", or the physical device on which you are working; it also returns the size of the font with which you are working. It returns the current VDI handle. *chwidth* and *chheight* point, respectively, to the character width and character height of the font being used. *bwidth* and *bheight* point to the width and height of the box in which a character is displayed. In effect, the difference between the character size and the box size governs how much "white space" surrounds each character. These values are set by GEM.

See Also
AES, TOS

graf_mbox—AES function (**libaes.a/graf_mbox**)
Move a box
#include <aesbind.h>
int graf_mbox(*width, height, fromx, fromy, tox, toy*)
int *width, height, fromx, fromy, tox, toy*;

graf_mbox is an AES routine that moves a box without changing its size. *width* and *height* are the dimensions of the box. *fromx* and *fromy* give the original position of the box; *tox* and *toy* the destination position of the box. Note that both of these pairs of coordinates refer to the upper left-hand corner of the box being moved. **graf_mbox** returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, TOS

graf_mkstate—AES function (**libaes.a/graf_mkstate**)
Get the current mouse state
#include <aesbind.h>
int graf_mkstate(*record*) **Mouse** *record*;

graf_mkstate is an AES routine that returns the current mouse state. *record* is declared to be of type **Mouse**, which is a structure of four pointers to integers that is declared in the header file **aesbind.h**, as follows:

x	X coordinate of mouse pointer
y	Y coordinate of mouse pointer
b	button state when event occurred
k	state of control, alt, and shift keys: 0x0: all keys up 0x1: right shift key down 0x2: left shift key down 0x4: control key down 0x8: alt key down

These values are set by GEM.

graf_mkstate always returns one.

See Also
AES, TOS

graf_mouse—AES function (**libaes.a/graf_mouse**)
Change the shape of the mouse pointer
#include <aesbind.h>
int graf_mouse(*form, definition***) int** *form*; **char** **definition*;

graf_mouse is an AES routine that changes the mouse pointer from the default arrow to another shape. *form* is an integer that indicates what new shape you want, as follows:

0	Arrow (default)
1	Vertical line
2	Bee
3	Hand with pointing finger
4	Hand with extended fingers
5	Thin cross hairs
6	Thick cross hairs
7	Outlined cross hairs
255	Use user-described shape
256	Hide mouse pointer
257	Display mouse pointer

definition points to a 35-word block in which the user has specified her new pointer shape. This argument is ignored if *form* is any value other than 255.

graf_mouse returns zero if an error occurred, and a number greater than zero if one did not.

Example
The following example cycles through the preset shapes for the mouse pointer.

```
#include <aesbind.h>

main() {
	int counter;
	int nowhere;			/* Someplace to point */

	appl_init();
	for (counter = 0; counter <=7; counter++) {
		graf_mouse(counter, &nowhere);
		evnt_keybd();		/* Halt until a key it typed */
	}

	appl_exit();
	exit(0);
}
```

For further examples, see the entries **evnt_multi**, **object**, **window**.

See Also
AES, **object**, **TOS**, **window**

graf_rubbox—AES function (**libaes.a/graf_rubbox**)
Draw a rubber box
#include <aesbind.h>
int graf_rubbox(*box, newwidth, newheight***)**
Rect *box*; **int** **newwidth*, **newheight*;

graf_rubbox is an AES routine that draws a "rubber box" on the screen; a rubber box is one whose dimensions can be altered by the user. *box* defines the initial dimensions of the rubber box. It is of the type **Rect**, which is defined in the header file **aesbind.h**. **Rect** consists of four elements:

x	X coordinate of rectangle
y	Y coordinate of rectangle
w	width of rectangle
h	height of rectangle

newwidth and *newheight* point to the values for width and height to be set by the user.

This routine can be used to define a block of screen area that can be copied elsewhere. For example, the GEM desktop routine that allows you to click more than one file at a time employs **graf_rubbox**.

graf_rubbox returns zero if an error occurred, and a number greater than zero if one did not.

Example
For an example of this routine, see the entry for **v_bar**.

See Also
AES, **TOS**

Notes
This routine is often called **graf_rubberbox** in other bindings.

graf_shrinkbox—AES function (**libaes.a/graf_shrinkbox**)
Draw a shrinking box
#include <aesbind.h>
int graf_shrinkbox(*smallbox, bigbox***) Rect** *smallbox, bigbox*;

graf_shrinkbox is an AES routine that draws a shrinking box on the screen. The box drawn by **graf_shrinkbox** does not stay on the screen; this routine is designed merely to add a "star wars"-style flourish to GEM programs. The arguments *smallbox* and *bigbox* are both defined as being of type **Rect**, which is defined in the header file **aesbind.h. Rect** consists of four elements:

x	X coordinate of rectangle
y	Y coordinate of rectangle
w	width of rectangle
h	height of rectangle

The box grows from the dimensions described in *smallbox* to those described in *large*. The unit of measure is the number of rasters for the screen, as follows:

Resolution	*Width*	*Height*
High	640	400
Medium	640	200
Low	320	200

graf_shrinkbox returns zero if an error occurred, and a number greater than zero if one did not.

Example
For an example of how to use this routine, see the entry for **window**.

See Also
AES, gem, graf_growbox, TOS

graf_slidebox—AES function (**libaes.a/graf_slidebox**)
Track the slider within a box
#include <aesbind.h>
#include <obdefs.h>
int graf_slidebox(*tree, parent, slider, direction***)**
char ***tree*; **int *parent, slider, direction*;

graf_slidebox is an AES routine that tracks the movement of the "slider" within a box. The "slider" is the area of the window that the user can click to scroll through the contents of the file or directory being displayed in the window.

tree points to the address of the object tree that contains the slider. *parent* is the index of the parent object within the object tree, and *slider* is the index of the slider object. *direction* is the direction of movement relative to the position of the parent object: zero indicates horizontal movement and one indicates vertical movement. **graf_slider** returns the position of the center of the slider relative to the parent object. If movement is vertical, then zero indicates the topmost position and 1,000 the bottom-most; and if movement is horizontal, then zero indicates the leftmost position and 1,000 the rightmost.

See Also
AES, TOS

graf_watchbox—AES function (**libaes.a/graf_watchbox**)
Draw a watched box
#include <aesbind.h>
#include <obdefs.h>
int graf_watchbox(*tree, object, insidepattern, outsidepattern***)**
OBJECT ***tree**; **int** *object, insidepattern, outsidepattern*;

graf_watchbox is an AES routine that draws a "watchable box", that is, a box that the screen manager can poll to see if the mouse pointer is inside it or outside it. The user must hold down the leftmost mouse button while moving the pointer; **graf_watchbox** returns the position the pointer was at when the button was released.

tree points to object tree that produces the box in question. *object* is the index of this object within the tree. *insidepattern* and *outsidepattern* indicate, respectively, the pattern used to fill the area within the box and outside the box, as follows:

1	normal
2	selected
3	crossed
4	checked
5	outlined
6	shadowed

graf_watchbox returns a value that indicates whether the mouse pointer was inside or outside the box when the button was released: zero indicates outside, and one indicates inside.

See Also
AES, TOS

handle—Definition

A **handle** is a generic term for a unique identifier used by TOS and GEM. Three types of handles are commonly used: file handles, workstation handles, and process handles.

A *file handle* is identifies a source of bits; it can refer either to a file on disk or to a character device. File handles are returned by **fopen**, **fcreat**, and **fdup**, and are used by **fwrite**, **fread**, and **fseek**.

A *workstation handle* is used by the GEM VDI to identify a virtual device. It is returned by the routines **v_opnwk** and **v_opnvwk**, and is always the first argument accepted by a VDI routine.

A *process handle* identifies a process that runs under the AES. At present, these handles have only limited use, because the AES currently can run only one process at a time.

See Also
AES, VDI, UNIX routines

header file—Definition

A *header file* is a file of text that contains definitions, declarations, and structures commonly used in a given situation. By tradition, a header file always has the suffix ".h". Header files are invoked within a C program by the command **#include**, which is read by **cpp**, the C preprocessor; for this reason, they are also called "include files".

Header files are one the most useful tools available to a C programmer. They allow you to put into one place all of the information that the different modules of your program share. Proper use of header files will make your programs much easier to maintain and port to other environments.

See Also
#include, math.h, portability, stdio.h

help—Command

Print concise description of command
help *command*

help prints a concise description of the options available for each specifed *command*. If the *command* is omitted, **help** prints a simple description of itself. The primary purpose of **help** is to refresh the memory of a user who has forgotten a *command* option.

Information used by help is kept in the file named **helpfile**. Information about a *command* begins with a line

#command

and ends with the next line beginning with '#'. If you wish, you can edit this file and add new descriptions for commands that you want to run under **msh**.

See Also
commands, msh

hidemouse—Command
Hide the mouse pointer
hidemouse

hidemouse is a command that uses the function **lineaa** to hide the mouse pointer. Note that if **hidemouse** is used when the mouse pointer is already hidden, the mouse pointer will need to be called twice before it reappears.

See Also
commands, Line A, showmouse, TOS

HOME—Environmental parameter
HOME names where the micro-shell **msh** should look for a file when no other directory is specified. For example, if you type the **cd** command without an argument, **msh** will change the directory to the one you named as the **HOME** directory.

It is set with the **setenv** command.

See Also
msh, setenv

horizontal tab—Definition
Mark Williams C recognizes the literal character '\t' for the ASCII horizontal tab character HT (octal 011). This character may be used as a character constant or in a string constant, like the other character constants: '\a', which rings the audible bell on the terminal; '\b', to backspace; '\f', to pass a formfeed command to the printer; '\r', for a carriage return; and '\v', for a vertical tab character.

See Also
ASCII, character constant

htom—Command
Redraw screen from high to medium resolution
htom

htom is a command that redraws the screen, moving from high to medium resolution.

See Also
commands, ltom, mtoh, mtol, TOS

hypot—Mathematics function (**libm.a/hypot**)
Compute hypotenuse of right triangle
#include <math.h>
double hypot(*x*, *y*) **double** *x*, *y*;

hypot computes the hypotenuse, or distance from the origin, of its arguments *x* and *y*. The result is the square root of the sum of the squares of *x* and *y*.

Example
For an example of this function, see the entry for **acos**. For an example of its use in a GEM-DOS application, see the entry for **v_circle**.

See Also
cabs, mathematics library

Ikbdws—xbios function 25 (**osbind.h**)

Write a string to the intelligent keyboard device

#include <osbind.h>
#include <xbios.h>
void Ikbdws(*number, buffer***) int** *number*; **char** **buffer*;

Ikbdws writes a string of characters to the intelligent keyboard. *number* is the number of characters to write, minus one, and *buffer* points to the buffer where these characters are kept.

The Atari ST's intelligent keyboard can accept many commands that affect the keyboard itself, the mouse, and the joystick. For more information on how the intelligent keyboard manipulates these devices, see the entry for **Kbdvbase**.

See Also
Gettime, Kbdvbase, Settime, TOS, xbios

INCDIR—Environmental parameter

INCDIR names the default directory in which **cc** looks for files to be included during compilation. The directory that contains the source files and directories named in the **-I** option to the **cc** command are also searched for include files. To set **INCDIR**, use the **setenv** command.

See Also
#include, msh, setenv

#include—Definition

#include <*file***.h>**
#include "*file***.h"**

#include is a statement processed by the C preprocessor **cpp**. Its operation is simple: the preprocessor replaces the **#include** statement with the contents of *file*.**h**.

The name of the file can be enclosed within angle brackets (**<***file***.h>**) or quotation marks ("*file*.h"). Angle brackets tell **cpp** to look for *file*.**h** in the directories named with the *-I* option to the **cc** command line, and then in the standard directory, in this instance the directory named by the **INCDIR** environmental parameter. Quotation marks tell **cpp** to look for *file*.**h** in the source file's directory, then in directories named with the **-I** option, and then in the standard directory.

Files that are called with **#include** statements are called *header files* or *include files*.

See Also
cpp, header file, msh
The C Programming Language, page 207

include file—Definition
Include file is another name for a **header file.**

See Also
header file

index—String function (**libc.a/index**)
Find a character in a string
char *index(*string, c***) char ****string***; char** *c***;**

index scans the given *string* for the first occurrence of character *c*. If *c* is found, **index** returns a pointer to it. If it is not found, **index** returns NULL.

See Also
string
The C Programming Language, page 67

Initmous—**xbios** function 0 (**osbind.h**)
Initialize the mouse
#include <osbind.h>
#include <xbios.h>
void Initmous(*type, parameter, vector***)**
int *type***; char ****parameter***; long** *vector***;**

Initmous initializes the mouse, and returns nothing.

type indicates the mode into which the mouse is to be set, as follows:

0	turn mouse off
1	enable in relative mode
2	enable in absolute mode
3	unused
4	enable in keycode mode

parameter is the address of the 14-byte parameter block. Bytes 0 through 3 are used under all modes; bytes 4 through 11 are used only if the mouse is initialized into absolute mode. The parameter block's bytes indicate the following:

0	non-zero, set Y axis 0 at bottom; zero, set Y axis 0 at top
1	set the parameter for command to set mouse buttons
2	set parameter for X axis threshhold-scale-delta
3	set parameter for Y axis threshhold-scale-delta

4	most significant byte (MSB) for mouse's absolute maximum position on X axis
5	least significant byte (LSB) for mouse's absolute maximum position on X axis
6	MSB for mouse's absolute maximum position on Y axis
7	LSB for mouse's absolute maximum position on Y axis
8	MSB for mouse's initial position on X axis
9	LSB for mouse's initial position on X axis
A	MSB for mouse's initial position on Y axis
B	LSB for mouse's initial position on Y axis

Finally, *vector* gives the mouse's interrupt vector routine.

See Also
TOS, xbios

int—Definition

An **int** is the most commonly used numeric data type, and is normally used to encode integers. On the 68000, as on most microprocessors, **sizeof int** equals 2, that is, two **chars** (15 bits plus a sign bit); therefore, an **int** can contain values from -32768 to +32767. An **int** normally is sign extended when cast to a larger data type; an **unsigned int**, however, will be zero extended.

See Also
data types, declarations, long

interrupt—Definition

An **interrupt** is an interruption of the sequential flow of a program. It can be generated by the hardware, from within the program itself, or from the operating system.

The functions **bios, gemdos**, and **xbios** all employ traps, a form of interrupt, to perform their respective tasks.

See Also
bios, gemdos, xbios

Iorec—xbios function 14 (osbind.h)

Set the I/O record
#include <osbind.h>
#include <xbios.h>
iorec *Iorec(*device***) int** *device***;**

Iorec returns a pointer to a serial device's input buffer record. *device* is an integer that encodes the serial device: the legal settings are 0, 1, or 2, for the RS-232 port, the keyboard, or the musical instrument device interface (MIDI) port, respectively.

As noted, **Iorec** returns a pointer to the device's input buffer record. The record is a structure that is laid out as follows:

```
struct iorec {
	char *io_buff;			/* Buffer */
	short io_bufsiz;		/* Buffer size in bytes */
	short io_head;			/* Current write pointer */
	short io_tail;			/* Current read pointer */
	short io_low;			/* Low water mark, unstop line */
	short io_high;			/* High water mark, stop line */
}
```

buffer points to the device's buffer. *size* is the buffer's size; *high* is its "high water mark", or where an XOFF is sent to the transmitting device; and *low* is its "low water mark", or the point where an XON is sent to the transmitting device. Finally, *head* is the head index and *tail* the tail index. Note that for the RS-232 port, the input-buffer record is followed by an output-buffer record that is structured exactly the same.

Example
This example examines all of the input devices and displays their buffers. For an example of using this function from the **\auto** directory, see the entry for **\auto**.

```
#include <osbind.h>
#include <xbios.h>

iodump(ptr)
register struct iorec *ptr; {
	int ccount;

	if ((ccount = ptr->io_tail - ptr->io_head) < 0)
		ccount += ptr->io_bufsiz;

	printf("Buffer at %lx has %d out of %d characters in it.\n",
		ptr->io_buff, ccount, ptr->io_bufsiz);
	printf("LWM at %d characters, HWM at %d characters\n",
		ptr->io_low, ptr->io_high);
}

main() {
	struct iorec *bp;

	bp = Iorec(0);			/* get I/O buffer for serial port */
	printf("Serial port input buffer:\n");
	iodump(bp);
	printf("Serial port output buffer:\n");
	bp++;
```

```
        iodump(bp);
        bp = Iorec(1);              /* Now for the keyboard */
        printf("Keyboard input buffer:\n");
        iodump(bp);

        bp = Iorec(2);              /* MIDI input buffer */
        printf("MIDI input buffer:\n");
        iodump(bp);
}
```

See Also
TOS, **xbios**

isalnum—ctype macro (**ctype.h**)
Check if a character is a number or letter
#include <ctype.h>
isalnum(*c*) int *c*;

isalnum tests whether the argument *c* is alphanumeric (**0-9**, **A-Z**, or **a-z**). It returns non-zero if *c* is of the desired type, zero if it is not. **isalnum** assumes that *c* is an ASCII character or EOF.

Example
For an example of how to use this macro, see the entry for ctype.

See Also
ctype

isalpha—ctype macro (**ctype.h**)
Check if a character is a letter
#include <ctype.h>
isalpha(*c*) int *c*;

isalpha tests whether the argument *c* is a letter (**A-Z** or **a-z**). It returns non-zero if *c* is, zero if it is not. **isalpha** assumes that *c* is an ASCII character or EOF.

Example
For an example of this macro, see the entry for **ctype**.

See Also
ctype

isascii—ctype macro (**ctype.h**)
Check if a character is an ASCII character
#include <ctype.h>
isascii(*c*) int *c*;

isascii tests whether the argument *c* is an ASCII character (0 <= *c* <= 0177). It returns non-zero if *c* is an ASCII character, zero if it is not. Many other **ctype** macros will fair if passed non-ASCII values other than EOF.

Example
For an example of how to use this macro, see the entry for **ctype**. For an example of its use in a TOS application, see the entry for **Fgetdta**.

See Also
ASCII, ctype

iscntrl—ctype macro (**ctype.h**)
Check if a character is a control character
#include <ctype.h>
iscntrl(*c*) int *c*;

iscntrl tests whether the argument *c* is a control character (including a newline character) or a delete character. It returns non-zero if *c* is of the desired type, zero if it is not. **iscntrl** assumes that *c* is an ASCII character or **EOF**.

Example
For an example of how to use this macro, see the entry for **ctype**.

See Also
ctype

isdigit—ctype macro (**ctype.h**)
Check if a character is a numeral
#include <ctype.h>
isdigit(*c*) int *c*;

isdigit tests whether the argument *c* is a numeral (**0-9**). It returns non-zero if *c* is of the desired type, zero if it is not. **isdigit** assumes that *c* is an ASCII character or **EOF**.

Example
For an example of how to use this macro, see the entry for **ctype**.

See Also
ctype

isleapyear—Time function (**libc.a/isleapyear**)
Indicate if a year was a leap year
#include <time.h>
int isleapyear(*year*) int *year*;

isleapyear indicates whether a given year A.D. is a leap year or not. *year* is the year A.D. in which you are interested. **isleapyear** returns zero if *year* was not a leap year, and a number greater than zero if it was.

See Also
dayspermonth, time, time.h

islower—ctype macro (**ctype.h**)
Check if a character is a lower-case letter
#include <ctype.h>
islower(*c*) int *c*;

islower tests whether the argument *c* is a lower-case letter (**a-z**). It returns non-zero if *c* is of the desired type, zero if it is not. **islower** assumes that *c* is an ASCII character or EOF.

Example
For an example of how to use this macro, see the entry for **ctype**.

See Also
ctype

isprint—ctype macro (**ctype.h**)
Check if a character is printable
#include <ctype.h>
isprint(*c*) int *c*;

isprint is a macro that tests whether the argument *c* is printable, i.e, whether it is neither a delete nor a control character. It returns non-zero if *c* is of the desired type, zero if it is not. **isprint** assumes that *c* is an ASCII character or EOF.

Example
For an example of how to use this macro, see the entry for **ctype**.

See Also
ctype

ispunct—ctype macro (**ctype.h**)
Check if a character is a punctuation mark
#include <ctype.h>
ispunct(*c*) int *c*;

ispunct tests whether the argument *c* is a punctuation mark, i.e., neither an alphanumeric character nor a control character. It returns non-zero if the character tested is of the desired type, zero if it is not. **ispunct** assumes that *c* is an ASCII character or EOF.

Example
For an example of how to use this macro, see the entry for **ctype**.

See Also
ctype

isspace—ctype macro (**ctype.h**)
Check if a character prints white space
#include <ctype.h>
isspace(*c*) int *c*;

isspace tests whether the argument *c* is a space, tab, newline, carriage return, or form-feed character. It returns non-zero if *c* is of the desired type, zero if it is not. **isspace** assumes that *c* is an ASCII character or EOF.

Example
For an example of how to use this macro, see the entry for **ctype**.

See Also
ctype

isupper—ctype macro (**ctype.h**)
Check if a character is an an upper-case letter
#include <ctype.h>
isupper(*c*) int *c*;

isupper tests whether the argument *c* is an upper-case letter (A-Z). It returns non-zero if *c* is of the desired type, zero if it is not. **isupper** assumes that *c* is an ASCII character or EOF.

Example
For an example of how to use this macro, see the entry for **ctype**. For an example of its use in a TOS application, see the entry for **Fgetdta**.

See Also
ctype

j0—Mathematics function (**libm.a/j0**)

Compute Bessel function

#include <math.h>
double j0(*z*) double *z*;

j0 takes the argument *z* and computes the Bessel function of the first kind for order 0.

Example

This example, called **bessel.c**, demonstrates all of the Bessel functions. Compile it with the following command line

```
cc -f bessel.c -lm
```

to include floating-point functions and the mathematics library.

```
#include <math.h>
dodisplay(value, name)
double value; char *name;
{
	if (errno)
		perror(name);

	else
		printf("%10g %s\n", value, name);
	errno = 0;
}

#define display(x) dodisplay((double)(x), "x")
main() {
	extern char *gets();
	double x;
	char string[64];

	for(;;) {
		printf("Enter number: ");
		if(gets(string) == 0)
			break;
		x = atof(string);

		display(x);
		display(j0(x));
		display(j1(x));
		display(jn(0,x));

		display(jn(1,x));
		display(jn(2,x));
		display(jn(3,x));
	}
}
```

See Also
j1, jn, mathematics library

j1—Mathematics function (**libm.a/j1**)
Compute Bessel function
#include <math.h>
double j1(*z*) double *z*;

j1 takes the argument *z* and computes the Bessel function of the first kind for order 1.

Example
For an example of this function, see the entry for **j0**.

See Also
j0, jn, mathematics library

jday_to_time—Time function (**libc.a/jday_to_time**)
Convert Julian date to system time
#include <time.h>
time_t jday_to_time(*time*) jday_t *time*;

jday_to_time converts Julian time to system time. *time* is the Julian time to be converted. It is of type **jday_t**, which is defined in the header file **time.h**. **jday_t** is a structure that consists of two **unsigned longs**. The first gives the number of the Julian day, which is the number of days since the beginning of the Julian calendar (January 1, 4713 B.C.). The second gives the number of seconds since midnight of the given Julian day.

jday_to_time returns the Julian time as converted to type **time_t**; this type is defined in the header file **time.h** as being equivalent to a **long**. Mark Williams C defines the current system time as being the number of seconds from January 1, 1970, 0h00m00s GMT, which is equivalent to the Julian day 2,440,587.5.

See Also
jday_to_tm, time, time.h, time_to_jday, tm_to_jday

Note
This function mainly is of use to astronomers, geographers, and historians.

jday_to_tm—Time function (**libc.a/jday_to_tm**)
Convert Julian date to system calendar format
#include <time.h>
tm_t *jday_to_tm(*time*) jday_t *time*;

jday_to_tm converts Julian time to the system calendar format. *time* is the Julian time to be converted. It is of type **jday_t**, which is defined in the header file **time.h**. **jday_t** is a structure that consists of two **unsigned longs**. The first gives the number of the Julian day, which is the number of days since the beginning of the Julian calendar (January 1, 4713 B.C.). The second gives the number of seconds since midnight of the given Julian day.

jday_to_tm returns a pointer to a copy of the structure **tm_t**, which is defined in the header file **time.h**. For more information on this structure, see the Lexicon entry for **time**.

See Also
jday_to_time, time, time.h, time_to_jday, tm_to_jday

Note
This function is of use mainly to astronomers, geographers, and historians.

Jdisint—xbios function 26 (**osbind.h**)
Disable interrupt on muli-function peripheral device
#include <osbind.h>
#include <xbios.h>
void Jdisint(*number***) int** *number*;

Jdisint disables an interrupt on the multi-function peripheral device, and returns nothing. *number* is the number of the interrupt to disable. For a table of interrupt codes, see the entry for **Mfpint**.

See Also
Jenabint, Mfpint, TOS, xbios

Jenabint—xbios function 27 (**osbind.h**)
Enable a multi-function peripheral port interrupt
#include <osbind.h>
#include <xbios.h>
void Jenabint(*number***) int** *number*;

Jenabint enables the multi-function peripheral (MFP) interrupt, and returns nothing. *number* is the number of the interrupt to disable. For a table of interrupts, see the entry for **Mfpint**.

See Also
Jdisint, Mfpint, TOS, xbios

jn—Mathematics function (**libm.a/jn**)
Compute Bessel function

#include <math.h>
double jn(*n*, *z*) int *n*; double *z*;

jn takes an argument *z* and computes the Bessel function of the first kind for order *n*.

Example
For an example of this function, see the entry for **j0**.

See Also
j0, j1, mathematics library

Kbdvbase—xbios function 34 (**osbind.h**)

Return a pointer to the keyboard vectors

#include <osbind.h>
#include <xbios.h>
kbdvbase *Kbdvbase()

Kbdvbase returns a pointer to a structure that holds the following elements:

```
struct kbdvbase {
        void (*kb_midivec)();          /* MIDI input data vector */
        void (*kb_vkbderr)();          /* keyboard error vector */
        void (*kb_vmiderr)();          /* MIDI error vector */
        void (*kb_statvec)();          /* keyboard status packet */
        void (*kb_mousevec)();         /* keyboard mouse packet */
        void (*kb_clockvec)();         /* keyboard clock packet */
        void (*kb_joyvec)();           /* keyboard joystick packet */
        void (*kb_midisys)();          /* system midi vector */
        void (*kb_kbdsys)();           /* system keyboard vector */
};
```

midivec points to a routine that moves data from the musical instrument digital interface (MIDI) into the MIDI buffer.

kb_vkbderr and **kb_vmiderr** point to routines that are called whenever an error condition is detected, respectively, on the intelligent keyboard or on the MIDI.

kb_statvec, kb_mousevec, kb_clockvec, and **kb_** point to routines that process data received from, respectively, the intelligent keyboard status handler, the mouse, the clock, and the joystick.

Finally, **kb_midisys** and **kb_ikbdsys** point to routines that call handlers when characters become available for, respectively, the MIDI and the intelligent keyboard.

Manipulating peripheral devices

By default, the keyboard reports each make/break contact on the joystick port, each make/break contact on the mouse buttons, and each movement of the mouse that exceeds a preset threshold. Each report consists of a "packet" of three bytes that indicate which device is changing and what change took place. Note that the packet for the joystick has been documented elsewhere as consisting of two bytes; this is incorrect.

The joystick packets consist of three bytes: The first is always 0xFF, which indicates joystick event on port 1; the second is filler, and is always 0x00; and third records the closed switches on the joystick as set bits in the low nybble. Technically, the high bit of the third byte should encode the state of the joystick fire button. In the default set-up, the fire button is set to the left mouse button. This will change if you instruct the keyboard to adopt some other reporting mode.

The mouse packets consist of three bytes: The first is 0xF8, which indicates relative mouse event and encodes the state of the mouse buttons and joystick fire button in the low bits of the low nybble The second and third encode, respectively, the relative X- and Y-axis motion as signed characters.

If you do not have a joystick, you can simulate one by plugging your mouse into the joystick port. The mouse quadrature signals show up as the *north south east west* switch closure bits in the joystick packet. In addition, the left mouse button still shows up as a mouse event, but the right button is inoperative.

Example
The following example monitors the keyboard's mouse and joystick vectors.

```
#include <osbind.h>
#include <bios.h>
#include <xbios.h>

union {
	char k_c[4];                    /* translate four-character packet ... */
	long k_s;                       /* ... into a long */
} kst;                                  /* one for joystick and mouse */
long ktm;                               /* packet time stamp */

kbdvec(p) char *p;
{
	kst.k_c[0] = *p++;              /* store four byte packet */
	kst.k_c[1] = *p++;              /* NB: 'p' could be an odd address */
	kst.k_c[2] = *p++;
	kst.k_c[3] = *p++;
	ktm = *((long *)0x4BA);         /* system 200hz clock tick */
}

main()
{
	register struct kbdvbase *kbp;
	register void (*xx_joyvec)(), (*xx_mousevec)();
	register long ks, kt;

	kbp = Kbdvbase();               /* keyboard vector table */
	xx_joyvec = kbp->kb_joyvec;     /* save old joystick vector */
	kbp->kb_joyvec = kbdvec;        /* install new joystick vector */
	xx_mousevec = kbp->kb_mousevec; /* ditto for mouse */
	kbp->kb_mousevec = kbdvec;
	ks = kst.k_s;                   /* initialize state record */
```

```
        while (Bconstat(BC_CON) == 0) {    /* i.e., until a key is struck */
                if (ks != kst.k_s) {        /* new event? */
                        ks = kst.k_s;       /* then report new state ... */
                        kt = ktm;           /* ... and timestamp */
                        printf("%08lx %lu\n", ks, kt);
                }
        }
        Bconin(BC_CON);                      /* clear keystroke */
        kbp->kb_joyvec = xx_joyvec;          /* !RESTORE VECTORS! */
        kbp->kb_mousevec = xx_mousevec;      /* !OR YOU BOMB ON THE NEXT EVENT! */
        return 0;
}
```

See Also
TOS, xbios

kbrate—Command
Reset the keyboard's repeat rate
kbrate *start, delay*

kbrate uses the **xbios** function **Kbrate** to reset the keyboard's repeat rate. *start* is the amount of time to pass before repeating begins, and *delay* is the time interval between repeats. Both are measured in "system ticks", each tick being 20 milliseconds long. For example, the command

```
kbrate 50 5
```

tells the system that a key must be held down half a second before repeating begins, and then repeating will occur ten times a second thereafter.

See Also
commands, TOS

Kbrate—xbios function 35 (**osbind.h**)
Get or set the keyboard's repeat rate
#include <osbind.h>
#include <xbios.h>
int Kbrate(*start, delay***) int** *start, delay*;

Kbrate gets or sets the keyboard's repeat rate. Rates are set as multiples of "system ticks"; each tick is 20 milliseconds long. *first* sets the number of ticks to wait before a key begins to repeat; *delay* sets the number of ticks to wait between repeats. If either variable is set to 0xFFFF (-1), that value is not changed. Kbrate returns an **int** that holds the previous setting of the keyboard rate: the value of *first* is written as the high byte, and the value of *delay* as the low byte.

Example
This example displays the keyboard repeat rate and delay period; it then sets them to unreasonable values, lets the user try them out, and finally resets the previous values. For an example of using this function from the **\auto** directory, see the entry for **\auto**.

```
#include <osbind.h>
#define DEL 10
#define RT 1

main() {
	int old_rate;
	int old_delay;
	char c;

	old_rate = Kbrate(DEL,RT); /* Set the new rate. */
	old_delay = (old_rate>>8)&0xFF;
	old_rate &= 0xFF;
	printf("The repeat delay is %d/50 seconds\n", old_delay);
	printf("and repeat rate is once every %d/50 seconds\n",
		old_rate);
	printf("Rates are changed to delay=%d, rate=%d\n", DEL, RT);
	printf("Try typing something--end with ^C.\n\n");
	while((c = Crawcin()) != '\03') {
		Crawio(c);
	}
	Kbrate(old_delay,old_rate);
	printf( "\nRates restored.\n" );
}
```

See Also
TOS, xbios

keyboard—Definition

The Atari keyboard is table-driven. The keyboard tables are vectors of byte values that are indexed by the scan code passed from the intelligent keyboard (IKBD). The table is zero-based, so the first entry is always **NULL**. The following display shows the layout of the keyboard, with the scan code each key generates being given in hexadecimal:

```
 ______________________________
/3B/3C/3D/3E/3F/40/41/42/43/44/
-------------------------------
+---------------------------------------------++--------++-----------+
|01|02|03|04|05|06|07|08|09|0A|0B|0C|0D|29|0E|| 62  |61 ||63|64|65|66|
|--+--+--+--+--+--+--+--+--+--+--+--+--+--+--||--------||--+--+--+--|
|0F|10|11|12|13|14|15|16|17|18|19|1A|1B|1C|53||52|48|47||67|68|69|4A|
|--+--+--+--+--+--+--+--+--+--+--+--+--+  +--||--+--+--||--+--+--+--|
|1D|1E|1F|20|21|22|23|24|25|26|27|28|     |2B||4B|50|4D||6A|6B|6C|4E|
|--+--+--+--+--+--+--+--+--+--+--+--+------++--------+|--+--+--+--|
|2A|60|2C|2D|2E|2F|30|31|32|33|34|35|36|               |6D|6E|6F|72|
|--+-----------------------------+--+--+               |-----+--|  |
|3B|               39            |3A|                  | 70  |71|  |
+--------------------------------+                     +-----------+
```

Note that the keyboard sold in the United States does not have the key with scan code 60. This key is sometimes called the "ISO Key", and is only on European models.

See Also
ASCII, Keytbl, TOS

Keytbl—**xbios** function 16 (**osbind.h**)
Set the keyboard's translation table
#include <osbind.h>
#include <xbios.h>
char *Keytbl(*unshifted, shifted, caplock***) char ****unshifted, shifted, caplock*;

Keytbl sets the keyboard's translation tables. On the Atari ST, each key generates three *scan codes*: one in normal mode, one in shifted mode, and one in caps-lock mode. Each scan code is then translated into an ASCII character by being looked up in the appropriate table. The variables *shifted*, *unshifted*, and *capslock* each point to a translation table for the indicated mode; each table must be 128 bytes long. **Keytbl** returns a pointer to the following structure:

```
struct keytbl {
	char *unshifted;
	char *shifted;
	char *capslock;
}
```

Example
This example prints out the default keyboard map in the form of a C source file. This example also demonstrates a good method of obtaining data from the Atari's memory.

```
#include <osbind.h>
#include <xbios.h>
```

```
showmap(map, p)
register char *map, *p;
{
	register int i, j;
	printf("char %s[128] = {		/* @%06lx */\n", map, p);

	for (i = 0; i < 8; i += 1) {
	putchar('\t');

	for (j = 0; j < 16; j += 1)
		if (*p < ' ' || *p >= 0177 || *p == '\'' || *p == '\\')
			printf("%3d,", *p++ & 0xFF);
		else
			printf("'%c',", *p++ & 0xFF);

	putchar('\n');
	}
	printf("};\n");
}

main() {
	struct keytbl *kp;
	kp = Keytbl(-1L, -1L, -1L);
	showmap("normal", kp->kt_normal);
	showmap("shifted", kp->kt_shifted);
	showmap("capslock", kp->kt_capslock);
	return 0;
}
```

See Also
Bioskeys, TOS, xbios

keyword—Definition

A **keyword** is a word that is reserved within C, and may not be used to name variables, functions, or macros. The following gives keywords recognized by Mark Williams C:

alien	**entry**	**return**
auto	**extern**	**short**
break	**float**	**sizeof**
case	**for**	**static**
char	**goto**	**struct**
continue	**if**	**switch**
default	**int**	**typedef**
do	**long**	**union**
double	**readonly**	**unsigned**
else	**register**	**while**

The keyword **entry** is not implemented. The proposed ANSI standard for C adds **const**, **signed**, and **volatile** to the above set, and deletes **entry** and **readonly**. Mark Williams C reserves the keywords **readonly** and **alien**, but these are not implemented on the 68000.

See Also
C language

Kgettime—Time function (**libc.a/Kgettime**)
Read time from intelligent keyboard's clock
#include <time.h>
tm_t *Kgettime();

Kgettime is a function that reads the time from the intelligent keyboard's clock. Note that this clock is maintained apart from the other clocks on the Atari ST. **Kgettime** returns a pointer to the structure **tm_t**, which it initializes. **tm_t** is defined in the header file **time.h**; for more information about it, see the entry for **time**.

See Also
Ksettime, time, time.h

Notes
Unlike the function **Gettime**, which deals in two-second increments, **Kgettime** allows the programmer to work with clock ticks.

kick—Command
Force TOS to reread the disk cache
kick *drive*

kick forces TOS to read a disk cache. *drive* is the name of the disk drive whose cache is to be read. **kick** should be used when disks are switched in a drive, to ensure that TOS has the correct form of the disk's root directory in memory.

See Also
commands, TOS

Ksettime—Time function (**libc.a/Ksettime**)
Set time in intelligent keyboard's clock
#include <time.h>
int Ksettime(*time***) tm_t ***time***;**

Ksettime is a function that sets the time on the intelligent keyboard's clock. Note that this clock is maintained apart from the other clocks on the Atari ST. *time* points to a copy of the structure **tm_t**, which is filled by the functions **gmtime** or **localtime**. This structure is defined in the header file **time.h**; for more information about it, see the entry for **time**.

See Also
Kgettime, time, time.h

Notes
Unlike the function **Settime**, which deals with two-second increments, **Ksettime** works directly with clock ticks.

lcalloc—General function (**libc.a/lcalloc**)

Allocate dynamic memory
char *lcalloc(*count*, *size***)**
unsigned long *count*, *size*;

lcalloc is one of a set of routines that helps you to manage the computer's free memory, or *arena*. **lcalloc** calls **lmalloc** to obtain a block large enough to contain *count* items of *size* bytes each; it then initializes the block to zeroes and returns a pointer to it. Dynamic memory that is no longer needed can be returned to the free memory pool with the function **free**.

Unlike the related function **calloc**, **lcalloc** takes arguments that are **unsigned longs**; therefore, it can allocate memory blocks that are larger than 64 kilobytes.

See Also
arena, calloc, free, lmalloc, lrealloc, malloc, notmem, realloc

Diagnostics
lcalloc returns NULL if insufficient memory is available.

ld—Command

Link relocatable object files
ld *[option ...] file ...*

A compiler translates a file of source code into a *relocatable object*. This relocatable object cannot be executed by itself, for calls to routines stored in libraries have not yet been resolved. **ld** combines, or *links*, relocatable object files with libraries produced by the archiver **ar** to construct an executable file. For this reason, **ld** is sometimes called a *linker*, a *link editor*, or a *loader*.

ld scans its arguments in order and interprets each option as described below. Each non-option argument is either a relocatable object file, produced by **cc**, **as**, or **ld**, or a library archive produced by **ar**. It rejects all other arguments and prints a diagnostic message.

Each relocatable file argument is bound into the output file if its machine type matches the machine type of the first file so bound; if it does not, a diagnostic message is generated. The symbol table of the file is merged into the output symbol table and the list of defined and undefined symbols updated appropriately. If the file redefines a symbol defined in an earlier bound module, the redefinition is reported and the link continues. The last such redefinition determines the value that the symbol will have in the output file, which may be acceptable but is probably an error.

Each library archive argument is searched only to resolve undefined references, i.e., if there are no undefined symbols, the linker goes to the next argument immediately. The library is searched from first module to last and any module that resolves one or more undefined symbols is bound into the output exactly as an explicitly named relocatable file is bound. The library is searched repeatedly until an entire scan adds nothing to the executable file.

The order of modules in a library is important in two respects: it will affect the time required to search the library, and, if more than one module resolves an undefined symbol, it can alter the set of library modules bound into the output.

A library will link faster if the undefined symbols in any given library module are resolved by a library module that comes later in the library. Thus, the low-level library modules, those with no undefined symbols, should come at the end of the library, whereas the higher-level modules, those with many undefined symbols, should come at the beginning. The library module **ranlib.sym**, which is maintained by the **ar s** modifier, provides **ld** with a compressed index to the symbols defined in the library. But even with the index, the library will link much faster if the modules occur in top-down rather than bottom-up order.

A library can be constructed to provide a type of "conditional" linking if alternate resolutions of undefined symbols are archived in a carefully thought-out order. For instance, **libc.a** contains the modules

> **finit.o**
> **exit.o**
> **_finish.o**

in precisely the order given, though some other modules may intervene. **finit.o** contains most of the internals of the STDIO library, **exit.o** contains the exit() function, and **_finish.o** contains an empty version of **_finish()**, the function that **exit()** calls to close STDIO streams before process termination. If a program uses any STDIO routines, macros, or data, then **finit.o** will be bound into the output with its version of **finish()**. If a program uses no STDIO, then the "dummy" **_finish.o** will be bound into the output because it is the first module that defines **_finish()** that the linker encounters after **exit.o** adds the undefined reference. This saves approximately 3,000 bytes. To set the order of routines within a library, use the archiver **ar**; this, of course, has its own entry in the Lexicon.

The available options are as follows:

-d Define common regions even if relocation information is retained. By default, **ld** leaves common areas undefined if there are undefined symbols or if the -r option is specified.

-k*filename*
Link with the object file *filename*. This option is used to link programs to access code or data at fixed locations outside the program being linked, such as a library burned into a ROM or the fixed low memory locations documented by Atari.

-l *name*
An abbreviation for the libraries named in the environmental variable **LIBPATH**. **ld** searches each directory named in **LIBPATH** for a file named **lib***name***.a**.

-o *file*
Write output to *file* (default, **l.prg**.)

-R *value*
Relocation base option. By default, **ld** links executeable files to run at the *user-base* for the computer. In almost all cases, the *user-base* is zero. If the **-R** option is used, **ld** will link the executeable to run at *value* instead of at zero. *value* can be set to any C-style constant, or to a symbol name that **ld** can find in the object files and archives being linked; remember that a C-accessible symbol *must* end with an underscore character '_'. This option is used primarily to produce output files that can be burned into ROM. These programs must make their own provisions for relocating initialized data and other tasks.

-r Retain relocation information in the output, and issue no diagnostic message for undefined symbols. By default **ld** discards relocation information from the output if there are no undefined symbols.

-s Strip the symbol table from the output. The same effect may be obtained by using **strip**. The **-s** and **-r** options are mutually exclusive.

-u *symbol*
Add *symbol* to the symbol table as a global reference, usually to force the linking of a particular library module.

-X Discard local compiler-generated symbols of the form '**L...**'.

-x Discard all local symbols.

See Also
ar, as, cc, commands, n.out

Notes
If you are linking a program by hand (that is, running **ld** independently from the **cc** command), be sure to include the appropriate run-time start-up routine with the **ld** command line; otherwise, the program will not link correctly.

ldexp—General function (**libc.a/ldexp**)
Separate mantissa and exponent
double ldexp(*m*, *e***) double** *m*; **int** *e*;

ldexp combines the mantissa *m* with the binary exponent *e* to return a floating point value *real* that satisfies the equation *real=m**2^*e*.

See Also
atof, ceil, fabs, floor, frexp, modf

Lexicon—Introduction

The Mark Williams Lexicon is a new approach to documentation of computer software. The Lexicon has been designed to improve documentation and eliminate some limitations found in more conventional documentation.

How to use the Lexicon

The Lexicon consists of one large document that contains all entries for every aspect of Mark Williams C. You will not have to search through a number of different manuals to find the entry you are looking for.

Every entry in the Lexicon has the same structure. The first line gives the name of the topic being discussed, followed by its type (e.g., **Mathematics function**) and, where appropriate, the file where it is kept.

The next lines briefly describes the item, then give the item's usage, where applicable. These are followed a brief discussion of the item, and an example.

Cross-references follow; these can be to other entries or to other texts, notably to *The Art of Computer Programming* or *The C Programming Language*. Diagnostics and notes, where applicable, conclude each entry.

Types of entries

There are several types of entries, as follows:

Command
Commands or utilities that run directly under the micro-shell **msh**, or from the GEM desktop.

Library functions
Functions or macros that are included with Mark Williams C; these include the following: **ctype macros** (a macro that checks the type of data being handled); **debugging macros**; **general functions** (non-specialized C functions and macros); **mathematics functions**; **STDIO functions**; **STDIO macros**; **string functions** (routines used to manipulate character strings); and **time functions** (routines used to manipulate the time setting rendered by TOS).

Definition
These entries define technical terms and provide backround information that is useful in C programming.

Overview
Give an overview of a group of routines.

Symbols and constants
Data elements that are used while compiling or running programs; these include **environmental parameters**, **linker-defined symbols**, and **manifest**

constants.

TOS support
Entries that give information useful in programming for the Atari ST; these include the following: **TOS devices** (logical devices used by TOS to describe its peripheral devices); **TOS functions**; and **TOS support** (routines designed to support the TOS operating system).

UNIX routines
A function, macro, or data item included to provide compatibility with UNIX, COHERENT, and related operating systems.

The **Overview** entries review an entire topic, and give full cross-references to all of the entries that belong to the category discussed. If you are unfamiliar with a particular variety of routine, be sure to check the **Overview** entry that discusses it. The following **Overview** entries are included in the Lexicon:

C language
commands
ctype
declarations
TOS functions
UNIX system calls

Use the Lexicon
If, while reading an entry, you encounter a technical term that you do not understand, be sure to look it up in the Lexicon. You should find an entry for it. For example, if a function is said to return a data type **float** and you do not know exactly what a **float** is, look it up. You will find it described in full. In this way, you should increase your understanding of Mark Williams C, and so make your programming easier and more productive.

We wish to hear your comments on the Lexicon; we especially wish to hear if you discover something wrong or if an entry that you looked for is missing.

libaes.a—Definition

libaes.a is the library that holds the GEM AES binding routines. AES stands for *application environment services*; the routines contained in **libaes.a** allow you to invoke the elements of the GEM graphics interface, such as icons, windows, and pull-down menus. See the entry for **AES** for a brief description of the routines in this library.

To alter **libaes.a** or print out its table of contents, use the archiver **ar**.

This library can be called on the **cc** command line in one of two ways. First, the **-VGEM** will automatically link it in, plus the library **libvdi.a** and the run-time startup module **crtsg.o**. Second, it can be included by itself with the library option **-laes**; note that this option must come at the *end* of the **cc** command line, or the library will not be linked in.

Example
For an example of a program that uses **libaes.a**, see the entry for **AES**.

See Also
AES, ar, crtsg.o, gemdefs.h, library, nm, TOS, vdibind.h

libc.a—Definition
libc.a is the archive file that holds the more commonly used C functions, system calls, and compiler run-time support routines. See the entries for **string**, **STDIO**, and **UNIX routines** for information about many of the routines within **libc.a**. For a complete listing of the modules within **libc.a**, pass the following command to **msh**:

```
ar t libc.a >foo
```

This writes a list of the library's contents into the file **foo**.

See Also
ar, library, nm

libm.a—Definition
libm.a is the archive file that holds the mathematics library.

See Also
ar, library, mathematics library, math.h, nm

LIBPATH—Environmental parameter
LIBPATH names the directories that **cc** searches for the compiler's executable programs and libraries. **make** also searches these directories for the files **mmacros** and **mactions**, and **ld** looks them for its libraries. For example, the command

```
setenv LIBPATH=a:\lib,,b:\lib
```

tells **cc** to look for the compiler's executable files first in directory **lib** on drive A:, then in the current directory (as indicated by the two commas with nothing between them), and finally in **lib** on drive B:.

It is set with the **setenv** command.

See Also
msh, setenv

library—Definition
A **library** is an archive file of commonly used functions that have been compiled, tested, and stored for inclusion in a program at link time. Normally, C

uses two libraries: **libc.a**, which holds most standard C functions, such as I/O function; and **libm.a**, which holds mathematical functions. Users, however, may create their own libraries of functions or purchase such libraries from elsewhere. Mark Williams C includes an archiver that allows you to create custom libraries.

The files in a library can be listed with **ar**; the sizes of the files can be listed with **size**; the symbol tables of the object files may be listed with **nm**.

See Also
ar, function, libaes.a, libc.a, libm.a, libvdi.a

libvdi.a—Definition

libvdi.a is the library that holds the GEM VDI routines. VDI stands for *virtual device interface*. These routines perform low-level GEM graphics tasks, and are kept in the library **libvdi.a**. For a brief summary of the routines in this library, see the entry for **VDI**.

libvdi.a's table of contents can be printed out with the command **nm**, and its contents can be altered with the archiver **ar**.

This library can be called on the **cc** command line in one of two ways. First, the **-VGEM** will automatically link it in, plus the library **libaes.a** and the run-time startup module **crtsg.o**. Second, it can be included by itself with the library option **-lvdi**; note that this option must come at the *end* of the **cc** command line, or the library will not be linked in.

See Also
AES, ar, crtsg.o, gemdefs.h, library, nm, TOS, vdibind.h

line feed—Definition

Mark Williams C recognizes the literal character '\n' for the ASCII line feed character LF (octal 013). This character may be used as a character constant or in a string constant, like the other character constants: '\a', which rings the audible bell on the terminal; '\b', to backspace; '\f', to pass a formfeed command to the printer; '\r', for a carriage return; '\t', for a horizontal tab character; and '\v', for a vertical tab character.

See Also
ASCII

Notes
On many systems, \n both feeds the line and tosses the carriage; however, on the Atari ST \n must be used with \r if the program does not work through STDIO.

Line A—Definition

Line A is the interface to the Atari ST's assembly-language-level graphics routines.

If the machine instructions of the 68000 are sorted by their bit patterns, they may be categorized into 16 "lines", according to the value of the high nybble of the instruction word. Lines 1, 2, and 3, for instance, give the **move** instructions. Lines A and F are not used by the 68000 instruction set, so the processor traps when it encounters instructions with these initial bit patterns. Line F is used by the Atari ROM to make GEM AES and VDI fit into the ROM. Line A is used to call the low-level graphics routines.

Each Line A function consists of few lines of assembly language, which save registers, load parameters, execute one of the unimplemented Line A instructions, restore registers, and return. These perform simple graphics functions, such as drawing lines, displaying characters, or drawing polygons. They underpin the GEM VDI routines.

Most functions pass their parameters through the structure **la_data**. **la_data** is referenced through a pointer in in the structure **la_init**, which is initialized by function **linea0**. The exceptions are **linea7**, which takes the structure **la_blit**; **lineac**, which takes a pointer; and **linead**, which takes two pointers. All functions and structures are declared in the header file **linea.h**, which also contains a number of macros used to access elements within the Line A structures.

The following briefly summarizes the Line A functions:

linea0	Initialize
linea1	Put pixel
linea2	Get pixel
linea3	Draw a line
linea4	Draw a horizontal line
linea5	Draw a filled rectangle
linea6	Draw a filled polygon
linea7	Bit blit
linea8	Text blit
linea9	Show the mouse's pointer
lineaa	Hide the mouse's pointer
lineab	Transform the mouse's pointer
lineac	Erase a sprite
linead	Draw a sprite
lineae	Copy a raster form
lineaf	Seedfill

Examples
The first example demonstrates **linea3**, **linea5**, and **linea8**. When compiled, it takes four arguments, in decimal: an ASCII character; a column number (0 through 79); a row number (0 through 23); and a mode number (0 through 63). The mode indicates how the character named in the first argument is displayed.

```
#include <stdio.h>
#include <linea.h>
struct la_font *fontp;   /* font pointer for linea interface */
char line[100], *p;
char scr_wrk[1024];       /* area for graphics */
int scr_fat, scr_chi;   /* length and disp for underline */

/*
 * Put a character on the screen.
 */
put_scr(c, x, y, mode)
int c;                        /* character to put out */
int x, y;                     /* x & y coordinates on 80*25 screen */
int mode;                     /* see vst_effects for list of codes */

{
	unsigned int tmp;
	static long patmsk = -1;

	tmp = c - fontp->font_low_ade;
	DELX = fontp->font_char_off[tmp+1] -
		(SRCX = fontp->font_char_off[tmp]);
	DSTX = x << 3;
	DSTY = y << 4;
	WMODE = 0;          /* replace mode */
	STYLE = (mode & 7);

	if(mode & 8) {                    /* reverse */
		X2 = (X1 = DSTX) + scr_fat;
		Y2 = (Y1 = DSTY) + scr_chi;
		PATPTR = &patmsk;
		PATMSK = 1;
		CLIP = 0;
		linea5();                 /* filled rectangle */
		WMODE = 2;                /* xor mode */
	}
```

```
	if(mode & 16) {			/* underline */
		X2 = (X1 = DSTX) + scr_fat;
		Y2 = Y1 = DSTY + scr_chi;
		linea8();
		LNMASK = -1;
		WMODE = 2;
		linea3();
	}
	else
		linea8();
}

/* initialize material for screen */
init_scr() {
	linea0();			/* initialize linea */
	lineaa();			/* hide mouse */
	fontp = la_init.li_a1[2];	/* 8x16 system font */
	FBASE = fontp->font_data;
	FWIDTH = fontp->font_width;
	TEXTFG = 1;			/* text forground white */
	SRCY = 0;
	DELY = fontp->font_height;
	scr_fat = fontp->font_fatest;
	scr_chi = fontp->font_height - 1;

	COLBIT0 = 1;
	COLBIT1 = 0;
	COLBIT2 = 0;
	COLBIT3 = 0;
	LITEMSK = 0x5555;
	SKEWMSK = 0x1111;
	SCRTCHP = scr_wrk;
	WEIGHT = 1;
	LSTLIN = -1;
}

init_msg() {
	printf("\033EProgram to demonstrate some linea capabilities\n");
	printf("Each line should have four decimal numbers or 'quit'\n");
	printf("The ASCII value of the char 'A'==65, etc.\n");
	printf("The x and y coordinates relative to a 25X80 screen\n");
	printf("The mode  1=thicken  2=grey 4=italic\n");
	printf("           8=reverse 16=underline\n");
	printf("Combinations work but some are weird\n\n");
}

main() {
	int c, x, y, m;
	init_scr();
	init_msg();
```

```
        for(;;) {
                printf("\033A\033K> ");
                fflush(stdout);
                gets(line);
                if(!strcmp(line, "quit"))
                        return(0);
                sscanf(line, "%d %d %d %d", &c, &x, &y, &m);
                put_scr(c, x, y, m);
        }
}
```

The second example uses **linea5** to draw a filled rectangle. Typing any key ends the display.

```
#include <linea.h>
#include <osbind.h>
box(i, j)
{
        long patmsk = -1;                       /* pattern all ones */

        WMODE = 2;                  /* xor mode */
        PATPTR = &patmsk;
        PATMSK = 1;                 /* sizeof pattern */
        CLIP = 0;                   /* no clipping */
        X1 = Y1 = i;
        X2 = Y2 = j;
        linea5();                   /* draw box */
}

main(){
        int i;
        linea0();
        lineaa();
        Cconws("\033E\033f Any key stops the display");

        for(;Cconis() == 0;)
                for(i = 50; i < 200; i++)
                        box(i, 400-i);

        Cconin();                   /* eat char */
        Cconws("\033e\n");
}
```

See Also
linea.h, TOS, VDI

Notes
Line A is described in chapter 3.4 of *Atari ST Internals*, and in unpublished Atari documentation. These functions are extremely complex, and are not thoroughly documented. Programmers who wish to use these routines are well advised to use the above example as a model for testing the Line A functions and studying how they manipulate the screen.

linea.h—Header file

linea.h is the header file that declares the the Atari's Line A routines. It also defines all specialized structures used by them.

See Also
header file, Line A, TOS

lmalloc—General Function (**libc.a/lmalloc**)

Allocate dynamic memory
char *lmalloc(*size*) unsigned long *size*;

lmalloc helps to manage an a program's arena. It uses a circular, first-fit algorithm to select an unused block of at least *size* bytes, marks the portion it uses, and returns a pointer to it. The function **free** can be used to return allocated memory to the free memory pool.

Unlike the related function **malloc**, **lmalloc** takes an unsigned long as its *size* argument, which allows allocation of memory blocks larger than 64 kilobytes.

Example
For an example of a related function, see **malloc**.

See Also
arena, calloc, free, lcalloc, lrealloc, malloc, notmem, realloc, setbuf

Diagnostics
lmalloc returns **NULL** if insufficient memory is available. It prints a message and calls **abort** if it discovers that the arena has been corrupted, which most often occurs by storing past the bounds of an allocated block. **lmalloc** will behave unpredictably if handed an unreliable *ptr*.

localtime—Time function (**libc.a/ctime**)

Convert TOS time to ASCII string
#include <time.h>
tm_t *localtime(*timep*) time_t *timep*;
tm_t *localtime(*timep*) long *timep*;

localtime converts the system's internal time into the form described in the structure **tm_t**.

timep points to the system time. It is declared to be of type **time_t**, which is defined in the header file **time.h** as being equivalent to a **long**. The system time, in turn, is returned by the function **time**. Mark Williams C defines the system time seconds since midnight January 1, 1970 0h00m00s GMT.

localtime returns a pointer to the structure, **tm_t**, which is also defined in **time.h**. **tm_t** breaks the system time down into integer years since 1900, the month, day of the month, the hour, the minute, the second, the day of the week, and yearday. The function **asctime** turns **tm_t** into an ASCII string that can be read by humans.

Unlike its cousin **gmtime**, **localtime** returns the local time, including conversion to daylight saving time, if applicable. The daylight saving time flag indicates whether daylight saving time is now in effect, *not* whether it is in effect during some part of the year. Note, too, that the time zone is set by **localtime** every time the value returned by

```
getenv("TIMEZONE")
```

changes.

Example
For an example of how to use this function, see the entry for **asctime**.

See Also
gmtime, time, TIMEZONE

Notes
localtime returns a pointer to a statically allocated data area that is overwritten by successive calls.

log—Mathematics function (**libm.a/log**)
Compute natural logarithm
#include <math.h>
double log(*z*) double *z*;

log returns the natural (base e) logarithm of its argument *z*.

Example
For an example of this function, see the entry for **exp**.

See Also
log10, mathematics library

Diagnostics
A domain error in **log** (*z* is less than or equal to 0) sets **errno** to **EDOM** and returns 0.

log10—Mathematics function (**libm.a/log10**)
Compute common logarithm
#include <math.h>
double log10(*z*) double *z*;

log10 returns the common (base 10) logarithm of its argument *z*.

Example
For an example of this function, see the entry for **exp**.

See Also
log, mathematics library

Diagnostics
A domain error in **log10** (*z* is less than or equal to 0) sets **errno** to **EDOM** and returns 0.

Logbase—**xbios** function 3 (**osbind.h**)
Read the logical screen's display base
#include <osbind.h>
#include <xbios.h>
long Logbase()

Logbase reads the screen's logical display base, and returns a pointer to it.

The logical base is where the screen-drawing primitives do their work. This is in contrast to the physical base, which is returned by **Physbase**; the latter is where the display hardware gets the image that is displayed on the monitor. This differentiation allows you to draw one pattern while displaying another.

Example
This example gets the logical and physical screen base addresses. If they are the same, it fills the top of the screen with the pattern 10101010; otherwise, it prints out each address. In the case of this program, they will generally be equal.

```
#include <osbind.h>

main() {
	long *lbase;
	long *pbase;
	int x;

	lbase = (long *) Logbase();		/* Get logical screen */
	pbase = (long *) Physbase();		/* Get physical screen */

	if(pbase == lbase) {
		for(x=0;x<0x1000;x++)
			*pbase++ = 0xAAAAAAAAL;
	} else {
		printf("The logical screen is at %lx\n", lbase);
		printf("The physical screen is at %lx\n", pbase);
	}
	exit();
}
```

See Also
Physbase, TOS, xbios

long—Definition

A **long** is a numeric data type. By definition, a **long** is the largest integer data type; it cannot be smaller than an **int**, although on some machines an **int** and a **long** will be the same size. On most machines, **sizeof long** will equal two machine words, or four **chars** (31 data bits plus a sign bit).

See Also
declarations, int

longjmp—General function (**libc.a/setjmp**)

Return from a non-local goto
#include <setjmp.h>
longjmp(*env*, *rval***) jmp_buf** *env*; **int** *rval*

The function call is the only mechanism that C provides to transfer control between functions. This mechanism is inadequate for some purposes, such as handling unexpected errors or interrupts at lower levels of a program. To answer this need, **longjmp** helps to provide a non-local *goto* facility.

longjmp restores an environment that had been saved by a previous **setjmp** call, and returns value *rval* to the caller of **setjmp**, just as if the **setjmp** call had just returned. **longjmp** must not restore the environment of a routine that has already returned. The type declaration for **jmp_buf** is in the header file **setjmp.h**. The environment saved includes the program counter, stack pointer, and stack frame. These routines do not restore register variables in the environment returned.

See Also
setjmp, setjmp.h

Notes
Programmers should note that many user-level routines cannot be interrupted and reentered safely. For that reason, improper use of **longjmp** and **setjmp** will result in the creation of mysterious and irreproducible bugs. Do not attempt to use **longjmp** within an exception handler.

lrealloc—General function (**libc.a/lrealloc**)

Reallocate dynamic memory
char *lrealloc(*ptr*, *size***)**
char ****ptr*; **unsigned long *size*;

lrealloc helps to manage a program's arena. It returns a block of *size* bytes that holds the contents of the old block, up to the smaller of the old and new sizes.

lrealloc tries to return the same block, truncated or extended; if *size* is smaller than the size of the old block, **lrealloc** will return the same *ptr*.

Unlike the related function **realloc**, **lrealloc** takes an unsigned long as its *size* argument, and therefore can reallocate a memory blocks that is larger than 64 kilobytes.

See Also
arena, calloc, free, lcalloc, lmalloc, malloc, notmem, realloc, setbuf

Diagnostics
lrealloc returns **NULL** if insufficient memory is available. It prints a message and calls **abort** if it discovers that the arena has been corrupted, which most often occurs by storing past the bounds of an allocated block. **lrealloc** will behave capriciously if handed a fallacious *ptr*.

ls—Command

List directory contents
ls [-adlrt] [*file* ... **]**

ls prints information about each *file*. Normally, **ls** sorts by file name and prints only the name of each *file*. If a directory name is given as an argument, **ls** sorts and lists its contents, not including '.' and '..'. If no *file* is named, **ls** lists the contents of the current directory.

The following options control how **ls** sorts and displays its output.

- **-a** Print all directory entries, including '.', '..', any hidden files, and volume ID's.
- **-d** Treat directories as if they were files.
- **-l** Print information in long format. The fields give mode bits, size in bytes, date of last update, and file name.
- **-r** Reverse the sense of the sort.
- **-t** Sort by time, newest first.

The mode field in the long list format consists of four characters. The first character will be one of the following:

-	regular file
d	directory
s	system file
v	volume identifier

The next two characters are **r** or **-** if the file is read-only, and **w** if the file can be written to. The fourth character is **h** if the file is hidden.

See Also
commands, msh

lseek—UNIX system call (**libc.a/lseek**)
Set read/write position
long lseek(*fd*, *where*, *how***)**
int *fd*, *how*; **long** *where*;

lseek changes the location where the next read or write operation occurs within the file identified by file descriptor *fd*. Each read or write procedure executes at the current seek position, and advances the seek position by the number of bytes successfully transferred. The *where* and *how* arguments specify the desired seek position. *where* indicates the new seek position in the file; it is measured from the beginning of the file if *how* is zero, from the current seek position if *how* is one, or from the end of the file if *how* is two. A successful call to **lseek** returns the new seek position.

See Also
STDIO, UNIX routines

Diagnostics
lseek returns **(long)-1** on an error, such as seeking to a negative position.

Notes
For any diagnostic error, **lseek** returns **-1**; otherwise, it returns 0. Note that if **lseek** goes beyond the end of the file, it will not return an error message until the corresponding read or write is performed.

ltom—Command
Redraw the screen from low to medium resolution
ltom

ltom redraws the screen, moving from low to medium resolution.

See Also
commands, htom, mtoh, mtol, TOS

lvalue—Definition
An **lvalue** is an expression that designates a region of storage. The name comes from the assignment expression **e1=e2;**, in which the left operand must be an lvalue.

An identifier has both an *lvalue* (its address) and an *rvalue* (its contents). Some C operators require lvalue operands; the left operand of an assignment must be an lvalue. Some operators give lvalue results; if *e* is a pointer expression, **e* is an lvalue that designates the object to which *e* points. The following example shows the use of both an lvalue and a rvalue:

```
int i, *ip;

ip = &i;          /* ip is an lvalue, i and &i are rvalues */
i = 3;            /* i is an lvalue, 3 is an rvalue */
*ip = 4;          /* *ip is an lvalue, 4 is an rvalue */
```

See Also
rvalue

macro—Definition

A **macro** is a collection of instructions that is given a name and can be referenced in a program. For example, **getchar()** is a macro that consists of the function call **getc(stdin)**. Note that because macros may employ an argument *n* times, any arguments that have side effects will have the side effect repeated *n* times as well, which may be undesirable.

See Also
function

main—Definition

A C program consists of a set of functions, one of which must be called **main**. This function is called from the runtime startoff after the runtime environment has been initialized.

Programs can terminate in one of two ways. The easiest is simply to have the **main** routine **return**. Control is passed back to the run-time start-up code, which performs cleanup operations and then returns control to the operating system, passing the returned value from **main** as exit status. In some situations (errors, for example), it may be necessary to stop a program, and you may not want (or even be able) to return to the **main** routine. Here, the **exit** routine can be used; it cleans up the debris left by the broken program and returns control to the operating system.

A second exit routine, called **_exit**, quickly returns control to the operating system without performing any cleanup. This routine should be used with care, because bypassing the cleanup will leave files open and buffers of data in memory.

Programs compiled by Mark Williams C return to the program that called them; if they return from **main** with a value or call **exit** with a value, that value is returned to their caller. Programs that invoke other programs through the **system**, **execve**, or **Pexec** functions check the returned value to see if these secondary programs terminated successfully.

See Also
argc, argv, envp, exit, _exit, runtime startup

make—Command

Program building discipline
make [*option* ...] [*argument* ...] [*target* ...]

make assists in building programs from more than one compilable module.

Complex programs are often constructed of several *object modules*, which are the product of compiling *source programs*. Source programs may refer to **include** files, which are subject to independent change. Recompiling and relinking complicated programs correctly can be difficult and tedious.

make regenerates a program, based upon a specification of the structure of the program in **makefile** and the modification times of the files involved; **make** will recompile a source file only if it is younger than the object module.

makefile has three types of lines: *macro definitions*, *dependency definitions*, and *commands*. Macro definitions contain the equal sign '='; dependency definitions have a target name at the beginning of a line followed by a colon; and command lines begin with a space or tab. Comments within lines begin with an unquoted pound sign '#', and end at the end of the line. Long non-comment lines may be broken with a quoted newline character. If no *target* is given on the command line, **make** assumes the target to be the first target in **makefile**.

Dependencies
makefile specifies which files depend upon other files, and how to recreate the dependent files. Each *target* file is followed by a colon, followed by a space-separated list of files upon which it depends. The commands to recreate the dependent file are on the following lines, each beginning with a tab or space. If the target file **test.o** depends upon the source file **test.c**, the dependency is illustrated by

```
test:  test.o
       cc -O test.o -o test
```

If **test.c** is modified or recreated, **make** will issue the **cc** command to regenerate the dependent file **test.o**.

make knows about common dependencies, e.g., that **.o** files depend upon **.c** files with the same base name. The target **.SUFFIXES** contains the suffixes **make** knows about. **make** also has a set of rules to regenerate dependent files. For example, for a source file with suffix **.c** and dependent file suffix **.o**, the target **.c.o** gives the regeneration rule:

```
.c.o:
       cc -O -c $<
```

Here **$<** stands for the name of the file that causes the action. The default suffixes and rules are kept in the files **mmacros** and **mactions**, which should be kept in one of the directories named in the **LIBPATH** environmental variable. The dependencies can be changed by editing these files.

Macros
To simplify the writing of complex dependencies, **make** provides a *macro* facility. To define a macro, write

NAME = string

The *string* is terminated by the end-of-line character, so it can contain blanks. To refer to the value of the macro, use a dollar sign '$' followed by the macro name enclosed in parentheses:

$(*NAME*)

If the macro name is one character, parentheses are not necessary. **make** uses macros in the definition of default rules:

```
.c.o:
	$(CC) $(CFLAGS) -c $<
```

where the macros are defined as

```
CC=cc
CFLAGS=-O
```

Other built-in macros used in interpretation of rules are:

$*	target name less suffix
$@	target name
$<	list of referred files
$?	referred files newer than target

Each command line *argument* should be a macro definition of the form

OBJECT=a.o b.o

Arguments that include spaces must be surrounded by quotation marks, because blanks are significant to the micro-shell **msh**.

Options
The following lists the options that can be passed to **make** on its command line.

-d (Debug) Give verbose printout of all decisions and information going into decisions.

-f *file*
file contains the **make** specification. If this option does not appear, **make** uses the file **makefile** or **Makefile** in the current directory.

-i Ignore error returns from commands and continue processing. Normally, **make** exits if commands return error status.

-n Test only; suppresses actual execution of commands.

-p Print all macro definitions and target descriptions.

-q Return a zero exit status if the targets are up to date. No commands are executed.

-r Do not use built-in rules describing dependencies.

-s Do not print command lines when executing them. Commands preceded by '@' are not printed, except under the **-n** option.

-t (Touch option) Force the dates of targets to be the current time, and bypass actual regeneration.

Invoking **make**
make can be used either from the micro-shell **msh**, or from the TOS desktop.

To use **make** from the TOS desktop, its suffix must be changed to **TOS** or **TTP**. Once this is done, you can invoke **make** simply pointing to the appropriate icon with your mouse and clicking it. When the **Open Application** box appears, enter the options and target you want. **make** reads whatever **makefile** is in the current directory, and executes its instructions. It cannot accept options from the desktop, however.

If you wish to use **make** from **msh**, simply invoke **msh** from TOS, then enter the **make** command as you normally would, including options and a path name for the **makefile**, should it be in a directory other than one that you have previously defined in the **PATH** environmental parameter.

See Also
commands, msh
See the tutorial *Building Programs with Make*, which is included at the end of this manual.

Diagnostics
make reports its exit status if interrupted or if an executed command returns error status. It replies "Target *name* not defined" or "Don't know how to make target *name*" if it cannot find appropriate rules.

Notes
Note that the order of items in **mmacros/.SUFFIXES** matters. The consequent of a default rule (e.g., **.obj**) must *precede* the antecedent (e.g., **.c**) in the entry **.SUFFIXES**. Otherwise, **make** will not work properly.

malloc—General function (**libc.a/malloc**)
Allocate dynamic memory
char *malloc(*size*) unsigned *size*;

malloc helps to manage an a program's arena. It uses a circular, first-fit algorithm to select an unused block of at least *size* bytes, marks the portion it uses, and returns a pointer to it. The function **free** can be used to return allocated memory to the free memory pool.

Example
This example reads from the standard input up to *NITEMS* items, each of which is up to *MAXLEN* long, sorts them, and writes the sorted list onto the standard output. It demonstrates the functions **qsort**, **malloc**, **free**, **exit**, and **strcmp**. You may want to use as input what the example for **Random** has output. For an example of how to use **malloc** in a TOS application, see the entry for **Fgetdta**.

```
#include <stdio.h>
#define NITEMS 512
#define MAXLEN 256
char *data[NITEMS];
char string[MAXLEN];

main() {
	register char **cpp;
	register int count;
	extern int compare();
	extern char *malloc();
	extern char *gets();

	for (cpp = &data[0]; cpp < &data[NITEMS]; cpp++) {
		if (gets(string) == NULL)
			break;
		if ((*cpp = malloc(strlen(string) + 1)) == NULL)
			exit(1);
		strcpy(*cpp, string);
	}
	count = cpp - &data[0];
	qsort(data, count, sizeof(char *), compare);
	for (cpp = &data[0]; cpp < &data[count]; cpp++) {
		printf("%s\n", *cpp);
		free(*cpp);
	}
	exit(0);
}

compare(p1, p2)
register char **p1, **p2;
{
	extern int strcmp();
	return(strcmp(*p1, *p2));
}
```

See Also
arena, calloc, free, lcalloc, lmalloc, lrealloc, notmem, realloc, setbuf

Diagnostics
malloc returns NULL if insufficient memory is available. It prints a message and calls **abort** if it discovers that the arena has been corrupted, which most often occurs by storing past the bounds of an allocated block.

The related function **lmalloc** takes an unsigned long as its *size* argument, and therefore can allocate memory blocks that are larger than 64 kilobytes.

Malloc—**gemdos** function 72 (**osbind.h**)
Allocate dynamic memory
#include <osbind.h>

long Malloc(*n*) long *n*;

Malloc allocates dynamic memory. *n* contains either the number of bytes to be allocated, or the number -1L (0xFFFFFFFF), which returns all available memory. If *n* contains the number of bytes to be allocated, **Malloc** returns a pointer to the starting address of the memory allocated; if *n* contains -1L, then **Malloc** returns the number of bytes available; in either case, **Malloc** returns 0 upon failure.

Example
This program displays the amount of free memory available.

```
#include <osbind.h>

main()
{
	printf("[%ld bytes of memory free]\n", Malloc(-1L));
}
```

See Also
gemdos, Mfree, Mshrink, TOS

Notes
As of this writing, **Malloc** appears to have some peculiarities. Always **Malloc** even-size blocks of memory. Always **Mfree** memory in the *reverse* order of allocation. Finally, try to **Malloc** a few pieces of memory; there appears to be an undocumented limit on the number of times **Malloc** can be called by a given program. Though large, this number is finite; when it is exceeded, **Malloc** will return **NULL** even though considerable amounts of memory are still available.

manifest constant—Definition

A **manifest constant** is a numeric constant that is referenced by a symbolic name, to allow it to be defined differently under different computing environments. An example is **EOF**, the end-of-file marker, which has wildly different representations under different operating systems.

The use of manifest constants in programs help to ensure that code is portable, by isolating the definition of these elements in a single header file, where they belong.

See Also
EOF, header file, NULL, portability

mantissa—Definition

In mathematics, a **mantissa** is the fractional part of a logarithm. In the context of C, "mantissa" refers to the fractional portion of a floating point number.

See Also
data formats, double, float, frexp

math.h—Header file
Header file for mathematics functions
#include <math.h>

math.h is the header file to be included with programs that use any of Mark Williams C's mathematics routines. It includes the following: definitions for mathematical functions; error return values, as used by the **errno** function; definitions of mathematical constants, e.g., **PI**; the definition of structure **cpx**, which describes complex variables; definitions of internal compiler functions; and, finally, declarations of mathematical functions.

See Also
libm.a, mathematics library

mathematics library—Overview
The following mathematics routines are available with Mark Williams C:

acos	inverse cosine
asin	inverse sine
atan	inverse tangent
atan2	inverse tangent of quotient
cabs	complex absolute value
cos	cosine
cosh	hyperbolic cosine
exp	exponent
fabs	absolute value function
floor	floor function
hypot	hypotenuse
j0	Bessel function, order 0
j1	Bessel function, order 1
jn	Bessel function, order *n*
log	natural logarithm
log10	common logarithm
pow	power
sin	sine
sinh	hyperbolic sine
sqrt	square root
tan	tangent
tanh	hyperbolic tangent

See Also
libm.a, Lexicon, math.h

Notes
When programs that contain mathematics routines are compiled, the mathematics libraries must be called specifically on the **cc** command line. For example, to compile the example presented under the entry for **acos**, use the following **cc** command line:

```
cc -f -o acos.prg acos.c -lm
```

The **-f** option links in the floating point routines for **printf**, while the **-lm** option links in the mathematics libraries. Note that the **-lm** option must come *last* on the **cc** command line, or the library will not be searched properly.

me—Command
Invoke MicroEMACS screen editor
me [-e] [*file* ...]

me is the command that invokes MicroEMACS, Mark Williams C's screen editor. With it, the user can insert text, delete text, move text, search for a string and replace it, and perform many other editing tasks. MicroEMACS reads text from files and writes edited text to files; it can edit several files simultaneously.

If the command **me** is used without arguments, MicroEMACS opens an empty buffer. If used with one or more file name arguments, MicroEMACS will to open each of the files named, and display its contents in a window. If a file cannot be found, MicroEMACS will assume that you are creating it for the first time, and create an appropriately named buffer and file descriptor for it.

The last line of the screen is used to print messages and inquiries. The rest of the screen is portioned into one or more *windows* in which text is displayed. The last line of each window shows whether the text has been changed, the name of the buffer, and the name of the file associated with the window.

MicroEMACS notes its *current position* and displays a cursor under the character to the right of that position. It remembers a position called the *mark*. Some commands manipulate the block of text between the current position and the mark.

The printable ASCII characters, from ' ' to '~', can be inserted at the current position. Control characters and escape sequences are recognized as *commands*, described below. A command character can be inserted into the text by prefixing it with **<ctrl-Q>**.

Commands remove text in two different ways. *Delete* commands remove text and throw it away, whereas *kill* commands remove text but save it in the *kill* buffer. Successive kill commands append text to the previous kill buffer.

Search commands prompt for a search string terminated by **<RETURN>** and then search for it. Case sensitivity for searching can be toggled with the command **<esc>@**. Typing **<RETURN>** instead of a search string tells MicroEMACS to use the previous search string.

Some commands manipulate words rather than characters; a word consists of upper-case and lower-case letters, '_', and '$'. Usually, a character command is a control character and the corresponding word command is an escape sequence. For example, **<ctrl-F>** moves forward one character and **<esc>F** moves forward one word. Note that the MicroEMACS commands are not case sensitive; for example, **<ctrl-F>** and **<ctrl-f>** are identical.

MicroEMACS can be invoked automatically by the compiler command **cc** to display for correction any errors that occurred during compilation. The **-A** option to **cc** will cause MicroEMACS to be invoked with error messages in one window, source code in the other, and with the cursor fixed at the line on which the first error occurred. When the text is altered, exiting from MicroEMACS will automatically cause the file to recompile. This cycle will continue either until the file compiles without error, or until you break the cycle by typing **<ctrl-U> <ctrl-X> <ctrl-C>**.

The option **-e** to the **me** command allows you to invoke the error buffer by hand; compilation can then be performed by passing a command to the shell using the **<ctrl-X>!** command.

The following list gives the MicroEMACS commands. They are grouped by function, e.g., *Moving the cursor*. An *argument* giving a repeat count can precede a command; the default argument is 1. **<ctrl-U>** introduces an argument. If it is followed by an optional minus sign '-' and decimal digits, the number gives the argument. If not, each **<ctrl-U>** multiplies the value of the argument by four.

Moving the cursor

<ctrl-A>	Move to start of line.
<ctrl-B>	(Back) Move backward by characters.
<esc>B	Move backward by words.
<ctrl-E>	(End) Move to end of line.
<ctrl-F>	(Forward) Move forward by characters.
<esc>F	(Forward) Move forward by words.
<esc>G	Go to an absolute line number in a file.
<ctrl-N>	(Next) Move to next line.

<ctrl-P> (Previous) Move to previous line.

<ctrl-V> Move forward by pages.

<esc>V Move backward by pages.

<ctrl-X>= Print the current position.

<ctrl-X>G Go to an absolute line number in a file. Can be used with an argument; otherwise, will prompt for a line number.

<esc>! Move to the line within the window given by *argument*; the position is in lines from the top if positive, in lines from the bottom if negative, and the center of the window if 0.

<esc>< Move to the beginning of the current buffer.

<esc>> Move to the end of the current buffer.

Killing and deleting

<ctrl-D> (Delete) Delete next character.

<esc>D Kill the next word.

<ctrl-H> If no argument, delete previous character. Otherwise, kill *argument* previous characters.

<ctrl-K> (Kill) With no argument, kill from current position to end of line; if at the end, kill the newline. With argument 0, kill from beginning of line to current position. Otherwise, kill *argument* lines forward (if positive) or backward (if negative).

<ctrl-W> Kill text from current position to mark.

<esc>W Kill text from mark to current position. Same as **<ctrl-W>**.

<ctrl-X><ctrl-O>
Kill blank lines at current position.

<ctrl-Y> (Yank) Copy the kill buffer into text at the current position; set current position to the end of the new text.

<esc><ctrl-H>
Kill the previous word.

<esc><DEL>
Kill the previous word.

<DEL> If no argument, delete the previous character. Otherwise, kill *argument* previous characters.

Windows

<ctrl-X>1 Display only the current window.

<ctrl-X>2 Split the current window; usually followed by **<ctrl-X>B** or **<ctrl-X><ctrl-V>**.

<ctrl-X>N (Next) Move to next window.

<ctrl-X>P (Previous) Move to previous window.

<ctrl-X>Z Enlarge the current window by *argument* lines.

<ctrl-X><ctrl-N>
Move current window down by *argument* lines.

<ctrl-X><ctrl-P>
Move current window up by *argument* lines.

<ctrl-X><ctrl-Z>
Shrink current window by *argument* lines.

Buffers

<ctrl-X>B (Buffer) Prompt for a buffer name, and display the buffer in the current window.

<ctrl-X>K (Kill) Prompt for a buffer name and delete it.

<ctrl-X><ctrl-B>
Display a window showing the change flag, size, buffer name, and file name of each buffer.

<ctrl-X><ctrl-F>
(File name) Prompt for a file name for current buffer.

<ctrl-X><ctrl-R>
(Read) Prompt for a file name, delete current buffer, and read the file.

<ctrl-X><ctrl-V>
(Visit) Prompt for a file name and display the file in the current window.

Saving text and exiting

<ctrl-X><ctrl-C>
Exit without saving text.

<ctrl-X><ctrl-S>
(Save) Save current buffer to the associated file.

<ctrl-X><ctrl-W>
(Write) Prompt for a file name and write the current buffer to it.

<ctrl-Z> Save current buffer to associated file and exit.

Compilation error handling

<ctrl-X>> Move to next error.

<ctrl-X>< Return to previous error.

Search and replace

<ctrl-R> (Reverse) Incremental search backward; a pattern is searched for as each character is typed in.

<esc>R (Reverse) Search towards the beginning of the file.

<ctrl-S> (Search) Incremental search forward; a pattern is searched for as each character is typed in.

<esc>S (Search) Search toward the end of the file.

<esc>% Search and replace. Prompt for two strings; then search for the first string and replace it with the second.

<esc>/ Search for next occurrence of a string entered with the <esc>S or **<esc>R** commands; this remembers whether the previous search had been forwards or backwards.

<esc>@ Toggle case sensitivity in search commands.

Keyboard macros

<ctrl-X>(Begin a macro definition. MicroEMACS collects everything typed until the end of the definition for subsequent repeated execution. **<ctrl-G>** breaks the definition.

<ctrl-X>) End a macro definition.

<ctrl-X>E (Execute) Execute macro.

Change case of text

<esc>C (Capitalize) Capitalize the next word.

<ctrl-X><ctrl-L>
(Lower) Convert from current position to mark into lower case.

<esc>L (Lower) Convert the next word to lower case.

<ctrl-X><ctrl-U>
(Upper) Convert from current position to mark into upper case.

<esc>U (Upper) Convert the next word to upper case.

White space

<ctrl-I> Insert a tab.

<ctrl-J> Insert a new line and indent to current level.

<ctrl-M> (Return) If the following line is not empty, insert a new line; if empty, move to next line.

<ctrl-O> (Open) Open a blank line; that is, insert newline after the current position.

<TAB> With argument, set tab indentation to *argument* characters. An argument of zero restores the default of eight characters.

Send commands to operating system

<ctrl-C> Suspend MicroEMACS and invoke a new copy of **msh**. Typing exit returns you to MicroEMACS and allows you to resume editing.

<ctrl-X>! Prompt for an **msh** command and execute it.

Setting the mark

<ctrl-@> Set mark at current position.

<esc>. Set mark at current position.

Miscellaneous

<ctrl-G> Abort a command.

<ctrl-L> Refresh the screen.

<ctrl-Q> (Quote) Insert the next character into text; used to insert control characters.

<esc>Q (Quote) Insert the next control character into the text; same as **<ctrl-Q>**.

<ctrl-T> (Transpose) Transpose the characters before and after the current position.

<ctrl-U> Specify a numeric argument, as described above.

<ctrl-X>F Set word wrap to *argument* column.

<ctrl-X><ctrl-X>
Exchange current position and mark.

Diagnostics
MicroEMACS prints error messages on the bottom line of the screen. It prints informational messages (enclosed in square brackets '[' and ']' to distinguish them from error messages) in the same place.

MicroEMACS manipulates text in memory rather than in a file. No changes to a file occur until the user writes edited text. MicroEMACS prints a warning and prompts the user whenever a command would cause it to lose changed text.

Because MicroEMACS keeps text in memory, it does not work for extremely large files. It prints an error message if a file is too large; when this happens, you should exit from the editor immediately, *without saving the file*. Otherwise, your file on disk will be truncated.

See Also
commands
See the accompanying tutorial *MicroEMACS Screen Editor Tutorial*.

Notes
This version of MicroEMACS does not include many facilities available in the original EMACS display editor, which was written by Richard Stallman at M.I.T. In particular, it does not include user-defined commands. It also does not include pattern search commands.

Note that the current version MicroEMACS, including source code, is proprietary to Mark Williams Company. The code may be altered or otherwise changed for your personal use, but may *not* be used for commercial purposes, and may not be distributed without prior written consent by Mark Williams Company.

MicroEMACS is based the public domain editor by David G. Conroy.

me.a—Archive

me.a is an archive that holds the source files for the Mark Williams proprietary version of the MicroEMACS screen editor. If you wish to recompile MicroEMACS, you must first extract the source files from the archive. Use the command **cd** to move to the directory where you have stored this archive, then give **msh** the following command:

```
ar xv me.a
```

See Also
ar, me

Mediach—**bios** function 9 (**osbind.h**)

Check whether disk has been changed
#include <osbind.h>
#include <bios.h>
long Mediach(*drive*) int *drive*;

Mediach checks whether a disk has been changed. *drive* is a number from zero to 15, and indicates which drive to check: zero indicates drive A, one indicates drive B, etc. **Mediach** returns zero if the medium has not been changed, one if it may have been changed, and two if it was changed.

Example
This example discovers whether the floppy disks have been changed.

```
#include <osbind.h>
main() {
	int d, ds;
	char *status[3] = { "not", "possibly", "definitely" };

	for (d = 0; d < 2; d += 1) {
		ds = Mediach(d);
		printf("drive %c has ", d+'a');

		if (ds < 0 || ds > 2)
			printf("bad status: %d\n", ds);
		else
			printf("%s changed\n", status[ds]);
	}
}
```

See Also
bios, TOS

memory allocation—Definition

The following diagram shows how Mark Williams C allocates memory.

```
                          ----------------------
                          |      VIDEO RAM     | highest address
                          ----------------------
                          |        ARENA       |
                          |         AND        |
                          |        FREE        |
                          |       MEMORY       |
                          ----------------------
                          |        STACK       |
                          ----------------------
Not in                    |                    | uninitialized
image                     | UNINITIALIZED DATA | instructions
file                      |                    | & data
                          ----------------------
                          |                    | private data,
In                        |  INITIALIZED DATA  | shared data,
                          |                    | strings
                          ----------------------
                          |                    |
image                     |      TEXT CODE     | instructions
                          |                    |
                          ----------------------
file                      |   RUNTIME STARTUP  |
                          ----------------------
                          |                    |
                          |      BASE PAGE     | low address
                          |                    |
                          ----------------------
```

The stack *descends* from the highest address in its space, toward the static data area; new arguments are placed on the stack in its *lowest* address. Everything from the top of the stack space to the end of the data segment is free to accept dynamically allocated data.

The size of the stack cannot be altered while a program is running. The amount of stack is set by the global variable **_stksize**. By default, the run-time start-up sets **_stksize** to two kilobytes. Note, however, that a highly recursive function may cause the stack to grow larger than two kilobytes, so that it overwrites other data areas. This will cause your program to work incorrectly.

Should your program need more than two kilobytes of stack, include in it the following global statement:

```
long _stksize = nL;
```

where *n* is a constant that specifies the number of bytes to allocate.

Example
The following example displays the "memory map" of a GEM-DOS process. It demonstrates **argc**, **argv**, **envp**, **environ**, **end**, **etext**, **edata**, and **_stksize**, as well as how to use the header file **basepage.h**.

```
#include <basepage.h>
dodisplay(value, name)
long value; char *name;
{
	printf("0x%08lx %s\n", value, name);
}

#define display(x) dodisplay((long)(x), "x")

main(argc, argv, envp)
int argc; char *argv[], *envp[];
{
	extern long _stksize;
	extern char **environ;
	extern char etext[], edata[], end[];

	display(BP->p_env);
	display(envp[0]);
	display(environ[0]);
	display(argv[0]);

	if (argv[1] != 0)
		display(argv[1]);
	if (argc > 2)
		display(argv[argc-1]);

	display(BP);
	display(BP->p_lowtpa);
	display(BP->p_cmdlin+1);
	display(_start);
	display(BP->p_tbase);

	display(etext);
	display(BP->p_tbase+BP->p_tlen);
	display(edata);
	display(BP->p_dbase+BP->p_dlen);
	display(end);

	display(BP->p_bbase+BP->p_blen);
	display(envp);
	display(environ);
	display(argv);
	display(argv+argc);
```

```
        display(_stksize);
        display(&argc);
        display(&argv);
        display(&envp);
        display(BP->p_hitpa);
}
```

See Also
C language, calling conventions, data format

menu—Definition

A **menu** is a graphics form that is used extensively in programs that run under TOS. It is a specialized form of AES object that uses the structure **OBJECT** described in the header file **obdefs.h**. For more information on this structure, see the entry for **object**.

Each menu's object tree must be built in a special way. The **root** object is a **G_IBOX** that is sized to dimensions of the screen. Note that in high resolution, the screen is 640 rasters wide by 400 high; in medium resolution, it is 640 rasters wide by 200 high; and in low resolution, it is 320 rasters wide by 200 high. The **root** has two children: the **bar** object and the **screen** object.

The root object's first child is the **bar** object. It describes the menu bar, at the top of the screen, and is of object type **G_BOX**. Its length is that of the screen, and its width is that of a normal character plus two rasters for gutter. In high and medium resolutions, a character is 16 rasters high; in low resolution, it is eight rasters high; thus, in high resolution the **bar** object is 18 rasters high; in medium and low resolutions, it is ten rasters high.

The **bar** object has one child: an **active** object, whose type is **G_IBOX**. The **active** box is sized to hold all of the titles that appear in the bar along the top of the screen.

The **active** box, in turn, has one or more children: the **title** strings, which are the titles of the menus. These strings are of the type **G_TITLE**; note that this type is used only with menus. By design, the first (leftmost) title must be called "Desk"; it triggers the drop-down menu that names the available GEM desk accessories.

The **screen** object is the object's other child. It is of type **G_IBOX**, and it is sized to cover the portion of the screen that is used by the drop-down menus. Thus, it should be as wide as the screen and as high as the longest drop-down menu. The **screen** object has one or more children; each child is a **box** that displays a drop-down menu. There should be one box for each drop-down menu; i.e., the number of **boxes** and **titles** must be the same. Each **box** is of type **G_BOX**; each must be wide enough and high enough to hold all of the text that will be written into it; for example, if the longest string to go into it is ten characters wide, then the box must be at least 64 rasters wide (in high resolution) or the string will splash over its edge. Each **box** should be aligned on the left with its corresponding **title**; there is no need, however, to keep the various **boxes** from overlapping. Note that the Atari ST has a buffer in which to store the portion of the screen that is overwritten by a drop-down menu, so that it can be restored when the menu is erased. This buffer can hold up to one quarter of the screen, or 64,000 pixels; no **box** should exceed this limit, or debris will be left on the screen when the menu is erased.

Each **box** can have one or more children, called **names**. Each **name** is of type **G_STRING**, and names the particular option that you are offering the user. Note that all the **names** must be as wide as the box; otherwise, the **box** will "leak", and cause more than one selection to be illuminated when the mouse pointer is moved into that **box**. The Y coordinate for each **name** must be increased by one line's height; for example, if a **box** has three **names**, the Y coordinate of the first should be zero, that of the second should be 16 (in high resolution, eight in medium or low resolution), and that of the third should be 32. This will keep the **names** from overlapping, which could possibly have disastrous results. As always, the X and Y coordinates of an object are relative to those of its parent.

The first (leftmost) **box** is special in that the AES can manipulate its **name** objects. By design, the first **box** must have eight **name** children. The first **name** can be defined by the user. The second **name** consists of a row of hyphens; its state is set to **DISABLED**, which causes it to be written in gray, rather than solid, letters. The next six **names** should point to empty strings. These will be filled in by the AES with the names of the available desk accessories. The AES will alter the size of the leftmost **box** if fewer than six are available.

The following "geneological table" shows the object tree for a menu that has two drop-down menus, the latter with three entries. The numbers indicate each element's place in the object tree, and are used to set the parent-child-sibling pointers. These are set by the order in which the elements are loaded into the object's array:

```
                  ----------------------
                  |      1. ROOT       |
                  ----------------------
                  /                    \
           --------                   -----------
           |2. BAR| ----------------> |6. SCREEN|
           --------                   -----------
              |                        /        \
          -----------            --------       ---------
          |3. ACTIVE|            |7. BOX| ------> |16. BOX|
          -----------            --------       ---------
          /         \            /      \        /       \
   ---------   ---------   -------   --------  --------  --------
   | 4. TL | --> | 5. TL |   |8. N1|...|15. N8|  |17. N1|...|19. N3|
   ---------   ---------   -------   --------  --------  --------
```

The menu should be invoked with the **menu_bar** call; the AES will handle the rest. Note that, as shown in the above example, **menu_bar** regards as significant the *order* in which the elements of a menu are loaded into the object array. The order should be as follows:

root
bar
active
title(s)
screen
first menu **box**
first **items**
...
last menu **box**
last **items**

When the mouse is used to select a menu entry, the AES generates a message that contains that object's index number within the menu tree; use **evnt_mesag** to receive the message and initiate the proper response. The AES will automatically handle all invocation of desk elements; you do not need to write code for them.

Example
This example clears the screen and displays a menu that lists all of the GEM desk accessories. Note that the objects are sized in rasters for a high-resolution screen.

```
#include <aesbind.h>
#include <obdefs.h>
#include <gemdefs.h>
```

```
#define SPEC 0x111C1L
/*
 *      I.e.:   (1 << 16) |      [Border 1 raster thick]
 *              (BLACK << 12) |  [Fill color; BLACK = 1]
 *              (BLACK << 8) |   [Text color]
 *             {(1 << 7) |       [Turn on replace bit]
 *               (4 << 4) |      [Fill pattern to gray]
 *                               [Together make one nybble] }
 *                BLACK          [Border color]
 */

#define COLOR 0x010F0L
/*
 *      I.e.,   (0 << 16)        [Border thickness 0]
 *              (BLACK << 12)    [Fill color; BLACK = 1]
 *              (WHITE << 8)     [Text color]
 *             {(1 << 7)         [Replace bit on]
 *              (7 << 4)         [Fill pattern to solid] [one nybble]}
 *               WHITE           [Border color; WHITE = 0]
 */

/* Strings used in the menu object */
char desk[] = "  Desk";
char quit[] = "  Quit";
char line[] = " ------------------ ";
char blank[] = "";

/* Define an object that masks the screen */
OBJECT mask[] = {
/* N/  H/  T/  type / flags /  state /specification/ X/ Y/  W /  H */
  -1, -1, -1, G_BOX, LASTOB,  NORMAL,     SPEC,      0, 0, 639, 399
};

/* Define the menu object */
OBJECT menu[] = {
/* N/  H/  T/    type /  flags /  state /specif./  X /  Y /  W /  H */
  -1,  1,  4,   G_IBOX,   NONE,   NORMAL,  0x0L,   0,   0, 639, 399, /* ROOT */
   4,  2,  2,    G_BOX,   NONE,   NORMAL, COLOR,   0,   0, 639,  18, /* BAR */
   1,  3,  3,   G_IBOX,   NONE,   NORMAL,  0x0L,   0,   0,  64,  18, /* ACTIVE */
   2, -1, -1,  G_TITLE,   NONE,   NORMAL,  desk,   0,   0,  64,  18, /* TITLE */
   0,  5,  5,   G_IBOX,   NONE,   NORMAL,  0x0L,   0,  18, 639, 140, /* SCREEN */
   4,  6, 13,    G_BOX,   NONE,   NORMAL, COLOR,   0,   0, 168, 140, /* BOX */
   7, -1, -1, G_STRING,   NONE,   NORMAL,  quit,   0,   0, 168,  16, /* N1 */
   8, -1, -1, G_STRING,   NONE, DISABLED,  line,   0,  16, 168,  16, /* N2 */
   9, -1, -1, G_STRING,   NONE,   NORMAL, blank,   0,  32, 168,  16, /* N3 */
  10, -1, -1, G_STRING,   NONE,   NORMAL, blank,   0,  48, 168,  16, /* N4 */
  11, -1, -1, G_STRING,   NONE,   NORMAL, blank,   0,  64, 168,  16, /* N5 */
  12, -1, -1, G_STRING,   NONE,   NORMAL, blank,   0,  80, 168,  16, /* N6 */
  13, -1, -1, G_STRING,   NONE,   NORMAL, blank,   0,  96, 168,  16, /* N7 */
   5, -1, -1, G_STRING, LASTOB,   NORMAL, blank,   0, 112, 168,  16  /* N8 */
};
```

```
main() {
        int buffer[8];
        int nowhere = 0;                /* Unused pointers point here */

        appl_init();                    /* Begin application */
        graf_mouse(ARROW, &nowhere);    /* Mouse ptr. to arrow */
        objc_draw(mask, ROOT, MAX_DEPTH, 0, 18, 639, 380);
                                        /* Mask the screen */
        menu_bar(menu, 1);              /* Show menu bar */

        for(;;) {
                evnt_mesag(buffer);
                if (buffer[0] == MN_SELECTED) {
                        switch(buffer[4]) {

                        case 6:                 /* i.e., object 6 clicked */
                                menu_bar(menu, 0);
                                appl_exit();
                                exit(0);
                        default:
                                break;
                        }
                }
        }
}
```

See Also
AES, object, TOS, window

menu_bar—AES function (**libaes.a/menu_bar**)
Show or erase the menu bar
#include <aesbind.h>
#include <obdefs.h>
int menu_bar(*tree, eraseshow***) OBJECT ****tree***; int** *eraseshow*;

menu_bar is an AES routine that shows or erases the menu bar; the menu bar is the bar that appears at the top of the screen and names the menus that are available to the user. *tree* is the name of the object tree being used. *eraseshow* indicates whether you want to show or erase the menu bar: zero indicates erase, and one indicates show. **menu_bar** returns zero if an error occurred, and a number greater than zero if one did not.

Example
For an example of how to use this routine, see the entry for **menu**.

See Also
AES, menu, object, TOS

menu_icheck—AES function (**libaes.a/menu_icheck**)
Write or erase a check mark next to a menu item
#include <aesbind.h>
#include <obdefs.h>
int menu_icheck(*tree, item, eraseshow***) OBJECT ***tree*; **int** *item, eraseshow*;

menu_icheck is an AES routine that draws or erases a check mark next to a selected menu entry. *tree* points to the object tree that holds the menu, and *object* is the object within the tree that is being handled. *eraseshow* indicates whether you want to show the check mark or erase it: zero indicates erase, and one indicates show. **menu_icheck** returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, menu, object, TOS

menu_ienable—AES function (**libaes.a/menu_ienable**)
Enable or disable a menu item
#include <aesbind.h>
#include <obdefs.h>
int menu_ienable(*tree, object, disable***)**
OBJECT ***tree*; **int *object, disable*;

menu_ienable is an AES routine that enables or disables a menu item. A disabled item is displayed in faint letters and cannot be clicked by the user. *tree* points to the object tree that contains the menu, and *object* is the number of the object within the tree. *disable* indicates whether the item should be enabled or disabled: zero indicates disable, and one indicates enable. **menu_ienable** returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, menu, object, TOS

menu_register—AES function (**libaes.a/menu_register**)
Add a name to the desk accessory menu list
#include <aesbind.h>
#include <obdefs.h>
int menu_register(*accessory, teststring***) int** *accessory*; **char ***teststring*;

menu_register is an AES routine that adds a name to the desk accessory menu list. *accessory* is the ID of the desk accessory. *teststring* points to the desk accessory's desk menu test string. The test string is a template used to check whether text typed by the user, if any, is of the correct type (e.g., lower-case letters only). For more information about the template, see the entry for **menu**.

menu_register returns the desk accessory's identifier, from zero through five.

See Also
AES, desk accessory, menu, object, TOS

Notes
Because only six desk accessories can be used at any one time, only six items can be displayed on the desk accessory menu.

menu_text—AES function (**libaes.a/menu_text**)
Replace text of a menu item
#include <aesbind.h>
#include <obdefs.h>
int menu_text(*tree, object, text***) OBJECT** **tree*; **char** **text*; **int** *object*;

menu_text is an AES routine that changes the text for a menu item. *tree* points to the object tree for the menu, and *object* is the number of the object within the tree that holds that particular menu entry. *text* points to the text string to be plugged into the menu. **menu_text** returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, menu, object, TOS

menu_tnormal—AES function (**libaes.a/menu_tnormal**)
Display menu title in normal or reverse video
#include <aesbind.h>
#include <obdefs.h>
int menu_tnormal(*tree, object, video***) OBJECT** **tree*; **int** *object, video*;

menu_tnormal is an AES routine that displays the menu title in normal or reverse video. *tree* points to the object tree that encodes the menu, and *object* is the number of menu title within the tree. *video* indicates whether you want the title to be in normal or reverse video: zero indicates reverse video, and one indicates normal. **menu_tnormal** returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, menu, object, TOS

metafile—Definition
A **metafile** is a file of VDI instructions that can be stored on disk and incorporated into other programs. This allows you to create "boiler-plate" images that are transferred easily.

Note that a metafile consists of a set of VDI instructions, rather than device-dependent bits. This allows you to edit such a file easily to alter its aspects. More importantly, because the elements of an image are described logically rather than absolutely, it allows the elements to be manipulated easily, and the image as a whole to be maneuvered. This allows you to create images independent of the the type or resolution of the device on which they are displayed.

Consider, for example, the example of the bouncing colored ball used in the Atari demonstration program. At present, that program has a set of "snapshots" of the ball in different positions; to animate the ball, the program simply cycles through the snapshots. If this program were stored in a VDI metafile, however, a programmer could describe how each plane on the surface of the ball is logically connected to its neighbors; by setting parameters, then, the entire ball in all of its aspects could be resized easily or moved about the screen. This, in turn, would allow the programmer to create a user interface, in which the user could "zoom in" toward the ball, zoom out, move the ball around the screen, change its rate or direction of rotation, etc.

Metafile structure
For a full description of the VDI metafile structure, see Appendix C to volume 1 of the *GEM Programmer's Guide*. The following briefly summarizes the metafile format.

Each metafile begins with a 16-integer header, structured as follows:

1 Always set to 0xFFFF.

2 VDI version number: 100 times the major version number, plus the minor version number.

3 Type of coordinates: zero indicates normalized device coordinates (NDC); two indicates raster coordinates (RC). One is reserved by TOS.

4-7 Respectively, minimum width and height, and maximum width and height required to display image in the file. These are set with the function **v_extent_meta**; otherwise, they are set to zero.

8-16 Reserved; always set to zeroes.

The header is follow by a series of VDI entries; each consists of an array of **ints**, in the following order:

0 The VDI function's opcode. See the list below for the appropriate opcode for each legal VDI routine.

1 The number of vertices (i.e., endpoints or corners) in the figure being drawn.

2 The number of integer parameters passed to the VDI routine.

3 The VDI routine's sub-opcode; see the table below for each routine's appropriate sub-opcode.

4-*n* The settings for each vertex. The number of vertices described corresponds to the value in 1.

n+4-*m* The values for each integer parameter. The number of parameters described corresponds to the value in **2**.

Finally, each metafile closes with an integer set to 0xFFFF.

Customized routines can be inserted into a metafile with the function **v_meta_write**.

Metafile routines
The following VDI library routines can be incorporated into metafiles. The first column gives the routine's opcode, the second gives its sub-opcode, the third gives its name, and the fourth a brief description of its action.

3	0	**v_clrwk**	clear a virtual device
4	0	**v_updwk**	update workstation (flush buffers)
5	2	**v_exit_cur**	exit from alphabetic mode
5	3	**v_enter_cur**	enter alphabetic mode
5	20	**v_form_adv**	advance page on hard-copy device
5	21	**v_output_window**	print portion of a virtual device
5	22	**v_clear_disp_list**	clear a printer's display list
5	23	**v_bit_image**	print a bit-image file
6	0	**v_pline**	draw a polyline
7	0	**v_pmarker**	draw a polymarker
8	0	**v_gtext**	output graphics text
9	0	**v_fillarea**	flood enclosed area with fill pattern
11	1	**v_bar**	draw an outlined, filled rectangle
11	2	**v_arc**	draw a circular arc
11	3	**v_pieslice**	draw a circular pie segment
11	4	**v_circle**	draw a circle
11	5	**v_ellipse**	draw an ellipse
11	6	**v_ellarc**	draw an elliptical arc
11	7	**v_ellpie**	draw an elliptical pie segment
11	8	**v_rbox**	draw rounded rectangle
11	9	**v_rfbox**	draw rounded rectangular fill area
12	0	**vst_height**	set graphics text height, in pixels
13	0	**vst_rotation**	set angle of graphics text
14	0	**vs_color**	set mix for a color
15	0	**vsl_type**	set polyline's pattern
16	0	**vsl_width**	set polyline width
17	0	**vsl_color**	set polyline color
18	0	**vsm_type**	query graphics text attributes
19	0	**vsm_height**	query character cell's height
20	0	**vsm_color**	query color settings
21	0	**vst_font**	set graphics text font
22	0	**vst_color**	set graphics text color
23	0	**vsf_interior**	set fill type
24	0	**vsf_style**	set fill style
25	0	**vsf_color**	set fill color

32	0	**vswr_mode**	set writing mode
39	0	**vst_alignment**	set graphics text alignment
104	0	**vsf_perimeter**	set drawing of perimeter
106	0	**vst_effects**	set graphics text special effects
107	0	**vst_point**	set graphics text height, in points
108	0	**vsl_ends**	set polyline end types
112	0	**vsf_udpat**	set user-defined fill pattern
113	0	**vsl_udsty**	set user-defined polyline style
114	0	**vr_recfl**	draw a rectangular fill area
129	0	**vs_clip**	clip an area of the virtual device

See Also
TOS, v_extent_meta, v_write_meta, VDI, vm_filename

Notes
Metafiles need the VDI's GDOS in their operation. They should not be used if the GDOS is not present in your edition of VDI.

mf—Command

Measure space left in RAM
mf

mf is a command that measures the amount of free space left in RAM for program execution. It takes no arguments.

See Also
commands, df, msh

Mfpint—xbios function 13 (osbind.h)

Initialize the MFP interrupt
#include <osbind.h>
#include <xbios.h>
void Mfpint(*interrupt, vector***) int** *interrupt*; **char** **vector*;

Mfpint initializes the multi-function peripheral (MFP) interrupt, and returns nothing. This routine allows a programmer to trap a hardware interrupt in her program. *interrupt* is the number of the interrupt to be set, 0 through 15, as follows, going from lowest to highest priority:

MFP_BIT0	0	I/O port bit 0
MFP_BIT1	1	undefined
MFP_BIT2	2	undefined
MFP_BIT3	3	undefined
MFP_TIMD	4	timer D, RS-232 baud rate generator
MFP_TIMC	5	timer C, system 200-hz clock
MFP_BIT4	6	I/O port bit 4
MFP_BIT5	7	undefined
MFP_TIMB	8	timer B
MFP_XERR	9	RS-232 transmit error
MFP_EMPT	10	RS-232 transmit buffer empty
MFP_RERR	11	RS-232 receive error
MFP_FULL	12	RS-232 receive buffer full
MFP_TIMA	13	timer A, user programmable
MFP_BIT6	14	I/O port bit 6
MFP_BIT7	15	I/O port bit 7

vector points to the interrupt routine to be set.

See Also
Jdisint, Jenabit, TOS, xbios

Mfree—**gemdos** function 73 (**osbind.h**)
Free allocated memory
#include <osbind.h>
long Mfree(*memory***) long** *memory*;

Mfree frees memory allocated by the function **Malloc**. *memory* points to the address of the memory to free. **Mfree** returns 0 if memory could be freed, and non-zero if it could not.

Example
The following example prints the number of bytes currently free and the number allocated.

```
#include <osbind.h>

main() {
	unsigned long memleft;
	unsigned long memhere;
	char *almem;

	/*
	 * This first 'printf' is needed to make the numbers
	 * look right, because printf malloc's memory for the
	 * FILE buffer
	 */

	printf("Test of Malloc(), Mfree() and Mshrink()\n");

	printf("%8lx bytes free, %8lx bytes allocated\n",
		(memleft = Malloc(-1L)), 0L);

	memhere = memleft>>1;
	almem = (char *) Malloc(memhere);
	printf("%8lx bytes free, %8lx bytes allocated (%8lx)\n",
		Malloc(-1L), memleft-Malloc(-1L), memhere);

	Mshrink(almem,0x1000L);
	printf("%8lx bytes free, %8lx bytes allocated (%8lx)\n",
		Malloc(-1L), memleft-Malloc(-1L), 0x1000L);
	Mfree(almem);
	printf("%8lx bytes free, %8lx bytes allocated (%8lx)\n",
		Malloc(-1L), memleft-Malloc(-1L), 0L);
}
```

See Also
gemdos, Malloc, Mshrink, TOS

Notes
Do not attempt to **Mfree** blocks of memory not directly allocated by **Malloc**. Memory freed by **Mfree** is not inserted into the arena used by **malloc**, but is returned to the system.

Midiws—**xbios** function 12 (**osbind.h**)

Write a string to the MIDI port
#include <osbind.h>
#include <xbios.h>
void Midiws(*count*, *pointer*) **int** *count*; **char** **pointer*;

Midiws writes a string to the musical instrument device interface (MIDI) port, and returns nothing. *count* gives the number of characters that will be sent, minus one; and *buffer* points to where the characters are stored. Note that this routine will transmit *count* characters; NUL characters will be used like any other character.

Example
This example plays some notes on a MIDI instrument connected to the ST through the MIDI-OUT plug.

```
#include <osbind.h>

/* MIDI status byte values */

#define NOTE_OFF (0x80)                /* Key off command */
#define NOTE_ON (0x90)                 /* Key on command */

/* Some useful things to know... */

#define MIDDLE_C (60)

#define C_OFFSET (0)
#define D_OFFSET (2)
#define E_OFFSET (4)
#define F_OFFSET (5)
#define G_OFFSET (7)
#define A_OFFSET (9)
#define B_OFFSET (11)
#define FLAT (-1)
#define SHARP (1)
#define OCTAVE_STEP (12)

unsigned char notes[128];              /* Note counters... */

key_down(note_offset)
int note_offset; {                     /* Note relative... */
                                       /* ...to middle C */
        int midi_note;
        char midi_buf[4];

        if ((midi_note=MIDDLE_C+note_offset) < 0 || midi_note >127)
                return;                /* Return if out of range */
        notes[midi_note]++;            /* Mark as key-down... */
        midi_buf[0]=NOTE_ON;           /* Note on... */
        midi_buf[1]=midi_note;         /* This one... */
        midi_buf[2]=0x40;              /* this fast... */
        Midiws(2, midi_buf);           /* Send message out */
}
```

```
key_up(note_offset)
int note_offset; {                              /* Note */
        int midi_note;
        char midi_buf[4];

        if ((midi_note=MIDDLE_C+note_offset) < 0 || midi_note > 127)
                return;                        /* Return if out of range */
        if (notes[midi_note]-- < 0)
                notes[midi_note] = 0;          /* Decrement down count */
        midi_buf[0]=NOTE_OFF;                  /* Note off... */
        midi_buf[1]=midi_note;                 /* This one... */
        midi_buf[2]=0x40;                      /* this fast... */
        Midiws(2, midi_buf);                   /* send message out */
}

clean_up() {
        char midi_buf[258];                     /* buffer for commands */
        char *mp;                              /* And a pointer. */
        int  i=0;                              /* A counter. */
        int  c=0;                              /* Another counter */

        mp = midi_buf;
        *mp++ = NOTE_OFF;
        while (i < 128) {
                while(notes[i] != 0) {
                        notes[i]--;
                        *mp++ = i;
                        *mp++ = 0x40;
                        c++;
                }
                i++;
        }
        if(c > 0)
                Midiws(c<<1, midi_buf);
}

/* Delay for a little while -- Use the vertical sync for timing.*/

delay(n)
int n; {
        int i;
        while(n-- > 0) {
                for(i=35 ; i>0 ; i--)
                        Vsync();
        }
}
```

```
main() {
	int i;
	int n;

	key_down(C_OFFSET);
	delay(2);
	key_down(E_OFFSET);
	delay(2);
	key_down(G_OFFSET);
	delay(2);
	key_down(C_OFFSET+OCTAVE_STEP);
	delay(5);
	key_up(E_OFFSET);
	key_up(G_OFFSET);
	key_down(F_OFFSET);
	key_down(A_OFFSET);
	delay(20);
	clean_up();
}
```

See Also
TOS, xbios

mkdir—Command
Create a directory
mkdir *directory*

mkdir creates *directory*. Files or directories with the same name as *directory* must not already exist. **directory** will be empty except for the entries '.', the directory's link to itself, and '..', its link to its parent directory.

See Also
commands, msh, rm, rmdir

mktemp—General function (**libc.a/mktemp**)
Generate a temporary file name
**char *mktemp(*pattern*) char **pattern*;

mktemp generates a unique file name, for such purposes as naming intermediate data files.

Note that the functions **tmpnam** and **tempnam** assemble temporary file name and then call **mktemp**. These routines ease somewhat the difficulty in creating a proper name for a temporary file.

See Also
msh, tempnam, tmpnam

modf—General function (**libc.a/modf**)
Separate integral part and fraction
double modf(*real*, *ip***) double** *real*, **ip*;

modf is the floating point modulus function. It returns the fractional part of its *real* argument, which is a value *f* in the range 0 <= *f* < 1. It also stores the integral part in the **double** location referenced by *ip*. These numbers satisfy the equation *real* = *f* + **ip*.

See Also
atof, ceil, fabs, floor, frexp, ldexp

modulus—Definition
Modulus is the operation whereby the remainder is derived from a division operation; for example, 12 modulo 4 equals 0, because when 12 is divided by 4 it leaves no remainder. The term "modulo" also refers to the product of a modulo operation; in the above example, the modulo is 0. In C, the modulo operation is indicated with a percent sign '%'; therefore, 12 modulo 4 is written **12%4**.

msh—Command
msh is the Mark Williams micro-shell, which is designed for use under TOS. It combines aspects of the Bourne shell and the Berkeley C shell into one command that is powerful and easy to use.

msh is a *command processor*. It finds commands and executes them either singly or in batches; and it allows the user to direct the output of a command to a device, into a new file, or to another command for further processing. It can replace text with symbols defined by the user, or with wildcards that are expanded according to carefully defined rules.

The simplest command consists of a list of words; the words are separated from each other by spaces or tab characters, and the list is terminated by a **<newline>** sequence. Each word may contain *history substitutions*, *variable substitutions*, *file name substitutions*, *quoted characters*, *quoted strings*, or *file redirection*. **msh** also supports aliasing, for use in batch files and scripts. These are discussed below.

Several commands may be placed on the same line; the commands are then separated with semicolons or other *command separators*; these are outlined below. A list of commands may be grouped into a single command by enclosing the list within parentheses.

Both simple commands and lists of commands be made to extend over more than one line by typing a slash '/' before pressing the <return> key.

History substitutions
A *history* substitution allows you to use all or part of a previously entered command or a shell variable in your present command. For example, typing

```
echo foo
!echo >bar
```

is equivalent to typing:

```
echo foo
echo foo >bar
```

The history substitution **!foo** tells **msh** to repeat all of the previous command **echo foo**; if **msh** does not find **foo** in **history**, it looks for the shell variable foo. Typing **!*n*** repeats the *n*th command before the present one. Note that you must tell **msh** how many commands to save; for example

```
set history=8
```

saves the last eight commands issued. The default setting is one. To see what commands have been stored in the history buffer, type:

```
set in history
```

History substitutions may be used anywhere on the command line. For example, typing

```
ls \documents\scripts\editors
echo foo ; !-1
```

is equivalent to typing

```
ls \documents\scripts\editors
echo foo ; ls \documents\scripts\editors
```

Note, too, that history substitutions can use variable names.

Variable substitution
A *variable* is a symbol defined by the user with the **set** command; for example, the command

```
set X="echo foo"
```

declares that **X** is a symbol equivalent to the string **echo foo**. When a variable is used in a **msh** command line, it must be preceded by a dollar sign '$' or an exclamation point '!'. For example, to call the variable set in the above example, type **$X** or **!X**. When it sees a token that begins with either of these punctuation marks, **msh** searches for it first on the list of variables that have been assigned with the **set** command, then on the list of those assigned with the **setenv** command, and finally on the list of tokens that it received from TOS or from the parent shell. For example, if you type

```
set esc="^["
set cls="echo ${esc}E"
```

(where <esc> indicates the escape character) and then type

```
$cls
```

msh will expand this variable into

```
echo ^[E
```

and then execute the **echo** command with the argument **<esc>**E, which in turn clears the screen.

The difference between $*name* and !*name* is that the latter may include command separators because it is rescanned as input, whereas the former is not rescanned. For example, the variable set with the command

```
set X="echo foo ; echo bar"
```

should be reference with the token **!X** rather than **$X**. Command separators are described in detail below.

File name substitutions
File name substitutions contain the punctuation marks [ ] ? * { }. The following notes what each punctuation mark does:

[*list*], [*a-z*]
Match any of the characters *l*, *i*, *s*, or *t*, or any character in the range **a-z**.

? Match any one character or no character.

* Match any character, any string of characters, or no character.

{*list*} Braces enclose a list of words that are each combined with the remainder of the word.

Character quotations
A *quotation* is used when you want **msh** to disregard the special meaning of a character and read it merely as a literal character. In general, preceding a character with a slash will remove the special meaning of a character, except under the following circumstances:

1. A slash followed by an end-of-file indicator is always an error.

2. A slash followed by a **<newline>** becomes a space and continues input on the next line.

3. When set between " "'s or ' ''s, a slash followed by a **<newline>** translates into **<newline>**, and /" becomes a literal quotation mark. All other characters quoted with '/' are left untouched.

4. Within literal quotations, '/' is literal.

Quoted strings
Strings may be quoted by enclosing them in apostrophes or quotation marks. Quoting a string means that **msh** or a command is to accept it literally. Note that quoting a string with apostrophes prevents any further expansion; all wildcards and variables will be treated as literal characters. Quoting a string with quotation marks, however, tells **msh** to treat white space as part of the string, but allows further expansion of variables. The following exercise will demonstrate how these forms of quotation differ:

```
set A="123"
set B="XYZ"
echo $A              $B
echo "$A             $B"
echo '$A             $B'
```

File redirection
The term *file redirection* means redirecting the input or output of a command into a file. The following redirection operators are recognized by **msh**:

> *file* Redirect output of a file into *file*. If *file* already exists, replace its contents with the output of the command.

>& *file*
Redirect the output of a command and any diagnostic messages it produces into *file*.

>> *file*
Append the output of a command onto *file*. If *file* does not exist, create it and fill it with the output of the command.

>>& *file*
Append the output of a command and all of the diagnostic messages it generates onto *file*. If *file* does not exist, create it and fill it with the output and diagnostic messages generated by the command.

< *file* Use the contents of *file* to control the execution of a command.

Separating and joining commands
Commands can be separated or joined on the same command line by using the following marks:

; Execute commands sequentially.

&& Execute commands sequentially until one terminates with non-zero exit status (i.e., until an error occurs in one).

| Form a *pipe* between the commands: feed the standard output of the command on the left of the '|' into the standard input of the command on the right.

|& Form a pipe that passes both the output of the command on the left and any diagnostic messages it produces as input to the command on the right.

| | Commands separated by '| |' are run sequentially until one terminates with zero exit status (i.e., executed without error).

Commands
Mark Williams C includes a number of commands that are designed to be used with **msh**. For a list of these commands and a brief description of each, see the entry for **commands**. If you need help with **msh** or any of its built-in commands, type **help** and the name of the command for which you need help. **msh** will print on the screen a summary of how to use that command.

Setting the environment
msh allows you to set a number of *environmental variables*. **msh** uses some of these variables, and makes all of them available to programs that run under it. A program can read these variables by using the function **getenv**. Environmental variables can be set or changed with the command **setenv**, and erased with the command **unsetenv**. Typing **setenv** without an argument will display the list of environmental variables plus their settings.

For Mark Williams C to work properly, the following *environmental parameters* must be set:

HOME The default directory: where **msh** performs a task when no other directory is named.

INCDIR Name the directory in which **cc** searches for header files and other text files to be included in compilation.

LIBPATH Name the path along which **cc** searches for the executable files for the compiler and the linker i.e., **cc0.prg**, **cc1.prg**, **cc2.prg**, **cc3.prg**, **crts0.o**, **ld.prg**, and the libraries.

PATH This environmental variable consists of a list of directory prefixes that are separated by commas. These prefixes name the directories that are searched in order for commands or batch files to be run. For example, typing

```
PATH = ,\bin,\lib
```

will ensure that **msh** will search the the current directory, then the directories **\bin** and **\lib**, in that order, to find the executable file named in a command.

SUFF This consists of a list of file-name suffixes that are separated by commas. These suffixes are appended to the given command name when searching the directories named in **${PATH}**.

TMPDIR Name the directory into which temporary files are written.

See the Lexicon entry **environment** for more information.

Shell variables
The following variables control the operation of **msh**. Some can be set with the set command. Typing **set** without an argument will display a list of all current variables, both those set by the user and those set by **msh**:

history Set the length of the history list. For example, to set the **history** variable to eight, type the following:

```
set history=8
```

This allows you to invoke any of the last eight commands by using the form **!-***n*.

cwd The current working directory. This variable cannot be reset by the user.

prompt This variable holds the prompt string. The default is '$'.

status This variable holds the exit status returned by the last command executed. It should not be reset by the user.

Command files
msh reserves the variables **$0** through **$9** for arguments passed on a command line. This allows you to write shell scripts whose variables can be set when you run the script.

For example, the following commands could be typed into the file **foo**:

```
cc -V -f $1 $2 $3 -lm
```

Thereafter, typing **foo** followed by the names of up to three C source files will compile the files with the floating point **printf** routines, and link in the mathematics library.

The **profile** *file*
Whenever you invoke **msh** from the GEM desktop, it automatically reads a file called **profile** and executes all of the commands that it finds therein. By altering your **profile**, you can customize **msh** to suit your preferences and tasks at hand.

See Also
commands, environment, set, setenv, wildcard, unset, unsetenv

Mshrink—**gemdos** function 74 (**osbind.h**)
Shrink amount of allocated memory
#include <osbind.h>
long Mshrink(*begin, length***) int** *n*; **long** *begin, length*;

Mshrink shrinks the amount of memory allocated by a program, and returns dynamic memory to the free memory pool. *begin* points to the beginning of the space to be returned, and *length* indicates the amount of memory to be returned. **Mshrink** returns zero if memory could be de-allocated, and non-zero if it could not.

Example
For an example of how to use this function, see the entry for **Mfree**.

See Also
gemdos, Malloc, Mfree, TOS

NOTES
The **gemdos** call has a third parameter that is always zero; the **Mshrink** macro inserts this parameter automatically.

msleep—Command

Stop executing for a specified time
msleep *milliseconds*

msleep suspends processing for a set time. *milliseconds* is the amount of time to suspend processing, in milliseconds.

See Also
commands, sleep, TOS

mtoh—Command

Redraw the screen from medium to high resolution
mtoh

mtoh redraws the screen, moving from medium to high resolution.

See Also
commands, htom, ltom, mtol, TOS

mtol—Command

Redraw the screen from medium to low resolution
mtol

mtol is a command that redraws the screen, moving from medium to low resolution.

See Also
commands, htom, ltom, mtoh, TOS

mtype.h—Header file

The header file **mtype.h** assigns a code number to each of the processors supported by Mark Williams C compilers. These include the Intel 8086, 8088, 80186, and 80286; the Zilog Z8001 and Z8001; the DEC PDP-11 and VAX; the

IBM 370, and the Motorola 68000.

See Also
header file

mv—Command
rename files or directories
mv *oldfile newfile*
mv *file ... directory*

mv renames files. In the first form above, it changes the name of *oldfile* to *newfile*. If *newfile* previously existed, **mv** deletes its former contents; if not, **mv** creates it. If *newfile* is a directory, **mv** places *oldfile* under that directory.

In the second form, **mv** moves each file argument into the directory argument. If the source and destination files are on different disk drives, **mv** copies the source to the destination and removes the source.

mv will not copy directories between devices and will not remove directories that occupy the destination of the command.

See Also
commands, cp, msh

nested comments—Definition

The C Programming Language declares that comments cannot be nested. The **-VCNEST** option to Mark Williams C allows nested comments, and prints a warning whenever they occur.

The following gives an example of properly nested comments:

```
/* nested /* comment */ */
```

Note that the open-comment and close-comment symbols are balanced. The following shows improperly "nested" comments:

```
/* not
/* nested
/* comment */
```

See Also

cc

Notes

A program with inappropriately nested comments will fail if the option is used, but will compile correctly if it is *not* used. A program with correctly nested comments, however, will compile correctly if is used, but will fail if it is not used. In general, it is best to use if you use nested comments in your program, to ensure correct compilation of your program.

newline—Definition

Mark Williams C recognizes the literal character '\n' for the ASCII newline character LF (\012). This normally feeds the line and returns the carriage. This character may be used as a character constant or in a string constant, like the other character constants: '\a', which rings the audible bell on the terminal; '\b', to backspace; '\f', to pass a formfeed command to the printer; '\r', for a carriage return; '\t', for a tab character; and '\v', for a vertical tab character.

See Also

ASCII, character constants

Notes

On the Atari ST, '\n' must be used with the carriage return character '\r' if the program does not go through STDIO.

nm—Command

Print a program's symbol table

nm [-adgnopru] *file ...*

nm prints the symbol table of each *file* in its argument list. Each *file* argument may be an Mark Williams C object module or an object library built with the archiver **ar**. If an argument is a library, **nm** prints the symbol table for each member of the library.

The first argument selects one of several options. It is optional; if present, it must begin with '-'. The options are as follows:

-a Print all symbols. Normally, **nm** prints names in a C-style format and ignores symbols with names inaccessible from C programs.

-d Print only defined symbol.

-g Print only global symbols.

-n Sort numerically rather than alphabetically. **nm** uses unsigned compares when sorting symbols with this option.

-o Prepend the file name to each output line.

-p Print symbols in symbol table order.

-r Sort in reverse order.

-u Print only undefined symbols.

By default, **nm** sorts symbol names alphabetically. Each symbol is followed by its value and its segment. For relocatable object modules, the letter 'U' appears in place of a value if the symbol is undefined. If the file is an executable program, the value is the address of the symbol. The segment type designations for global symbols are shown below.

SI	shared instructions
PI	private instructions
BI	instruction space BSS
SD	shared data
PD	private data
BD	data space BSS
D	debug table
A	absolute
C	common

Static symbols have the same segment type descriptors in lower-case letters.

See Also
cc, commands, ld

notmem—General function (**libc.a/notmem**)
Check if memory is allocated
int notmem(*ptr***) char ***ptr*

notmem checks if a memory block has been allocated by **malloc**, **lmalloc**, **realloc**, **calloc**, or **lcalloc**. The pointer *ptr* indicates the block to be checked. **notmem** searches the arena for *ptr*; it returns one if *ptr* has not been allocated, and one if it has.

See Also
arena, calloc, free, malloc, realloc, setbuf

n.out—Definition

n.out is the format used by the Mark Williams C compiler, assembler and linker to generate their output.

n.out first gives global information and information about the size of each segment. Segments of the indicated size follow the header in a fixed order. **n.out** defines the header structure for the 68000 as follows:

```
struct ldheader {
	short	l_magic;
	short	l_flag;
	short	l_machine;
	short	l_tbase;
	size_t l_ssize[NLSEG];
	long	l_entry;
};
```

All elements of the **nout** header are stored in canonical byte order. **l_magic** is the "magic number" that identifies a load module; it always contains 0407. **l_flag** contains flags that indicate the type of the object module. **l_machine** is the processor identifier. **l_tbase** is the start of the text segment. **l_entry** contains the machine address where execution of the module commences. **l_ssize** gives the size of each segment.

size prints the segment sizes of the **n.out** format header, **nm** lists the symbols, and **strip** will remove the symbols.

See Also
as, ld, nm, size, strip

NUL—Definition

NUL is the character ASCII 0 and, in C, signals the end of a string. It is represented as '\0'. Note that NUL is defined as part of the string it is terminating; therefore, a string that is defined to be 50 characters long in fact holds 49 printable characters plus NUL.

See Also
ASCII, string, NULL

NULL—Definition

NULL is defined in the header file **stdio.h.** It is the null pointer **(char *)0**, which is a pointer filled with zeros. Numerous routines return this value to indicate failure; it is useful as a return value because it points nowhere, and so removes the possibility of accidentally destroying a section of memory after failure.

See Also
manifest constant, NUL, pointer, stdio.h

Notes
References through **NULL** on the Atari ST cause a *bus error*, i.e., two cherry bombs appear on the screen.

nybble—Definition

A **nybble** is four bits, or half of an eight-bit byte. The term is generally used to refer to the low four bits or the high four bits of a byte; thus, a byte may be said to have a "low nybble" and a "high nybble". One nybble encodes one hexadecimal digit.

See Also
bit, byte

obdefs.h—Header file

TOS header file

#include <obdefs.h>

obdefs.h is a header file that contains TOS common object definitions and structures. It defines numerous elements used in programs written for the Atari ST, such as definitions of color settings, editable fields, and fonts.

See Also

header file, object, TOS

objc_add—AES function (**libaes.a/objc_add**)

Redefine a child object within an object tree

#include <aesbind.h>
#include <obdefs.h>
int objc_add(*tree, parent, child***) OBJECT** **tree*; **int** *parent, child*;

objc_add is an AES routine that redefines a child object within an object tree; specifically, it redefines an object as being the offspring of a specified parent. *tree* points to the object tree being modified. *child* is the number of the object being redefined, and *parent* is the number of the object being made *child*'s parent.

objc_add returns zero if an error occurred, and a number greater than zero if one did not.

See Also

AES, object, TOS

objc_change—AES function (**libaes.a/objc_change**)

Change an object's state within a clipping rectangle

#include <aesbind.h>
#include <obdefs.h>
int objc_change(*tree, object, junk, rectangle, newstate, redraw***)**
OBJECT **tree*; **int** *object, junk, newstate, redraw*; **Rect** *rectangle*;

objc_change is an AES routine that changes the state of an object within a named clipping rectangle. *tree* points to the object tree being modified, and *object* is the number of the object within the object tree. *junk* is reserved, and must be zero.

rectangle is the clipping rectangle being used. It is of the type **Rect**, which is defined in the header file **aesbind.h**. **Rect** consists of four elements:

x	X coordinate of rectangle
y	Y coordinate of rectangle
w	width of rectangle
h	height of rectangle

state indicates the new state for the object, as follows:

0x00	normal
0x01	selected
0x02	cross-hatched
0x04	checked
0x08	disabled
0x10	outlined
0x20	shadowed

Finally, *redraw* indicates whether or not to redraw the object being modified: zero indicates not to redraw, and one indicates redraw.

objc_change returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, object, TOS

objc_delete—AES function (**libaes.a/objc_delete**)

Delete an object from an object tree
#include <aesbind.h>
#include <obdefs.h>
int objc_delete(*tree, object***) OBJECT ****tree*; **int** *object*;

objc_delete is an AES routine that deletes an object from an object tree. *tree* points to the object tree being modified, and *object* is the number of the object within the object tree. **objc_delete** returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, object, TOS

objc_draw—AES function (**libaes.a/objc_draw**)

Draw an object
#include <aesbind.h>
#include <obdefs.h>
int objc_draw(*tree, object, depth, rectangle***)**
OBJECT ****tree*; **int *object, depth*; **Rect** *rectangle*;

objc_draw is an AES routine that draws an object. *tree* points to the object tree that contains the object in question. *object* is the number of the object within the object tree. *depth* indicates how many levels deep the object should be drawn: zero, draw only the object itself; one, draw the object plus its children; two, draw the object and its children and grandchildren; through eight (which is called **MAX_DEPTH** in **obdefs.h**), which draws the object and all of its descendents. Thus, setting *object* to zero (the root object within the tree) and setting *depth* to **MAX_DEPTH** will draw the entire object.

rectangle defines the clipping rectangle to be used in drawing the object. It is of the type **Rect**, which is defined in the header file **aesbind.h**. **Rect** consists of four elements:

x	X coordinate of rectangle
y	Y coordinate of rectangle
w	width of rectangle
h	height of rectangle

objc_draw returns zero if an error occurred, and a number greater than zero if one did not.

Example
For an example of this routine, see the entry for **object**.

See Also
AES, object, TOS

objc_edit—AES function (**libaes.a/objc_edit**)
Edit a text object
#include <aesbind.h>
#include <obdefs.h>
int objc_edit(*tree, object, character, oldindex, kind, &newindex***)**
OBJECT ***tree*; **int *object, character, oldindex, kind, newindex*;

objc_edit is an AES routine that edits a text object within an object tree. The object being edited must be either of type **G_TEXT** or **G_BOXTEXT**. *tree* points to the object tree that contains the object being edited, and *object* is the number of that object within the tree. *character* is the character to be inserted into the text. *oldindex* is the index of the character being replaced. *kind* is the type of replacement you want performed, as follows:

0	Reserved
1	Move input text into template; turn on cursor
2	Compare input with validation string; update text; display string
3	Turn off cursor

newindex is the index of character that follows the one edited. This value is set by AES.

objc_edit returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, TOS

objc_find—AES function (**libaes.a/objc_find**)
Find if mouse pointer is over particular object
#include <aesbind.h>
#include <obdefs.h>
int objc_find(*tree, object, depth, mousex, mousey***)**
OBJECT **tree*; **int** *object, depth, mousex, mousey*;

objc_find is an AES routine that finds whether the mouse pointer is positioned over a particular object. *tree* points to the object tree that holds the object in question, and *object* is its number within the object tree. *depth* is the depth to which the object tree should be searched, as follows: zero, search only for *object*; one, search for *object* and its children; two, search for the object plus its children and grandchildren; through eight (which is called **MAX_DEPTH** in **obdefs.h**), which searches for the object and all of its descendents. **objc_find** returns the number of the object over which the mouse pointer was found to be positioned, or -1 if it was found not to be positioned over any requested object.

See Also
AES, object, TOS

objc_order—AES function (**libaes.a/objc_order**)
Reorder a child object within the object tree
#include <aesbind.h>
#include <obdefs.h>
int objc_order(*tree, object, newposition***)**
OBJECT **tree*; **int** *object, newposition*;

objc_order is an AES routine that moves a child object to a new position within the object tree. *tree* points to the object tree that holds the object to be moved, and *object* is its number within the object tree. *newposition* gives the new position for this object in the list of its siblings: zero indicates the bottom of the list, one indicates one from the bottom, and so on; -1 indicates the top of the list. **objc_order** returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, object, object, TOS

objc_set—AES function (**libaes.a/objc_set**)
Calculate an object's absolute screen position
#include <aesbind.h>
int objc_set(*tree, object, &xcoord, &ycoord***)**
OBJECT **tree*; **int** *object, xcoord, ycoord*;

objc_set is an AES routine that returns the absolute position on the screen of a given object. *tree* points to the object tree that holds the object in question, and *object* is its number within the tree. *xcoord* and *ycoord* give, respectively, the X and Y coordinates of the object; these are set by AES. **objc_set** returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, object, TOS

Notes
Other sets of bindings call this routine **objc_offset**.

object—Definition
An **object** is an AES data form that encodes an element to be displayed on the screen. An object can be a rectangle, a text string, a box, a bit-mapped picture, a combination of any of these, or (most importantly) a number of such elements linked together in the form of an object tree.

The object tree
An *object tree* is a group of visual elements that are linked together to form a tree. One object is the tree's *root object*; it can have one or more *child objects* and each child object can have one or more *siblings* and children.

Consider the following example, for the object tree **foo**. Like all object trees, **foo** has a root object, **foo[0]**. This object, in turn, has three children: **foo[1]**, **foo[2]**, and **foo[3]**. Each of these three children has two siblings; e.g., **foo[2]**'s siblings are **foo[1]** and **foo[3]**. Each of these children can, in turn, have its own children, each of which can have siblings and children of its own.

As you can see, the name **foo** points to an array of objects; each object's subscript depends on the order in which it is read into memory. If you wish to write an object tree by hand, it is up to you to know each object's subscript in order to write the tree correctly.

Each object within the tree contains three "pointers" in its description. These are not true C pointers (i.e., memory addresses), but integers that are used by the AES to orient each object within its tree. The first pointer, **next**, points to the object's next sibling. For example, the **next** pointer for **foo[1]** is **2**, which points to **foo[2]**. If an object is the last of its siblings or if it has no siblings, then **next** must point to the object's *parent* object. The only exception is the root object, which has no sibling and no parent; its **next** pointer is always set to -1. The second pointer and third pointers, **head** and **tail**, point respectively to the object's first child and its last child. For example, **foo[0]** has a head pointer of 1, which indicates that **foo[1]** is the first of its children, and a tail pointer of 3, which indicates that **foo[3]** is the last of its children. If an object has only one child, then the head and tail pointers must both point to it; and if an object has no children, then both pointers must be set to -1. Note, however, that if object 1's **head** is set to 2 and its **tail** is set to 7, this does *not* mean that objects 3 through 6 are all children of object 1. It only means that the first of its chain of children is 2 and the last is 7; the members of object 1's "family" are indicated by the **next** pointers of the children themselves.

The OBJECT structure

Each object in an object tree must be described with the **OBJECT** structure that is declared in the header file **obdefs.h**. This structure is declared as follows:

```
typedef struct object
{
	int ob_next;			/* Object's next sibling */
	int ob_head;			/* Head of object's children */
	int ob_tail;			/* Tail of object's children */
	unsigned int ob_type;		/* Type of object */
	unsigned int ob_flags;		/* Flags */
	unsigned int ob_state;		/* Status */
	long ob_spec;			/* Object's specification */
	int ob_x;			/* X coordinate of object */
	int ob_y;			/* Y coordinate of object */
	int ob_width;			/* Width */
	int ob_height;			/* Height */
} OBJECT;
```

An object, as can be seen, is built out of following 11 elements:

ob_next The **next** pointer.

ob_head The **head** pointer.

ob_tail The **tail** pointer.

ob_type This indicates the object's type. The different types of object will be discussed below.

ob_flags This field encodes one of a set of flags for the object. The allowable flags are as follows:

0x000	NONE	No flags selected
0x001	SELECTABLE	Selectable by user
0x002	DEFAULT	Default (e.g., for buttons)
0x004	EXIT	If selected, ends dialogue
0x008	EDITABLE	Editable by user (e.g., string)

0x010	RBUTTON	Radio button
0x020	LASTOB	Last object in tree
0x040	TOUCHEXIT	Click once to end dialogue
0x080	HIDETREE	Hide object from searches
0x100	INDIRECT	Redirect to another object

Not every flag applies to every type of object. Some flags are mutually exclusive, e.g., EXIT and TOUCHEXIT; both force an exit from a dialogue, but the former requires that the button be clicked twice and the latter requires only one click.

ob_state This indicates the object's status, i.e., how the object is to be displayed. The status codes are as follows:

0x00	NORMAL	Normal display
0x01	SELECTED	Displayed in reverse video
0x02	CROSSED	Draw an 'X' in object; used with rectangles only
0x04	CHECKED	Draw check mark next to object
0x08	DISABLED	Draw in shading rather than solid
0x10	OUTLINED	Draw border around object
0x20	SHADOWED	Draw shadow on object

Note that this specification can be changed as the program runs; for example, in the specification in a menu object can change to indicate that the item is disabled or has been selected.

ob_spec The object's specification. This field, which is the only long field in the **OBJECT** structure, can hold a pointer to a string, a pointer to a structure, or a bit map, depending on the type of object being described. Which specification belongs with which object will be described below.

ob_x X coordinate of the object. In the root object, this value is an absolute value, in rasters; for each subordinate object, this value is relative to the X value of its parent. This allows the entire object tree to be repositioned on the screen simply by redefining the X coordinate of the root object.

ob_y Y coordinate of the object. In the root object, this value is an absolute value, in rasters; for each subordinate object, this value is relative to the Y value of its parent.

ob_width The object's width. This is always an absolute value.

ob_height The object's height. This is always an absolute value.

Types of objects

The following table lists the available types of objects. As noted above, each type of object used the field **ob_spec** in a different way; the specification is also given:

G_BOX Draw a rectangle on the screen. The field **ob_spec** holds a bit map that describes the box's color and the thickness of its border, as follows:

high word The high byte is not used. The low byte holds the thickness of the border, from -127 to 127. Negative numbers draw the border outwards from the edge of the rectangle, whereas positive numbers draw the border inwards.

low word The high nybble of the high byte holds the color of the interior of the rectangle, from one to 15, as follows:

0	WHITE
1	BLACK
2	RED
3	GREEN
4	BLUE
5	CYAN
6	YELLOW
7	MAGENTA
8	WHITE
9	GRAY
10	LRED
11	LGREEN
12	LBLUE
13	LCYAN
14	LYELLOW
15	LMAGENTA

The names in capital letters are mnemonics that are defined in the header file **obdefs.h**; this means that you can use these mnenomics in your program, without having to remember the numeric code of each color.

The low nybble of the high byte encodes the color of any text shown, as above.

In the low byte, the first bit of the high nybble indicates whether or not the object should be transparent; zero indicates that the object is transparent and one indicates that it is not. The next three bits hold the fill pattern, from zero through seven. Zero indicates hollow; seven indicates solid; and one through six indicate gradations of shading, with the higher numbers indicating increasing darkness.

Finally, the low nybble the low word indicates the color of the border, as above.

Example To set a figure with a border width of one raster, an inside color of white, a text color of black, the transparent bit off, the fill pattern of solid, and the border color of black, use the following C code:

```
((1<<16)|(WHITE<<12)|(BLACK<<8)|(1<<7)|(7<<4)|BLACK)
```

This translates into the hexadecimal number 0x101F1.

G_BOXCHAR
This draws a rectangle with a single character inside it. It is used for elements like the "fuller" button on GEM windows. **ob_spec** points to a string that must be only one character long.

G_BOXTEXT
This draws a box and writes text inside it. **ob_spec** points to the structure **TEDINFO**, which is described below.

G_BUTTON This draws a button, which AES handles in its usual manner. **ob_spec** points to the string that is written inside the button.

G_FTEXT This draws a string on the screen that can be edited by the user in the form of a dialogue. This is demonstrated in the second example, below. **ob_spec** points to the structure **TEDINFO**, which is described below.

G_FBOXTEXT
This draws an editable string, like **G_FTEXT**, but surrounds it with a box as well. **ob_spec** points to the structure **TEDINFO**, which is described below.

G_IBOX This draws an "invisible box" on the screen. This box is used to connect a number of elements without changing the appearance of the object. For example, if you wished to reverse a large section of the screen when an icon is clicked, you would overlay the icon with an invisible box sized to the dimensions of the area you wished to reverse; when the icon was clicked, the entire area within the invisible box would be reversed, not just the icon itself. **ob_spec** encodes the color information, as in **G_BOX**.

G_ICON This draws an icon on the screen. **ob_spec** points to the structure **ICONBLK**, which is described below.

G_IMAGE This draws a user-defined shape on the screen. **ob_spec** points to the structure **BITBLK**, which is described below.

G_PROGDEF
This is an object defined by the programmer. **ob_spec** points to the structure **USERBLK**, which is described below

G_STRING This writes a string. **ob_spec** points to the string being written.

G_TEXT This writes formatted text on the screen. **ob_spec** points to the structure **TEDINFO**, which is described below.

G_TITLE This is used to create a title on the menu bar. **ob_spec** points to the string to be written. As indicated above, four specialized structures are used by the set of objects: **BITBLK**, **ICONBLK**, **TEDINFO**, and **USERBLK**.

The BITBLK structure
The **BITBLK** structure is defined in the header file **obdefs.h** as follows:

```
typedef struct bit_block
{
        int *bi_pdata;        /* Points to bit map */
        int bi_wb;            /* Width of bit map in bytes */
        int bi_hl;            /* Height in lines */
        int bi_x;             /* Source X in bit form */
        int bi_y;             /* Source Y in bit form */
        int bi_color;         /* Color of blt */
} BITBLK;
```

bi_pdata points to an array of integers that encode the object's bit map. **bi_wb** gives the width of the bit map, in bytes. Note that the value of this variable must be even, to align along word boundaries. **bi_hl** gives the height of the bit map, in rasters. **bi_x** and **bi_y** give, respectively, the X and Y coordinates of the bit map. Finally, **bi_color** gives the object's color, encoded as above.

The ICONBLK structure
The structure **ICONBLK** is defined in the header file **obdefs.h** as follows:

```
typedef struct icon_block
{
        int *ib_pmask;          /* Points to icon mask */
        int *ib_pdata;          /* Points to icon description */
        char *ib_ptext;         /* String to appear in icon */
        int ib_char;            /* Character to appear in icon */
        int ib_xchar;           /* X location of character */
        int ib_ychar;           /* Y location of character */
        int ib_xicon;           /* X location of icon */
        int ib_yicon;           /* Y location of icon */
        int ib_wicon;           /* Width of icon */
        int ib_hicon;           /* Height of icon */
        int ib_xtext;           /* X location of text */
        int ib_ytext;           /* Y location of text */
        int ib_wtext;           /* Width of text */
        int ib_htext;           /* Height of text */
} ICONBLK;
```

ib_pmask points to an array of integers that describe the icon mask. **ib_pdata** points to an array of integers that describe the icon itself. **ib_text** points to a string to be written into the icon; **ib_char** points to a single character to be drawn on the icon. **ib_xchar** and **ib_ychar** give, respectively, the X and Y coordinates of the character. **ib_xicon**, **ib_yicon**, **ib_wicon**, and **ib_yicon** give, respectively, the X coordinate, the Y coordinate, the width, and the height of the icon; and **ib_xtext**, **ib_ytext**, **ib_wtext**, and **ib_htext** give, respectively, the X coordinate, the Y coordinate, the width, and the height of the text string within the icon.

The TEDINFO structure
This structure is used to create an editable dialogue. It is defined in the header file **obdefs.h** as follows:

```
typedef struct text_edinfo
{
        long te_ptext;          /* Points to text */
        long te_ptmplt;         /* Points to template */
        long te_pvalid;         /* Points to validation chars */
        int te_font;            /* Font */
        int te_junk1;           /* Junk word */
        int te_just;            /* Justification */
        int te_color;           /* Color */
        int te_junk2;           /* Junk word */
        int te_thickness;       /* Border thickness */
        int te_txtlen;          /* Length of text string */
        int te_tmplen;          /* Length of template string */
} TEDINFO;
```

te_ptext points to a string to be displayed within the object. The text typed by the user will be written over this string. If you do not want text to be displayed, replace it with a string of '@' characters as long as the maximum length of the string to be input.

te_ptmplt points to a template that will be used to input data. The template consists of a prompt, plus a string of underbar characters that is as long as the maximum length of the string that the user can input. The following is an example of a template string:

```
ENTER FILE NAME: ________.___
```

te_pvalid points to a string of validation characters. This string must be as long as the string that the user can input. Each character input by the user is checked against its corresponding validation character to ensure that it is of the right type. The validation characters are as follows:

9	All numerals, zero through nine
a	All alphabetic characters plus space
n	Alphabetic characters, numerals, space
p	Valid TOS path name characters
A	Upper-case alphabetic characters plus space
N	Upper-case alphabetic characters, space, numerals
F	TOS file name characters, question mark, asterisk, colon
P	TOS path name characters, question mark, asterisk, colon
X	Anything

Note that use of any validation character besides **F** *or* **X** *will cause a catastrophic system error.*

te_font indicates which font you want. **te_junk1** and **te_junk2** are reserved; they can be set to any value. **te_just** indicates how you want the text to be justified: **TE_LEFT** indicates left justification; **TE_RIGHT**, right justification; and **TE_CNTR**, centering. **te_color** indicates the color of the object; the color codes are the same as for **G_BOX**.

te_thickness is the thickness of the border; it uses the same values as **G_BOX**. Finally, **te_txtlen** and **te_tmplen** give, respectively, the length of the user input string and the length of the template, each in bytes. The length of each should be one byte longer than the strings pointed to by **te_ptext** and **te_ptmplt**, to allow the addition of the NUL character at the end of each.

The USERBLK structure
The **USERBLK** structure can also be called the **APPLBLK** or **APPL_BLK** structure in other bindings. It is defined in the header file **obdefs.h** as follows:

```
typedef struct user_blk
{
        long ub_code;           /* points to user's code */
        long ub_parm;           /* points to parameter */
} USERBLK;
```

This structure allows the programmer to define her own object or routine; **ub_code** points to the routine in question, which can be specialized code written in C or assembly language to do specific tasks beyond the scope of the normal AES routines. **ub_parm** points to the parameter to be passed to the routine named in **ub_code**. To use this structure, a programmer must have a sophisticated grasp of the AES.

Designing objects
Designing an object by hand is difficult. If possible, you should use a resource construction set (RCS) in designing screen elements; however, it is best to know how to modify the output of the RCS in order to gain exactly the results you want.

Before beginning, you should do the following: First, draw a picture of the object on graph paper. For text, each cell on the graph paper can considered equivalent to one character cell, i.e., the space taken up by one standard character on the screen (in high resolution, a character is eight rasters wide by 16 high; in medium resolution, it is eight rasters wide by eight high; and in low resolution, it is four wide by eight high). Otherwise, each cell can be considered equivalent to a pixel. Drawing the picture may seem tedious, but will save you time over trying to draw it "on the fly" on the screen.

Second, draw a "geneological table" of all the objects within the object tree. This will ensure that you set the **next**, **head**, and **tail** pointers for each object correctly. An example of such a table appears in the entry for **menu**.

Examples
The first example draws a set of seven nested rectangles on the screen. Typing any key returns you to **msh**. Note that all objects are sized in rasters, for a high-resolution screen.

```
#include <aesbind.h>
#include <obdefs.h>
#include <gemdefs.h>
#define SPEC1 0x100F1L
/*
 *    I.e.:      (1 << 16) |       [Border 1 raster thick]
 *               (WHITE << 12) |  [Border color; WHITE = 0]
 *               (WHITE << 8) |   [Text color]
 *              ((1 << 7) |        [Turn on replace]
 *               (7 << 4) |        [Fill pattern to solid] [one nybble])
 *                BLACK            [Fill color; BLACK = 1]
 */
```

```
#define SPEC2 0x111F0L
/*
 *      I.e.:   (1 << 16) |        [Border 1 raster thick]
 *              (BLACK << 12) |    [Border color]
 *              (BLACK << 8) |     [Text color]
 *             ((1 << 7) |         [Turn on replace]
 *              (7 << 4) |         [Fill pattern to solid] [one nybble])
 *               WHITE             [Fill color]
 */

OBJECT fill[] = {
/*    next/head/tail/type/  flags / state /specif./ X / Y  / W  / H  */
        -1,  1,  1, G_BOX, DEFAULT, NORMAL, SPEC1,  0,  0, 639, 399,
         0,  2,  2, G_BOX, DEFAULT, NORMAL, SPEC2, 50, 30, 539, 339,
         1,  3,  3, G_BOX, DEFAULT, NORMAL, SPEC1, 50, 30, 439, 279,
         2,  4,  4, G_BOX, DEFAULT, NORMAL, SPEC2, 50, 30, 339, 219,
         3,  5,  5, G_BOX, DEFAULT, NORMAL, SPEC1, 50, 30, 239, 159,
         4,  6,  6, G_BOX, DEFAULT, NORMAL, SPEC2, 50, 30, 139,  99,
         5, -1, -1, G_BOX, DEFAULT, NORMAL, SPEC1, 50, 30,  39,  39
};
/*
 * Note: X, Y are absolute for root object; for all others, X & Y are
 * relative to the parent object.  W & H are absolute for all objects.
 * All values in rasters; calculated for high-resolution screen.
 */

main() {
        int nowhere = 0;                        /* For unused pointers */

        appl_init();                            /* Begin application */
        graf_mouse(M_OFF, &nowhere);            /* Turn off mouse pointer */
        objc_draw(fill, ROOT, MAX_DEPTH, 0, 0, 639, 399);
        evnt_keybd();                           /* Wait for keybd event */
        graf_mouse(M_ON, &nowhere);             /* Turn on mouse pointer */
        appl_exit();                            /* Exit from application */
        exit(0);
}
```

The second example presents a brief dialogue, and demonstrates the **TEDINFO** structure. Note that all objects are sized in rasters, for a high-resolution screen.

```
#include <aesbind.h>
#include <obdefs.h>
#include <gemdefs.h>
```

```
#define SPEC 0x510F1L
/*
 *     I.e.:  (5 << 16) |   [Border 5 rasters thick]
 *            (BLACK << 12) |[Border color; BLACK = 1]
 *            (WHITE << 8) |[Text color; WHITE = 0]
 *            {(1 << 7) |   [Turn on replace bit]
 *            (7 << 4) |    [Fill pattern solid] [Together one nybble] }
 *            BLACK         [Fill color]
 */

#define B1FLAGS 0x5                     /* I.e.: SELECTABLE | EXIT */
#define B2FLAGS 0x7                     /* I.e.: SELECTABLE | DEFAULT | EXIT */

/* Strings and structure used with dialogue */
char input[] =  "@@@@@@@@@@";
char template[] = "YOUR NAME: __________";
char check[] = "FFFFFFFFFF";
char button1[] = "OK";
char button2[] = "EXIT";

TEDINFO text[] = {
/*  pointer to text ('@' indicates no text)
 *  |    pointer to text template
 *  |    |     pointer to validation string
 *  |    |     |    font code character
 *  |    |     |    |   reserved integer (takes anything)
 *  |    |     |    |   |  justification code character
 *  |    |     |    |   |  |     color code
 *  |    |     |    |   |  |     |    reserved integer (takes anything)
 *  |    |     |    |   |  |     |    |   border thickness (rasters)
 *  |    |     |    |   |  |     |    |   | input string (chars)
 *  |    |     |    |   |  |     |    |   | | template string (chars)
 *  v    v     v    v   v  v     v    v v v v   */
  input,template,check,IBM,1,TE_CNTR,WHITE,5,1,11,22
};

/* Define the dialogue object */
OBJECT dialogue[] = {
/*   next/head/tail/  type  /   flags  / state/ specif./ X /  Y/   W /  H */
       -1,   1,   3,    G_BOX,     NONE, NORMAL,    SPEC,   0,    0, 600, 250,
        2,  -1,  -1,  G_FTEXT, EDITABLE, NORMAL,    text, 100,   50, 400, 100,
        3,  -1,  -1, G_BUTTON,  B1FLAGS, NORMAL, button1, 230, 200,  20,  20,
        0,  -1,  -1, G_BUTTON,  B2FLAGS, NORMAL, button2, 300, 200,  40,  20
};

/* Rectangles used in ensuing fracas */
Rect tempbox = { 0, 0, 0, 0 };
Prect tempptr = { &tempbox.x, &tempbox.y, &tempbox.w, &tempbox.h };
```

```
main() {
        int nowhere = 0;                    /* For orphaned pointers */
        int quit;
        char newstring[29] = "YOUR NAME IS ";

        appl_init();                        /* Begin application */
        graf_mouse(ARROW, &nowhere);        /* Mouse ptr. to arrow */

        form_center(dialogue, tempptr);     /* Center dialogue box */
        form_dial(0, 1, 1, 1, 1, tempbox); /* Get screen area */
        form_dial(1, 1, 1, 1, 1, tempbox); /* "Star wars" effect */
        objc_draw(dialogue, ROOT, MAX_DEPTH, tempbox);
                                            /* Draw dialogue object */

        for (;;) {
                quit = form_do(dialogue, 1);
                if (quit == 2) {
                        strcat(newstring, input);
                        strcpy(template, newstring);
                        objc_draw(dialogue, ROOT, MAX_DEPTH, tempbox);
                }

                if (quit == 3) {
                        form_dial(2, 1, 1, 1, 1, tempbox);
                        form_dial(3, 1, 1, 1, 1, tempbox);
                        appl_exit();
                        exit(0);
                }
        }
}
```

See Also
AES, menu, obdefs.h, TOS, window

object format—Definition

An **object format** describes the form of compiled program that still contains relocation information. The linker **ld** reads file in object format to create executable files.

Mark Williams C creates object modules that are in the format **n.out**, which differs somewhat from other formats used on the Atari ST.

See Also
ld, n.out

od—Command

od [-bcdox] [*file*] [[+] *offset*[.][b]]

od prints the specified *file* as a sequence of octal numbers, or machine words. If no *file* is specified, **od** dumps the standard input.

The following options allow the user to select the output format:

-b	bytes in hexadecimal
-c	bytes in ASCII characters
-d	words in decimal
-o	words in octal

Dumping can start at *offset* into the file. The specified *offset* is octal unless the '.' suffix is present to signify decimal. The *offset* is in bytes unless the b suffix is present to signify 512-byte blocks.

See Also
ASCII, commands, db, msh

Offgibit—**xbios** function 29 (**osbind.h**)
Clear a bit in the sound chip's A port
#include <osbind.h>
#include <xbios.h>
void **Offgibit(***mask***) char** *mask*;

Offgibit manipulates the sound chip's register A (also called the "A port"). This port controls the disk drives.

Offgibit reads the contents of register A; it then ANDs this value with *mask*; and it writes the result back into register A. The bits in this register are bound to various control lines within the Atari ST. For a table of which bits bind which lines, see the entry for **Ongibit**.

Example
The following example demonstrates **Ongibit** and **Offgibit**:

```
#include <osbind.h>

main() {
	unsigned char a;

	Cconws("Wait for both floppy drives to stop and type a key\r\n");
	Cnecin();

	a = Giaccess(0, 14);		/* save the original value... */

	Offgibit(0xF9);			/* turn off bits 1 and 2 */
	Cconws("Both floppy drive lights on...\n\r");
	Cnecin();
```

```
	Ongibit(0x02);				/* turn on bit 1 */
	Cconws("Drive A light off...\n\r");
	Cnecin();

	Ongibit(0x04);				/* turn on bit 2 */
	Cconws("Drive B light off...\n\r");
	Cnecin();

	Giaccess(a,0x80|14);			/* restore original contents */
	Pterm0();
}
```

See Also
Giaccess, Ongibit, TOS, xbios

Ongibit—xbios function 30 (**osbind.h**)
Turn on a bit in the sound chip's A port
#include <osbind.h>
#include <xbios.h>
void **Ongibit(***mask***) char** *mask*;

Ongibit manipulates the sound chip's register A (also called the "A port").

Ongibit first reads the contents of register A; it then ORs with *mask*; and finally it writes the result back into register A.

The bits in register A are bound to various control lines within the Atari ST, as follows:

0	side of the floppy disk (0/1)
1	drive A (selected when clear)
2	drive B (selected when clear)
3	RS-232 request-to-send (RTS) line
4	RS-232 data-terminal-ready (DTR) line
5	Centronics data strobe
6	general purpose output (GPO) on video connector
7	unused

number should be set the bit that corresponds to the desired line.

Example
For an example of this function, see the entry for **Offgibit**.

See Also
Giaccess, Offgibit, TOS, xbios

open—UNIX system call (**libc.a/open**)
Open a file
open(*file*, *type*) char ****file*; **int *type*;

open prepares a *file* to be written into, or to have its data read. When successful, **open** returns a file descriptor, which is a small, positive integer that identifies the open *file* for subsequent calls to **read**, **write**, **close**, **dup**, or **dup2**, The *type* argument can be set to zero for reading, one for writing, or two for both reading and writing. After a *file* is opened, reading or writing will begin at byte 0.

Example
This example copies **argv[1]** to **argv[2]** by using UNIX-style routines. It demonstrates the functions **open**, **close**, **read**, **write**, and **creat**.

```
#include <stdio.h>
#define BUFSIZE (20*512)
char buf[BUFSIZE];

main(argc, argv) int argc; char *argv[]; {
	register int ifd, ofd;
	register unsigned int n;

	if (argc != 3)
		fatal("Usage: copy source destination");
	if ((ifd = open(argv[1], 0)) == -1)
		fatal("cannot open input file");
	if ((ofd = creat(argv[2], 0)) == -1)
		fatal("cannot open output file");
	while ((n = read(ifd, buf, BUFSIZE)) != 0) {
		if (n == -1)
			fatal("read error");
		if (write(ofd, buf, n) != n)
			fatal("write error");
	}
	if (close(ifd) == -1 || close(ofd) == -1)
		fatal("cannot close");
	exit(0);
}

fatal(s) char *s;
{
	fprintf(stderr, "copy: %s\n", s);
	exit(1);
}
```

See Also
STDIO, UNIX routines

Diagnostics
open returns -1 if the file is nonexistent, or if a system resource is exhausted.

Notes
open is a low-level call that passes data directly to TOS. It should be intermixed cautiously with high-level calls, such as **fread**, **fwrite**, or **fopen**.

operator—Definition

An **operator** relates one operand to another. For example, the statement

```
1+2
```

relates **1** and **2** through the operation of addition; on the other hand, the statement

```
A>B
```

relates **A** and **B** logically, by asserting that the former is greater than the latter; whereas

```
A=B
```

relates **A** and **B** by assigning the value of the latter to the former. The following is a table of C's operators:

*	multiplication
/	division
%	remainder
+	addition
-	subtraction
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
&&	logical AND
!=	inequality
!	logical negation
||	logical OR

=	assign
+=	increment and assign
-=	decrement and assign
*=	multiply and assign
/=	divide and assign
%=	modulo and assign
++	increment
--	decrement
==	equivalence
&	bitwise AND
^	bitwise exclusive OR
|	bitwise inclusive OR
<<	shift left
>>	shift right
*	indirection
&	render an address
()	function indicator
[]	array indicator
->	structure pointer
.	structure member
?:	conditional expression

See Also
precedence, sizeof
The C Programming Language, page 49.

osbind.h—Header file
#include <osbind.h>

osbind.h is a header file that declares the functions **bios()**, **gemdos()**, and **xbios()**. It also defines numerous macros that ease the use of these functions. The text of **osbind.h** is included with your copy of Mark Williams C.

See Also
bios, gemdos, header file, xbios, TOS

path—Definition

A **path** is the full name of a file, including the names of all the directories within which it resides. Conventions for naming paths vary among operating systems; for example, the file **foobar** that is in the directory **computer** that, in turn, is owned by user **anne** could be listed as follows under the COHERENT system:

```
/usr/anne/computer/foobar
```

but as follows under MS-DOS or TOS:

```
computer\foobar
```

The latter two operating systems do not use user names in constructing the path name. Note, too, that MS-DOS and TOS use the backslash '\' rather than the slash '/' to separate the elements of a path name.

See Also
directory

PATH—Environmental parameter

PATH names directories that **msh** searches when looking for files that you have asked it to execute. For example, typing

```
setenv PATH=.bin,\bin,,\lib
```

tells **msh** to search for executable files first in its set of built-in commands (as indicated by **.bin**), then in the directory **\bin**, then in the current directory (as indicated by the two commas with nothing between them), and finally in the directory **lib**.

It is set with the **setenv** command.

See Also
msh, setenv

patterns—Definition

A **pattern** is any combination of ASCII characters and wildcards that can be interpreted by a command.

See Also
egrep, wildcard

peekb—Library function (**libc.a/peekb**)

Extract a byte from memory
**int peekb(*bp*) char ****bp*;

peekb examines an arbitrary location in memory. It reads a byte located at the address *bp*. **peekb** circumvents the system's memory protection by temporarily entering supervisor mode.

See Also
peekl, peekw, pokeb, pokel, pokew

peekl—Library function (**libc.a/peekl**)
Extract a **long** from memory
**long peekl(*lp*) long ****lp*;

peekl returns the **long** (four bytes) at *lp*. **peekl** circumvents the system's memory protection by temporarily entering supervisor mode.

See Also
peekb, peekw, pokeb, pokel, pokew

Notes
peekl does not test for odd addresses, and will generate a bus error if given such an address. In general, be careful about what you **peek** and **poke.**

peekw—Library function (**libc.a/peekw**)
Extract a word from memory
**int peekw(*wp*) int ****wp*;

peekw returns the **word** (two bytes) at *wp*. **peekw** circumvents the system's memory protection by temporarily entering supervisor mode.

See Also
peekb, peekl, pokeb, pokel, pokew

Notes
peekw does not test for odd addresses, and will generate a bus error if given such an address. In general, be careful about what you **peek** and **poke.**

perror—General function (**libc.a/perror**)
System call error messages
#include <errno.h>
perror(*string*)
char ****string*; **extern int *sys_nerr*; **extern char ****sys_errlist[ ]*;

perror prints an error message on the standard error device. The message consists of the argument *string*, followed by a brief description of the last system call that failed. The external variable **errno** contains the last error number. Normally, *string* is the perror of the command that failed or a file perror.

The external array **sys_errlist** gives the list of messages used by **perror**. The external **sys_nerr** gives the number of messages in the list.

See Also
errno, errno.h, error codes

Pexec—**gemdos** function 75 (**osbind.h**)
Load or execute a process
#include <osbind.h>
long Pexec(*mode, path, tail, env***) int** *mode* **;**
char **path*, **tail*, **env*;

Pexec loads or executes a process. *mode* equals zero if the process is to be loaded and executed, or three if the process is to be loaded but not executed; the latter mode is used with overlays. *path* points to the path name of the file to be loaded; it must be a NUL-terminated string. *tail* points to the command tail, which included redirection information. *env* points to a block of strings that define the environment. Each string must terminate with a NUL character, and the block as a whole must terminate in NULL.

If *mode* equals zero, **Pexec** returns the child process's exit status when the child process exits; if *mode* equals three, it returns the address of the base page of the loaded process. In either instance, it returns a negative error code if it cannot load the process.

Example
This example times the execution speed of a program. It also demonstrates the time function **clock**.

```
#include <osbind.h>
#include <time.h>

main(argc, argv)
int argc; char *argv[];
{
	char program[80];
	char command[256];
	int x;
	clock_t timer;
	int status;

	if (argc < 2) {
		printf("usage: time command [ args ... ]\n");
		exit(1);
	}
```

```
        strcpy(program,argv[1]);
        strcat(program,".PRG");
        command[0] = 0;

        for (x=2 ; x < argc ; x++) {
                strcat( command, " ");
                strcat( command, argv[x]);
        }

        timer = clock();
        status = Pexec(0, program, command, "PATH=\0");
        timer = clock() - timer;

        printf("%ld.%03ld seconds\n",
                timer/CLK_TCK, (timer%CLK_TCK) * (1000/CLK_TCK));
        return status;
}
```

See Also

argv, gemdos, TOS

Physbase—**xbios** function 2 (**osbind.h**)

Read the physical screen's display base

#include <osbind.h>
#include <xbios.h>
long Physbase()

Physbase reads the physical screen's display base, and returns a pointer to the display base. The physical screen base is the location in memory currently displayed.

For examples of this function, see the entries for **Logbase** and **Prtblk**.

See Also

Logbase, Setscreen, TOS, xbios

picture—Example

Format numbers under mask

double picture(*number, mask, output***)**
double *number*; **char** **mask, *output*;

picture uses a mask to format a double-precision number; it returns any overflow. It is designed to be used with accounting programs, and other utilities that require precise formatting of printed numbers.

picture formats a given number by using a mask string. The mask may contain any characters; however, only a few have special significance. Non-special characters in the mask body are printed if, during execution, they come to be preceded by one or more numerals. Trailing non-special characters print if the displayed number is negative.

The following lists the special characters that control formatting within a mask:

9 Provides a slot for a number. For example, **5** with mask **999 CR** gives **005<sp><sp><sp>**, whereas printing **-5** with mask **999 CR** gives **005 CR**. Note that 'C' and 'R' are not special characters, but are taken literally.

Z Provide a slot for a number, but supress leading zeroes. For example, printing **1034** with mask **ZZZ,ZZZ** gives **<sp><sp>1,034**. Note that the comma is not a special character.

J Provide a slot for a number, but shrink out leading zeroes. For example, printing **1034** with mask **JJJ,JJJ** gives **1,034**.

K Provide a slot for a number, but shrink out any zeroes. For example, printing **070884** with mask **K9/K9/K9** gives **7/8/84**.

$ Float a dollar sign to the front of the displayed number. For example, printing **105** with mask **$Z,ZZZ** gives **<sp><sp>$105**.

. Separate the number between decimal and integer portions. For example, printing **105.67** with mask **ZZZ.999** gives **105.670**.

T Provide a slot for a number, but supress trailing zeroes. For example, printing **105.670** with mask **ZZ9.9TT** gives **105.67<sp>**.

S Provide a slot for a number but shrink out trailing zeroes. For example, printing **105.600** with mask **ZZ9.9SS** gives **105.6**.

\- This character, if placed to the left of the mask, floats to the front like the '$', but only if the number is negative. For example, printing **105** with mask **-Z,ZZZ** gives **<sp><sp>105**, whereas printing **-105** gives **<sp><sp>-105**.

(This character acts like the minus sign '-', but prints a '('. For example, printing **105** with mask **(ZZZ)** gives whereas printing **-5** gives **<sp><sp>(5)**.

\+ If placed to the left of the mask, this character floats to the front like the minus sign '-', but is replaced by a '-' if the number is minus. For example, printing **5** with mask **+ZZZ** gives **<sp><sp>5+**, whereas printing **-5** gives **<sp><sp>-5**. Placed behind the mask, it is printed if the number is positive, but is replaced by a minus sign '-' if the number is negative. For example, printing **5** with mask **ZZZ+** gives whereas printing **-5** gives **<sp><sp>5-**.

* When placed to the left of the mask, this character fills all leading spaces to its right. For example, printing **104.10** with mask ***ZZZ,ZZZ.99** gives *****104.10, and printing **104.10** with mask ***$ZZ,ZZZ.99** gives ****$104.10.

See Also
commands, STDIO

Diagnostics
picture returns all overflow as a **double**. For example, attempting to print -1234 with mask **(ZZZ)** gives **(234)** and returns -1.

Notes
For the source code of **picture**, see the file **picture.c**.

pnmatch—General function (**libc.a/pnmatch**)
Match string pattern
int pnmatch(*string*, *pattern*, *flag*)
char **string*, **pattern*; **int** *flag*;

pnmatch matches *string* with *pattern*, which is a regular expression. **pnmatch** returns 1 if *pattern* matches *string*, and 0 if it does not. Each character in *pattern* must exactly match a character in *string*; however, the wildcards '*', '?', '[', and ']' can be used in *pattern* to expand the range of matching. The *flag* argument must be either 0 or 1: 0 means that *pattern* must match *string* exactly, whereas 1 means that *pattern* can match any part of *string*. In the latter case, the wildcards '^' and '$' can also be used in *pattern*.

Example
This example looks for pattern **argv[1]** in standard input or in file **argv[2]**. It demonstrates the functions **pnmatch**, **fgets**, and **freopen**.

```
#include <stdio.h>
#define MAXLINE 128
char buf[MAXLINE];

main(argc, argv) int argc; char *argv[]; {
	if (argc != 2 && argc != 3)
		fatal("Usage: pnmatch pattern [ file ]");
	if (argc == 3 && freopen(argv[2], "r", stdin) == NULL)
		fatal("cannot open input file");
	while (fgets(buf, MAXLINE, stdin) != NULL) {
		if (pnmatch(buf, argv[1], 1))
			printf("%s", buf);
	}
	if (!feof(stdin))
		fatal("read error");
	exit(0);
}

fatal(s) char *s; {
	fprintf(stderr, "pnmatch: %s\n", s);
	exit(1);
}
```

See Also
egrep, msh, string

Notes
flag must be zero or one for **pnmatch** to yield predictable results.

pointer—Definition

A **pointer** is a data type that consists of the address of another item of data; therefore, it is said to "point" to that item of data.

The physical size of the pointer data type is determined entirely by the microprocessor. Pointers are 16 bits long on the i8086, SMALL model, non-segmented Z8000, and on the PDP-11; they are 32 bits long on the i8086, LARGE model, segmented Z8000, the 68000, and the VAX.

Note that failure to declare a function that returns a pointer (most commonly, a **char ***) will result in that function being implicitly declared as an **int**. This will not cause an error on microprocessors in which an **int** and a pointer both have the same size; transporting this code to a microprocessor in which an **int** consists of 16 bits and a pointer consists of 32 bits will result in the pointers being truncated to 16 bits and the program probably failing.

C allows pointers and integers to be compared or converted to each other without restriction. Mark Williams C flags such conversions with the strict message

```
integer pointer pun
```

and comparisons with the strict message

```
integer pointer comparison
```

These problems should be corrected if you want your code to be portable to other computing environments.

Casting a pointer from one data type to another may result in the loss of precision when alignment restrictions are taken into account. These sorts of data transformations should be done with great care to ensure that code remains portable.

See Also
data formats, declarations, pun

pokeb—Library function (**libc.a/pokeb**)

Insert a byte into memory
int pokeb(*bp*, *b***) char ****bp*; **int** *b*;

pokeb writes the character *b* at an arbitrary location *bp* in memory. **pokeb** circumvents the system's memory protection by temporarily entering supervisor mode. **pokeb** returns its argument **b**.

See Also
peekb, peekl, peekw, pokel, pokew

pokel—Library function (**libc.a/pokel**)
Insert a **long** into memory
long **pokel**(*lp, l*) **long** **lp, l*;

pokel writes the **long** *l* (four bytes) at an arbitrary location *lp* in memory. **pokel** circumvents the system's memory protection by temporarily entering supervisor mode.

See Also
peekb, peekl, peekw, pokeb, pokew

Notes
pokel does not test for odd addresses, and will generate a bus error if given such an address. In general, be careful about what you **peek** and **poke**.

pokew—Library function (**libc.a/pokew**)
Insert a **long** into memory
int pokew(*wp, l*) **int** **wp, w*;

pokew writes the word *w* (two bytes) at an arbitrary location *wp* in memory. **pokew** circumvents the system's memory protection by temporarily entering supervisor mode.

See Also
peekb, peekl, peekw, pokeb, pokel

Notes
pokew does not test for odd addresses, and will generate a bus error if given such an address. In general, be careful about what you **peek** and **poke**.

port—Definition
A **port** passes data to and receives data from remote devices.

See Also
aux:, fclose, FILE, fopen, prn:, stream

portability—Definition
Portability means that code can be recompiled and run under different computing environments without modification. Although true portability is an ideal that is difficult to realize, you can take a number of practical steps to ensure that your code is portable:

1. Do not assume that an integer and a pointer have the same size. Remember that undeclared functions are assumed to return an **int**. If a function returns a pointer, declare it so.
2. Do not write routines that depend on particular order of code evaluation, particular byte ordering, or particular length of data types.
3. Do not write routines that play tricks with a machine's "magic numbers"; for example, writing a routine that depends on a file's ending with **<ctrl-Z>** instead of **EOF** ensures that that code can run only under operating systems that recognize this magic number.
4. Always use manifest constants, such as **EOF**, and make full use of **#define** statements.
5. Use header files to hold all machine-dependent code.
6. Declare everything explicitly. In particular, be sure to declare functions to **void** if they do not return a value; this avoids unforeseen problems with undefined return values.

See Also
#define, header file, manifest constant, pointer, pun, void

pow—Mathematics function (**libm.a/pow**)
Compute a power of a number
#include <math.h>
double pow(*z*, x) double *z*, *x*;

pow returns *z* raised to the power of *x*, or z^x.

Example
For an example of this function, see the entry for **exp**.

See Also
mathematics library

Diagnostics
pow indicates overflow by an **errno** of **ERANGE** and a huge returned value.

pr—Command
Paginate and print files
pr [*options*] [*file* ...]

pr paginates each named *file* and sends it to the standard output. The file name '-' means standard input. If no *file* is specified, **pr** reads the standard input.

Each page has a header that gives the date, file name, and page and line numbers. **pr** may be used with the following options.

+n Skip the first *n* pages of each input file.

-n Print the text in *n* columns. This is used to print out material that was typed in one or more columns.

-h *header*
Use *header* in place of the text name in the title. If *header* is more than one word long, it must be enclosed within quotation marks.

-l*n* Set the page length to *n* lines (default, 66).

-m Print the texts simultaneously, in separate columns. Each text will be assigned an equal amount of width on the page; any lines longer than that will be truncated. This is used to print several similar texts or listings simultaneously.

-n Number each line as it is printed.

-s*c* Separate each column by the character *c*. You can separate columns with a letter of the alphabet, a period, or an asterisk. Normally, each column is left justified in a fixed-width field.

-t Suppress the printing of the header on each page, and the header and footer space.

-w*n* Set the page width to *n* columns (default, 80). Text lines are truncated to fit the column width. The maximum width is 256 columns.

Example
To print a numbered listing of a text file, do the following: First, plug a printer into your Atari ST and turn it on. Second, type this command:

```
pr -n filename >prn:
```

where *filename* is the name of the file you wish to print.

See Also
commands, prn:

precedence—Definition
Precedence refers to the property of each C operator that determines priority of execution; operators are executed in order of their degree of precedence, from highest to lowest. The order of precedence for operators is summarized on page 49 of *The C Programming Language*.

See Also
operators

printf—General function (**libc.a/printf**)

Format output

#include <stdio.h>

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

printf uses the *format* string to specify an output format for each *arg*, which it then writes on the standard output. **printf** reads characters from *format* one at a time; any character other than a percent sign '%' or a string that is introduced with a percent sign is copied to the output directly. '%' tells **printf** that what follows specifies how the corresponding *arg* is to be formatted; the characters that follow '%' can set the output width and the type of conversion desired. The following modifiers, in this order, may precede the conversion type:

1. A minus sign '-' will left-justify the output field, instead of the default right justify.
2. A string of digits gives the *width* of the output field. Normally, the field is padded with spaces to the field width; it is padded on the left unless left justification is specified with a '-'. If the field width begins with '0', the field is padded with '0' characters instead of spaces; the '0' does not cause the field width to be taken as an octal number. If the width specification is an asterisk '*', the routine uses the next *arg* as an integer that gives the width of the field.
3. A period '.' followed by a string of digits indicates the *precision*. For floating point (**e**, **f**, and **g**) conversions, the precision is the number of digits printed after the decimal point. For string (**s**) conversions, the precision is the maximum number of characters used from the string. If the precision specification is given as an asterisk '*', the routine uses the next *arg* as an integer giving the precision.
4. The letter '**l**' before any integer conversion (**d**, **o**, **x**, or **u**) indicates that the argument is a **long** rather than an **int**. Capitalizing the conversion type has the same effect; note, however, that capitalized conversion types are *not* compatible with all C compiler libraries.

The following format conversions are recognized:

% Output a '%' character. No arguments are processed.

c Convert the **int** argument to a character.

d Convert the **int** argument to signed decimal.

D Convert the **long** argument to signed decimal.

e Convert the **float** or **double** argument to exponential form. The format is: *d.dddddesdd*, where there is always one digit before the decimal point and as many as the *precision* after it (the default is six). The exponent sign *s* may be either '+' or '-'.

f Convert the **float** or **double** argument to a representation with an optional leading minus sign '-', at least one decimal digit, a decimal point ('.'), and optional decimal digits after the decimal point. The number of digits after the decimal point is the *precision* (default, six).

g Convert the **float** or **double** argument to whichever of the formats **d**, **e**, or **f** loses no significant precision and takes the least space.

o Convert the **int** argument to unsigned octal.

O Convert the **long** argument to unsigned octal.

r The next argument points to an array of new arguments that may be used recursively. The first argument of the list is a **char** * that contains a new format string. When the list is exhausted, the routine continues from where it left off in the original format string.

s Output the string to which the **char** * argument points. Reaching either the end of the string, indicated by a NUL character, or the specified *precision* will terminate output. If no *precision* is given, only the end of the string will terminate.

u Convert the **int** argument to unsigned decimal.

U Convert the **long** argument to unsigned decimal.

x Convert the **int** argument to unsigned hexadecimal.

X Convert the **long** argument to unsigned hexadecimal.

Example

The following example uses **printf** to print the location of the mouse pointer on the screen. The code \033H tells **printf** to output an **<esc>** character and the letter 'H', which tells TOS to home the cursor.

```
#include <gemdefs.h>
#include <aesbind.h>

#define CLICKS 1                    /* no. of clicks expected on mouse button */
#define BUTTON 1                    /* which button; 1 = leftmost */
#define DOWN 1                      /* i.e., the mouse button is down */

/* throw-away declarations, to keep system from scribbling over itself */
int nowhere = 0;
Rect norect = { 0, 0, 0, 0 };

main() {
/* declarations used by evnt_multi() */
      int selection;                /* code for event that occurred */
      unsigned int which = (MU_KEYBD | MU_BUTTON);
      int buffer[11];               /* place to write AES messages */
      int mousex;                   /* mouse X coordinate */
      int mousey;                   /* mouse Y coordinate */
```

```
/* OK, here we go ... */
	appl_init();
	graf_mouse(ARROW, &nowhere);

	for(;;) {
		selection = evnt_multi(which, CLICKS, BUTTON, DOWN,
			0, norect, 0, norect, buffer, 0, 0, &mousex, &mousey,
			&nowhere, &nowhere, &nowhere, &nowhere);

		switch(selection) {
		case MU_KEYBD:
			appl_exit();
			exit(0);

		case MU_BUTTON:
			graf_mouse(M_OFF, &nowhere);
			printf("\033HX: %03d Y: %03d\n", mousex, mousey);
			graf_mouse(M_ON, &nowhere);
			break;

		default:
			break;
		}
	}
}
```

See Also
fprintf, putc, puts, scanf, screen control, sprintf, write

Notes
Because C does not perform type checking, it is essential that each argument match its specification in the format string.

The use of upper-case format characters to specify long arguments is not standard, and will be phased out to conform with the ANSI standard. Use the 'l' modifier.

prn:—TOS device 0
TOS logical device for parallel port

TOS gives names to its logical devices. Mark Williams C uses these names, to allow the **STDIO** library routines to access these devices via TOS. **prn:** is the logical device for the the parallel port.

Example

```
#include <stdio.h>
main(){
        FILE *fp, *fopen();
        if ((fp = fopen("prn:","w")) != NULL)
                fprintf(fp,"prn: enabled.\n");
        else printf("prn: cannot open.\n");
}
```

See Also
aux:, con:

process—Definition
A process is a program in the state of execution.

Protobt—xbios function 18 (osbind.h)
Generate a prototype boot sector
#include <osbind.h>
#include <xbios.h>
void Protobt(*buffer, serialno, type, flag***)**
char ****buffer*; **long *serialno*; **int** *type, flag*;

Protobt generates a prototype boot sector, and returns nothing. *buffer* points to a 512-byte buffer; this buffer may already contain an image of a boot sector, but whether it does or not is irrelevant. *serialno* is a serial number that will be stamped into the boot sector; setting *serialno* to -1 leaves the boot sector's serial number unchanged, whereas setting it to any number higher than 0x01000000 creates a random serial number that will be stamped into the boot sector. *type* is an integer that encodes the type of disk being worked with, as follows:

0	40 tracks, single sided
1	40 tracks, double sided
2	80 tracks, single sided
3	80 tracks, double sided

Setting *type* to -1 retains the current disk type.

Finally, *flag* indicates whether the boot sector is executable or non-executable: zero indicates executable; one, non-executable; and -1, retain the current type.

Example
For an example of how to use this macro, see the entry for **Flopfmt**.

See Also
TOS, xbios

Prtblk—TOS macro (**osbind.h**)

Print a dump of the screen

#include <osbind.h>
#include <xbios.h>
**int Prtblk(*p*) struct prtblk ****p*;

Prtblk is a macro that uses the TOS function **xbios**. It prints out a block of memory; it returns 0 if the printing was successful, and nonzero if it was not. *p* points to a specialized structure, which is defined in the header file **xbios.h**.

Prtblk can also be used to print text strings.

Example

This example demonstrates the functions **Prtblk**, **Setprt**, **Physbase**, **Getrez**, and **Setcolor**.

```
#include <osbind.h>
#include <xbios.h>

main() {
	struct prtblk pb;
	int palette[16];
	register int i;

	/* Determine printer characteristics */
	i = Setprt(-1);
	if (i & PR_DAISY)
		pb.pb_type = PB_DAISY;
	else if (i & PR_MONO)
		pb.pb_type = PB_MONO160;

	else if (i & PR_EPSON)
		pb.pb_type = PB_MONO120;
	else
		pb.pb_type = PB_COLOR160;

	pb.pb_port = (i & PR_SERIAL) ? PB_AUX : PB_PRT;
	pb.pb_dstres = (i & PR_FINAL) ? PB_FINAL : PB_DRAFT;

	/* Print the screen */
	if (pb.pb_type != PB_DAISY) {
		pb.pb_blkptr = Physbase();

		switch (pb.pb_srcres = Getrez()) {
		case 0:		pb.pb_width = 320;
			pb.pb_height = 200;
			break;

		case 1:		pb.pb_width = 320;
			pb.pb_height = 400;
			break;
```

```
        case 2:      pb.pb_width = 640;
               pb.pb_height = 400;
               break;
        }

        pb.pb_colpal = &palette[0];
        for (i = 0; i < 16; i += 1)
               palette[i] = Setcolor(i, -1);

        pokew(0x4EEL,1);              /* Set prtcnt, locks out Scrdmp() */
        if (Prtblk(&pb) != 0)
                                      Cconws("Screen print failed.\r\n");

    } else
        Cconws("Cannot print graphics on daisy wheel printer.\r\n");

    /* Print a text string */
    pb.pb_blkptr = "\r\nThis is a string.\r\n";
    pb.pb_width = strlen(pb.pb_blkptr);
    pb.pb_height = 0;
    pokew(0x4EEL, 1);

    if (Prtblk(&pb) != 0)
        Cconws("Text print failed.\r\n");
    return 0;
}
```

See Also
TOS, xbios, xbios.h

Pterm—**gemdos** function 76 (**osbind.h**)
Terminate a process
#include <osbind.h>
void Pterm(*status***) int** *status***;**

Pterm terminates the current process, and returns control to the parent process. *status* can be a status code that can be interpreted by the parent process. **Pterm** returns non-zero in the unlikely event that the process could not be terminated.

Example
This program exits with a non-zero status.

```
#include <osbind.h>

main() {
    Pterm(2);                   /* Exit with return code set to 2 */
}
```

See Also
gemdos, Pexec, Pterm, Ptermres, TOS

Pterm0—**gemdos** function 0 (**osbind.h**)

Terminate an TOS process

#include <osbind.h>
void Pterm0()

Pterm0 terminates a TOS process, and never returns.

Example
For an example of this function, see the entry for **Bconin**.

See Also
gemdos, Pterm, Ptermres, TOS

Ptermres—**gemdos** function 49 (**osbind.h**)

Terminate a process but keep it in memory

#include <osbind.h>
void Ptermres(*n, code***) long** *n*; **int** *code*;

Ptermres terminates a process in TOS, but retains *n* bytes of the process in memory. *code* is the exit code for the process being terminated; it is returned to the process that invoked the current process.

Example
For an example of this function, see the entry for **\auto**.

See Also
gemdos, Pexec, Pterm, Pterm0, TOS

Notes
Programs that use this macro may not be portable to future versions of TOS, but they are interesting to work with in the meantime.

pun—Definition

In the context of C, a **pun** occurs when a programmer uses one data form interchangeably with another. These puns are supported by the language's willingness to apply implicit conversion rules.

A pun most often occurs unintentionally when the programmer fails to declare a function as returning a pointer; by default, what the function returns is assumed to be an **int**, thus creating a pun between the **int** and pointer into which the function's return value is stored. No trouble will arise if the program is run on a machine that defines an **int** and a pointer to be the same length; however, such code cannot be transported to an environment in which this is not the case.

See Also
pointer, portability

Puntaes—xbios function 39 (**osbind.h**)
Disable AES
#include <osbind.h.h>
#include <xbios.h>
void **Puntaes()**

Puntaes disables the AES. Note that this function may not work if the AES is in ROM.

See Also
TOS, xbios

putc—STDIO macro (**libc.a/putc**)
Write character to stream
#include <stdio.h>
int putc(*c*, *fp*) char *c*; FILE **fp*;

putc is a macro that writes a character *c* onto file stream *fp*, and returns that character upon success.

Example
The following example demonstrates **putc**.

```
#include <stdio.h>
main(){
	FILE *fp;
	int ch;
	int filename[20];
	printf("Enter file name: ");
	gets(filename);

	if ((fp = fopen(filename,"r")) != NULL) {
		while ((ch = fgetc(fp)) != EOF)
			putc(ch, stdout);
	}
	else
		printf("Cannot open %s.\n", filename);
	fclose(fp);
}
```

See Also
fputc, putchar, STDIO
The C Programming Language, pages 152, 166

Diagnostics
EOF is returned when a write error occurs.

Notes
Because **putc** is a macro, arguments with side effects may not work as expected.

putchar—STDIO macro (**stdio.h**)
Write a character to standard output
#include <stdio.h>
int putchar(*c*) char *c*;

putchar is a macro that expands to **putc(*c*, stdout)**; it writes a character onto the standard output.

Example

```
#include <stdio.h>
main(){
	FILE *fp;
	int ch;
	int filename[20];
	printf("Enter file name: ");
	gets(filename);
	if ((fp = fopen(filename,"r")) != NULL) {
		while ((ch = fgetc(fp)) != EOF)
			putchar(ch);
	}
	else
		printf("Cannot open %s.\n", filename);
	fclose(fp);
}
```

See Also
fputc, putc, STDIO
The C Programming Language, pages 144, 152

Diagnostics
EOF is returned when a write error occurs.

Notes
Because **putchar** is a macro, arguments with side effects may not work as expected.

puts—STDIO function (**libc.a/puts**)
Write string to standard output
#include <stdio.h>
puts(*string*) char **string*

puts appends a newline character to the argument *string* and writes the result on the standard output.

See Also
fputs, STDIO

putw—STDIO macro (**stdio.h**)
Write word to stream
#include <stdio.h>
putw(*word*, *fp***) int** *word*; **FILE ****fp*;

The macro **putw** writes *word* (an **int**) to the stream *fp*, and returns the value written.

See Also
ferror, STDIO

Diagnostics
putw returns **EOF** when an error occurs. A call to **ferror** may be needed to distinguish this value from a valid data item.

Notes
Because **putw** is a macro, arguments with side effects may not work as expected. The bytes of *word* are written in the natural byte order of the machine.

pwd—Command
Print the name of the current directory
pwd

pwd prints the name of the current working directory.

See Also
cd, commands, msh

qsort—General function (**libc.a/qsort**)

Sort arrays in memory

qsort(*data*, *n*, *size*, *comp*) **char** **data*; **int** *n*, *size*; **int** (**comp*)();

qsort is a generalized algorithm for sorting arrays of data in primary memory. It uses C. A. R. Hoare's "quicksort" algorithm. **qsort** works with a sequential array of memory called *data*, which is divided into *n* parts of *size* bytes each. In practice, *data* is usually an array of pointers or structures, and *size* is the **sizeof** the pointer or structure. Each routine compares pairs of items and exchanges them as required. The user-supplied routine to which *comp* points performs the comparison. It is called repeatedly, as follows:

```
(*comp)(p1, p2)
char *p1, *p2;
```

Here, *p1* and *p2* each point to a block of *size* bytes in the *data* array. In practice, they are usually pointers to pointers or pointers to structures. The comparison routine must return a negative, zero, or positive result depending on whether *p1* is logically less than, equal to, or greater than *p2*, respectively.

Example

For an example of how to use this function, see the entry for **malloc**.

See Also

shellsort, strcmp, strncmp

The Art of Computer Programming, vol. 3

Notes

qsort uses a recursive algorithm that may overflow the default stack allocated; however, this is unlikely.

rand—General function (**libc.a/rand**)
Generate pseudo-random numbers
int rand()

rand generates linear, congruential, pseudo-random numbers. It returns integers in the range 0 to 2^15-1, and purportedly has a period of 2^32.

Example
This example tests the functions **rand** and **srand**. It uses a threshold level that is passed in **argv[1]** (default, MAXVAL/2), the number of trials passed in **argv[2]** (default, 1,000), and a seed passed in **argv[3]** (default, no seeding).

```
#define MAXVAL 32767            /* range of rand(): [0, 2^15-1] */
main(argc, argv) int argc; char *argv[]; {
                                register int i, hits, threshold, ntrials;

        hits = 0;
        threshold = (argc > 1) ? atoi(argv[1]) : MAXVAL/2;
        ntrials = (argc > 2) ? atoi(argv[2]) : 1000;
        if (argc > 3)
                srand(atoi(argv[3]));

        for (i = 1; i <= ntrials; i++)
                if (rand() > threshold)
                        ++hits;

        printf("%d values above %d in %d trials (%D%%).\n",
                hits, threshold, ntrials, (100L*hits)/ntrials);
}
```

See Also
srand
The Art of Computer Programming, vol. 2

Random—xbios function 17 (**osbind.h**)
Generate a 24-bit pseudo-random number
#include <osbind.h>
#include <xbios.h>
long Random()

Random generates and returns a 24-bit pseudo-random number. The generator is seeded from the frame-counter, and is likely to be different every time the computer is turned on.

Example
The following example generates an array of random numbers. You may wish to use this as input for the example in **malloc**, which demonstrates sorting.

```
#include <osbind.h>
main() {
	int i;
	for (i=100;i>0;i--) {
		printf("%8ld ", Random());
		if ( i%4 == 0 )
			printf( "\n" );
	}
}
```

See Also
TOS, xbios
The Art of Computer Programming, vol. 2

Notes
The lowest bit has a distribution of exactly 50%.

random access—Definition

In the context of computing, **random access** means that an entity, such as memory, can be accessed at any point, not just at the beginning. This means that all points within memory can be accessed equally quickly. This contrasts with *sequential access*, in which entities must be accessed in a particular order, so that some entities take longer to access than do others.

A tape drive is an example of a sequential access device, i.e., the order in which they stream past the tape head. Random-access memory (RAM) demonstrates random access. Hard disks and floppy disks are combine elements of random access and sequential access.

RAM, which usually consists of semiconductor integrated circuits, is also strictly random access. In this regard, the term "RAM" is slightly misleading, and should be called "RWM", for read/write memory, contrasting it with read-only memory (ROM), which is also *random access* memory.

See Also
read-only memory

ranlib—Definition

The **ranlib** is a "directory" that appears at the beginning of each library. It contains the name of each global symbol (i.e., function name) that appears within the library, and a pointer to the module in which that symbol is defined. Thus, the ranlib eliminates the need for the linker to search the entire library sequentially to find a given global symbol, which speeds up linking noticeably.

If the date on the library file is later than that in the ranlib header, the linker will ignore the ranlib and perform a sequential search through the library; the linker will also send the warning message

```
Outdated ranlib
```

to the standard error device. This is done to prevent the accidental use of an outdated ranlib, which could be disastrous. When you use the archiver **ar** to update a library or to create a new library, be sure to employ the options that update the ranlib as well as modify or create the library.

See Also
ar, date, ld, touch

Notes
Under certain circumstances, it was possible to generate the **Outdated ranlib** error message even though the ranlib was in fact up to date. In previous releases of Mark Williams C, this occurred when it was installed on a system with the date set to the current date, rather than not set, as requested in the installation procedures. Installing Mark Williams C with the date set on the system had the effect of updating the date stamp on the library files, which put the date on the ranlib header and that of its library file out of synch. The linker thus thought that the ranlib was outdated, when it was in fact correct. This problem was fixed on a previous release.

rational number—Definition
A **rational number** is the quotient of two integers.

See Also
integer, real number

rc_copy—AES function (**libaes.a/rc_copy**)
Copy a rectangle
#include <aesbind.h>
int rc_copy(*oldrect, newrect***) Rect** **oldrect, *newrect;*

rc_copy is an AES routine that copies a rectangle from one part of the screen to another. *oldrect* and *newrect* point, respectively, to the rectangle being copied and the area to which it is being copied. Each is defined as pointing to a structure of the type **Rect**, which is defined in the header file **aesbind.h**, as follows:

x	X coordinate of rectangle
y	Y coordinate of rectangle
w	width of rectangle
h	height of rectangle

rc_copy returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, TOS

Notes
A clipping rectangle should be set using the VDI function **vs_clip** before this routine is used.

rc_equal—AES function (**libaes.a/rc_equal**)
Compare two rectangles
#include <aesbind.h>
int rc_equal(*rect1, rect2***) Rect ****rect1*, ******rect2*;

rc_equal is an AES routine that compares two rectangles. *rect1* and *rect2* point to the two rectangles being compared. Each is declared as pointing to a structure of the type **Rect**, which is defined in the header file **aesbind.h**, as follows:

x	X coordinate of rectangle
y	Y coordinate of rectangle
w	width of rectangle
h	height of rectangle

rc_equal returns zero if the rectangles are not identical, and one if they are.

See Also
AES, TOS

rc_intersect—AES function (**libaes.a/rc_intersect**)
Check if two rectangles intersect
#include <aesbind.h>
int rc_intersect(*rect1, rect2***) Rect ****rect1*, ******rect2*;

rc_intersect is an AES routine that check to see if two rectangles intersect. *rect1* and *rect2* point to the two rectangles being compared. Each is declared as pointing to a structure of the type **Rect**, which is defined in the header file **aesbind.h**, as follows:

x	X coordinate of rectangle
y	Y coordinate of rectangle
w	width of rectangle
h	height of rectangle

The values of the structure to which *rect2* points will be changed to the coordinates of the area common to both rectangles, or to nonsense if they do not intersect. **rc_intersect** returns zero if the rectangles do not intersect, and one if they do.

See Also
AES, TOS

rc_union—AES function (**libaes.a/rc_union**)
Calculate overlap between two rectangles
#include <aesbind.h>
void rc_union(*rect1, rect2***) Rect ***rect1*, ***rect2***;**

rc_union is an AES routine that computes a rectangle that encloses two overlapping rectangles. *rect1* and *rect2* point to the two overlapping rectangles. Each is declared as pointing to a structure of the type **Rect**, which is defined in the header file **aesbind.h**, as follows:

x	X coordinate of rectangle
y	Y coordinate of rectangle
w	width of rectangle
h	height of rectangle

The values of the structure to which *rect2* points will be changed to the coordinates of the rectangle that encloses the overlapping rectangles; these variables are set to nonsense if the rectangles do not intersect. **rc_union** returns nothing.

See Also
AES, TOS

Notes
This routine should be used only if you are certain that the rectangles in question do overlap. The routine **rc_intersect** returns a value that indicates if the rectangles do in fact overlap.

read—UNIX system call (**libc.a/read**)
Read from a file
read(*fd*, *buffer*, *n***) int** *fd*; **char ***buffer*; **int** *n*;

read reads up to *n* bytes of data from the file descriptor *fd* and places them into the data segment at address *buffer*. The amount of data actually read may be less than that requested if EOF is encountered. The data are read at the current seek position in the file, which was set by the most recently executed **read** or **lseek** routine. **read** advances the seek pointer by the number of characters actually read.

Example
For an example of how to use this function, see the entry for **open**.

See Also
UNIX routines, STDIO

Diagnostics
With a successful call, **read** returns the number of bytes read; thus, zero bytes signals the end of the file. It returns -1 if an error occurs: bad file descriptor, bad *buffer* address, and physical read error are among the possibilities.

Notes
read is a low-level call that passes data directly to TOS. It should not be intermixed with high-level calls, such as **fread**, **fwrite**, or **fopen**, without caution.

read-only memory—Definition

As its name suggests, **read-only memory**, or ROM, is memory that can be read but not written to. It most often is used to store material that is used frequently or in key situations, such as a language interpreter or a boot routine.

See Also
random access

real numbers—Definition

A **real number** is any number of the set of rational numbers or irrational numbers.

See Also
float, rational number, integer, irrational number

realloc—General function (**libc.a/realloc**)

Reallocate dynamic memory
char *realloc(*ptr*, *size***) char ****ptr*; **unsigned** *size*;

realloc helps to manage a program's arena. It returns a block of *size* bytes that holds the contents of the old block, up to the smaller of the old and new sizes. **realloc** tries to return the same block, truncated or extended; if *size* is smaller than the size of the old block, **realloc** will return the same *ptr*.

See Also
arena, calloc, free, lcalloc, lmalloc, lrealloc, malloc, notmem, setbuf

Diagnostics
realloc returns **NULL** if insufficient memory is available. It prints a message and calls **abort** if it discovers that the arena has been corrupted, which most often occurs by storing past the bounds of an allocated block. **realloc** will behave unpredictably if handed an incorrect *ptr*.

The related function **lrealloc** takes an unsigned long as its *size* argument, and therefore can reallocate memory blocks that are larger than 64 kilobytes.

record—Definition

A **record** is a set of data of a fixed length that has been given a unique identifier, and whose structure conforms to an exact description. An example of a record is an entry in an entry in a file of names and addresses: each entry has a fixed length, is marked by a unique identifier, and has a fixed number of bytes set aside in fixed order to record name, address, city, state, and ZIP code.

Note, too, that what is called a "record" in Pascal is called a "structure" in C.

See Also
field, structure

Rect—Definition

Rect is a structure that is defined in the header file **aesbind.h**. It defines a rectangle in a manner that can be understood by an AES routine. It consists of four integers, as follows:

x	X coordinate of rectangle
y	Y coordinate of rectangle
w	width of rectangle
h	height of rectangle

Because Mark Williams C allows you to pass a structure directly, this structure can be placed in the argument list of AES functions to replace the four arguments that indicate coordinates, height, and width of a rectangle.

See Also
AES, aesbind.h, struct, TOS

register—Definition

A **register** is memory within a microprocessor within which data can be stored and modified. The size and the configuration of a microprocessor's registers affect its computing potential. Registers can be manipulated much faster than RAM.

See Also
register variable

register variable—Definition

register is a C storage class. A **register** declaration tells the compiler to try to keep the defined local data item in a machine register. Under Mark Williams C, the **int foo** can be declared to be a register variable with the following statement:

```
register int foo;
```

On the i8086, two registers are available to accept register variables; if more than two are declared, all after the first two will be treated as ordinary **autos**. On the 68000, eight registers are available to accept register variables: three address registers and five data registers.

By definition of the C language, registers have no addresses, so pointers to registers cannot be passed as function arguments. Placing heavily-used local variables into registers often improves performance, but in some cases declaring **register** variables can degrade performance somewhat.

See Also
auto, extern, static, storage class
The C Programming Language, page 81

rewind—STDIO function (**libc.a/rewind**)
Reset file pointer
#include <stdio.h>
rewind(*fp***) FILE ****fp***;**

rewind resets the file pointer to the beginning of stream *fp*; it is a synonym for **fseek(***fp***, 0L, 0)**.

See Also
fseek, STDIO

rindex—String function (**libc.a/rindex**)
Find a character in a string
char *rindex(*string***,** *c***) char ***string***; char** *c***;**

rindex scans *string* for the last occurrence of character *c*. If *c* is found, **rindex** returns a pointer to it. If it is not found, **rindex** returns NULL.

Example
This example, when handed a path name, returns a pointer to a file name with the leading directory information stripped away.

```
#define PATHSEP '\\'             /* path name separator */
extern char *rindex();

basename(path) register char *path;
{
        register char *cp;
        return ((cp = rindex(path, PATHSEP)) == NULL) ? path : ++cp);
}
```

See Also
index, string

rm—Command
Remove files
rm *file ...*

rm removes each *file*, and frees data blocks associated with it.

See Also
commands, msh, rmdir

rmdir—Command
Remove a directory
rmdir *directory ...*

rmdir removes each *directory*. This will not be allowed if a *directory* is the current working directory or is not empty.

rmdir will not allow you to remove the current working directory.

See Also
commands, mkdir, msh, rm

rsconf—Command
Reconfigure the serial port
rsconf *speed, flow, UCR, RSR, TSR, SCR*

rsconf is a command that uses the **xbios** function **Rsconf** to reconfigure the serial port. *speed* is the baud rate to which the port will be set; *flow* sets the form of flow control. *UCR* is a bit map that sets the control register; *RSR* is a bit map that sets the receive status; *TSR* is a bit map that sets the transmission status; and *SCR* sets the synchronous character register. For details on the values for these arguments, see the entry for **Rsconf**.

See Also
commands, Rsconf, TOS

Rsconf—xbios function 15 (**osbind.h**)
Configure the serial port
#include <osbind.h>
#include <xbios.h>
void Rsconf(*speed, flow, UCR, RSR, TSR, SCR***)**
int *speed, flow, UCR, RSR, TSR, SCR*;

Rsconf configures the serial port, and returns nothing. *speed* is an integer that sets the baud, as follows:

0	19,200	8	600
1	9600	9	300
2	4800	10	200
3	3600	11	150
4	2400	12	134
5	2000	13	110
6	1800	14	75
7	1200	15	50

flow is an integer that sets the flow control, as follows:

0	None (the default)
1	XON/XOFF (**<ctrl-S>**/**<ctrl-Q>**)
2	Request to send/clear to send (RTS/CTS)
3	XON/XOFF and RTS/CTS

UCR stands for USART control register. (USART, in turn, means universal synchronous-asynchronous receiver-transmitter). This variable is an byte-length bit map that controls the operation of the serial port. Its bits encode the following information:

Bit 0	unused
Bit 1	0 indicates odd parity; 1, even parity
Bit 2	0 indicates no parity; 1, parity as set in bit 1
Bits 3,4	Start/stop bits and format: 00 synchronous; start=0; stop=0 10 asynchronous; start=1; stop=1 01 asynchronous; start=1; stop=1.5 11 asynchronous; start=1; stop=2
Bits 5,6	Word length: 00 8 bits 10 7 bits 01 6 bits 11 5 bits
Bit 7	0=Use frequency from transmit control and receive control directly 1=Divide frequency by 16

RSR is a byte-length bit map that controls the receive status register; setting the bits sets the following conditions:

Bit 0	Enable reception
Bit 1	In synchronous mode, enable comparison of character in SCR with character in receive buffer
Bit 2	In synchronous mode, signal that character identical to character in SCR may be received; in asynchronous mode, signal reception of start bit
Bit 3	In synchronous mode, signal that character identical to character in SCR has been received; in asynchronous mode, signal reception of BREAK
Bit 4	Signal frame error: stop bit is a NUL, but byte received is not
Bit 5	Signal parity error
Bit 6	Signal buffer overrun
Bit 7	Signal buffer full

TSR is a byte-length bit map that controls the transmitter status register. The bits in this map indicate the following:

Bit 1	Enable transmission
Bits 2,3	High or low output mode: 00 High 10 High 01 Low 11 Loop-back mode
Bit 3	In synchronous mode, not used; in asynchronous, sends break condition
Bit 4	Send end-of-transmission character after current character
Bit 5	Switch to reception immediately after end of transmission
Bit 6	Send character in sender floating register before writing new character into send buffer
Bit 7	Buffer empty

Finally, *SCR* initializes the synchronous character register; this variable should be set to zero.

Note that setting *UCR*, *RSR*, *TSR*, or *SCR* to -1 will cause it to be ignored by TOS.

Example
This example sets the serial port to 4800 baud with XON/XOFF flow control. For an example of using this function from the \auto directory, see the entry for \auto.

```
#include <osbind.h>

#define BR_4800 (2)                    /* 4800 baud */
#define FC_XON (1)                     /* XON/XOFF */

main() {
        Rsconf(BR_4800, FC_XON, -1, -1, -1, -1);
        Cconws("Serial port set to 4800 baud, XON/XOFF\n\r");
}
```

See Also
TOS, xbios

Notes
Resetting the speed, even if there is no change, will transmit an ASCII DEL across the serial line. This may be intended to help remote systems or modems to determine line speed.

rsrc_free—AES function (libaes.a/rsrc_free)

Free memory allocated to a set of resources
#include <aesbind.h>
int rsrc_free()

rsrc_free is an AES routine that frees the random-access memory that had been allocated to a set of resources by the routine **rsrc_load**. Because the contents of only one resource file can be kept in memory at any given time, this routine should be employed before loading a second resource file. **rsrc_load** returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, TOS

rsrc_gaddr—AES function (libaes.a/rsrc_gaddr)

Get the address of a resource object
#include <aesbind.h>
**int rsrc_gaddr(*type, index, address*) int *type, index*; char ****address*;

rsrc_gaddr is an AES routine that gets the address of a given resource object. *type* indicates the type of object being sought, as follows:

0	object tree
1	object within a tree
2	text (**TEDINFO**)
3	icon (**ICONBLK**)
4	predefined bit pattern (**BITBLK**)
5	string
6	image data
7	object specification
8	pointer to text (**TEDINFO**)
9	pointer to text template (**TEDINFO**)
10	pointer to text validation string (**TEDINFO**)
11	pointer to mask for icon image (**ICONBLK**)
12	pointer to data for icon image (**ICONBLK**)
13	pointer to icon text (**ICONBLK**)
14	pointer to bit image (**BITBLK**)
15	address of pointer to free string
16	address of pointer to free image

index gives the index number of the object within the object tree. *address* points to the address of the data sought; this value is set by the routine. **rsrc_gaddr** returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, TOS

rsrc_load—AES function (**libaes.a/rsrc_load**)
Load a resource file into memory
#include <aesbind.h>
int rsrc_load(*filename***) char ***filename*;

rsrc_load is an AES routine that loads a resource file into memory. *filename* points to the name of the file to be loaded. Note that by convention, the name of the file must have the suffix **.rsc**.

Note that only one resource file can be loaded into memory at any given time; **rsrc_load** automatically calls **rsrc_free** to free the memory allocated to any previously loaded resource file. **rsrc_load** returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, TOS

rsrc_obfix—AES function (**libaes.a/rsrc_obfix**)
Change the form of an object's coördinates
#include <aesbind.h>
#include <obdefs.h>
int rsrc_obfix(*tree, object***) char ****tree***; int** *object*;

rsrc_obfix is an AES routine that changes the form the coordinates for an object that is stored in a resource file. A resource file encodes an object's coordinates in the form of character coordinates, not pixel coordinates; these character coordinates are transformed into pixel coordinates when the resource file is loaded, when the resolution of the screen is known. *tree* points to the address of the tree that contains the object in question, and *object* is the number of the object within the tree. **rsrc_obfix** always returns one.

See Also
AES, TOS

rsrc_saddr—AES function (**libaes.a/rsrc_saddr**)
Store address of a free string or a bit image
#include <aesbind.h>
int rsrc_saddr(*type, index, address***) int** *type, index*; **char ****address*;

rsrc_saddr is an AES routine that copies into an object the address of a pointer to either the free string or the free image of another object within the object tree. *type* denotes the type of pointer whose address is being stored: 15 indicates a pointer to a free string, and 16 indicates a pointer to a bit image. **rsrc_saddr** returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, TOS

runtime startup—Definition
The C runtime startup is an initialization routine that is linked with a C program as the first part of an executable object program. It performs the functions necessary to start and terminate the C environment. At a minimum, it initializes the stack, calls **main**, and calls **exit** with the return value from **main**.

Three C runtime startup routines are available on Mark Williams C for the Atari ST: **crts0.o**, the normal runtime startup; **crtsg.o**, the runtime startup for the GEM environment; and **crtsd.o**, which is used to create a GEM desktop application. The default is **crts0.o**, which is appropriate for most uses. You can call **crtsg.o** on the **cc** command line in either of two ways: with the switch **-VGEM**, or with the **name** option **Nrcrtsg.o**. The **crtsd.o** start-up routine can be called with the option **-VGEMACC** or with the **name** option **Ncrtsd.o**.

See Also
cc, crts0.o, crtsd.o, crtsg.o, stack, _stksize

rvalue—Definition

An **rvalue** is the value of an expression. The name comes from the assignment expression **e1=e2;**, in which the right operand is an rvalue.

See Also
lvalue

Rwabs—bios function 4 (**osbind.h**)

Read or write data on a disk drive
#include <osbind.h>
#include <bios.h>
long Rwabs(*rorw, buffer, n, rec, drive***)**
int *rorw, n, rec, drive*; **char** **buffer*;

Rwabs reads from or writes data to a disk drive. *rorw* indicates whether the process will read or write; zero indicates read, and one indicates write. *n* is the number of sectors to transfer; *rec* is the number of the first record to transfer; and *drive* is the name of the disk drive to use: zero indicates drive A, one indicates drive B, etc. *buffer* points to the area to which the data are to be written, or from which they are to be read.

See Also
bios, TOS

scanf—STDIO function (**libc.a/scanf**)

Format input

#include <stdio.h>

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

scanf reads the standard input, and uses the string *format* to specify a format for each *arg*, each of which must be a pointer. **scanf** reads one character at a time from *format*; white space characters are ignored. The percent sign character '%' marks the beginning of a conversion specification. '%' may be followed by characters that indicate the width of the input field and the type of conversion to be done. The following modifiers, in this order, may precede the conversion type:

1. The asterisk '*', which indicates that the next input field should be skipped rather than assigned to the next *arg*.

2. A string of decimal digits, which specifies a maximum field width.

3. An **l**, which specifies that the next input item is a **long** object rather than an **int** object. Capitalizing the conversion character has the same effect.

The following conversion characters are recognized:

c Assign the next input character to the next *arg*, which should be of type **char** *.

d Assign the decimal integer from the next input field to the next *arg*, which should be of type **int** *.

D Assign the decimal integer from the next input field to the next *arg*, which should be of type **long** *.

e Assign the floating point number from the next input field to the next *arg*, which should be of type **float** *.

E Assign the floating point number from the next input field to the next *arg*, which should be of type **double** *.

f Same as **e**.

F Same as **E**.

o Assign the octal integer from the next input field to the next *arg*, which should be of type **int** *.

O Assign the octal integer from the next input field to the next *arg*, which should be of type **long** *.

s Assign the string from the next input field to the next *arg*, which should be of type **char** **. The array to which the **char** ** points should be long enough to accept the string and a terminating NUL character.

x Assign the hexadecimal integer from the next input field to the next *arg*, which should be of type **int** *.

X Assign the hexadecimal integer from the next input field to the next *arg*, which should be of type **long** *.

See Also
STDIO
The C Programming Language, page 147

Notes
Because C does not perform type checking, it is essential that an argument match its specification; for that reason, **scanf** is best used to process only data that you are certain is in the correct data format. The use of upper-case format characters to specify long arguments is not standard; use the 'l' modifier for portability.

Scrdmp—**xbios** function 20 (**osbind.h**)
Print a dump of the screen
#include <osbind.h>
#include <xbios.h>
void Scrdmp()

Scrdmp dumps the screen to the printer port, and returns nothing. Note that at present this routine works only with the monochrome monitor.

Example
This example dumps the screen to a printer. Be sure that before you use this example, your printer is plugged into your computer, properly described to TOS, and turned on.

```
#include <osbind.h>
#include <bios.h>

main() {
	if(Bcostat(BC_PRT) == 0 )
		Cconws( "The printer is not ready.\n\r" );

	else {
		Cconws( "The screen is being printed... Please wait.\n\r" );
		Scrdmp();
		Cconws( "The screen is printed.\n\r" );
	}
	return(0);
}
```

See Also
TOS, xbios

screen control—TOS data

The Atari ST uses the following escape sequences to control the terminal screen. These can be passed by the macro **Cconout**, as well as by numerous other output routines, to manipulate the Atari ST's screen:

Note that <esc> represents the escape character, ASCII 033.

<esc>A	Cursor up
<esc>B	Cursor down
<esc>C	Cursor forward
<esc>D	Cursor backward
<esc>E	Clear screen, home cursor
<esc>H	Home cursor
<esc>I	Return to same position on previous line
<esc>J	Erase to the end of the page
<esc>K	Clear to the end of the line
<esc>L	Insert line
<esc>M	Delete line
<esc>Y *row col*	Position cursor at *row*, *col*, which are row/column numbers plus 040 (space character)
<esc>b*c*	Set foreground color to *c*
<esc>c*c*	Set background color to *c*
<esc>d	Erase beginning of display
<esc>e	Make cursor visible
<esc>f	Make cursor invisible
<esc>j	Save cursor position
<esc>k	Restore cursor position
<esc>l	Erase a line
<esc>o	Erase from beginning of line to cursor
<esc>p	Enter reverse video mode
<esc>q	Exit reverse video mode
<esc>v	Wrap text at end of line
<esc>w	Discard text at end of line

For the sequences <esc>b and <esc>c, the variable *c* is the color index plus 040. In monochrome mode, the color index can be zero or one; in medium resolution, it can be zero through three; and in low resolution, it can be one through 15.

Example

The following example clears the screen and homes the cursor, then moves the cursor to row 12, column 6 on the screen.

```
main() {
        char row = 12+'\040';
        char column = 6+'\040';

        printf("\033E");
        printf("\033Y%c%c", row, column);
}
```

See Also
Cconout, gemdos, TOS

scrp_read—AES function (**libaes.a/scrp_read**)
Read the scrap directory
#include <aesbind.h>
int scrp_read(*buffer*) char **buffer*;

The "scrap" feature provides a way for applications to pass information among themselves.

The information to be passed is written into a file, which is always called **scrap.***xxx*. The suffix indicates what type of information the file contains: text (.txt), a GEM metafile (**.gem**), a bit image (**.img**), or spreadsheet data (**.dif**).

The name of the directory that holds the scrap file is written into a static buffer, or *clipboard*. The clipboard contains only the name of the directory in which the information is kept, not the information itself. The clipboard is overwritten each time it is used, so in effect only one scrap file can be used at any given time. AES provides routines for reading and writing to the clipboard; it is up to you to see to it that the scrap file is correctly written and read.

scrp_read is an AES routine that reads the clipboard. *buffer* points to the name of a buffer into which the contents of the clipboard will be written. **scrp_read** returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, TOS

scrp_write—AES function (**libaes.a/scrp_write**)
Write to the scrap directory
#include <aesbind.h>
int scrp_write(*directory*) char **directory*;

scrp_write is an AES routine that writes the name of the scrap directory onto the clipboard. *directory* is the name of the scrap directory. **scrp_write** returns zero if an error occurred, and a number greater than zero if one did not. For more information on using the clipboard, see the entry for **scrp_read**.

See Also
AES, scrp_read, TOS

set—Command
Set an msh variable
set [*VARIABLE=value*]

set sets sets the **msh** *VARIABLE* to *value*. For example, the command

```
set b="b:\bin"
```

tells **msh** that the variable **b** is equivalent to **b:\bin**; thus, typing

```
cd $b
```

is equivalent to typing

```
cd b:\bin
```

Typing **set** without an argument displays all the variables that have been set. Typing

```
set in history
```

lists the contents of the shell's history buffer. Typing

```
set in .bin
```

lists the installed built-in functions; **.bin** is **msh**'s internal directory, which points to areas in absolute memory where commands are stored.

Additional forms of the built-in functions can be installed into **.bin** with the **set** command. For example, the command

```
set in .bin off="cursconf 3"
```

installs the command **off** into **.bin**, and declares it to be equivalent to the command **cursconf 3**. **cursconf** is a command that is built into the micro-shell, and uses the TOS function **Cursconf** to manipulate the system cursor. This command turns off the cursor blink.

See Also
commands, msh, unset

setbuf—STDIO function (**libc.a/setbuf**)
Set alternative stream buffers
#include <stdio.h>
setbuf(*fp, buffer***) FILE ***fp*; **char ***buffer*;

The standard I/O library STDIO automatically buffers all data read and written in streams, with the exception of streams to terminal devices. STDIO normally uses **malloc** to allocate the buffer, which is a **char** array **BUFSIZ** characters long; **BUFSIZ** is defined in the header file **stdio.h**. **setbuf's** arguments are the stream pointer *fp* and a *buffer* to be associated with the stream. The call should be issued after the stream has been opened, but before any input or output request has been issued. The *buffer* passed to **setbuf** may be **NULL**, in which case the stream will be unbuffered, or must contain at least **BUFSIZ** bytes.

See Also
STDIO

setcol—Command
Reset a color
setcol *color*, *value*

setcol is a command that uses the **xbios** function **Setcolor** to reset a color. *color* is the entry in the color palette that you wish to reset, from zero through 15. *value* is a three-digit octal number that indicates the color to which you wish to set *color*.

See Also
commands, getcol, TOS

Setcolor—xbios function 7 (**osbind.h**)
Set one color
#include <osbind.h>
#include <xbios.h>
int Setcolor(*number, value*) **int** *number, value*;

Setcolor sets one color. *number* is the element on the color palette that is being redefined; it can be any number from zero to 15. *value* is the color value to which *number* is being reset; setting any *number* to a negative value ensures that no change is made.

On monochrome monitors,

```
Setcolor(0, 0);
```

gives a black background and white letters, whereas

```
Setcolor(0, 1);
```

switches the screen to a white background and black letters.

Setcolor returns the old value of *number*. The change will be made during the next vertical blank.

Examples
The first example reads and prints out the values of the color map. For another example, see the entry for **Setcolor**.

```
#include <osbind.h>

color_disp(indx, val)
int indx;
int val;
{
	int red, green, blue;

	red = (val>>8) & 7;		/* Red value in bits 8-10 */
	green = (val>>4) & 7;		/* Green value in bits 4-6 */
	blue = val & 7;			/* Blue value in bits 0-2 */
	printf( " %2d  : %1d %1d %1d\n", indx, red, green, blue );
}

main() {
	int i;
	printf( "Entry  R G B\n" );
	for ( i=0; i<16 ; i++ )
	color_disp( i, Setcolor( i, -1 ));
}
```

The second example works with a monochromatic monitor. It reverses the colors of the characters and background.

```
#include <osbind.h>
main() {
	int color = Setcolor(0, -1);
	Setcolor(0, ++color%2);
}
```

See Also
TOS, **xbios**

setenv—Command
Set an environmental variable
setenv [*VARIABLE=value*]

setenv sets an environmental variable. Environmental variables are those that are *exported*, or handed to other programs for their use at run time. For example, the environmental variable **TIMEZONE** is read by the C routine **ctime** as part of its time-handling work; whereas the environmental variable **LIBPATH** is read by the linker **ld** to locate its libraries.

You are free to define new environmental variables within your programs, and use **setenv** to define them on your system. Note that it is traditional to spell environmental variable with capital letters.

Typing **setenv** without any arguments displays all of the environmental variables that have been set so far.

See Also
commands, msh, unsetenv

Setexc—bios function 5 (osbind.h)
Get or set an exception vector
#include <osbind.h>
#include <bios.h>
long Setexc(*number, address***) int** *number*; **char** **address*;

Setexc gets or sets an exception vector. Vectors 0x00 through 0xFF are defined by the 68000 hardware; the extended vectors are defined in the header file **signal.h**, as follows:

0x100	timer tick
0x101	critical error handler
0x102	terminate handler
0x103-0x1FF	reserved for future use by TOS
0x200-0x2FF	reserved for future use by users

number is the number of the exception vector to be read or set. *address* is the address to be set into the exception table; -1 indicates that the vector is to be read rather than set. **Setexc** returns either the previous address if it is setting the vector, or the current address if is reading the vector.

Example
This example shows how to use **Setexc** to trap divide-by-zero errors. Note that this program calls the routine **setrte**, which is included with Mark Williams C in the file **setrte.s**. To compile, use the command line

```
cc -o Setexc.prg Setexc.c setrte.s
```

The following gives the text of **Setexc.c**:

```
#include <osbind.h>
#define DIV0 (5)                /* Divide by 0 vector number */

diverr() {
        setrte();               /* Make this an exception routine */

        Cconws("\r\nDivision by 0\r\n");
}
```

```
main() {
	register unsigned long oldvec;
	int a = 0;
	int b;

	oldvec = (unsigned long)Setexc(DIV0, diverr);
					/* Set the exception */
	printf("This is a test of divide by 0...\n");
	b = 133/a;			/* Generate error */
	printf("The result of 133/%d is %d\n", a, b);
	Setexc(DIV0, oldvec);		/* Set vector back */
	exit(0);			/* Return to system */
}
```

See Also
bios, signal.h, TOS

Notes
TOS does not reset exception vectors on process termination; therefore, you must reset them yourself or face the consequences.

setjmp—General function (**libc.a/setjmp**)
Perform non-local goto
#include <setjmp.h>
setjmp(*env*) jmp_buf *env*;

The function call is the only mechanism that C provides to transfer control between functions. This mechanism is inadequate for some purposes, such as handling unexpected errors or interrupts at lower levels of a program. To answer this need, **setjmp** helps to provide a non-local *goto* facility. **setjmp** saves a stack context in *env*, and returns value zero. The stack context can be restored with the function **longjmp**. The type declaration for **jmp_buf** is in the header file **setjmp.h**. The context saved includes the program counter, stack pointer, and stack frame. This routine does not restore register variables, but other variables are not affected.

See Also
getenv, longjmp, setjmp.h

Notes
Programmers should note that many user-level routines cannot be interrupted and reentered safely. For that reason, improper use of **setjmp** and **longjmp** will result in the creation of mysterious and irreproducible bugs. The use of **longjmp** to exit interrupt exception or signal handlers is particularly hazardous.

setjmp.h—Header file
Header file for **setjmp** and **longjmp** functions
#include <setjmp.h>

setjmp.h defines the structure **jmp_buf** for a **setjmp** environment.

See Also
header file, longjmp, setjmp

setpal—Command
Reset the color palette
setpal

setpal is a command that uses the **xbios** function **Setpallete** (*sic*) to reset the system's color palette.

See Also
commands, getpal, TOS

Setpallete—**xbios** function 6 (**osbind.h**)
Set the screen's color palette
#include <osbind.h>
#include <xbios.h>
void Setpallete(*palette*) int *palette*[16];

Setpallete (*sic*) sets the screen's color palette, and returns nothing. *palette* points to an array of 16 hexadecimal integers, each of which indicates a different color. The palette is implemented at the next vertical blank interval.

Example
This example sets the color palette. A palette is a table of 16 words containing the definitions for 16 colors as indexed by set bits in the "planes".

```
#include <osbind.h>

short ugly[] = {
	0x000, 0x111, 0x222, 0x333,
	0x444, 0x555, 0x666, 0x777,
	0x007, 0x070, 0x700, 0x707,
	0x770, 0x077, 0x737, 0x337
	};

main() {
	Setpallete( ugly );
}
```

See Also
TOS, xbios

setphys—Command
Reset physical screen's display space
setphys *address*

setphys is a command that resets the physical screen's display base. *address* is the address of the new display base.

See Also
commands, getphys, TOS

setprt—Command
Reset the printer port
setprt *configuration*

setprt is a command that uses the **xbios** function **Setprt** to reconfigure the printer port. *configuration* is an integer that indicates the port's new configuration. For a table of the configuration codes, see the entry for **Setprt**.

See Also
commands, Setprt, TOS

Setprt—**xbios** function 33 (**osbind.h**)
Get or set the printer's configuration
#include <osbind.h>
#include <xbios.h>
int Setprt(*configuration***) int** *configuration*;

Setprt gets or sets the configuration of the printer port. *configuration* is a 16-bit map that configures the port. If it is set to 0xFFFF (-1), the port's current configuration is read; otherwise, its value is used to set the port, as follows:

0x01	daisywheel printer
0x02	monochrome printer
0x04	if set, Epson-type dot-matrix printer; if not, Atari printer
0x08	if set, final mode; if not, draft mode
0x10	if set, printer uses serial port; if not, printer port
0x20	if set, uses single sheets; if not, uses fanfold paper

Bits 6 through 14 are reserved, and bit 15 must be zero. These values are defined in the header file **xbios.h**.

Setprt returns the printer port's current configuration when *configuration* is set to -1; otherwise, it returns a meaningless value.

Example
For examples of this function, see the entries for **\auto** and **prtblk**.

See Also
Prtblk, TOS, xbios, xbios.h

setrez—Command
Reset the screen resolution
setrez *resolution*

setrez is a command that resets the screen's resolution. *resolution* indicates the new screen resolution, as follows: zero, high resolution; one, medium resolution; and two, low resolution. Note that using this command inappropriately (e.g., resetting a monochromatic monitor to low resolution) will cause a meaningless jumble to appear on the screen.

See Also
commands, getrez, Getrez, TOS

Setscreen—xbios function 5 (**osbind.h**)
Set the video parameters
#include <osbind.h>
#include <xbios.h>
void Setscreen(*log, phys, res***) char ****log*, **phys*; **int** *res*;

Setscreen sets the video parameters, and returns nothing. *log* and *phys* are the bases of the logical and physical screen displays. *res* is the new screen resolution:

0	high resolution
1	medium resolution
2	low resolution

Setting any variable to a negative number ensures that that variable will be ignored.

Example
This example demonstrates **Setscreen**.

```
#include <osbind.h>
#include <bios.h>

main() {
	char *newscr, *oldscr, *memblk;
	int x, y;
	Cconws("Working...\n");
	oldscr = (char *) Physbase();

	if((memblk = (char *)Malloc(32*1024L)) == 0) {
		printf("Malloc of %ld bytes failed.\n", 32*1024L);
		Pterm(1);
	}
```

```
        newscr = (char *) (((long) memblk + 0xFFL) & ~(0xFFL));
        Setscreen(newscr,-1L,-1);           /* Change logical base */
        Cconws("\033H\033J");               /* Clear logical screen */

        for (y=0; y<24; y++) {               /* for 20 rows... */
                for (x=0; x<39; x++) {      /*  39 times each... */
                        Bconout(BC_RAW, 0x0E);
                        Bconout(BC_RAW, 0x0F);
                }
                Cconws("\r\n");
        }

        Setscreen(-1L,newscr,-1);           /* Move physical base... */
        Cconin();
        Setscreen(oldscr,oldscr,-1);        /* Restore addresses... */
        return 0;
}
```

See Also
Getrez, Logbase, Physbase, TOS, xbios

Settime—xbios function 22 (osbind.h)
Set the current time
#include <osbind.h>
#include <xbios.h>
void Settime(*datetime*) long *datetime*;

Settime sets the current time and date for the intelligent keyboard (IKBD), and returns nothing. *datetime* is a 32-bit mask whose bits indicate the following:

0-4	no. of two-second increments (0-29)
5-8	no. of minutes (0-59)
9-15	no. of hours (0-23)
16-20	day of the month (1-31)
21-26	month (1-12)
27-31	year (0-119, 0 indicates 1980)

Example
This examples sets the IKBD time. Note that this does not affect the current GEM-DOS time.

```
#include <osbind.h>

main() {
	register unsigned long time;
	int seconds;
	int minutes;
	int hours;
	int day;
	int month;
	int year;

	printf("Enter the date and time (MM/DD/YYYY HH:MM): ");
	scanf("%d/%d/%d %d:%d", &month, &day, &year, &hours, &minutes );
	seconds = 0;
	if(year < 100)
		year += 1900;
	time =  ((unsigned long)(year-1980)<<25)
		|((unsigned long)month<<21)
		|((unsigned long)day<<16)
		|((unsigned long)hours<<11)
		|((unsigned long)minutes<<5)
		|((unsigned long)seconds>>1);
	timeprint("We are setting the time to", time );
	Settime(time);

/* Verify what we did. */

	time = Gettime();
	timeprint("What we get is", time);
}

void fixdig(buf, onumber, size)
char *buf;
int onumber;
int size;
{
	register long limit;
	register long number;
	int o;

	number = onumber;

	limit = 10;
	for (o = 1; o < size ; o++)
		limit *= 10;
```

```
        if ((number >= limit)||(number <0)) {
                for (o = 0; o < size; o++)
                        *buf++ = '*';
                *buf = 0;
                return;
        }
        for (o = 0; o < size; o++) {
                limit /= 10;
                *buf++ = '0'+number/limit;
                number = number%limit;
        }
        *buf = '\0';
}

timeprint(string, time)
char *string;
register unsigned long time;
{
        int seconds;
        int minutes;
        int hours;
        int month;
        int day;
        int year;
        char mins[3];
        char secs[3];

        seconds = (time & 0x001F) << 1;         /* Bits 0:4 */
        minutes = (time >> 5)  & 0x3F;          /* Bits 5:10 */
        hours = (time >> 11) & 0x1F;            /* Bits 11:15 */

        day = (time >> 16) & 0x1F;              /* Bits 16:20 */
        month = (time >> 21) & 0x0F;            /* Bits 21:24 */
        year = ((time >> 25) & 0x7F)+1980;      /* Bits 25:31 */

        fixdig(mins, minutes, 2);
        fixdig(secs, seconds, 2);
        printf("%s %d:%s:%s on %d/%d/%d\n", string, hours, mins,
                secs, month, day, year);
}
```

For another example of this function, see the entry for **time**.

See Also
Gettime, Ksettime, time, TOS, xbios

Notes
The time data in the bit map used by **Settime** is in exactly the reverse order of the data used by the **gemdos** functions.

shel_envrn—AES function (**libaes.a/shel_envrn**)

Search for an environmental variable

#include <aesbind.h>
int shel_envrn(*parameter, name***) char ***parameter*, **name*;

shel_envrn is an AES routine that searches for a particular environmental variable. *name* points to the name of the variable for whose value you want; note that the name must end with an equal sign '='. *parameter* points to the byte immediately *following* the value of the variable. **shel_envrn** always returns one.

See Also
AES, TOS

shel_find—AES function (**libaes.a/shel_find**)

Search **PATH** for file name

#include <aesbind.h>
int shel_find(*pathname***) char ***pathname*;

shel_find is an AES routine that does searches for a file in the directories named in the **PATH** environmental variable. *pathname* points to the name of the file being sought; **shel_find** changes this name to the full path name of the file if it is found. **shel_find** returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, PATH, TOS

shel_read—AES function (**libaes.a/shel_read**)

Let an application identify the program that called it

#include <aesbind.h>
int shel_read(*command, tail***) char ***command*, **tail*;

shel_read is an AES routine that returns the name of the command that invoked the current AES application. *command* points to the name of the command, and *tail* points to its tail; the values of both are set by this routine. **shel_read** returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, TOS

shel_write—AES function (**libaes.a/shel_write**)

Run another application

#include <aesbind.h>
int shel_write(*flag, graphic, gem, command, tail***)**
int *flag, graphic, gem*; **char ***command*, **tail*;

shel_write is an AES routine that tells AES whether to run another application, and, if necessary, which application to run. *flag* indicates whether to run another application: zero, exit to the operating system; one, run another application. *graphic* indicates if the application to be run is a graphics application: zero indicates no, and one indicates yes. *gem* indicates if the application to be run is an AES application: zero indicates no, and one indicates yes.

Finally, *command* and *tail* point, respectively, to the command's name and tail. **shel_write** returns zero if an error occurred, and a number greater than zero if one did not.

See Also
AES, TOS

shellsort—General function (**libc.a/shellsort**)
Sort arrays in memory
shellsort(*data*, *n*, *size*, *comp***)**
char **data*; **int** *n*, *size*; **int** (**comp*)();

shellsort is a generalized algorithms for sorting arrays of data in primary memory. **shellsort** uses D. L. Shell's sorting method. **shellsort** works with a sequential array of memory called *data*, which is divided into *n* parts of *size* bytes each. In practice, *data* is usually an array of pointers or structures, and *size* is the **sizeof** the pointer or structure. Each routine compares pairs of items and exchanges them as required. The user-supplied routine to which *comp* points performs the comparison. It is called repeatedly, as follows:

```
(*comp)(p1, p2)
char *p1, *p2;
```

Here, *p1* and *p2* each point to a block of *size* bytes in the *data* array. In practice, they are usually pointers to pointers or pointers to structures. The comparison routine must return a negative, zero, or positive result depending on whether *p1* is less than, equal to, or greater than *p2*, respectively.

Example
For an example of how to use this routine, see the entry for **string**.

See Also
ctype, qsort
The Art of Computer Programming, vol. 3, pp. 84*ff*, 114*ff*

Notes
shellsort is an iterative algorithm; it does not use much stack.

short—Definition
A **short** is a numeric data type. By definition, it cannot be longer than an **int** or a **long**. For Mark Williams C, a **short** is equal to an **int**; that is, **sizeof short** e-

quals two **chars**, or 15 bits plus a sign. A **short** normally is sign extended when cast to a larger data type; however, an **unsigned short** will be zero extended when cast.

See Also
declarations

show—Command
Display a stored screen image
show *screenfile*

show displays a screen image that has been stored with the command **snap**. *screenfile* is the name of the file in which the screen image is stored. **show** checks to see that *screenfile* is the correct size, i.e., large enough to hold an entire screen image (32 kilobytes). If the file is of the wrong size, **show** exits silently.

See Also
commands, snap, TOS

showmouse—Command
Redisplay the mouse pointer
showmouse

showmouse uses the function **linea9** to redisplay the mouse pointer.

See Also
commands, hidemouse, Line A, TOS

signal.h—Header file
TOS header file
#include <signal.h>

signal.h is a header file that defines signals used on the Atari ST. These include 68000 machine exceptions, trap instructions, and GEM-DOS aliases.

See Also
bombs, header file, TOS

sin—Mathematics function (**libm.a/sin**)
Calculate sine
#include <math.h>
double sin(*radian***) double** *radian*;

sin calculates the sine of its argument *radian*, which must be in radian measure.

Example
For an example of this function, see the entry for **acos**.

See Also
mathematics library

sinh—Mathematics function (**libm.a/sinh**)
Calculate hyperbolic sine
#include <math.h>
double sinh(*radian*) double *radian*;

sinh calculates the hyperbolic sine of *radian*, which is in radian measure.

Example
For an example of this function, see the entry for **cosh**.

See Also
mathematics library

size—Command
Print the size of an object module
size [-act] *file...*

size prints the size of each segment of each given Mark Williams C object module *file* in decimal, plus the total of all the segments in both decimal and hexadecimal. All sizes are in bytes. Each *file* must be a Mark Williams C object module.

The options are as follows:

-a Print the size of debug, symbol, and relocation segments as well.

-c Print the total size of all common areas in each relocatable object module.

-t At the end, print the total size of each segment summed over all the files; no total is printed if only one *file* is specified. The segmented listed are the following:

.shri	shared instruction
.prvi	private instruction (usually zero)
.bssi	uninitialized instruction (usually zero)
.shrd	shared data
.prvd	private data
.bssd	uninitialized data

See Also
cc, commands, cpp, nm, strip

sizeof—Definition

sizeof is C operator that returns a constant **int** the size of any given data element. The element examined may be a data object or a piece of a data object, or a type cast. **sizeof** returns the size of the element in **chars**.

Note that **sizeof** is especially useful in **malloc** routines, and to specify byte counts to I/O routines. Using it to set the size of data types instead of using a predetermined value will increase the portability your code.

See Also
data types, operators
The C Programming Language, page 188

sleep—Command

Stop executing for a specified time
sleep *seconds*

sleep suspends execution for a specified number of *seconds*. This routine is especially useful with other commands to the shell **msh**. For example, typing

```
sleep 3600; echo coffee break time
```

will execute the **echo** command in one hour (3,600 seconds) to indicate an important appointment. **sleep** operates in two-second increments under TOS.

See Also
commands, msh, msleep

snap—Command

Save a screen image
snap *screenfile*

snap takes a "snapshot" of the screen's image, and writes it into *screenfile*. Note that *screenfile* is always 32 kilobytes long; if the disk drive does not have enough space to hold a file of this size, **snap** exits without an error message.

See Also
commands, show, TOS

sort—Command

Sort lines of text
sort [-bcdfimnru] [-t *c*] **[-o** *outfile*] **[-T** *dir*] [+*beg*[-*end*]][*file* ...]

sort reads lines from each *file* specified, or the standard input if none. It writes to the standard output in sorted order. The order into which the output is sorted is determined by comparing a *key* from each line; the key is all or part of an input line, depending upon options are selected. By default, the key is the entire input record (line) and ordering is by the ASCII collating sequence, i.e., lower-valued ASCII characters sorted before higher-valued.

The following options affect how the key is constructed or how the output is ordered.

-b Ignore leading white space (blanks or tabs) in key comparisons.

-d Dictionary ordering; only letters, blanks, and digits are considered in key comparisons. This is essentially the ordering used to sort telephone directories.

-f Fold upper-case letters to lower case for comparison purposes.

-i Ignore all characters outside of the printable ASCII range (040-0176).

-n This option tells **sort** that the key is a numeric string, which consists of optional leading blanks and optional minus sign followed by any number of digits with an optional decimal point. The ordering is by the numeric, as opposed to alphabetic, value of the string.

-r Reverse the ordering, i.e., **sort** from largest to smallest.

As noted above, the key compared from each line need not be the entire input line. The option *+beg* indicates the beginning position of the key field in the input line, and the optional *-end* indicates that the key field ends just before the *end* position. If no *-end* is given, the key field ends at the end of the line. Each of these positional indicators has the form *+m.nf* or *-m.nf*, where *m* is the number of fields to skip in the input line and *n* is the number of characters to skip after skipping fields. Optional flags *f* are chosen from the above key flags (**bdfinr**) and are local to the specified field.

The following additional options control how **sort** works.

-c Check the input to see if it is sorted. Print the first out of order line found.

-m Merge the input files. **sort** assumes each *file* to be sorted already. For large files, it runs much faster with this option.

-o *outfile*
Put the output into *outfile* rather than on the standard output. This allows **sort** to work correctly if the output file is one of the input files.

-t*c* Use the character *c* to separate fields rather than the default blanks and tabs.

-u Suppress multiple copies of lines with key fields that compare equally.

See Also
commands

Diagnostics
sort returns a nonzero exit status if file opening errors or other internal problems occurred, or if the file was not correctly sorted in the case of the -c option.

sprintf—STDIO function (**libc.a/printf**)
Format output
#include <stdio.h>
sprintf(*string*, *format* [, *arg*] ...)
**char ****string*, ***format*;

sprintf uses the string *format* to specify an output format for each *arg*; it then writes every *arg* into *string*, which it ends with NUL. For a detailed discussion of **sprintf**'s formatting codes, see **printf**.

See Also
printf, sprintf, STDIO
The C Programming Language, page 150

Notes
The output *string* passed to **sprintf** must be large enough to hold all output characters. Because C does not perform type checking, it is essential that each argument match its format specification.

sqrt—Mathematics function (**libm.a/sqrt**)
Compute square root
#include <math.h>
double sqrt(*z***) double** *z*;

sqrt returns the square root of *z*.

Example
For an example of this function, see the entry for **ceil**.

See Also
mathematics library

Diagnostics
A domain error in **sqrt** (*z* is negative) sets **errno** to **EDOM** and returns 0.

srand—General function (**libc.a/srand**)
Seed random number generator
srand(*seed***) int** *seed*;

srand uses *seed* to initialize the sequence of pseudo-random numbers returned by **rand**. Unequal values of *seed* initialize different sequences.

Example
For an example of how to use this function, see the entry for **rand**.

See Also
rand
The Art of Computer Programming, vol. 2

sscanf—STDIO function (**libc.a/scanf**)
Format input
#include <stdio.h>
sscanf(*string*, *format* [, *arg*] ...)
char **string*; **char** **format*;

sscanf reads the argument *string*, and uses *format* to specify a format for each *arg*, each of which must be a pointer. For more information on **sscanf**'s conversion codes, see **scanf**.

See Also
STDIO
The C Programming Language, page 150

Notes
Because C does not perform type checking, an argument must match its format specification. **sscanf** is best used only to process data that you are certain is in the correct data format, such data that were previously written out with **sprintf**.

stack—Definition
The **stack** is the segment of memory that holds function arguments, local variables, function return addresses, and stack frame linkage information. Neither the 68000 nor the Atari ST support dynamic stack resizing, so programs run on the ST have a fixed segment allocated to the stack at run time.

The Mark Williams C runtime startup routine allocates **_stksize** bytes of stack when a program is executed, and sets the 68000 stack pointer register, **a7**, to point at the highest address in this segment. **_stksize** is then assigned a pointer to the lowest address that the stack pointer may reach before the stack begins to overwrite program data. **_stksize** is set to two kilobytes by the Mark Williams C library. It may be set to another value by including an initialized declaration for it in your program; for example

```
long _stksize = 16000;
```

sets the stack size to 16,000 bytes.

The value of **_stksize** must be even. The size of the stack cannot change once your program has begun to execute because the allocation must be made before the stack is used and your program uses stack as soon as it begins to execute.

If your program uses recursive algorithms, or declares large amounts of automatic data, or simply contains many levels of functions calls, the stack may "overflow", and overwrite the program data. You can check for stack overflow very simply. The runtime startup reinitializes the **long _stksize** to point to an address that the stack should not reach. You can compare **_stksize** to the address of the last automatic variable in any function; as long as **_stksize** is *less* than the address of that automatic function, you are safe.

Example
This example checks for stack overflow; it aborts the program and prints a message when overflow occurs. The **main** routine prints the location of its arguments, calls the stack overflow routine, and then calls itself recursively. For another example, see the entry for **Fgetdta**.

```
_stktest(){
        int i;
        if ((long)&i <= _stksize) {
                puts ("Stack overflow!");
                exit(1);
        }
}

main(argc) int argc; {
        extern long _stksize;
        printf("argc at %lx\n", &argc);
        _stktest();
        main(argc);
}
```

See Also
_stksize

Notes
TOS pushes data onto the user stack; therefore, you should make sure that your stack has a cushion of at least 128 bytes to hold these data when your program enters the system.

standard input—Definition

The **standard input** is the device from which data are accepted by default; it is defined in the header file **stdio.h** under the abbreviation **stdin**, and will be the computer's keyboard unless redirected by **msh** or **freopen**.

See Also
freopen, header file, msh, standard output, stdio.h

standard output—Definition
The **standard output** is the peripheral device upon which programs write output by default. It is defined in the file **stdio.h** under the abbreviation **stdout**, and in most instances is defined to be the computer's monitor.

See Also
header file, standard input, stdio.h

stat.h—Header file
Definitions and declarations used to obtain file status
#include <stat.h>

stat.h is a header file that contains the declarations of several structures used by the routines **fstat** and **stat**, which return information about a file's status.

See Also
fstat, header file, stat

stat—General function (**libc.a/stat**)
Find file attributes
#include <stat.h>
stat(*file, statptr***)**
char ****file*; struct stat ****statptr***;**

stat returns a structure that contains the full GEM-DOS attributes of a file; note that the listing shown by the command does not describe attributes fully. *file* points to the path name of file, and *statptr* points to a structure of the type **stat**, as defined in the header file **stat.h**.

The following summarizes the structure **stat** and defines the permission and file type bits.

```
struct stat {
        dev_t st_dev;
        int_t st_ino;
        unsigned short st_mode;
        short st_nlink;
        short st_uid;
        short st_gid;
        dev_t st_rdev;
        size_t st_size;
        time_t st_atime;
        time_t st_mtime;
        time_t st_ctime;
};

#define S_IJRON 0x01        /* Read-only */
#define S_IJHID 0x02        /* Hidden from search */
#define S_IJSYS 0x04        /* System, hidden from search */
#define S_IJVOL 0x08        /* Volume label in first 11 bytes */
#define S_IJDIR 0x10        /* Directory */
#define S_IJWAC 0x20        /* Written to and closed */
```

Entries in the structure **stat** are there to preserve compatibility with the COHERENT operating system. Most return meaningless values when used on the Atari ST, with the following exceptions: **st_atime**, **st_mtime**, and **st_ctime** all return the time that the file or directory was last modified; **st_size** gives the size of the file, in bytes; and **st_mode** gives the mode of the file, as described in the entry for **ls**.

See Also
fstat, ls, msh, open, stat.h

Diagnostics
stat returns -1 if the file is not found.

static—Definition

static is a C storage class.

A **static** variable resembles an **extern** in that it does not disappear when its calling function exits. Unlike an **extern**, however, a **static** variable is "private": when internal to a function, it can be accessed only by that function; when used external to a function, it can be accessed only by functions that are defined within the same source file as that variable. This helps to avoid name conflicts; for example, if a program consists of two files, each of which has a variable named **foo**, declaring each **foo** to be **static** keeps them from being written into each other.

Functions that are used locally can also be declared to be **static**; this helps to prevent name conflicts when assembling programs from a number of different sources, such as libraries from a variety of vendors and modules written by different programmers.

See Also
auto, extern, register variable, storage class
The C Programming Language, page 80

stdin—Definition

stdin is an abbreviation for *standard input*. It is defined in the header file **stdio.h**.

See Also
stdio.h, standard input

STDIO—Overview

STDIO is an abbreviation for *standard input and output*. It refers to a set of standard library functions that accompany all C compilers and that govern input and output with peripheral devices.

Mark Williams C includes the following STDIO routines:

clearerr	Present status stream
exit	Leave a program gracefully
fclose	Close a stream
fdopen	Open a stream for I/O
feof	Discover a stream's status
ferror	Discover a stream's status
fflush	Flush a buffer
fgetc	Get a character
fgets	Get a string
fgetw	Get a word
fileno	Get a file descriptor
fopen	Open a stream
fprintf	Format and print to a file
fputc	Output a character
fputs	Output a string
fputw	Output a word
fread	Read a stream
freopen	Open a stream
fscanf	Format and read from a file
fseek	Seek in a stream
ftell	Return file pointer position
fwrite	Write to a stream
getc	Get a character
getchar	Get a character
gets	Get a string
getw	Get a word

putc	Output a character
putchar	Output a character
puts	Output a string
putw	Output a word
rewind	Reset a file pointer
scanf	Format and input from standard input
setbuf	Set alternative stream buffers
sprintf	Format and print to a string
sscanf	Format and read from a string
ungetc	Return character to input stream

Note that STDIO routines are buffered by default.

See Also
buffer, FILE, Lexicon, stdio.h, stream
The C Programming Language, page 166

stdio.h—Header file

stdio.h is a header file that defines several manifest constants used in I/O, such as **NULL** and **FILE**, declares the **STDIO** functions, and defines numerous I/O macros.

See Also
header file, manifest constant, STDIO

stdout—Definition

stdout is an abbreviation for *standard output*; it is defined in the header file **stdio.h**.

See Also
standard output, stdio.h

stime—Time function

Set the time
#include <time.h>
stime(*timep***) time_t ***timep***;**

stime sets the system time, which Mark Williams C defines as being the number of seconds since midnight of January 1, 1970, 0h00m00s GMT. The argument *timep* points to the new system time, which is of the type **time_t**; this is defined in the header file **time.h** as being equivalent to a a **long**.

Example
For an example of using this function from the **\auto** directory, see the entry for **\auto**.

See Also
date, time

Diagnostics
stime returns -1 on error, zero otherwise.

_**stksize**—External data

_**stksize** is an external symbol that sets the size of the stack. It is defined in the Mark Williams Company libraries as being equal to two kilobytes, which is more than enough stack for most applications.

If you wish to have more stack, insert into **main** the declaration

```
long _stksize = n;
```

where *n* is the number of bytes required. *n* must be even.

Example
For an example of how to use this variable in a program, see the entry for **memory allocation**. For an example of a program that uses _**stksize** to check for stack overflow, see the entry for **Fgetdta**.

See Also
ld, stack

storage class—Definition

Storage class refers to the part of a declaration that indicates how data are to be stored. The legal storage classes are as follows:

auto
extern
register
static

typedef is technically defined as a storage class as well, but it does not actually indicate how data are stored. The default class is **auto**.

See Also
auto, extern, register, static, typedef
The C Programming Language, page 192

strcat—String function (**libc.a/strcat**)

Append one string to another
char *strcat(*string1*, *string2***) char ***string1*, ****string2*;

strcat appends all characters in *string2* onto the end of *string1*. It returns the modified *string1*.

Example
See **string**.

For an example of this function in a TOS application, see the entry for **Fgetdta**.

See Also
string, strncat
The C Programming Language, page 44

Notes
string1 must contain enough space to hold itself and *string2*.

strcmp—String function (**libc.a/strcmp**)
Compare two strings
strcmp(*string1*, *string2***) char** **string1*, **string2*;

strcmp compares *string1* with *string2* lexicographically. It returns zero if the strings are identical, -1 if *string1* occurs earlier alphabetically than *string2*, and one if it occurs later. This routine is compatible with the ordering routine needed by **qsort**.

Example
See **string** and **malloc**.

See Also
qsort, string, strncmp
The C Programming Language, page 101

strcpy—String function (**libc.a/strcpy**)
Copy one string into another
char *strcpy(*string1*, *string2***) char** **string1*, **string2*;

strcpy copies the contents of *string2*, up to the NUL character, into *string1*, and returns *string1*. The order of the arguments is reminiscent of an assignment statement.

Example
See **string**.

For an example of using this function in a TOS application, see the entry for **Fgetdta**.

See Also
string, strncpy
The C Programming Language, page 100

Notes
string1 must contain enough space to hold *string2*.

stream—Definition

The term **stream** applies to any entity that can be named and through which bits can flow, such as a device or a file. The name "stream" reflects the fact that in the C programming environment eschews record descriptors and other devices that predetermine what form data assumes; rather, data, from whatever source, are seen merely to be a flow of bytes whose significance is imposed entirely by the context that the calling program creates.

See Also
bit, byte, data formats, file

string—Overview

The character string is a common structure in C programs. The runtime representation of a string is an array of ASCII characters that is terminated by a NUL character ('\0'). Mark Williams C uses this representation when a program contains a string constant, for example:

```
"I am a string constant"
```

The address of the first character in the string acts as the starting point, or "handle", of the string; note that a pointer to a string is nothing more than this address. Note, too, that an array of 20 characters holds a string of 19 (*not* 20) non-NUL characters; the 20th character is the NUL that terminates the string.

The following routines are available to help manipulate strings:

index	search for a character
rindex	search for a character
strcat	concatenate a string
strcmp	compare two strings
strcpy	copy a string
strlen	measure a string
strncat	concatenate a string
strncmp	compare two strings
strncpy	copy a string

Example
This example reads from **stdin** up to *NNAMES* names, each of which is no more than *MAXLEN* characters long. It then removes duplicates names, sorts the names, and writes the sorted list to the standard output. It demonstrates the functions **strcat, strcmp, strcpy,** and **strlen.**

```
#include <stdio.h>

#define NNAMES 512
#define MAXLEN 60

char *array[NNAMES];
char first[MAXLEN], mid[MAXLEN], last[MAXLEN];
char *space = " ";

extern int strcomp();
extern char *strcat();

main() {
	register int index, count, inflag;
	register char *name;

	count = 0;
	while (scanf("%s %s %s\n", first, mid, last) == 3) {
		strcat(first, space);
		strcat(mid, space);
		name = strcat(first, (strcat(mid, last)));
		inflag = 0;
		for (index=0; index < count; index++)
			if (strcmp(array[index], name) == 0)
				inflag = 1;
		if (inflag == 0) {
			array[count] = malloc(strlen(name) + 1);
			strcpy(array[count], name);
			count++;
		}
	}

	shellsort(array, count-1, sizeof(char *), strcomp);
	for (index=0; index < count; index++)
		printf("%s\n", array[index]);
	exit(0);
}

strcomp(s1, s2)
register char **s1, **s2;
{
	extern int strcmp();
	return(strcmp(*s1, *s2));
}
```

See Also
ASCII, Lexicon

strip—Command
Strip symbol table from object file
strip -drs *file ...*

strip removes the symbol table, relocation information, and debug tables from each object *file* specified. **strip** effects reasonable savings on systems where file space is at a premium.

strip recognizes the following options:

- **-d** Keep debug information.
- **-r** Keep relocation information.
- **-s** Keep symbols.

See Also
cc, commands, ld, nm

Notes
strip should be used only on fully linked files.

strlen—String function (**libc.a/strlen**)
Measure the length of a string
strlen(*string***) char ***string***;**

strlen measures *string*, and returns its length in bytes, not including the NUL terminator. This may be useful in determining how much storage to allocate for a string.

Example
For an example of how to use this function, see the entry for **string**. For an example of using this function in a TOS application, see the entry for **Fgetdta**.

See Also
string
The C Programming Language, page 95

strncat—String function (**libc.a/strncat**)
Append one string to another
char *strncat(*string1, string2, n***)**
char ***string1*, *****string2*; **unsigned *n*;

strncat copies up to *n* characters from *string2* onto the end of *string1*. It stops when *n* characters have been copied or it encounters a NUL character in *string2*, whichever occurs first, and returns the modified *string1*.

See Also
strcat, string

Notes
string1 should contain enough space to hold itself and *n* characters of *string2*.

strncmp—String function (**libc.a/strncmp**)

Compare two strings
strncmp(*string1*, *string2*, *n***)**
char **string1*, **string2*; **unsigned** *n*;

strncmp compares lexicographically the first *n* bytes of *string1* with *string2*. Comparison ends when *n* bytes have been compared, or a NUL character encountered, whichever occurs first. **strncmp** returns zero if the strings are identical, -1 if *string1* occurs earlier alphabetically than *string2*, and one if it occurs later. This routine is compatible with the ordering routine needed by **qsort**.

Example
For an example of the related string-handling function **strcmp**, see the entry for **string**.

See Also
strcmp, string

strncpy—String function (**libc.a/strncpy**)

Copy one string into another
char ***strncpy(***string1*, *string2*, *n***)**
char **string1*, **string2*; **unsigned** *n*;

strncpy copies up to *n* bytes of *string2* into *string1*, and returns *string1*. Copying ends when *n* bytes have been copied or a NUL character has been encountered, whichever comes first. If *string2* is less than *n* characters in length, *string2* is padded to length *n* with one or more NUL bytes. The order of the arguments is reminiscent of an assignment statement.

Example
For an example of the related string-handling function **strcpy**, see the entry for **string**.

See Also
strcpy, string

Notes
string1 should have enough space to hold itself and *n* characters of *string2*.

struct—Definition

struct is a C keyword that introduces a structure. The following is an example of how **struct** can be used in the description of a name and address file:

```
struct address {
        char firstname[10];
        char lastname[15];
        char street[25];
        char city[10];
        char state[2];
        char zip[5];
        int  salescode;
};
```

The definition of C in *The C Programming Language* prohibits the assignment of structures, the passing of structures to functions, and the returning of structures by functions. Mark Williams C allows structures to be assigned, provided they are of the same type, and allows structures to be passed and returned from functions. These features are supported by most compilers, but users should be aware that their use can cause problems in porting code to some compilers.

See Also
array, field, structure
The C Programming Language, page 119

structure—Definition

A **structure** is a set of variables that has been given a name and can be worked with as a single entity. The variables may be of different data types. Structures are a convenient way to deal with data elements that belong together, such as names and addresses, employee descriptions, or sales and inventory information.

See Also
field, record, struct
The C Programming Language, page 119

structure assignment—Definition

The C Programming Language forbids structure assignment, the passing of structures to functions, and returning structures from functions (as opposed to the passing or returning of *pointers* to structures). Mark Williams C lifts these restrictions.

Some other C compilers modify structure arguments and structure returns to be structure pointers. Note that the use of structure assignment, structure arguments, or structure returns may create problems when porting the code to another computing environment.

See Also
structure

SUFF—Environmental parameter

SUFF names a set of suffixes that **msh** will automatically append to command names. The suffixes are appended to the given command name when searching the directories named in the **PATH** environmental variable. For example, typing

```
setenv PATH=\bin,,\lib
setenv SUFF=,.prg,.tos,.ttp
```

means that when you give **msh** the command

```
foo
```

it will look for a file with one of the following names:

```
\bin\foo
\bin\foo.prg
\bin\foo.tos
\bin\foo.ttp
foo
foo.prg
foo.tos
foo.ttp
\lib\foo
\lib\foo.prg
\lib\foo.tos
\lib\foo.ttp
```

The file names are searched for in the order given above, and **msh** stops searching after finding the first file that matches the requested pattern.

It is set the with **setenv** command.

See Also
msh, setenv

Super—**gemdos** function 32 (**osbind.h**)
Enter supervisor mode
long Super(*stack***) char ****stack***;**

Super manipulates the Atari ST's supervisor mode, which, in theory, must be obtained before the extended BIOS routines can be used. *stack* points to a new supervisor stack. If the machine is presently set in user mode, it switches to supervisor mode; if in supervisor mode, it returns to user mode.

Example
This example changes the floppy write verify flag so floppy writes are not automatically verified. This speeds up processing, but can be dangerous, and is not recommended.

```
#include <osbind.h>
#define FVERIFY ((short *) 0x0444L)

main() {
	long save_ssp;

	save_ssp = Super(0L);		/* Switch to system mode */
	*FVERIFY = 0;			/* Clear the word. */
	Super(save_ssp);		/* Restore system */
}
```

See Also
gemdos, TOS

Notes
Super has been documented elsewhere as returning the supervisor/user mode flag if *stack* is set to -1L; however, it crashes the system instead. With systems that have TOS in ROMs, *stack* should be set to one to perform this task.

Supexec—xbios function 38 (osbind.h)

Run a function under supervisor mode
#include <osbind.h>
#include <xbios.h>
unsigned long Supexec(*address*)
int *_address_;

Supexec invokes supervisor mode, and allows you to run a routine under it. *address* is the address of the function to be run.

The **Supexec** function has two features that are not widely known but could prove useful in your programs.

The first is that any value returned by function run under under **Supexec** is returned untouched by the **xbios** trap.

Example
The following example uses the return value of a function run under **Supexec** to time execution speeds:

```
/* Redefine Supexec() function to get long return value */
#include <osbind.h>
#undef Supexec
#define Supexec(a) xbios(38,a)

/* Return the system 200 hz timer tick count */
long read_ticks() { return *((long *)0x4ba); }
```

```
/* Return microseconds that (*f)() takes to execute */
long time_function(f) int (*f)();
{
	register int ntimes = 4*5*1000;
	long tstart = Supexec(read_ticks);
	while (--ntimes >= 0) (*f)();
	return (Supexec(read_ticks) - tstart + 2) >> 2;
}

/* Some functions to time */
null_function() { return; }
int ia = 0x0123, ib = 0x3210;
int iret_function() { return ia,ib; }
int iadd_function() { return ia+ib; }
int isub_function() { return ia-ib; }
int imul_function() { return ia*ib; }
int idiv_function() { return ia/ib; }

long la = 0x01234567L, lb = 0x76543210L;
long lret_function() { return la,lb; }
long ladd_function() { return la+lb; }
long lsub_function() { return la-lb; }
long lmul_function() { return la*lb; }
long ldiv_function() { return la/lb; }

double da = 12340.0, db = 4321.0;
double dret_function() { return da,db; }
double dadd_function() { return da+db; }
double dsub_function() { return da-db; }
double dmul_function() { return da*db; }
double ddiv_function() { return da/db; }

/* Report the times for the functions */
main() {
	printf("null %ld microseconds\n", time_function(null_function));
	printf("iret %ld microseconds\n", time_function(iret_function));
	printf("iadd %ld microseconds\n", time_function(iadd_function));
	printf("isub %ld microseconds\n", time_function(isub_function));
	printf("imul %ld microseconds\n", time_function(imul_function));
	printf("idiv %ld microseconds\n", time_function(idiv_function));

	printf("lret %ld microseconds\n", time_function(lret_function));
	printf("ladd %ld microseconds\n", time_function(ladd_function));
	printf("lsub %ld microseconds\n", time_function(lsub_function));
	printf("lmul %ld microseconds\n", time_function(lmul_function));
	printf("ldiv %ld microseconds\n", time_function(ldiv_function));
```

```
        printf("dret %ld microseconds\n", time_function(dret_function));
        printf("dadd %ld microseconds\n", time_function(dadd_function));
        printf("dsub %ld microseconds\n", time_function(dsub_function));
        printf("dmul %ld microseconds\n", time_function(dmul_function));
        printf("ddiv %ld microseconds\n", time_function(ddiv_function));
        return 0;
}
```

The second feature is that a function run under **Supexec** can be passed parameters by including them in the call to the **xbios** trap. The first parameter to the function will always be a long pointer to itself. Any subsequent parameters will be available if they are declared in normal C style.

Example
The following example passes three arguments to a function run under **Supexec** to copy a block of low memory to a user-supplied buffer.

```
/* Redefine Supexec() to pass 3 arguments */
#include <osbind.h>
#undef Supexec
#define Supexec(a,b,c,d) xbios(38,a,b,c,d)

/* Word copy function with dummy parameter */
supercopy(self,destp,srcp,nwds) register int (*self)(), *destp, *srcp, nwds;
{
        while (--nwds >= 0) *destp++ = *srcp++;
}

/* Copy the process dump area to our data space and print it */
main() {
        int proc[64]; /* More or less */
        Supexec(supercopy,proc,0x380L,64);
        for (i = 0; i < 64; i += 4)
                printf("%04x %04x %04x %04x\n", proc[i], proc[i+1], proc[i+2],
                        proc[i+3]);
        return 0;
}
```

See Also
TOS, xbios

Sversion—**gemdos** function 48 (**osbind.h**)
Get the version number of TOS
#include <osbind.h>
int Sversion()

Sversion gets and returns the current TOS version number.

Example
This example prints the TOS version number on the standard output.

```
#include <osbind.h>

main() {
	union {
		struct {
		unsigned minor:8;
		unsigned major:8;
		} braker;
		int all;
	} versn;

	versn.all = Sversion();
	printf("TOS/GEMDOS version %2X revision %2x.\n",
		versn.braker.major, versn.braker.minor);
}
```

See Also
gemdos, TOS

swab—General function (**libc.a/swab**)
Swap a pair of bytes
swab(*src*, *dest*, *nb***) char** **src*, **dest*; **unsigned** *nb*;

The ordering of bytes within a word differs from machine to machine. This may cause problems when moving binary data between machines. **swab** interchanges each pair of bytes in the array *src* that is *n* bytes long, and places the result into the array *dest*. The length *nb* should be an even number, or the last byte will not be touched. *src* and *dest* may be the same place.

See Also
byte ordering

system—General function (**libc.a/system**)
Pass a command to TOS for execution
int system(*commandline***) char** **commandline*;

system passes *commandline* to the Mark Williams shell, which loads it into memory and executes it. **system** executes commands exactly as if they had been typed directly into the shell.

Example
This example uses the system function to list all C programs in the present directory.

```
main() {
        extern int system();
        system("echo [a-z]*.c");
}
```

See Also
exit, msh, Pexec

Notes
No shell variable that has been set with the **set** command is duplicated.

system variables—Definition

The TOS operating system uses a number of "magic locations" where it stores key system variables. By using the **peek** and **poke** routines included with Mark Williams C, you can alter these variables directly, to customize TOS more closely to your needs and tastes.

Note that you can safely manipulate the address 0x0 to 0x800 only when your program is in supervisor mode; you can enter supervisor mode by calling the **gemdos** function **super**.

The following table gives each "magic location", the common Atari mnemonic for it (should you wish to build a header file to work with these locations), the length of the system variable, and a brief description.

0x400/etv_timer/long
Points to the timer event handler.

0x404/etv_critic/long
Points to the critical error handler.

0x408/etv_term/long
Points to routine that ends a program.

0x420/memvalid/int
Check if the memory controller's configuration is valid.

0x424/memctrl/int
Copy of configuration value in memory controller.

0x426/resvalid/long
If proper value given, jump is made to reset routine pointed to by address **0x42A**.

0x42A/resvector/long
Address of reset routine.

0x42E/phystop/long
Top of RAM.

0x432/_membot/long
Points to beginning of transient program area.

0x436/_memtop/long
Points to end of transient program area.

0x43A/memval2/long
This if set properly, declares memory configuration to be valid.

43E/flock/int
If set to a value other than zero, disk access is in progress.

0x440/seekrate/int
Set disk drive seek rate, as follows: zero, six milliseconds; one, 12 milliseconds; two, two milliseconds; and three, three milliseconds.

0x442/_timer_ms/int
Clock rate, in microseconds.

0x444/_fverify/intn
If set to a value other than zero, every disk write access is verified.

0x446/_bootdev/int
Number of disk drive from which operating system was loaded.

0x448/palmode/int
If set to a value other than zero, system is in PAL mode (50 Hz); otherwise, system is in NTSC mode.

0x44A/defshiftmod/int
If Atari shifted from monochrome to color, new resolution is set here: zero indicates low resolution; one, medium resolution.

0x44C/sshiftmod/int
Screen resolution, as follows: zero, low resolution; one, medium resolution; two, high resolution.

0x44C/_v_bas_ad/long
Points to logical screen base. Address always begins on a 256-byte boundary.

0x452/vblsem/int
If set to zero, vertical blank routines are not executed.

0x454/nvbls/int
Number of vertical blank routines queued for execution.

0x456/_vblqueue/long
Points to the list of routines queued to be executed during vertical blanking.

0x45A/colorptr/long
If other than zero, holds pointer to color palette to be executed during next vertical blank.

0x45E/screenpt/long
Point to beginning of video RAM.

0x462/_vbclock/long
Number of vertical blank interrupt routines.

0x466/_freclock/long
Number of vertical blank routines executed.

0x46A/hdv_init/long
Point to hard-disk initialization.

0x46E/swv_vec/long
Point to routine to change screen resolution.

0x472/hdv_bpb/long
Point to fetch BIOS parameter block for hard disk.

0x476/hdv_rw/long
Point to read/write routine for hard disk.

0x47A/hdv_boot/long
Point to routine to reboot hard disk.

0x47E/hdv_mediach/long
Point to routine to handle medium change for hard disk.

0x482/_comload/int
If set to a value other than zero, system will attempt to load file **command.prg** after TOS has been loaded.

0x484/conterm/char
Set console attributes. This is a byte-length bit map, whose first four bits signify the following: bit 0, toggle key click; bit 1, toggle key repeat; bit 2, toggle bell when **<ctrl-G>** is typed; and bit 3, toggle returning **Kbshift** in bits 24-31 for the function **Conin**.

0x486/trp14ret/long
Return address for call to trap 14.

0x48A/criticret/long
Return address of critical error handler.

0x48E/themd/4 longs
Memory descriptor filled by function **Getmpb**.

0x4A2/savptr/long
Pointer to save area for process registers after a BIOS call.

0x4A6/_nflops/int
Number of floppy disk drives.

0x4A8/con_state/long
Point to screen output routine.

0x4AC/save_row/int
Save cursor line temporarily when moving cursor with **<esc>**Y.

0x4AE/sav_context/long
Point to temporary areas used by exception-handling routines.

0x4B2/_bufl/2 longs
Pointers to heads of buffer lists: first points to head of data sector list; second points to head of FAT (file allocation table).

0x4BA/_hz_200/long
Counter for 200-Hz system clock.

0x4BC/the_env/4 chars
Default environment string, four NULs.

0x4C2/_drvbits/long
Bit map indicating connected drives: bit zero indicates drive A:, bit one indicates drive B:, etc.

0x4C6/_dskbufp/long
Pointer to 1,024-byte disk buffer.

0x4CA/_autopath/long
Pointer to autoexecute path.

0x4CE/_vbl_list/8 longs
List of pointers to standard vertical blank routines.

0x4EE/_dumpflg/int
If set to one, a dump of the current screen is sent to the printer port. Dump can be aborted by typing **help** and **alt** keys simultaneously.

0x4F0/_prtabt/int
Printer abort flag due to time-out.

0x4F2/_sysbase/long
Pointer to beginning of operating system.

0x4F6/_shell_p/long
Pointer to global shell information.

0x4FA/end_os/long
Pointer to end of operating system.

0x4FE/exec_os/long
Pointer to start of AES.

Example
The following example **pokes** address **0x484** to turn off the key click:

```
main() {
	pokeb(0x484L, peekb(0x484L) & ~1);
}
```

See Also
memory allocation, peekb, peekl, peekw, pokeb, pokel, pokew, TOS

tail—Command

Print the end of a file

tail [+*n*[**bcl**]] [*file*]
tail [-*n*[**bcl**]] [*file*]

tail copies the last part of the specified *file*, or of the standard input if none, to the standard output.

The given *number* tells **tail** where to begin to copy the data. Numbers of the form +*number* measure the starting point from the beginning of the file; those of the form -*number* measure from the end of the file.

A specifier of blocks, characters, or lines (**b**, **c**, or **l**, respectively) may follow the number. If no *number* is specified, a default of -10 is assumed.

See Also
commands

Notes
As **tail** buffers data measured from the end of the file, large counts may not work.

tan—Mathematics function (**libm.a/tan**)

Calculate tangent

#include <math.h>
double tan(*radian*) double *radian*;

tan calculates the tangent of its argument *radian*, which must be in radian measure.

Example
For an example of this function, see the entry for **acos**.

See Also
mathematics library

Diagnostics
tan returns a very large number where it is singular, and sets **errno** to **ERANGE**.

tanh—Mathematics function (**libm.a/tanh**)

Calculate hyperbolic cosine

#include <math.h>
double tanh(*radian*) double *radian*;

tanh calculates the hyperbolic tangent of *radian*, which is in radian measure.

Example
For an example of this function, see the entry for **cosh**.

See Also
mathematics library

Diagnostics
tanh sets **errno** to **ERANGE** when an overflow occurs.

tempnam—General function (**libc.a/tempnam**)
Generate a unique name for a temporary file
char *tempnam(*directory, name***)**
char ****directory*, **name*;**

tempnam constructs a unique temporary name that can be used with your program.

directory points to the name of the directory in which you want the temporary file written. If this variable is **NULL**, **tempnam** reads the environmental variable **TMPDIR** and uses it for *directory*. If neither *directory* nor **TMPDIR** is given, **tempnam** uses **\tmp.**

name points to a string of letters that you want to prefix the temporary name; this string should not be more than a few characters, to prevent truncation or duplication of temporary file names. If *name* is **NULL**, **tempnam** will set it to t.

tempnam uses **malloc** to allocate a buffer for the temporary file name it returns. If all goes well, it returns a pointer to the temporary name it has written; otherwise, it returns **NULL** if the allocation fails or if it cannot build a temporary file name successfully.

See Also
environment, mktemp, msh, tmpnam

tetd_to_tm—Time function (**libc.a/tetd_to_tm**)
Convert IKBD time to system calendar format
#include <time.h>
tm_t *tetd_to_tm(*time***) tetd_t** *time***;**

tetd_to_tm converts the time setting for the intelligent keyboard, as returned by the function **Gettime**, into the system's calendar format.

time is of type **tetd_t**, which is defined in the header file **time.h** as being equivalent to an **unsigned long**. It holds the 32-bit map returned by **Gettime**. For information on what the bits of this map signify, see the entry for **Gettime**.

tetd_to_tm returns a pointer to the structure **tm_t**, which is defined in the header file **time.h**. For more information on this structure, see the entry for **time**.

See Also
time, time.h, tm_to_tetd

Tgetdate—gemdos function 42 (osbind.h)

Get the current date
#include <osbind.h>
int Tgetdate()

Tgetdate gets the current date from TOS. It returns an integer whose bits indicate the following:

0-4	day (1-31)
5-8	month (1-12)
9-15	year (0-119, 0=1980)

Example
This examples demonstrates both **Tgetdate** and **Tgettime**. Note that the time returned by this example will be one hour earlier than the time returned by **msh** if the latter is adjusting for daylight savings time.

```
#include <osbind.h>

main() {
	unsigned int date;
	unsigned int time;

	date = Tgetdate();			/* Get system date */
	time = Tgettime();			/* Get system time */
	timeprint("The TOS time is", time);
	dateprint("The TOS date is", date);
}

void fixdig(buf, onumber, size)
char *buf;
int onumber;
int size;
{
	register long limit;
	register long number;
	int o;

	number = onumber;

	limit = 10;
	for (o = 1; o < size ; o++)
		limit *= 10;
```

```
        if ((number >= limit)||(number <0)) {
                for (o = 0; o < size; o++)
                        *buf++ = '*';
                *buf = 0;
                return;
        }
        for (o = 0; o < size; o++) {
                limit /= 10;
                *buf++ = '0'+number/limit;
                number = number%limit;
        }
        *buf = '\0';
}

timeprint(string, time)
char *string;
register unsigned int time;
{
        int seconds;
        int minutes;
        int hours;
        char mins[3];
        char secs[3];

        seconds = (time & 0x001F) << 1;           /* Bits  0:4 */
        minutes = (time >> 5)  & 0x3F;            /* Bits  5:10 */
        hours = (time >> 11) & 0x1F;              /* Bits 11:15 */

        fixdig(mins, minutes, 2);
        fixdig(secs, seconds, 2);
        printf("%s %d:%s:%s\n", string, hours, mins, secs);
}

dateprint(string, date)
char *string;
unsigned int date;
{
        int year;
        int month;
        int day;

        day = date & 0x1F;
        month = (date>>5) & 0x0F;
        year = ((date>>9) & 0x7F) + 1980;
        printf("%s %d/%d/%d\n", string, month, day, year);
}
```

For another example of this function, see the entry for **time**.

See Also
gemdos, time, Tsetdate, TOS

Tgettime—**gemdos** function 44 (**osbind.h**)
Get the current time
#include <osbind.h>
int Tgettime()

Tgettime obtains the current time from the operating system. It returns the time encoded in the form of an integer whose bits mean the following:

0-4	number of two-second increments (0-29)
5-10	number of minutes (0-59)
11-15	number of hours (0-23)

Example
For example of how to use this function, see the entries for **Tgetdate** and **time**.

See Also
gemdos, time, Settime, TOS

Tickcal—**bios** function 6 (**osbind.h**)
Return system timer's calibration.
#include <osbind.h>
#include <bios.h>
long Tickcal()

Tickcal returns the system timer's calibration, rounded to the nearest millisecond.

Example
This example demonstrates **Tickcal**. Also see the example in the entry for **time**.

```
#include <osbind.h>

main()
{
	printf("System clock ticks once every %ld msec.\n", Tickcal());
}
```

See Also
bios, time, TOS

time—Time function (**libc.a/time**)
Get current time
#include <timeb.h>
time_t time(*tp*) time_t ****tp*;**

time reads the current system time. It is a simpler version of the function **ftime**. *tp* is a pointer of the type **time_t**, which is defined in the header file **time.h** as being equivalent to a **long**. Note that Mark Williams C defines the current system time as being the number of seconds since January 1, 1970, 0h00m00s GMT.

Example
For an example of this function, see the entry for **asctime**.

See Also
time (overview)

time—Overview
Mark Williams C includes a number of routines that allow the user to set and manipulate time, as recorded on the system's clock, into a variety of formats. These routines should be adequate for nearly any task a programmer has that involves temporal calculations or the maintenance of data gathered over a long period of time.

All functions, global variables, and manifest constants used in connection with time are defined and described in the header file **time.h**.

The ANSI Draft Time Standard
The draft ANSI standard for the C language describes functions designed to be used with calendar time (i.e., the Gregorian calendar), local time, and daylight savings time.

The basic unit of time is defined as the **CLK_TCK**, which is defined as one tick of the system clock. On the Atari ST, the **CLK_TCK** is equivalent to five milliseconds. Three types are declared:

clock_t
This is an implementation-specific type that is capable of encoding clock time. On the Atari ST, this is set to an **unsigned long**.

time_t
This is an implementation-specific type that can represent time. On Mark Williams C, **time_t** is defined as a 32-bit number that holds the number of seconds since January 1, 1970, 0h00m00s GMT.

struct tm or tm_t
This structure encodes the elements of calendar time. It is defined as follows:

```
typedef struct tm {
        int tm_sec;         /* second [0-59] */
        int tm_min;         /* minute [0-59] */
        int tm_hour;        /* hour [0-23]: 0 = midnight */
        int tm_mday;        /* day of the month [1-28,29,30,31] */
        int tm_mon;         /* month [0-11]: 0=January */
        int tm_year;        /* year since 1900 A.D. */
        int tm_wday;        /* day of week [0-6]: 0=Sunday */
        int tm_yday;        /* day of the year [0-365,366] */
        int tm_isdst;       /* daylight savings time flag */
} tm_t;
```

The ANSI standard also describes a number of time functions, as follows:

asctime	convert time to ASCII string
clock	return time since system was turned on
ctime	output an ASCII string that gives the time
difftime	compute difference between calendar times
gmtime	return Greenwich Mean Time
localtime	return local time
stime	set time (UNIX/COHERENT-compatible)

Extensions to the ANSI Standard
Mark Williams C includes a number of extensions to the ANSI standard. These are designed to increase the scope and accuracy of the standard, and to ease calculation of some time elements.

To begin, Mark Williams C includes three variables that are used by the function **localtime**; it parses the environmental variable **TIMEZONE** into the following:

timezone	seconds from GMT to give local time
dstadjust	seconds to local standard, if any
tzname	array with names of standard and daylight times

The following functions return information about the calendar:

isleapyear	is this year AD a leap year?
dayspermonth	how many days in this historical month?

Time on Mark Williams C is modelled after time on the COHERENT operating system. As noted above, the variable **time_t** is defined as the number of seconds since January 1, 1970, 0h00m00s GMT; this moment, in turn, is rendered as day 2,440,587.5 on the Julian calendar. This allows accurate calculation of time as far back as January 1, 4713 B.C.

Conversion to the Gregorian calendar is set to October 1582, when it was first adopted in Rome. The issue of conversion of when a nation changed from the Julian to the Gregorian calendar is moot in the United States, Canada (except Quebec), Asia, Africa, Australia, and the Middle East; however, users in

Quebec, Latin America, Europe, the Soviet Union, and European-influenced areas of Asia (e.g., India) may wish to to write their own functions to convert historical data properly from the Julian to the Gregorian calendar.

The following functions assist in conversion from Julian to Gregorian time:

time_to_jday	convert **time_t** to the Julian date
jday_to_time	convert Julian date to **time_t**
tm_to_jday	convert **tm_t** structure to Julian date
jday_to_tm	convert Julian date to **tm_t** structure

Atari ST Time Functions
The Atari ST's ROM BIOS contains a number of functions that manipulate system time. This task is complicated by the fact that the ST has several clocks, which do not reference each other; each can be set independently, and each is used under different circumstances.

The following functions convert between standard time and TOS time:

tm_to_tetd	convert **tm_t** to TOS time
tetd_to_tm	convert TOS time to **tm_t**

The intelligent keyboard (IKBD) keeps time to the second, but it not supported by either the **xbios** or the **gemdos** functions. The following two functions convert between time as encoded in **tm_t** and the IKBD clock:

Kgettime	turn IKBD time to **tm_t**
Ksettime	turn **tm_t** to IKBD time

Finally, the Atari **gemdos** and **xbios** routines include a number of functions that directly manipulate system time, as follows:

Fdatime	get/set a file's time and date stamp
Gettime	get the system time (**xbios**)
Settime	set the system time (**xbios**)
Tgettime	get the system time (**gemdos**)
Tgetdate	get the system date (**gemdos**)
Tsettime	get the system time (**gemdos**)
Tsetdate	set the system date (**gemdos**)

Example
For an examle of time functions, see the entry for **asctime**. The following example demonstrates the header file **time.h**, and the functions **Gettime**, **Kgettime**, **Ksettime**, **Settime**, **stime**, **tetd_to_tm**, **Tgetdate**, **Tgettime**, **time**, **time_to_tm**, **tm_to_tetd**, **tm_to_time**, **Tsetdate**, and **Tsettime**.

```
#include <time.h>
tm_t getdate(p)
char *p;
{
	static tm_t t;

	sscanf(p,"%4d%2d%2d%2d%2d.%2d", &t.tm_year, &t.tm_mon, &t.tm_mday,
		&t.tm_hour, &t.tm_min, &t.tm_sec);
	t.tm_year -= 1900;
	t.tm_mday -= 1;
	return &t;
}

dodisplay(tp, name)
tm_t *tp char *name;
{
	printf("%4d%02d%02d%02d%02d.%02d %s\n",
		tp->tm_year+1900, tp->tm_mon+1, tp->tm_mday,
		tp->tm_hour, tp->tm_min, tp->tm_sec, name);
}

#define display(x) dodisplay((tm_t *)(x), "x");

main(argc, argv)
int argc; char *argv[];
{
	tm_t *tp;
	tetd_t td;
	time_t t;

	if (argc > 1) {
		tp = getdate(argv[1]);
		td = tm_to_tetd(tp);
		t = tm_to_time(tp);
		stime(&t);

		Ksettime(tp);
		Settime(td);
		Tsetdate(td.g_date);
		Tsettime(td.g_date);
	}

	display(time_to_tm(time(0L)));
	display(Kgettime());
	display(tetd_to_tm(Gettime()));
	display(tetd_to_tm(((long)Tgetdate()<<16)|(unsigned)Tgettime()));
}
```

See Also
Lexicon

time_to_jday—Time function (**libc.a/time_to_jday**)

Convert system time to Julian date

#include <time.h>

jday_t time_to_jday(*time*) time_t *time*;

time_to_jday converts system time to Julian days. *time* is the current system time. It is declared to be of type **time_t**, which is defined in the header file **time.h** as being equivalent to a **long**. Mark Williams C defines the current system time as being the number of seconds from January 1, 1970, 0h00m00s GMT. The function **time** returns the current system time in this format.

time_to_jday returns the structure **jday_t**, which is defined in the header file **time.h**. **jday_t** consists of two unsigned **ints**. The first gives the number of the Julian day, which is the number of days since the beginning of the Julian calendar (January 1, 4713 B.C.). The second gives the number of seconds since midnight of the given Julian day.

See Also

jday_to_time, jday_to_tm, time, time.h, tm_to_jday

time.h—Header file

Header file with time-description structure

#include <time.h>

time.h is a header file that contains descriptions and declarations for elements used to manipulate time under TOS.

See Also

time, timeb.h

timezone—Time library data

timezone helps to convert TOS time to a form readable by humans. It is an external variable that contains the number of seconds to be subtracted from GMT to obtain local standard time.

Example

For an example of how to use this routine, see the entry for **time**

See Also

settz, time, TIMEZONE

TIMEZONE—Environmental parameter

Time zone environmental parameter

TIMEZONE=*standard*:*offset*[:*daylight*: *date*:*date*:*hour*:*minutes*]

TIMEZONE is an environmental parameter that is set to information about the user's time zone. This information is used by **ctime** to construct its description of the current time and day. To set the **TIMEZONE** parameter, use the set command, as follows:

```
setenv TIMEZONE=[description]
```

where `[description]` describes your time zone. Most users write this command into the file **profile**, so that the **TIMEZONE** parameter is set automatically whenever they reboot their system.

A TIMEZONE description contains at least two fields that are separated by colons; the first gives the name of the standard time zone and second its offset from Greenwich Mean Time in minutes. Offsets are positive for time zones west of Greenwich and negative for time zones east of Greenwich.

Fields 3 through 7 are optional. Field 3 gives the name of the local daylight saving time zone. The absence of this field indicates that no daylight saving time correction should be made. If **TIMEZONE** contains no additional fields, the changes between standard time and daylight saving time occur at the times currently legislated in the United States: at 2 A.M. standard time on the last Sunday in April, and at 2 A.M. daylight saving time on the last Sunday in October.

Fields 4 and 5 specify the dates on which daylight saving time begins and ends. Each consists of three numbers separated by periods. The first number specifies which occurrence of the weekday in the month marks the change, counting positive occurrences from the beginning of the month and negative occurrences from the the end of the month. The second number specifies a day of the week, numbering Sunday as one. The third number specifies a month of the year, numbering January as one.

Finally, fields 6 and 7 specify the hour of the day at which daylight saving time begins and ends, and the number of minutes of adjustment.

Example
The following are possible descriptions of Central Standard Time:

```
TIMEZONE=CST:360
TIMEZONE=CST:360:CDT
TIMEZONE=CST:360:CDT:-1.1.4:-1.1.10
TIMEZONE=CST:360:CDT:-1.1.4:-1.1.10:2:60
```

The first setting provides conversions to standard time only, a convention used by many farmers. The last three settings provide conversions to daylight time and specify the default conversion rules in increasing detail.

Note that under the microshell **msh**, it usually not necessary to set the offset field, unless you wish to keep your system set to Greenwich Mean Time.

For an example of this variable's use in a program, see the entry for **asctime**.

See Also
environment, setenv, time

Notes
The time zone that **time** and **ftime** depends on how the time zone was originally set. If **date** and **TIMEZONE** has the correct offset from Greenwich, then the system time is GMT; however, if the time was set on the GEM desktop, or if **TIMEZONE** has set the offset from Greenwich incorrectly, then the system time is not GMT.

The default **profile** included with your copy of Mark Williams C has a **TIMEZONE** setting for Central Standard Time (CST/CDT). Users who live outside that time zone may wish to edit **TIMEZONE** to reflect their time zone.

tm_to_jday—Time function (**libc.a/tm_to_jday**)
Convert calendar format to Julian time
#include <time.h>
**jday_t tm_to_jday(*time*) tm_t ****time*;

tm_to_jday converts the system time, as described in the system calendar format, to Julian time. *time* points to a copy of the structure **tm_t**, which is defined in the header file **time.h**. The functions **gmtime** and **localtime** returns the current time in this format. For more information on **tm_t**, see the Lexicon entry for **time**.

tm_to_jday returns the structure **jday_t**, which is defined in the header file **time.h**. **jday_t** consists of two unsigned **ints**. The first gives the number of the Julian day, which is the number of days since the beginning of the Julian calendar (January 1, 4713 B.C.). The second gives the number of seconds since midnight of the given Julian day.

See Also
jday_to_time, jday_to_tm, time, time.h, time_to_jday

tm_to_tetd—Time function (**libc.a/tm_to_tetd**)
Convert system calendar format to IKBD time
#include <time.h>
**tetd_t tm_to_tetd(*time*) tm_t ****time*;

tm_to_tetd converts the system calendar structure, as returned by the functions **gmtime** and **localtime**, into a form that can be used by the Atari function **Settime** to set the intelligent keyboard's clock.

time points to a copy of the structure **tm_t**, which is defined in the header file **time.h**. For more information on this structure, see the entry for **time**.

tm_to_tetd returns a data element of the type **tetd_t**, which is defined in the header file **time.h** as being equivalent to an **unsigned long**. It holds the 32-bit map used by **Settime** to set the intelligent keyboard's clock. For information on what the bits of this map signify, see the entry for **Settime**.

See Also
tetd_to_tm, time, time.h

TMPDIR—Environmental parameter

TMPDIR directive names the directory into which **msh** and its commands write their temporary files.

It is set with the **setenv** command.

See Also
msh, setenv

tmpnam—General function (**libc.a/tmpnam**)

Generate a unique name for a temporary file
#include <stdio.h>
char ***tmpnam**(*name*) **char** **name*;

tmpnam constructs a unique temporary name that can be used with your program. *name* is the name of a buffer into which **tmpnam** writes the temporary name. If *name* is **NULL**, **tmpnam** writes the name into an internal buffer that is overwritten each time it is called.

tmpnam assumes that the temporary file will be written into directory **\tmp** and builds the name accordingly. It returns the address of the internal buffer.

See Also
mktemp, tempnam

toascii—ctype macro (**ctype.h**)

Convert characters to ASCII
#include <ctype.h>
toascii(*c*) **int** *c*;

toascii takes any integer value *c* and keeps the low seven bits. If *c* is already a valid ASCII character, it is unchanged.

See Also
ctype

tolower—ctype macro (**ctype.h**)

Convert characters to lower case
tolower(*c*) **int** *c*;

tolower converts the letter *c* to lower case. If *c* is not a letter, the result is undefined.

Example
The following demonstrates **tolower**.

For an example of its use in a TOS application, see the entry for **Fgetdta**.

```
#include <ctype.h>
#include <stdio.h>
main(){
	FILE *fp;
	int ch;
	int filename[20];
	printf("Enter name of file to use: ");
	gets(filename);
	if ((fp = fopen(filename,"r")) != NULL) {
		while ((ch = fgetc(fp)) != EOF)
			putchar(isupper(ch) ? tolower(ch) : ch);
	}
	else printf("Cannot open %s.\n", filename);
}
```

See Also
ctype, toupper

_tolower—ctype macro (**ctype.h**)
Convert letter to lower case
#include <ctype.h>
_tolower(*c***)**
int *c*;

_tolower is a macro that returns *c* converted to lower case. If *c* is not a letter, the result is undefined.

See Also
ctype

tos—Command
Execute GEM-DOS program
tos *program options*

tos allows you to run under **msh** a program that uses unredirected GEM-DOS file handles. It resets file handle 2 to the **aux:** device; unlike its cousin, the **gem** command, **tos** does not enable the mouse cursor. *program* is the name of the program you wish to execute; note that you should give the full path name of the program and its full name, including suffix. *options* are a list of options that are passed directly to the program to be executed.

See Also
commands, gem

TOS—Overview

TOS is the operating system for the ATARI ST. It includes a number of components, including Digital Research's Graphics Environment Manager (GEM) and the GEM-DOS disk operating system.

The following entries in the Lexicon describe features of TOS:

AES This describes the GEM Application Environment System (AES), which allows the programmer to use predefined windows, icons, pull-down menus, and other GEM elements. It also lists and briefly describes all of the AES routines; each AES routine has its own entry within the Lexicon.

bios This entry describes the TOS function **bios**, and introduces the functions that use it to manipulate the Atari ST's BIOS.

desk accessory
This entry describes how to compile a GEM desk accessory.

error codes
This lists and defines the error codes that can be returned by TOS.

gemdos This entry describes the TOS function **gemdos**, and introduces the functions that use it to manipulate GEM-DOS.

keyboard
This describes the layout of the Atari ST keyboard, with the codes generates by each key.

Line A This describes briefly the Atari "Line A" interface routines, which allow the creation and manipulation of graphics displays.

screen control
This entry lists the escape sequences used to control text on the Atari ST's screen.

system variables
This entry lists all of the "magic locations" within memory where TOS stores its key elements.

VDI This describes the GEM Virtual Device Interface (VDI), which gives the user access to basic graphics routines. It also lists and describes briefly all of the VDI routines; each VDI routine also has its own entry within the Lexicon.

xbios This entry describes the TOS function **xbios**, and introduces the function that use it to manipulate the Atari ST's extended BIOS.

A number of header files are also used with TOS. These include the following:

aesbind.h	bindings for GEM AES routines
basepage.h	TOS basepage structure
bios.h	declarations for **bios** functions
errno.h	**gemdos/bios/xbios** error number enumeration
gemdefs.h	miscellaneous declarations
gemout.h	TOS executable and archive file formats
linea.h	ST linea interface header
obdefs.h	miscellaneous object and variable definitions
osbind.h	bindings for bios/gemdos/xbios functions
signal.h	ST processor exception, extended trap vectors
stat.h	TOS DMABUFFER structure and file attributes
time.h	time and date services
vdibind.h	bindings for GEM VDI routines
xbios.h	declarations for **xbios** functions

Compiling TOS programs

You can include the AES/VDI libraries in your compilations in any of three ways.

First, you can include the libraries with the *library* option to the **cc** command line. To compile the program **sample.c**, use the following form of the **cc** command line:

```
cc sample.c -laes -lvdi
```

The -l option is described in the Lexicon entry for **cc**.

The other two methods involve using a switch on the **cc** command line. -VGEM is used to create an ordinary GEM program. It automatically links in the AES and VDI libraries, and calls the special run-time start-up routine **crtsg.o**. For example, to use the **-VGEM** option to compile **sample.c**, use the following command line:

```
cc -VGEM sample.c
```

crtsg.o has the advantage of being smaller, faster, and simpler than the default run-time start-up routine, **crts0.o**. Note, however, that it differs from the default runtime startup **crts0.o** in the following ways:

1. **argv**, **argc**, and **envp** are all set to zero.

2. **getenv** is not enabled; this means programs that use **crtsg.o** cannot read environmental parameters.

3. **stderr** will send error messages to the auxiliary ports rather to the console.

-VGEMACC is used to create a GEM desktop accessory. It works in much the same way as **-VGEM**, except that it uses the run-time start-up routine **crtsd.o** instead of **crtsg.o**.

The source files for **crtsd.o** and **crtsg.o** are included with your copy of Mark Williams C, should you wish to enhance it.

Finally, **libaes.a** uses the routine **crystal.o** to call traps. This routine is *never* called by the programmer, but it is automatically linked with **libaes.a**.

See Also
AES, bios, crtsg.o, gem, gemdos, keyboard, Lexicon, Line A, screen control, VDI, xbios

touch—Command
Update modification time of a file
touch [-c] *file ...*

TOS keeps track of when each file was last modified. **touch** changes the modification time of each *file* to the current time, but does not modify its contents. By default, **touch** creates *file* if it does not already exist; the **-c** flag suppresses this.

See Also
commands, make, msh

toupper—ctype macro (**ctype.h**)
Convert characters to upper case
#include <ctype.h>
toupper(*c***) int** *c*;

toupper is a macro that converts the letter *c* to upper case. If *c* is not a letter or is already upper case, the result is undefined.

Example
This example demonstrates **toupper** and **putchar**.

```
#include <ctype.h>
#include <stdio.h>
main(){
	FILE *fp;
	int ch;
	int filename[20];
	printf("Enter file name: ");
	gets(filename);
	if ((fp = fopen(filename,"r")) != NULL) {
		while ((ch = fgetc(fp)) != EOF)
			putchar(islower(ch) ? toupper(ch) : ch);
	}
	else printf("Cannot open %s.\n", filename);
}
```

See Also
ctype, tolower

_toupper—ctype macro (**ctype.h**)
Convert letter to upper case
#include <ctype.h>
_toupper(*c*)
int *c*;

_toupper is a macro that returns *c* converted to upper case. If *c* is not a letter, the result is undefined.

See Also
ctype

Tsetdate—gemdos function 43 (**osbind.h**)
Set a new date
#include <osbind.h>
long Tsetdate(*i*) int*i*;

Tsetdate sets a new date. The 16 bits of the integer *i* encode the date, as follows:

0-4	day (1-31)
5-8	month (1-12)
9-15	year (0-119, 0=1980)

Example
This example demonstrates the macros **Tsetdate** and **Tsettime**, and also uses the macros **Tgetdate** and **Tgettime**. For another example of this function, see the entry for **time**.

```
#include <osbind.h>

main() {
	unsigned int date;
	unsigned int time;
	int seconds;
	int minutes;

	int hours;
	int day;
	int month;
	int year;

	printf("Enter the date and time (MM/DD/YYYY HH:MM): ");
	scanf("%d/%d/%d %d:%d", &month, &day, &year, &hours, &minutes);
	seconds = 0;

	if (year < 100)
		year += 1900;
	date = ((unsigned)(year-1980)<<9)
		|((unsigned)month<<5)
		|(unsigned)day;

	time = ((unsigned)hours<<11)
		|((unsigned)minutes<<5)
		|((unsigned)seconds>>1);

	timeprint("About to set the TOS time to", time);
	dateprint("About to set the TOS date to", date);
	Tsetdate(date);
	Tsettime(time);

	date = Tgetdate();  /* Get the system date */
	time = Tgettime();  /* Get the system time */
	timeprint("Now the TOS time is", time);
	dateprint("Now the TOS date is", date);
}

void fixdig(buf, onumber, size)
char *buf;
int onumber;
int size;
{
	register long limit;
	register long number;
	int o;

	number = onumber;

	limit = 10;
	for (o = 1; o < size ; o++)
		limit *= 10;
```

```
        if ((number >= limit)||(number <0)) {
                for (o = 0; o < size; o++)
                        *buf++ = '*';
                *buf = 0;
                return;
        }

        for (o = 0; o < size; o++) {
                limit /= 10;
                *buf++ = '0'+number/limit;
                number = number%limit;
        }
        *buf = '\0';
}

timeprint(string, time)
char *string;
register unsigned int time;
{
        int seconds;
        int minutes;
        int hours;
        char mins[3];
        char secs[3];

        seconds = (time & 0x001F) << 1;   /* Bits  0:4 */
        minutes = (time >> 5)  & 0x3F;    /* Bits  5:10 */
        hours = (time >> 11) & 0x1F;      /* Bits 11:15 */

        fixdig(mins, minutes, 2);
        fixdig(secs, seconds, 2);
        printf("%s %d:%s:%s\n", string, hours, mins, secs);
}

dateprint(string, date)
char *string;
unsigned int date;
{
        int year;
        int month;
        int day;

        day = date & 0x1F;
        month = (date>>5) & 0x0F;
        year = ((date>>9) & 0x7F) + 1980;
        printf("%s %d/%d/%d\n", string, month, day, year);
}
```

See Also

gemdos, Tgetdate, time, TOS

Tsettime—**gemdos** function 45 (**osbind.h**)

Set a new time

#include <osbind.h>

long Tsettime(*time***) int** *time*;

Tsettime sets a new system time. The argument **time** is an integer whose bits encode the time, in the following manner:

0-4	two-second increments (0-29)
5-10	minutes (0-59)
11-15	hours (0-23)

Example

For examples of this function, see the entries for **time** and **Tsetdate**.

See Also

gemdos, Tgettime, time, TOS

type promotion—Definition

In arithmetic expressions, Mark Williams C promotes signed types to signed types by sign extension and unsigned types to unsigned types by zero padding. For example, **char** promotes to **int** by sign extension, while **unsigned char** promotes to **unsigned int** by zero padding.

See Also

data formats, declarations

type checking—Definition

Every expression has a *type*, such as **int**, **char**, or **double**. C is not strongly typed, and allows different types to be mixed relatively freely. This gives a programmer freedom to write programs of great power and scope, which is consistent with the C philosophy of paying out plenty of rope to a programmer; whether she uses that rope to pull herself out of a bog or to hang herself is entirely up to her. Mark Williams C checks types more strictly than the C standard implies, which most users appreciate. Mark Williams C's type checking can be enabled or disabled in degrees, using **-VSTRICT** and other "variant" options with the **cc** command.

See Also

cc

typedef—Definition

typedef is a C facility that allows programmers to define new data types. Such definitions are always made in terms of existing data types; for example,

```
typedef long FOO;
```

establishes a data type called **FOO**, and defines it to be equivalent to a **long**. Note that, by convention, programmer-defined data types are written in capital letters.

Judicious use of the **typedef** facility can make programs easier to maintain, and improve their portability.

See Also
declarations, manifest constants, portability, storage class
The C Programming Language, page 140

ungetc—STDIO function (**libc.a/ungetc**)
Return character to input stream
#include <stdio.h>
ungetc (*c*, *fp*) **int** *c*; **FILE** **fp*;

ungetc returns the character *c* to the stream *fp*. *c* can then be read by a subsequent call to **getc**, **gets**, **getw**, **scanf**, or **fread**. Exactly one character at a time can be pushed back into any stream. A call to **fseek** will nullify the effects of a previous **ungetc**.

Example

```
#include <stdio.h>
main() {
	FILE *fp;
	int ch, nlines, nsents;
	int filename[20];
	nlines = nsents = 0;
	printf("Enter name of file to check: ");
	gets(filename);

	if ((fp = fopen(filename,"r")) != NULL) {
		while ((ch = fgetc(fp)) != EOF) {
			if (ch == '\n') ++nlines;

			else if (ch == '.' || ch == '!' || ch == '?') {
				if ((ch = fgetc(fp)) != '.') {
					++nsents;
					ungetc(ch, fp);
				}

				else for(ch='.'; (ch=fgetc(fp))=='.';)
					;
			}
		}
		printf("%d line(s), %d sentence(s).\n", nlines, nsents);
	}
	else printf("Cannot open %s.\n", filename);
}
```

See Also
fgetc, getc, STDIO
The C Programming Language, page 156

Diagnostics
ungetc normally returns *c*; it returns **EOF** if the character cannot be pushed back.

union—Definition

A **union** describes an area of storage that accepts any one of a number of heterogeneous data elements. For example, a **union** may be declared to consist of an **int**, a **double**, and a **char** *; any one of these three elements can be held by the **union** at a time, and will be handled appropriately by it.

unions are helpful in dealing with heterogeneous data, especially within structures; however, the programmer is responsible for keeping track of what data type the **union** is holding at any given time. Passing a **double** to a **union** and then reading the **union** as though it held an **int** will yield results that are unpredictable, and probably unwelcome.

Example

```
union {
        int number;
        double bignumber;
        char *stringptr;
} unionname;
```

See Also

struct, structure

The C Programming Language, page 138

uniq—Command

Remove/count repeated lines in a sorted file

uniq [-cdu] [-*n***] [+***n***] [***infile***[***outfile***]]**

uniq normally reads input line by line from *infile* and writes all non-duplicated lines to *outfile*. The input file must be sorted. **uniq** uses the standard input or output if either *infile* or *outfile* is omitted. The following describes the available options:

-c Print each line once, discarding duplicate lines; before each line, print the number of times it appears within the file.

-d Print only lines that are duplicated within the file; print each line only once; do not print any counts.

-u Print only lines that are *not* duplicated within the file.

uniq by default behaves as if both **-u** and **-d** were specified, so it prints each unique line once.

Optional specifiers allow **uniq** to skip leading portions of the input lines when comparing for uniqueness.

-*n* Skip *n* fields of each input line, where a field is any number of non-white space characters surrounded by any number of white space characters (blank or tab).

+*n* Skip *n* characters in each input line, after skipping fields as above.

See Also
commands

UNIX **routines**—Overview

Mark Williams C includes a number of routines that were originally written for the UNIX system and related operating systems; these allow Mark Williams C to compile programs that were originally written for these systems.

The routines are as follows:

close	close a file
creat	create/truncate a file
dup	duplicate a file descriptor
dup2	duplicate a file descriptor
errno	integer returned by error routine
_exit	exit directly from a program
lseek	set read/write position
open	open a file
read	read from a file
unlink	remove a file
write	write to a file

See Also
Lexicon

unlink—UNIX system call (**libc.a/unlink**)

Remove a file
unlink(*fp***) FILE ****fp*;

unlink removes the directory entry for the given file *fp*. The name is a historical artifact.

Example
This example removes a file named on the command line.

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

	for (i = 1; i < argc; i++) {
		if (unlink(argv[i]) == -1) {
			printf("cannot unlink \"%s\"\n", argv[i]);
			exit(1);
		}
	}
	exit(0);
}
```

See Also
UNIX routines, STDIO

Diagnostics
unlink returns -1 if there are any errors, and zero otherwise.

unset—Command
Discard a shell variable
unset *VARIABLE*

unset discards a variable that had been set with the **set** command. For example, if you wished to discard the the variable **b**, simply type

```
unset b
```

and it will be erased.

See Also
commands, msh, set

unsetenv—Command
Discard an environmental variable
unsetenv *VARIABLE*

unset discards an environmental variable. For example, if you wish for some reason to discard the **TMPDIR** variable, type

```
unsetenv TMPDIR
```

See Also
commands, msh, setenv

unsigned—Definition
The **unsigned** modifier tells the compiler to treat the variable as an unsigned value. This in effect doubles the largest positive value storage in that type, and changes the lowest storage value to zero. Note that the 68000 uses "two's com-

plement" storage, not sign magnitude.

See Also
data type
The C Programming Language, page 34

v_arc—VDI function (**libvdi.a/v_arc**)

Draw a circular arc

#include <aesbind.h>
#include <vdibind.h>
void v_arc(*handle, xcoord, ycoord, radius, beginangle, endangle***)**
int *handle, xcoord, ycoord, radius, beginangle, endangle***;**

v_arc is a VDI routine that draws a circular arc. *handle* is the virtual device's VDI handle. *xcoord* and *ycoord* give, respectively, the X and Y coordinates of the imaginary center of the circle of which **v_arc** is drawing a section. *radius* is the radius of the imaginary circle. These measurements will differ, depending on whether the device has been set as using normalized device coordinates (NDC) or raster coordinates (RC). Finally, *beginangle* and *endangle* give, respectively, the beginning and end angles of the arc, measured in tenths of a degree. Counting on an imaginary clock, zero degrees is at 3 o'clock, 90 degrees (900) at noon, 180 degrees (1800) at 9 o'clock, and 270 degrees (2700) at 6 o'clock.

See Also
TOS, v_circle, VDI

v_bar—VDI function (**libvdi.a/v_bar**)

Draw a rectangle

#include <aesbind.h>
#include <vdibind.h>
void v_bar(*handle, xyarray***) int** *handle, xyarray***[4];**

v_bar is a VDI routine that draws a rectangle. Unlike its cousin **vr_recfl**, **v_bar** can draw a perimeter as well the preset fill pattern.

handle is the virtual device's VDI pattern. *xyarray* sets the X and Y coordinates from which to construct the rectangle; the even-numbered entries indicate the X coordinates, and the odd-numbered entries the Y coordinates. Which corner of the rectangle each pair of coordinates indicates will differ depending on whether the virtual device has been set to normalized device coordinates (NDC) or to raster coordinates (RC). On an NDC device, the first pair points to the lower left-hand corner and the second pair to the upper right-hand corner; whereas on an RC device, the first pair points to the upper left-hand corner and the second pair to the lower right-hand corner.

Note that to use this routine, the fill type must be set with **vsf_interior**, the fill style by **vsf_style**, the perimeter flag by **vsf_perimeter**, and the fill color by **vsf_color**. To output a complex polygon (i.e., a shape other than a rectangle), use the routine **v_fillarea**.

Example

The following program draws a filled rectangle onto the screen. By clicking the mouse's left button and dragging the mouse, you can draw a rectangle on the screen. Pressing the 'T' key changes the rectangle's *type* of fill; and pressing the 'S' key changes its *style*. Pressing **<return>** exits.

```
#include <gemdefs.h>
#include <aesbind.h>
#include <vdibind.h>

#define RETURN 0x1C0D                     /* scan code for <return> key */
#define T_KEY 0x1474                      /* scan code for T key */
#define S_KEY 0x1F73                      /* scan code for S key */

/* global line A variables used by vdi; MUST be included */
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

/* array used by v_bar() */
int xyarray[] = { 1, 1, 1, 1 };

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 639, 399 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* throw-away declarations, to keep system from scribbling over itself */
int nowhere = 0;
Rect norect = { 0, 0, 0, 0 };

main() {
/* declarations used by evnt_multi() */
        int selection;                    /* code for event that occurred */
        unsigned int which = (MU_KEYBD | MU_BUTTON);
        int clicks = 1;                   /* no. of clicks expected on mouse button */
        int button = 1;                   /* which button; 1 = leftmost */
        int buttonstate = 1;              /* button state expected; 1 = down */
        int buffer[11];                   /* place to write AES messages */
        int mousex;                       /* mouse X coordinate */
        int mousey;                       /* mouse Y coordinate */
        unsigned key;                     /* key typed by user */

/* misc declarations */
        int vdihandle;                    /* virtual device's handle */
        int type = 0;                     /* type of fill */
        int style = 1;                    /* style of fill */
        int width;                        /* width of rubberbox user draws */
        int depth;                        /* depth of rubberbox user draws */
```

```
/* OK, here we go ... */
	appl_init();
	graf_mouse(ARROW, &nowhere);
	vdihandle = graf_handle(&nowhere, &nowhere, &nowhere, &nowhere);
	v_opnvwk(work_in, &vdihandle, work_out);
	vs_clip(vdihandle, 1, cliparray);
	vsf_perimeter(vdihandle, 1);

	for(;;) {
		selection = evnt_multi(which, clicks, button, buttonstate,
			0, norect, 0, norect, buffer, 0, 0, &mousex, &mousey,
			&nowhere, &nowhere, &key, &nowhere);

		switch(selection) {
		case MU_KEYBD:
			switch(key) {
			case RETURN:
				v_clsvwk(vdihandle);
				appl_exit();
				exit(0);

			case T_KEY:
				type = (++type%5);
				vsf_interior(vdihandle, type);
				graf_mouse(M_OFF, &nowhere);
				v_bar(vdihandle, xyarray);
				graf_mouse(M_ON, &nowhere);
				break;

			case S_KEY:
				style = ((++style%24)+1);
				vsf_style(vdihandle, style);
				graf_mouse(M_OFF, &nowhere);
				v_bar(vdihandle, xyarray);
				graf_mouse(M_ON, &nowhere);
				break;
			}
			break;

		case MU_BUTTON:
			graf_rubbox(mousex, mousey, 3, 3, &width, &depth);
			xyarray[0] = mousex;
			xyarray[1] = mousey;
			xyarray[2] = (mousex+width);
			xyarray[3] = (mousey+depth);
			graf_mouse(M_OFF, &nowhere);
			v_bar(vdihandle, xyarray);
			graf_mouse(M_ON, &nowhere);
			break;
```

```
		default:
			break;
		}
	}
}
```

See Also
TOS, vr_recfl, VDI

v_bit_image—VDI function (**libvdl.a/v_bit_image**)
Print a bit image file
#include <aesbind.h>
#include <vdibind.h>
void v_bit_image(*handle, filename, aspect, scaling, points, xyarray***)**
int *handle, aspect, scaling, points, xyarray***[4]; char** **filename***;**

v_bit_image is a VDI routine that prints a bit image file. *handle* is the virtual device's VDI handle. *filename* points to the name of the file that holds the bit image; note that this name must be terminated with a NUL character.

aspect gives the code for the aspect ratio used to transfer the bit image onto paper, as follows: zero indicates that aspect ratio should be ignored; one, honor pixel aspect ratio; and two, honor page aspect ratio. *Pixel aspect ratio* ensures that the figures within the bit image remain constant, e.g., that a circle will remain circular; this may involve some cropping or shrinking of the image when printing. *Page aspect ratio* ensures that one full page in the bit image file is always printed as one full page of paper; this may result in some distortion of the figures within the bit image, however.

scaling describes how the bit image should be scaled onto to the page being printed; zero indicates that the X and Y coordinates should be scaled together, whereas one indicates that they should be scaled separately. Note that this argument is meaningful only if the variables in *xyarray* are set. If the X and Y coordinates are scaled together, the printed image may not fully occupy the rectangle defined by *xyarray* on the output device. If they are scaled separately, the bit image will entirely fill the area defined by *xyarray*, but the setting of *aspect* will be ignored.

Finally, *xyarray* defines the upper left-hand and lower right-hand corners of the area on the page into which the bit image will be printed.

See Also
TOS, VDI

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

v_cellarray—VDI function (**libvdi.a/v_cellarray**)

Draw a table of colored cells

#include <aesbind.h>
#include <vdibind.h>
void v_cellarray(*handle, xyarray, rowlength, cells, rows, mode, cellarray***)**
int *handle, xyarray, rowlength*[4], *cells, rows, mode, cellarray*[*n*];

v_cellarray is a VDI routine that draws a table of colored cells. *handle* is the virtual device's VDI handle. *xyarray* gives the X and Y coordinates for the rectangle in which the table will be drawn. Note that these values will vary, depending on whether the device is set to normalized device coordinates (NDC) or raster coordinates (RC). On NDC devices, *xyarray[0]* and *xyarray[1]* give, respectively, the X and Y coordinates of the lower left-hand corner of the rectangle, whereas *xyarray[2]* and *xyarray[3]* give the coordinates for the upper right-handle corner. On RC devices, *xyarray[0]* and *xyarray[1]* give, respectively, the X and Y coordinates of the upper left-hand corner, whereas *xyarray[2]* and *xyarray[3]* give the X and Y coordinates of the lower right-hand corner.

rowlength gives the horizontal length of the table to be shown, in NDCs or RCs. *cells* is the number of cells to be drawn in each row, and *rows* is the number of rows of cells to draw. *mode* is the writing mode in which the cells will be drawn: one indicates replace mode; two, transparent mode; three, XOR (exclusive or); and four, reverse transparent mode.

Finally, *cellarray* gives the array of colors to be shown in the cells. *n* must be equal to cells times rows.

See Also
TOS, VDI, vq_cellarray

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

v_circle—VDI function (**libvdi.a/v_circle**)

Draw a circle

#include <aesbind.h>
#include <vdibind.h>
void v_circle(*handle, xcoord, ycoord, radius***) int** *handle, xcoord, ycoord, radius*;

v_circle is a VDI routine that draws a circle. *handle* is the virtual device's VDI handle. *xcoord* and *ycoord* give, respectively, the X and Y coordinates of the circle's center. *radius* gives the circle's radius. These measurements will vary, depending on whether the device has been defined as using normalized device coordinates (NDC) or raster coordinates (RC).

Example
The following program, called **circle.c**, draws a circle on screen. The first mouse click sets the circle's center; the second mouse click sets its radius. The 'W' key cycles through the available write modes, for truly psychedelic effects. Pressing **<return>** exits. Compile it with the command line

```
cc -V -VGEM circle.c -lm
```

to include the necessary mathematics routine.

```
#include <gemdefs.h>
#include <aesbind.h>
#include <vdibind.h>

#define ASTERISK 3                      /* code for drawing an asterisk marker */
#define UP 0                            /* mouse button is up */
#define DOWN 1                          /* mouse button is down */
#define W_KEY 0x1177                    /* scan code returned by W key */
#define RETURN 0x1C0D                   /* scan code returned by <return> key */
#define XOR 3                           /* XOR mode for writing mouse pointer */

/* global line A variables used by vdi; MUST be included */
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

/* array used to calculate radius */
int xyarray[4];

/* array used by v_pmarker() */
int xymarker[2];

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 639, 399 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* throw-away declarations, to keep system from scribbling over itself */
int nowhere = 0;
Rect norect = { 0, 0, 0, 0 };

main() {
/* declarations used by evnt_multi() */
        int selection;                  /* code for event that occurred */
        unsigned int which = (MU_KEYBD | MU_BUTTON);
        int clicks = 1;                 /* no. of clicks expected on mouse button */
        int button = 1;                 /* which button; 1 = leftmost */
        int buffer[11];                 /* place to write AES messages */
        int mousex;                     /* mouse X coordinate */
        int mousey;                     /* mouse Y coordinate */
        unsigned key;                   /* key typed by user */
```

```
/* misc declarations */
	int vdihandle;			/* virtual device's handle */
	int writectr = 0;		/* used to cycle through write modes */
	int fillctr = 1;		/* used to cycle through circle fill styles */
	int n = 0;			/* used to keep track of xyarray */

/* OK, here we go ... */
	appl_init();
	graf_mouse(ARROW, &nowhere);
	vdihandle = graf_handle(&nowhere, &nowhere, &nowhere, &nowhere);
	v_opnvwk(work_in, &vdihandle, work_out);
	vs_clip(vdihandle, 1, cliparray);

	vsf_interior(vdihandle, 2);
	vsf_perimeter(vdihandle, 1);
	vsm_height(vdihandle, 3);
	vsm_type(vdihandle, ASTERISK);

	for(;;) {
		selection = evnt_multi(which, clicks, button, DOWN,
			0, norect, 0, norect, buffer, 0, 0, &mousex, &mousey,
			&nowhere, &nowhere, &key, &nowhere);

		switch(selection) {
		case MU_KEYBD:
			if (key == RETURN) {
				v_clsvwk(vdihandle);
				appl_exit();
				exit(0);
			}
			if (key == W_KEY)
				writectr++;
			break;

		case MU_BUTTON:
			evnt_button(clicks, button, UP,
				&nowhere, &nowhere, &nowhere, &nowhere);
			if (n == 0) {
				/* draw center marker in XOR mode */
				xymarker[0] = mousex;
				xymarker[1] = mousey;
				graf_mouse(M_OFF, &nowhere);
				vswr_mode(vdihandle, XOR);
				v_pmarker(vdihandle, 1, xymarker);
				graf_mouse(M_ON, &nowhere);
			}
```

```
                xyarray[n++] = mousex;
                xyarray[n++] = mousey;
                if (n > 3) {
                        n = 0;
                        fillctr++;
                        /* XOR-away the center marker ... */
                        v_pmarker(vdihandle, 1, xymarker);
                        /* ... and set new drawing mode */
                        vswr_mode(vdihandle, (writectr%4)+1);

                        vsf_style(vdihandle, (fillctr%24)+1);
                        drawcircle(vdihandle);
                }
                break;

        default:
                break;
        }
    }
}

drawcircle(handle)
int handle;
{
        int leg1;                       /* first leg of triangle to compute radius */
        int leg2;                       /* second leg */
        int radius;                     /* radius of circle = hypotenuse */
        extern double hypot();          /* declare hypot() function */

/*
 * Calculate two legs of right triangle, then use Pythagorean theorem
 * to compute hypotenuse, which equals radius of circle to be drawn.
 * Note necessary casts of variables.
 */

        leg1 = abs(xyarray[2] - xyarray[0]);
        leg2 = abs(xyarray[3] - xyarray[1]);
        radius = (int) hypot( (double) leg1, (double) leg2);

/* now, draw the circle */
        graf_mouse(M_OFF, &nowhere);
        v_circle(handle, xyarray[0], xyarray[1], radius);
        graf_mouse(M_ON, &nowhere);
        return;
}
```

See Also
TOS, v_ellipse, VDI

v_clear_disp_list—VDI function (libvdi.a/v_clear_disp_list)
Clear a printer's display list

#include <aesbind.h>
#include <vdibind.h>
void v_clear_disp_list(*handle***) int** *handle***;**

v_clear_disp_list is a VDI routine that clear's a printer's display list. Unlike the related function **v_clrwk**, it does not set a new page.

See Also
TOS, v_form_adv, v_clrwk, VDI

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

v_clrwk—VDI function (**libvdi.a/v_clrwk**)
Clear the virtual workstation
#include <aesbind.h>
#include <vdibind.h>
void v_clrwk(*handle***) int** *handle***;**

v_clrwk is a VDI routine that clears the virtual workstation. It is executed automatically after a device is opened. It clears the screen device by setting it to the background color, and clears a hard-copy device (e.g., printer, plotter) by sending a new-page signal. *handle* is the device's VDI handle.

Example
For an example of this function, see the entry for **v_gtext**.

See Also
TOS, v_clear_disp_list, v_form_adv, VDI

v_clsvwk—VDI function (**libvdi.a/v_clsvwk**)
Close the screen virtual device
#include <aesbind.h>
#include <vdibind.h>
void v_clsvwk(*handle***) int** *handle***;**

v_clsvwk is a VDI routine that closes the screen virtual device. It also flushes all appropriate buffers, frees the space assigned to the screen's device driver, and otherwise performs other tasks to ensure that the device is closed gracefully. *handle* is the screen's VDI handle.

Example
For an example of this routine, see the entry for **v_pline**.

See Also
TOS, **VDI**, **v_clswk**, **v_opnvwk**, **v_opnwk**

v_clswk—VDI function (**libvdi.a/v_clswk**)
Close a virtual workstation
#include <aesbind.h>
#include <vdibind.h>
void v_clswk(*handle***) int** *handle*;

v_clswk is a VDI routine that closes a virtual workstation. It also flushes the any associated buffers and frees the memory allocated to the workstation's driver, to conclude matters gracefully. *handle* is the device's VDI handle.

See Also
TOS, **VDI**, **v_opnvwk**, **v_opnwk**

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI. To close the screen device, use the related function **v_clsvwk**.

v_contourfill—VDI function (**libvdi.a/v_contourfill**)
Fill an outlined area
#include <aesbind.h>
#include <vdibind.h>
void v_contourfill(*handle, xcoord, ycoord, color***)**
int *handle, xcoord, ycoord, color*;

v_contourfill is a VDI routine that fills an outlined area with a fill pattern. Note that the fill type must be set by the function **vsf_interior**, and the fill style by the function **vsf_style**.

handle is the virtual device's VDI handle. *xcoord* and *ycoord* give, respectively, the X and Y coordinates of the point at which filling is to begin. Finally, *color* is the code for the color at which filling stops. For a table of color settings, see the entry for **v_opnwk**.

Example
The following example lets the user draw a number of "rubber lines" on the screen. The 'W' key floods an enclosed area with the fill pattern. Pressing **<return>** exits.

```
#include <gemdefs.h>
#include <aesbind.h>
#include <vdibind.h>
```

```
#define RETURN 0x1C0D                    /* scan code returned by return key */
#define W_KEY 0x1177                     /* scan code returned by W key */
#define UP 0                             /* mouse button is up */
#define DOWN 1                           /* mouse button is down */
#define CLICKS 1                         /* no. of clicks expected on mouse button */
#define BUTTON 1                         /* which button; 1 = leftmost */
#define XOR 3                            /* make writing mode XOR */
#define REPLACE 1                        /* make writing mode REPLACE */
#define FUJI 4                           /* set fill type to little "fujis" */
#define BLACK 1                          /* code for color black */

/* global line A variables used by vdi; MUST be included */
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

/*
 * array used by vs_clip(); MUST be set, or images that extend
 * beyond the screen perimeters will write over low-level memory
 * (e.g., RAM disks, spoolers, etc.)
 */
int cliparray[] = { 1, 1, 639, 399 };

/* array used by evnt_multi(), for writing AES messages */
int buffer[11];

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* array used by v_pline() */
int xyarray[4];

/* throw-away declarations, to keep system from scribbling over itself */
int nowhere = 0;
Rect norect = { 0, 0, 0, 0 };

main() {
/* declarations used by evnt_multi() */
        int selection;                   /* code for event that occurred */
        unsigned key;                    /* scan code of key pressed by user */
        int mousex;                      /* mouse X coordinate */
        int mousey;                      /* mouse Y coordinate */
        int vdihandle;                   /* virtual device's handle */
        int flag = 0;                    /* has line been drawn yet? */

/* OK, here we go ... */
        appl_init();
        graf_mouse(ARROW, &nowhere);
        vdihandle = graf_handle(&nowhere, &nowhere, &nowhere, &nowhere);
        v_opnvwk(work_in, &vdihandle, work_out);
```

```
vs_clip(vdihandle, 1, cliparray);
vsf_interior(vdihandle, FUJI);
vswr_mode(vdihandle, XOR);

    for(;;) {
        selection = evnt_multi(MU_KEYBD | MU_BUTTON, CLICKS, BUTTON,
                DOWN, 0, norect, 0, norect, buffer, 0, 0, &mousex,
                &mousey, &nowhere, &nowhere, &key, &nowhere);

        switch(selection) {
        case MU_KEYBD:
                if (key == RETURN) {
                        v_clsvwk(vdihandle);
                        appl_exit();
                        exit(0);
                }

                if (key == W_KEY) {
                        graf_mouse(M_OFF, &nowhere);
                        v_contourfill(vdihandle, mousex, mousey, BLACK);
                        graf_mouse(M_ON, &nowhere);
                }
                break;

        case MU_BUTTON:
                /* "rubberline" routine */
                if (flag > 0) {
                        /* if line has moved ... */
                        if ((xyarray[2] != mousex) || (xyarray[3] != mousey)) {

                                /* ... undraw old line ... */
                                graf_mouse(M_OFF, &nowhere);
                                v_pline(vdihandle, 2, xyarray);
                                graf_mouse(M_ON, &nowhere);

                                /* ... change far endpoint ... */
                                xyarray[2] = mousex;
                                xyarray[3] = mousey;

                                /* ... and draw new line */
                                graf_mouse(M_OFF, &nowhere);
                                v_pline(vdihandle, 2, xyarray);
                                graf_mouse(M_ON, &nowhere);
                        }
                }
```

```
                        if (flag == 0) {
                                /* redraw line in REPLACE mode */
                                vswr_mode(vdihandle, REPLACE);
                                graf_mouse(M_OFF, &nowhere);
                                v_pline(vdihandle, 2, xyarray);
                                graf_mouse(M_ON, &nowhere);
                                vswr_mode(vdihandle, XOR);

                                /* reset endpoints */
                                xyarray[0] = mousex;
                                xyarray[1] = mousey;
                                xyarray[2] = mousex;
                                xyarray[3] = mousey;
                                flag = 1;
                        }
                        flag = check();
                        break;

                default:
                        break;
                }
        }
}

check() {
        int buttonstate = 1;                    /* button state */

        evnt_multi(MU_TIMER, CLICKS, BUTTON,
                UP, 0, norect, 0, norect, buffer, 0, 0, &nowhere,
                &nowhere, &buttonstate, &nowhere, &nowhere, &nowhere);

        return(buttonstate);
}
```

See Also
TOS, VDI

Notes
Due to the way the AES routine reads the mouse buttons, the example will not always notice that the mouse button has returned to the up position.

v_curdown—VDI function (**libvdi.a/v_curdown**)
Move text cursor down one row
#include <aesbind.h>
#include <vdibind.h>
void v_curdown(*handle*) int *handle*;

v_curdown is a VDI routine that moves the text cursor down one row. It does not affect the cursor's horizontal position. Note that the virtual device must first be put into text mode with the function **v_enter_cur** before this function can be used. *handle* is the virtual device's VDI handle.

See Also
TOS, **v_curhome, v_curleft, v_curright, v_curup, VDI**

v_curhome—VDI function (**libvdi.a/v_curhome**)
Move text cursor to the home position
#include <aesbind.h>
#include <vdibind.h>
void v_curhome(*handle*) int *handle*;

v_curhome is a VDI routine that moves the text cursor to the home position, i.e., to the upper left-hand corner. Note that the virtual device must first be put into text mode with the function **v_enter_cur** before this function can be used. *handle* is the virtual device's VDI handle.

See Also
TOS, **v_curdown, v_curleft, v_curright, v_curhome, VDI**

v_curleft—VDI function (**libvdi.a/v_curleft**)
Move text cursor left one column
#include <aesbind.h>
#include <vdibind.h>
void v_curleft(*handle*) int *handle*;

v_curleft is a VDI routine that moves the text cursor one column to the left. It does not affect the cursor's vertical position. Note that the virtual device must first be put into text mode with the function **v_enter_cur** before this function can be used. *handle* is the virtual device's VDI handle.

See Also
TOS, **v_curdown, v_curhome, v_curright, v_curup, VDI**

v_curright—VDI function (**libvdi.a/v_curright**)
Move text cursor right one column
#include <aesbind.h>
#include <vdibind.h>
void v_curright(*handle*) int *handle*;

v_curright is a VDI routine that moves the text cursor one column to the right. It does not affect the cursor's vertical position. Note that the virtual device must first be put into text mode with the function **v_enter_cur** before this function can be used. *handle* is the virtual device's VDI handle.

See Also
TOS, v_curdown, v_curhome, v_curleft, v_curup, VDI

v_curtext—VDI function (**libvdi.a/v_curtext**)
Write alphabetic text
#include <aesbind.h>
#include <vdibind.h>
void v_curtext(*handle, string***) int** *handle*; **char** ***string**;

v_curtext is a VDI routine that writes alphabetic text on the virtual device. Note that to use this routine, the virtual device must first be placed in text mode, using the routine **v_enter_cur**. *handle* is the virtual device's VDI handle. *string* points to the NUL-terminated string of alphabetic characters to be written.

See Also
TOS, VDI

v_curup—VDI function (**libvdi.a/v_curup**)
Move text cursor up one row
#include <aesbind.h>
#include <vdibind.h>
void v_curup(*handle***) int** *handle*;

v_curup is a VDI routine that moves the text cursor up one row. It does not affect the cursor's horizontal position. Note that the virtual device must first be put into text mode with the function **v_enter_cur** before this function can be used. *handle* is the virtual device's VDI handle.

See Also
TOS, v_curdown, v_curhome, v_curleft, v_curright, VDI

v_dspcur—VDI function (**libvdi.a/v_dspcur**)
Move mouse pointer to point on screen
#include <aesbind.h>
#include <vdibind.h>
void v_dspcur(*handle, xcoord, ycoord***) int** *handle, xcoord, ycoord*;

v_dspcur is a VDI routine that moves the mouse pointer to a specified point on the screen. *handle* is the virtual device's VDI handle. *xcoord* and *ycoord* are, respectively, the X and Y coordinates to which the mouse cursor will be moved.

See Also
TOS, VDI

v_eeol—VDI function (**libvdi.a/v_eeol**)

Erase text from cursor to end of screen

#include <aesbind.h>
#include <vdibind.h>
void **v_eeol(*handle*) int** *handle*;

v_eeol is a VDI routine that erases alphabetic text from the position of the text cursor to the end of the line. Note that the virtual device must first be put into text mode with the function **v_enter_cur** before this function can be used. *handle* is the virtual device's VDI handle.

See Also
TOS, **v_eeos, VDI**

v_eeos—VDI function (**libvdi.a/v_eeos**)

Erase from text cursor to end of screen

#include <aesbind.h>
#include <vdibind.h>
void **v_eeos(*handle*) int** *handle*;

v_eeos is a VDI routine that erases a virtual device from the position of the text cursor to the end. Note that the virtual device must first be put into text mode, with the function **v_enter_cur** before this function can be used. *handle* is the virtual device's VDI handle.

See Also
TOS, **v_eeol, VDI**

v_ellarc—VDI function (**libvdi.a/v_ellarc**)

Draw an elliptical arc

#include <aesbind.h>
#include <vdibind.h>
void **v_ellarc(*handle, xcoord, ycoord, xradius, yradius, beginangle, endangle*)**
int *handle, xcoord, ycoord, xradius, yradius, beginangle, endangle*;

v_ellarc is a VDI routine that draws an elliptical arc. *handle* is the virtual device's VDI handle. *xcoord* and *ycoord* give, respectively, the X and Y coordinates of the center of the imaginary ellipse of the curve being drawn is a part. *xradius* is the horizontal radius of the ellipse, and *yradius* is the vertical radius. Note that all of these values will vary, depending on whether the virtual device uses normalized device coordinates (NDC) or raster coordinates (RC). Finally, *beginangle* and *endangle* represent the beginning and end angles of the ellipse, in tenths of a degree. On an imaginary clock, zero degrees is at 3 o'clock, 90 degrees at noon, 180 degrees at 9 o'clock, and 270 degrees at 6 o'clock.

Example
The following program uses STDIO routines to create a "rough-and-ready" dialogue; the user sets the X radius, the Y radius, the beginning angle, and the end angle, which are then used to draw an elliptical arc on the screen.

```
#include <gemdefs.h>
#include <aesbind.h>
#include <vdibind.h>

#define ESCAPE 0x1B                    /* scan code returned by escape key */
#define ROUNDED 2                      /* put rounded ends on lines */
#define REPLACE 1                      /* code for REPLACE writing mode */
#define XOR 3                          /* code for XOR writing mode */

/* global line A variables used by vdi; MUST be included */
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

/*
 * array used by vs_clip(); MUST be set, or images that extend
 * beyond the screen perimeters will write over low-level memory
 * (e.g., RAM disks, spoolers, etc.)
 */
int cliparray[] = { 1, 1, 639, 399 };

/* arrays used by drawline() */
xyvert[] = { 320, 1, 320, 399 };
xyhoriz[] = { 1, 200, 639, 200 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* throw-away declaration, to keep system from scribbling over itself */
int nowhere = 0;

main() {
        unsigned key;                  /* key returned by user */
        int vdihandle;                 /* virtual device's handle */
        int xradius;                   /* length of X radius */
        int yradius;                   /* length of Y radius */
        int beginangle;                /* beginning angle */
        int endangle;                  /* end angle */

/* OK, here we go ... */
        appl_init();
        graf_mouse(M_OFF, &nowhere);
        vdihandle = graf_handle(&nowhere, &nowhere, &nowhere, &nowhere);
        v_opnvwk(work_in, &vdihandle, work_out);
        vs_clip(vdihandle, 1, cliparray);
        vsl_ends(vdihandle, ROUNDED, ROUNDED);
```

```
	for(;;) {
		printf("Type <return> to continue, <esc> to exit.\n");
		key = evnt_keybd();
		switch((char)key) {

		case ESCAPE:
			v_clsvwk(vdihandle);
			appl_exit();
			exit(0);

		case '\n':
			drawlines(vdihandle);
		/* Enter X radius */
			xradius  = getdata("Enter X radius (screen, 0-320)");
			xyhoriz[0] = 320 - xradius;
			xyhoriz[2] = 320 + xradius;
			drawlines(vdihandle);

		/* Enter Y radius */
			yradius = getdata("Enter Y radius (screen, 0-200)");
			xyvert[1] = 200 - yradius;
			xyvert[3] = 200 + yradius;
			drawlines(vdihandle);

		/* Enter beginning angle */
			beginangle = getdata("Enter beginning angle (0-360)") * 10;
			drawlines(vdihandle);

		/* Enter end angle */
			endangle = getdata("Enter end angle (0-360)") * 10;
			drawlines(vdihandle);

		/* And now, draw the elliptical arc */
			vsl_width(vdihandle, 5);
			v_ellarc(vdihandle, 320, 200, xradius, yradius,
				beginangle, endangle);
			break;

		default:
			break;
		}
	}
}
```

```
drawlines(handle)
int handle;
{
	printf("\033E\n");
	vsl_width(handle, 1);
	v_pline(handle, 2, xyvert);
	v_pline(handle, 2, xyhoriz);
	return;
}

getdata(message)
char *message;
{

	for(;;) {
		char string[20];		/* string used with user input */
		int value;			/* value user intended */

		   printf("%s: ", message);
		fflush(stdout);
		if((value = atoi(gets(string))) >= 0)
			return(value);
	}
}
```

See Also
TOS, VDI, v_ellipse

v_ellipse—VDI function (**libvdi.a/v_ellipse**)
Draw an ellipse
#include <aesbind.h>
#include <vdibind.h>
void v_ellipse(*handle, xcoord, ycoord, xradius, yradius***)**
int *handle, xcoord, ycoord, xradius, yradius*;

v_ellipse is a VDI routine that draws an ellipse. *handle* is the virtual device's VDI handle. *xcoord* and *ycoord* give, respectively, the X coordinates and Y coordinates of the ellipse's center. Note that these measurements will change, depending on whether the virtual device is set to normalized device coordinates (NDC) or raster coordinates (RC). Finally, *xradius* gives the ellipse's horizontal radius, and *yradius* gives its vertical radius.

Example
The following example draws ellipses on the screen. Clicking the mouse draws a rubber box; releasing the mouse fixes the box, whose dimensions are used to calculate the ellipse. Pressing the 'W' key cycles through the available write modes, for truly psychedelic effects. Pressing **<return>** exits.

```
#include <gemdefs.h>
#include <aesbind.h>
#include <vdibind.h>

#define DOWN 1                          /* mouse button is down */
#define W_KEY 0x1177                    /* scan code returned by W key */
#define RETURN 0x1C0D                   /* scan code returned by <return> key */

/* global line A variables used by vdi; MUST be included */
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 639, 399 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* throw-away declarations, to keep system from scribbling over itself */
int nowhere = 0;
Rect norect = { 0, 0, 0, 0 };

main() {
/* declarations used by evnt_multi() */
      int selection;                    /* code for event that occurred */
      unsigned int which = (MU_KEYBD | MU_BUTTON);
      int clicks = 1;                   /* no. of clicks expected on mouse button */
      int button = 1;                   /* which button; 1 = leftmost */
      int buffer[11];                   /* place to write AES messages */
      int mousex;                       /* mouse X coordinate */
      int mousey;                       /* mouse Y coordinate */
      unsigned key;                     /* key typed by user */

/* misc declarations */
      int vdihandle;                    /* virtual device's handle */
      int writectr = 0;                 /* used to cycle through write modes */
      int fillctr = 1;                  /* used to cycle through circle fill styles */
      int width;                        /* box width set by graf_rubbox */
      int height;                       /* box height set by graf_rubbox */
      int xcoord;                       /* X coordinate of ellipse's center */
      int ycoord;                       /* Y coordinate of ellipse's center */
      int xradius;                      /* X radius of ellipse */
      int yradius;                      /* Y radius of ellipse */

/* OK, here we go ... */
      appl_init();
      graf_mouse(ARROW, &nowhere);
      vdihandle = graf_handle(&nowhere, &nowhere, &nowhere, &nowhere);
      v_opnvwk(work_in, &vdihandle, work_out);
      vs_clip(vdihandle, 1, cliparray);
```

```
	vsf_interior(vdihandle, 2);
	vsf_perimeter(vdihandle, 1);
	vsm_height(vdihandle, 3);

	for(;;) {
		selection = evnt_multi(which, clicks, button, DOWN,
			0, norect, 0, norect, buffer, 0, 0, &mousex, &mousey,
			&nowhere, &nowhere, &key, &nowhere);

		switch(selection) {
		case MU_KEYBD:
			if (key == RETURN) {
				v_clsvwk(vdihandle);
				appl_exit();
				exit(0);
			}

			if (key == W_KEY) {
				writectr++;
				vswr_mode(vdihandle, (writectr%4)+1);
			}
			break;

		case MU_BUTTON:
			fillctr++;
			vsf_style(vdihandle, (fillctr%24)+1);
			graf_rubbox(mousex, mousey, 0, 0, &width, &height);
			xcoord = mousex+(width/2);
			ycoord = mousey+(height/2);
			xradius = width/2;
			yradius = height/2;
			v_ellipse(vdihandle, xcoord, ycoord, xradius, yradius);
			break;

		default:
			break;
		}
	}
}
```

See Also
TOS, **VDI**, **v_ellarc**, **v_ellpie**

Notes
v_ellipse can only create ellipses that are oriented horizontally or vertically. It cannot create ellipses that are oriented diagnonally.

v_ellpie—VDI function (**libvdi.a/v_ellpie**)
Draw an elliptical pie slice
#include <aesbind.h>
#include <vdibind.h>

void **v_ellpie(***handle, xcoord, ycoord, xradius, yradius, beginangle, endangle***)**
int *handle, xcoord, ycoord, xradius, yradius, beginangle, endangle*;

v_ellpie is a VDI routine that draws an elliptical pie slice. *handle* is the virtual device's VDI handle. *xcoord* and *ycoord* give, respectively, the X and Y coordinates of the imaginary ellipse of which **v_ellpie** draws a part. *xradius* gives the imaginary ellipse's horizontal radius, and *yradius* gives its vertical radius. Finally, *beginangle* and *endangle* give, respectively, the beginning and end angles of the pie slice, in tenths of a degree. On an imaginary clock, zero degrees is at 3 o'clock, 90 degrees at noon, 180 degrees at 9 o'clock, and 270 degrees at 6 o'clock.

See Also
TOS, VDI, v_ellipse

v_enter_cur—VDI function (libvdi.a/v_enter_cur)

Enter text mode
#include <aesbind.h>
#include <vdibind.h>
void **v_enter_cur(***handle***)** int *handle*;

v_enter_cur is a VDI routine that moves a virtual device into text mode. It hides the mouse pointer and draws the text cursor. *handle* is the virtual device's VDI handle.

See Also
TOS, v_exit_cur, VDI

v_exit_cur—VDI function (libvdi.a/v_exit_cur)

Exit from text mode
#include <aesbind.h>
#include <vdibind.h>
void **v_exit_cur(***handle***)** int *handle*;

v_exit_cur is a VDI routine that forces a virtual device to exit from text mode and return to graphics mode, should these modes be separate on that device. It removes the text cursor from the device and restores the mouse pointer, should the virtual device support them. *handle* is the virtual device's VDI handle.

See Also
TOS, v_enter_cur, VDI

v_fillarea—VDI function (libvdi.a/v_fillarea)

Draw a complex polygon
#include <aesbind.h>

#include <vdibind.h>
void v_fillarea(*handle, count, xyarray*) int *handle, count, xyarray*[*n*];

v_fillarea is a VDI routine that draws and fills a complex polygon. Note that to use the full power of this routine, you must first set the fill type with **vsf_interior**, the fill style with **vsf_style**, and the fill color with **vsf_color**.

handle is the virtual device's VDI handle. *count* is the number of corners on the polygon. *xyarray* gives the X and Y coordinates for each of the corners: all of the even-numbered entries hold X coordinates, and all the odd-numbered entries hold Y coordinates. Note that the value of *n* must be exactly double that of *count*.

Example
The following program draws a filled polygon on screen. Use mouse to set markers on the screen, with a maximum of 40 points. Pressing **<esc>** "connects the dots" to draw and fill the polygon. Pressing the 'T' key cycles through types of fill; pressing the 'S' key cycles through styles of fill. Pressing **<return>** exits.

```
#include <gemdefs.h>
#include <aesbind.h>
#include <vdibind.h>

#define RETURN 0x1C0D                    /* scan code for <return> key */
#define ESC 0x11B                        /* scan code for <esc> key */
#define T_KEY 0x1474                     /* scan code for T key */
#define S_KEY 0x1F73                     /* scan code for S key */
#define ASTERISK 3

/* global line A variables used by vdi; MUST be included */
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

/* array used by v_pmarker() */
int xymarker[2];

/* array used by v_fillarea() */
int xypoly[80];

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 639, 399 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* throw-away declarations, to keep system from scribbling over itself */
int nowhere = 0;
Rect norect = { 0, 0, 0, 0 };
```

```
main() {
/* declarations used by evnt_multi() */
        int selection;                          /* code for event that occurred */
        unsigned int which = (MU_KEYBD | MU_BUTTON);
        int clicks = 1;                         /* no. of clicks expected on mouse button */
        int button = 1;                         /* which button; 1 = leftmost */
        int buttonstate = 1;                    /* button state expected; 1 = down */
        int buffer[11];                         /* place to write AES messages */
        int mousex;                             /* mouse X coordinate */
        int mousey;                             /* mouse Y coordinate */
        unsigned key;                           /* key typed by user */

/* misc declarations */
        int vdihandle;                          /* virtual device's handle */
        int type = 0;                           /* type of fill */
        int style = 1;                          /* style of fill */
        int n = 0;                              /* used with xyarray[] */
        int flag = 0;                           /* has polygon been drawn yet? */

/* OK, here we go ... */
        appl_init();
        graf_mouse(ARROW, &nowhere);
        vdihandle = graf_handle(&nowhere, &nowhere, &nowhere, &nowhere);
        v_opnvwk(work_in, &vdihandle, work_out);
        vs_clip(vdihandle, 1, cliparray);

        vsm_height(vdihandle, 3);
        vsm_type(vdihandle, ASTERISK);

        for(;;) {
                selection = evnt_multi(which, clicks, button, buttonstate,
                        0, norect, 0, norect, buffer, 0, 0, &mousex, &mousey,
                        &nowhere, &nowhere, &key, &nowhere);

                switch(selection) {
                case MU_KEYBD:
                        switch(key) {
                        case RETURN:
                                v_clsvwk(vdihandle);
                                appl_exit();
                                exit(0);

                        case ESC:
                                graf_mouse(M_OFF, &nowhere);
                                v_fillarea(vdihandle, n/2, xypoly);
                                graf_mouse(M_ON, &nowhere);
                                flag = 1;
                                break;
```

```
                case T_KEY:
                        if (flag == 0) {
                                break;
                        } else {
                                type = (++type%5);
                                vsf_interior(vdihandle, type);
                                graf_mouse(M_OFF, &nowhere);
                                v_fillarea(vdihandle, n/2, xypoly);
                                graf_mouse(M_ON, &nowhere);
                        }
                        break;

                case S_KEY:
                        if (flag == 0) {
                                break;
                        } else {
                                style = ((++style%24)+1);
                                vsf_style(vdihandle, style);
                                graf_mouse(M_OFF, &nowhere);
                                v_fillarea(vdihandle, n/2, xypoly);
                                graf_mouse(M_ON, &nowhere);
                        }
                        break;
                }
                break;

        case MU_BUTTON:
                if (flag > 0) {
                        n = 0;
                        flag = 0;
                }
                xymarker[0] = mousex;
                xymarker[1] = mousey;
                if (n <= 79) {
                        xypoly[n] = mousex;
                        n++;
                        xypoly[n] = mousey;
                        n++;
                }

                graf_mouse(M_OFF, &nowhere);
                v_pmarker(vdihandle, ASTERISK, xymarker);
                graf_mouse(M_ON, &nowhere);
                break;
        default:
                break;
        }
    }
}
```

See Also
TOS, **v_bar**, **VDI**, **vr_recfl**

v_form_adv—VDI function (**libvdi.a/v_form_adv**)
Advance the page on a printer
#include <aesbind.h>
#include <vdibind.h>
void **v_form_adv**(*handle*) **int** *handle*;

v_form_adv is a VDI routine that advances the page on a printer. Unlike the related function **v_clrwk**, **v_form_adv** does not erase material that has not yet been written onto the printer.

See Also
TOS, **v_clear_disp_list**, **v_clrwk**, **VDI**

v_get_pixel—VDI function (**libvdi.a/v_get_pixel**)
See if a given pixel is set
#include <aesbind.h>
#include <vdibind.h>
void **v_get_pixel**(*handle*, *xcoord*, *ycoord*, *&flag*, *&color*)
int *handle*, *xcoord*, *ycoord*, *flag*, *color*;

v_get_pixel is a VDI routine that indicates whether or not a pixel is set. *handle* is the virtual device's VDI handle. *xcoord* and *ycoord* are, respectively, the X and Y coordinates of the pixel in question. *flag* is set by **v_get_pixel**; zero indicates that the pixel is not set, whereas one indicates that it is set. Finally, *color* is set by **v_get_pixel**; if the pixel is set, this variable holds the code of the color to which it is set. For a table of color codes, see the entry for **v_opnwk**.

See Also
TOS, **VDI**

v_gtext—VDI function (**libvdi.a/v_gtext**)
Draw graphics text
#include <aesbind.h>
#include <vdibind.h>
void **v_gtext**(*handle*, *xcoord*, *ycoord*, *text*) **int** *text*, *xcoord*, *ycoord*; **char** **text*;

v_gtext is a VDI routine that draws graphics text on the screen. *handle* is the virtual device's VDI handle. *xcoord* and *ycoord* are, respectively, the X and Y coordinates of the point on the screen where the drawing of the string will begin. Note that these values will change, depending upon the virtual device has been set to normalized device coordinates (NDC) or raster coordinates (RC). Finally, *text* points to the string to drawn.

The font of the string drawn, its size, its color, the angle at which it is displayed, and the manner of its alignment can all be set with separate VDI calls. See the entries given below for more information.

Example
The following example draws cross-hairs on the screen, and then aligns the string "Mark Williams C" against them. Pressing the 'E' key cycles through the available special effects; 'H', the available horizontal alignments; 'R', the text rotation; 'S', the available font sizes; and 'V', the vertical alignments. Typing **<return>** exits from the program

```
#include <gemdefs.h>
#include <aesbind.h>
#include <vdibind.h>

#define RETURN 0x1C0D                 /* scan code returned by return key */
#define E_KEY 0x1265                  /* scan code returned by E key */
#define H_KEY 0x2368                  /* scan code returned by H key */
#define R_KEY 0x1372                  /* scan code returned by R key */
#define S_KEY 0x1F73                  /* scan code returned by S key */
#define V_KEY 0x2F76                  /* scan code returned by V key */
#define RESERVED 0                    /* used by system for its own purposes */
#define ESCAPE 0x1B                   /* ASCII code for <esc> */

/* global line A variables used by vdi; MUST be included */
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 639, 399 };

/* arrays used by drawline() */
xyvert[] = { 320, 1, 320, 399 };
xyhoriz[] = { 1, 200, 639, 200 };

/* string used by drawtext() */
char *text = "Mark Williams C";

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* throw-away declaration, to keep system from scribbling over itself */
int nowhere = 0;

main() {
        unsigned key;                 /* key typed by user */
        int vdihandle;                /* virtual device's handle */
        int size = 1;                 /* text's size, in rasters */
        int effect = 1;               /* text's special effect used */
        int halign = 0;               /* text's horizontal alignment */
        int valign = 0;               /* text's vertical alignment */
        int angle = 0;                /* angle at which text is drawn */
```

```
/* OK, here we go ... */
	appl_init();
	graf_mouse(M_OFF, &nowhere);
	vdihandle = graf_handle(&nowhere, &nowhere, &nowhere, &nowhere);
	v_opnvwk(work_in, &vdihandle, work_out);
	vs_clip(vdihandle, 1, cliparray);
	drawtext(vdihandle);

	for(;;) {
		key = evnt_keybd();
		switch(key) {

		case RETURN:
			graf_mouse(M_ON, &nowhere);
			v_clsvwk(vdihandle);
			appl_exit();
			exit(0);

		case E_KEY:
			vst_effects(vdihandle, effect);
			if (++effect > 32)
				effect = 1;
			drawtext(vdihandle);
			break;

		case H_KEY:
			halign++;
			vst_alignment(vdihandle, (halign%3), (valign%6),
				&nowhere, &nowhere);
				/* legal H value 0-2, V value 0-5 */
			drawtext(vdihandle);
			break;

		case R_KEY:
			/* Note: ST draws text only at right angles */
			angle += 900;
			if (angle > 3500)
				angle = 0;
			vst_rotation(vdihandle, angle);
			drawtext(vdihandle);
			break;

		case S_KEY:
			vst_height(vdihandle, size, &nowhere,
				&nowhere, &nowhere, &nowhere);
				/* character size in rasters */
			if (++size > 26)
				size = 1;
			drawtext(vdihandle);
			break;
```

```
                case V_KEY:
                        valign++;
                        vst_alignment(vdihandle, (halign%3), (valign%6),
                                &nowhere, &nowhere);
                                /* legal H value 0-2, V value 0-5 */
                        drawtext(vdihandle);
                        break;

                default:
                        break;
                }
        }
}

drawlines(handle)
int handle;
{
        v_pline(handle, 2, xyvert);
        v_pline(handle, 2, xyhoriz);
        return;
}

drawtext(handle)
int handle;
{
        v_clrwk(handle);
        drawlines(handle);
        v_gtext(handle, 320, 200, text);
        return;
}
```

See Also
TOS, v_justified, VDI, vqt_extent, vqt_name, vqt_width, vst_alignment, vst_color, vst_effects, vst_height, vst_load_fonts, vst_point, vst_rotation, vst_unload_fonts

v_hardcopy—VDI function (**libvdi.a/v_hardcopy**)
Write the screen to a hard-copy device
#include <aesbind.h>
#include <vdibind.h>
void v_hardcopy(*handle*)

v_hardcopy is a VDI routine that writes a copy of the virtual device to a printer or other attached hard-copy device. *handle* is the virtual device's VDI handle.

See Also
TOS, VDI

Notes
The printer must be installed with TOS before this routine will work properly.

v_hide_c—VDI function (**libvdi.a/v_hide_c**)
Hide the mouse pointer
#include <aesbind.h>
#include <vdibind.h>
void v_hide_c(*handle***) int** *handle*;

v_hide_c is a VDI routine that hides the mouse pointer. This routine should be invoked when your program redraws the screen; if the pointer is not hidden, it will leave a grayish patch on the screen when it is moved.

See Also
TOS, v_show_c, VDI

v_justified—VDI function (**libvdi.a/v_justified**)
Justify graphics text
#include <aesbind.h>
#include <vdibind.h>
void v_justified(*handle, xcoord, ycoord, string, length, charsp, wordsp***)**
int *handle, xcoord, ycoord, length, charsp, wordsp*; **char** **string*;

v_justified is a VDI routine that justifies a string a text on a preset line length. *Justification* means that slivers of space are inserted between words or characters to ensure that each string fills exactly the same space. This paragraph is an example of justified text.

handle is, as always, the virtual device's VDI handle. *xcoord* and *ycoord* give, respectively, the X and Y coordinates of the point where the text is to begin printing. *length* is the length to which you want the text set; this value will vary, depending on whether the virtual device is set to normalized device coordinates (NDC) or to raster coordinates (RC). *string* points to the string you want to set. Finally, *charsp* and *wordsp* are flags that indicate whether you want spacing altered between words or characters when performing justification; zero turns off spacing, and one turns it on. Therefore, setting both *charsp* and *wordsp* to zero effectively turns off justification.

Note that if the string is too long to fit into *space*, the characters will overlap.

See Also
TOS, v_gtext, VDI

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

v_meta_extents—VDI function (**libvdi.a/v_meta_extents**)

Update extents header of metafile

#include <aesbind.h>
#include <vdibind.h>
void v_meta_extents(*handle, minx, miny, maxx, maxy***)**
int *handle, minx, miny, maxx, maxy*;

v_meta_extents is a VDI routine that updates the extents header of a metafile. The extents header gives the minimum space needed to draw all of the VDI primitives contained in the metafile; it is used by some routines in allocating space. *handle* is the virtual device's VDI handle. *minx* and *miny* give, respectively, the minimum width and height of the area needed to hold the VDI primitives contained within the metafile; whereas *maxx* and *maxy* give, respectively, the maximum width and height.

See Also
TOS, v_write_meta, VDI, vm_filename

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

If this routine is not used when an item is added to a metafile, the extents parameters will be set to zero.

v_opnvwk—VDI function (**libvdi.a/v_opnvwk**)

Open the virtual screen device

#include <aesbind.h>
#include <vdibind.h>
void v_opnvwk(*work_in, handle, work_out***)**
int *work_in*[11], **handle, work_out*[57];

v_opnvwk is a VDI routine that opens the virtual screen device. *work_in* is an array of 11 integers that must be set before invoking **v_opnvwk**. These are described in the entry for **v_opnwk**.

handle is the device handle for the screen. Because the desktop has already opened the physical screen device, you must use the AES routine **graf_handle** to obtain the VDI handle for the screen, as follows:

```
foo = graf_handle(&xcoord, &ycoord, &width, &height);
```

In this example, *foo* is the VDI handle, which is returned by **graf_handle**; *xcoord*, *ycoord*, *width*, and *height* give the dimensions of a character cell in the screen device, and are set by **graf_handle**.

work_out is an array of 57 integers that are set by **v_opnvwk**. Your program may need to interrogate this array for information; what each integer encodes is described in the entry for **v_opnwk**.

Example
For an example of this routine, see the entry for **v_pline**.

See Also
TOS, v_opnwk, VDI

Notes
At present, device attributes cannot be set through the *work_in* array. With the exception of *work_in[10]*, they are all ignored and should be set to zero. To set device attributes, use the appropriate attribute functions, as listed in the entry for **VDI**.

v_opnwk—VDI function (**libvdi.a/v_opnwk**)
Open a virtual workstation
#include <aesbind.h>
#include <vdibind.h>
void v_opnwk(*work_in, handle, work_out***)**
int *work_in***[11], ****handle, work_out***[57];**

v_opnwk is a VDI routine that opens a virtual workstation. This routine should used to open all virtual workstations *except* the screen, because the screen is already set by GEM when it boots. To open the screen, use **v_opnvwk**.

work_in is an array of 11 integers that must be set before **v_opnwk** is invoked. Their values are as follows:

work_in[0] Device number, as follows:

1	screen
11	plotter
21	printer
31	metafile
41	camera
51	tablet

work_in[1] Line type, as follows:

1	solid
2	long dashes
3	dots
4	dashes plus dots
5	short dashes
6	dash, dot, dot
7	user-defined
8-*n*	device-independent

work_in[2] Line color, as follows:

0	WHITE
1	BLACK
2	RED
3	GREEN
4	BLUE
5	CYAN
6	YELLOW
7	MAGENTA
8	WHITE
9	BLACK
10	LRED
11	LGREEN
12	LBLUE
13	LCYAN
14	LYELLOW
15	LCYAN
16-*n*	device-independent

Note that the names in capital letters are mnemonics that are defined in the header file **obdefs.h**.

work_in[3] Marker type, as follows:

1	dot
2	plus sign
3	asterisk
4	square
5	diagonal cross
6	diamond
7	device-independent

work_in[4] Marker color; same as above.

work_in[5] Text face. These can vary greatly, depending on the device being opened. For a list of the code and names of the the fonts available on a virtual device, use the function **vqt_font_info**.

work_in[6] Text color; same as above.

work_in[7] Fill type, as follows:

0	hollow
1	solid
2	patterned
3	cross-hatched
4	user-defined

work_in[8] Fill style. There are 24 styles of patterned fill, and 12 styles of cross-hatching. See the entry for **vsf_interior** for a program that displays the fill styles.

work_in[9] Fill color; same as above.

work_in[10] Coordinate system. Zero indicates normalized device coordinates (NDC). This is a system in which the screen is divided into a grid of 32,768 by 32,768 points, with the beginning point in the lower left-hand corner. Two indicates raster coordinates (RC). This uses the absolute number of rasters on the screen, counting from the upper left-hand corner of the screen. Note that the number of rasters varies with screen resolution: high resolution is 640 wide by 400 high; medium resolution, 640 wide by 200 high; and low resolution, 320 wide by 200 high. At present, the Atari ST can accept only raster coordinates.

handle is the device's VDI handle, and is set by TOS.

work_out is an array of 57 integers that is filled in by **v_opnwk**, as follows:

0	width of device, in rasters (no. of X coordinates)
1	height of device, in rasters (no. of Y coordinates)
2	uses precision scaling? (0=yes, 1=no)
3	width of one pixel, microns
4	height of one pixel, microns
5	no. of possible character heights (0=continuous scaling)
6	no. of line types
7	no. of possible line widths (0=continuous scaling)
8	no. of marker types
9	no. of possible marker sizes (0=continuous scaling)
10	no. of text fonts available
11	no. of fill styles available
12	no. of cross-hatching styles available
13	no. of colors that can be shown simultaneously
14	no. of generalized drawing primitives (GDI's)
15-24	first 10 GDI's supported (-1=end of list): 1=rectangle, 2=curve, 3=circle segment, 4=circle, 5=ellipse, 6=elliptical arc, 7=elliptical segment, 8=rounded rectangle, 9=filled, rounded rectangle, 10=justified text
25-34	attribute of corresponding GDI from **work_out[15]-[24]** (-1=end of list): 0=line, 1=marker, 2=text, 3=area fill, 4=no attribute
35	color capability? (0=no, 1=yes)
36	text rotatable? (0=no, 1=yes)
37	can fill areas? (0=no, 1=yes)
38	supports cell arrays? (0=no, 1=yes)

39 no. of colors supported:
0=more than 32,767; 1=monochrome; >2=no. of colors
40 Cursor control devices: 1=keyboard only; 2=keyboard and mouse
41 no. of mappable devices: 1=keyboard, 2=another device
42 no. of choice devices: 1=function keys, 2=another key field
43 no. of string devices: 1=keyboard
44 workstation type: 0=output only; 1=input only; 2=input/output; 3=reserved; 4=metafile
45 minimum character width
46 minimum character height
47 maximum character width
48 maximum character height
49 minimum visible line width
50 reserved (always zero)
51 maximum line width in X axis
52 reserved (always zero)
53 minimum marker width
54 minimum marker height
55 maximum marker width
56 maximum marker height

See Also
TOS, VDI, v_opnvwk

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI. To open the screen device, use the related function **v_opnvwk**.

As of this writing, a virtual device cannot have its attributes set through the *work_in* array. *work_in[0]* through *work_in[9]* should be set to one, and *work_in[10]* should be set to two. Any other settings will either be ignored or will cause system errors.

v_output_window—VDI function (**libvdi.a/v_output_window**)
Dump a portion of a virtual device to a printer
#include <aesbind.h>
#include <vdibind.h>
void v_output_window(*handle, xyarray***) int** *handle, xyarray***[4];**

v_output_window is a VDI routine that dumps a portion of a virtual device to the printer. *handle* is the virtual device's VDI handle. *xyarray* gives the two corners of the area to be dumped. On devices set to normalized device coordinates (NDC), *xyarray[0]* and *xyarray[1]* give, respectively, the X and Y coordinates of the lower left-hand corner, and *xyarray[2]* and *xyarray[3]* give the coordinates of the upper right-hand corner. On devices set to raster coordinates (RC), the first two array elements give the coordinates for the upper left-hand corner, and the latter two elements the coordinates of the lower right-hand corner.

See Also
TOS, VDI

Notes
The printer must be correctly described to TOS before this routine will work.

v_pieslice—VDI function (**libvdi.a/v_pieslice**)
Draw a circular pie slice
#include <aesbind.h>
#include <vdibind.h>
void v_pieslice(*handle, xcoord, ycoord, radius, beginangle, endangle***)**
int *handle, xcoord, ycoord, radius, beginangle, endangle*;

v_pieslice is a VDI routine that draws a circular pie slice. *handle* is the virtual device's VDI handle. *xcoord* and *ycoord* give, respectively, the X and Y coordinates for the imaginary circle of which the pie slice is a part. *radius* gives the imaginary circle's radius. Note that these measurements vary, depending on whether the device uses normalized device coordinates (NDC) or raster coordinates (RC). Finally, *beginangle* and *endangle* represent the beginning and end angles of the pie slice, given in tenths of a degree. Counting on an imaginary clock, zero degrees is at 3 o'clock; 90 degrees at noon; 180 degrees at 9 o'clock; and 270 degrees at 6 o'clock.

See Also
TOS, v_circle, VDI

v_pline—VDI function (**libvdi.a/v_pline**)
Draw a line
#include <aesbind.h>
#include <vdibind.h>
void v_pline(*handle, howmany, xyarray***)**
int handle, howmany, xyarray[*n***];**

v_pline is a VDI routine that draws a line. Note that for VDI, a line is built out of one or more line segments, each end of which has its own pair of X and Y coordinates. Thus, it is possible to use **v_pline** to draw polygons or other figures on the screen.

handle is the virtual device's VDI handle. *count* is the number of line segments to be drawn. *xyarray* is an array of integers that holds the X and Y coordinates for the ends of the line segments; *n* is exactly double the value of *count*. Note that each even value in the array encodes an X coordinate, and each odd value a Y coordinate.

Example
The following example allows you draw lines on the screen while using the mouse. Click the left button to draw a line; holding down the left button lets you draw a continuous squiggle. Exit by typing any key.

The program is called **line.c**. Compile it with the following command line:

```
cc -V line.c -laes -lvdi
```

This program should *not* be compiled with the **-VGEM** because it uses **argv** and **argc**.

The command line takes four arguments: the width of the line being drawn (from one to 99); the type of line being draw (from one to six); and the styles endpoints (from zero to two). For example, to invoke the program from **msh** to draw solid lines three rasters wide, with arrowheads at each end, type:

```
gem line 3 1 1 1
```

This will allow you to experiment with various widths and styles of lines. Passing an incorrect value or an incorrect number of variables causes it to exit with an error message that, unfortunately, flashes very briefly on the screen.

```
#include <aesbind.h>
#include <gemdefs.h>
#include <vdibind.h>

/* global line A variables used by vdi; MUST be included */
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

/* array used by v_pline() */
int xyarray[] = { 1, 1, 1, 1 };

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 639, 399 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* throw-away declarations, to keep system from scribbling over itself */
int nowhere = 0;
Rect norect = { 0, 0, 0, 0 };
```

```
main(argc, argv)
int argc; char *argv[]; {
/* declarations used by evnt_multi() */
        int selection;                  /* code for event that occurred */
        unsigned int which = (MU_KEYBD | MU_BUTTON);
        int clicks = 1;                 /* no. of clicks expected on mouse button */
        int button = 1;                 /* which button; 1 = leftmost */
        int buttonstate = 1;            /* button state expected; 1 = down */
        int buffer[11];                 /* place to write AES messages */
        int mousex;                     /* mouse X coordinate */
        int mousey;                     /* mouse Y coordinate */

/* misc declarations */
        int vdihandle;
        int width;
        int type;
        int end1;
        int end2;

/* check if command line arguments are OK */
        if ((argc-1) != 4) {
                quit ("Usage: line [width, type, end1, end2]");
        }
        width = atoi(argv[1]);
        type = atoi(argv[2]);
        end1 = atoi(argv[3]);
        end2 = atoi(argv[4]);

        if (width < 1 || width > 99) {

                quit ("Width [argv[1]] has incorrect value");
        }

        if (type < 1 || type > 6 ) {
                quit ("Type [argv[2]] has incorrect value");
        }

        if (end1 < 0 || end1 > 2) {
                quit ("First line end [argv[3]] has incorrect value");
        }

        if (end2 < 0 || end1 > 2) {
                quit ("Second line end [argv[4]] has incorrect value");
        }
```

```
/* OK, here we go ... */
	appl_init();
	graf_mouse(ARROW, &nowhere);
	vdihandle = graf_handle(&nowhere, &nowhere, &nowhere, &nowhere);
	v_opnvwk(work_in, &vdihandle, work_out);
	vs_clip(vdihandle, 1, cliparray);
	vsl_width(vdihandle, width);
	vsl_type(vdihandle, type);
	vsl_ends(vdihandle, end1, end2);

	for(;;) {
		selection = evnt_multi(which, clicks, button, buttonstate,
			0, norect, 0, norect, buffer, 0, 0, &mousex, &mousey,
			&nowhere, &nowhere, &nowhere, &nowhere);

		switch(selection) {
		case MU_KEYBD:
			v_clsvwk(vdihandle);
			appl_exit();
			exit(0);

		case MU_BUTTON:
			xyarray[0] = xyarray[2];
			xyarray[1] = xyarray[3];
			xyarray[2] = mousex;
			xyarray[3] = mousey;
			graf_mouse(M_OFF, &nowhere);
			v_pline(vdihandle, 2, xyarray);
			graf_mouse(M_ON, &nowhere);
			break;

		default:
			break;
		}
	}
}

quit(message)
char *message;
{
	printf("%s\n", message);
	exit(0);
}
```

See Also

TOS, **VDI**, **vql_attributes**, **vsl_color**, **vsl_ends**, **vsl_type**, **vsl_udsty**, **vsl_width**

v_pmarker—VDI function (**libvdi.a/v_pmarker**)

Draw a marker

#include <aesbind.h>
#include <vdibind.h>

void v_pmarker(*handle, count, array*) int *handle, count, array*[*n*];

v_pmarker is a VDI routine that draws one or more markers on a virtual device. *handle* is the virtual device's VDI handle. *count* is the number of markers you want to draw. *array* is an array of X and Y coordinates that locate each marker on the screen; *n*, therefore, must be exactly double the size of *count*. Every even number in this array indicates an X coordinate, and every odd number a Y coordinate. Note that the values for each coordinate will differ, depending on whether the device is set to normalized device coordinates (NDC) or raster coordinates (RC).

Example
For an example of this routine, see the entry for **v_circle**.

See Also
TOS, VDI, vqm_attributes, vsm_color, vsm_height, vsm_type

v_rbox—VDI function (**libvdi.a/v_rbox**)
Draw a rounded rectangle
#include <aesbind.h>
#include <vdibind.h>
void v_rbox(*handle, xyarray*) int *handle, xyarray*[4];

v_rbox is a VDI routine that draws a rectangle with rounded corners. *handle* is, as always, the virtual device's VDI handle. *xyarray* gives the X and Y coordinates of the two corners that define the rectangle; the even entries in the array give the X coordinates, and the odd entries the Y coordinates. Note that these values will change, depending on whether the virtual device is defined as using normalized device coordinates (NDC) or raster coordinates (RC). On an NDC device, *xyarray[0]* and *xyarray[1]* encode the lower left-hand corner, where on an RC device they encode the upper left-hand corner; likewise, on an NDC device *xyarray[2]* and *xyarray[3]* represent the upper right-hand corner, whereas on an RC device they represent the lower right-hand corner.

Example
The following example draws filled, rounded rectangles on the screen. Use mouse to draw rubberboxs on the screen. Pressing any key exits.

```
#include <gemdefs.h>
#include <aesbind.h>
#include <vdibind.h>

/* global line A variables used by vdi; MUST be included */
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];
```

```
/* array used by v_rfbox() */
int xyarray[] = { 1, 1, 1, 1 };

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 639, 399 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* throw-away declarations, to keep system from scribbling over itself */
int nowhere = 0;
Rect norect = { 0, 0, 0, 0 };

main() {
/* declarations used by evnt_multi() */
	int selection;			/* code for event that occurred */
	unsigned int which = (MU_KEYBD | MU_BUTTON);
	int clicks = 1;			/* no. of clicks expected on mouse button */
	int button = 1;			/* which button; 1 = leftmost */
	int buttonstate = 1;		/* button state expected; 1 = down */
	int buffer[11];			/* place to write AES messages */
	int mousex;			/* mouse X coordinate */
	int mousey;			/* mouse Y coordinate */
	unsigned key;			/* key typed by user */

/* misc declarations */
	int vdihandle;			/* virtual device's handle */
	int width;			/* width of rubberbox user draws */
	int depth;			/* depth of rubberbox user draws */

/* OK, here we go ... */
	appl_init();
	graf_mouse(ARROW, &nowhere);
	vdihandle = graf_handle(&nowhere, &nowhere, &nowhere, &nowhere);
	v_opnvwk(work_in, &vdihandle, work_out);
	vs_clip(vdihandle, 1, cliparray);
	vsf_perimeter(vdihandle, 1);

	for(;;) {
		selection = evnt_multi(which, clicks, button, buttonstate,
			0, norect, 0, norect, buffer, 0, 0, &mousex, &mousey,
			&nowhere, &nowhere, &key, &nowhere);

		switch(selection) {
		case MU_KEYBD:
			v_clsvwk(vdihandle);
			appl_exit();
			exit(0);
			break;
```

```
		case MU_BUTTON:
			graf_rubbox(mousex, mousey, 3, 3, &width, &depth);
			xyarray[0] = mousex;
			xyarray[1] = mousey;
			xyarray[2] = (mousex+width);
			xyarray[3] = (mousey+depth);
			graf_mouse(M_OFF, &nowhere);
			v_rfbox(vdihandle, xyarray);
			graf_mouse(M_ON, &nowhere);
			break;

		default:
			break;
		}
	}
}
```

See Also
TOS, VDI, v_rfbox

v_rfbox—VDI function (**libvdi.a/v_rfbox**)
Draw a filled, rounded rectangle
#include <aesbind.h>
#include <vdibind.h>
void v_rfbox(*handle, xyarray***) int** *handle, xyarray***[4];**

v_rfbox is a VDI routine that draws a rectangle with rounded corners. It uses the functions **vsf_interior** and **vsf_style**, which set, respectively, the type and style of the interior fill.

handle is, as always, the virtual device's VDI handle. *xyarray* gives the X and Y coordinates of the two corners that define the rectangle; the even entries in the array give the X coordinates, and the odd entries the Y coordinates. Note that these values will change, depending on whether the virtual device is defined as using normalized device coordinates (NDC) or raster coordinates (RC). On an NDC device, *xyarray[0]* and *xyarray[1]* encode the lower left-hand corner, where on an RC device they encode the upper left-hand corner; likewise, on an NDC device *xyarray[2]* and *xyarray[3]* represent the upper right-hand corner, whereas on an RC device they represent the lower right-hand corner.

Example
The following example draws filled, rounded rectangles on screen. Use mouse to draw a rubberbox on screen. Pressing any key exits.

```
#include <gemdefs.h>
#include <aesbind.h>
#include <vdibind.h>
#define RETURN 0x1C0D

/* global line A variables used by vdi; MUST be included */
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

/* array used by v_rfbox() */
int xyarray[] = { 1, 1, 1, 1 };

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 639, 399 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* throw-away declarations, to keep system from scribbling over itself */
int nowhere = 0;
Rect norect = { 0, 0, 0, 0 };

main() {
/* declarations used by evnt_multi() */
	int selection;			/* code for event that occurred */
	unsigned int which = (MU_KEYBD | MU_BUTTON);
	int clicks = 1;			/* no. of clicks expected on mouse button */
	int button = 1;			/* which button; 1 = leftmost */
	int buttonstate = 1;		/* button state expected; 1 = down */
	int buffer[11];			/* place to write AES messages */
	int mousex;			/* mouse X coordinate */
	int mousey;			/* mouse Y coordinate */
	unsigned key;			/* key typed by user */

/* misc declarations */
	int vdihandle;			/* virtual device's handle */
	int width;			/* width of rubberbox user draws */
	int depth;			/* depth of rubberbox user draws */

/* OK, here we go ... */
	appl_init();
	graf_mouse(ARROW, &nowhere);
	vdihandle = graf_handle(&nowhere, &nowhere, &nowhere, &nowhere);
	v_opnvwk(work_in, &vdihandle, work_out);
	vs_clip(vdihandle, 1, cliparray);
	vsf_perimeter(vdihandle, 1);

	for(;;) {
		selection = evnt_multi(which, clicks, button, buttonstate,
			0, norect, 0, norect, buffer, 0, 0, &mousex, &mousey,
			&nowhere, &nowhere, &key, &nowhere);
```

```
			switch(selection) {
			case MU_KEYBD:
				v_clsvwk(vdihandle);
				appl_exit();
				exit(0);

			case MU_BUTTON:
				graf_rubbox(mousex, mousey, 3, 3, &width, &depth);
				xyarray[0] = mousex;
				xyarray[1] = mousey;
				xyarray[2] = (mousex+width);
				xyarray[3] = (mousey+depth);
				graf_mouse(M_OFF, &nowhere);
				v_rfbox(vdihandle, xyarray);
				graf_mouse(M_ON, &nowhere);
				break;

			default:
				break;
			}
		}
}
```

See Also
TOS, VDI, v_rbox

v_rmcur—VDI function (**libvdi.a/v_rmcur**)
Remove last mouse pointer from the screen
#include <aesbind.h>
#include <vdibind.h>
void v_rmcur(*handle*) int *handle*;

v_rmcur is a VDI routine that removes the last mouse pointer from the screen. Note that this routine removes only the *last* mouse pointer to have been invoked. If the mouse pointer has been invoked several times, this routine must be called as many times before the mouse pointer finally disappears.

See Also
TOS, VDI

v_rvoff—VDI function (**libvdi.a/v_rvoff**)
End reverse video for alphabetic text
#include <aesbind.h>
#include <vdibind.h>
void v_rvoff(*handle*) int *handle*;

v_rvoff is a VDI routine that turns off reverse-video display for all alphabetic text written subsequently. *handle* is the virtual device's VDI handle.

See Also
TOS, v_rvon, VDI

v_rvon—VDI function (**libvdi.a/v_rvon**)
Display alphabetic text in reverse video
#include <aesbind.h>
#include <vdibind.h>
void v_rvon(*handle*) int *handle*;

v_rvon is a VDI routine that causes all subsequent alphabetic text to appear in reverse video. *handle* is the virtual device's VDI handle.

See Also
TOS, v_rvoff, VDI

v_show_c—VDI function (**libvdi.a/v_show_c**)
Show the mouse cursor
#include <aesbind.h>
#include <vdibind.h>
void v_show_c(*handle, ignore*) int *handle, ignore*;

v_show_c is a VDI routine that reshows the mouse cursor after it has been hidden. Due to a peculiarity in the VDI, this or a similar routine must be invoked the same number of times that the mouse pointer has been hidden. For example, if the mouse pointer was hidden three times in a row without being redisplayed, it must be recalled three times with **v_show_c** or a similar routine (e.g., **graf_mouse**) before it will reappear.

handle is the virtual device's VDI handle. *ignore* is a flag that sets the VDI's hide-mouse counting feature: zero indicates that the number of times the mouse pointer was hidden should be ignored, whereas one means that it should be honored.

See Also
TOS, v_hide_c, VDI

v_updwk—VDI function (**libvdi.a/v_updwk**)
Update a virtual workstation
#include <aesbind.h>
#include <vdibind.h>
void v_updwk(*handle*) int *handle*;

v_updwk is a VDI routine that updates a virtual workstation. *handle* is the virtual device's VDI handle.

This routine is used with virtual devices that have buffered output, e.g., printers and plotters, and so is somewhat analogous to the STDIO function **fflush**. Note that this function merely executes the commands in buffer, but does not clear the workstation. To clear the workstation, use the function **v_clrwk**.

See Also
TOS, v_clrwk, VDI

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

v_write_meta—VDI function (**libvdi.a/v_write_meta**)
Write a metafile item
#include <aesbind.h>
#include <vdibind.h>
void v_write_meta(*handle, numintin, intin, numptsin, ptsin***)**
int *handle, numintin, intin*[*numintin*], *numptsin, ptsin*[*numptsin*];

v_write_meta is a VDI routine that writes a VDI item into a metafile. The item is assigned an opcode by the VDI. *handle* is the virtual device's VDI handle. The *intin* and *ptsin* arrays hold the information needed to build the metafile item. The first entry must hold an opcode that describes the type of item being built. The next 100 entries are reserved by the system. The sub-opcodes used to build the item are entries 101 and higher.

See Also
metafile, TOS, v_meta_extent, VDI, vm_filename

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

VDI—Definition
VDI stands for *virtual device interface*. The VDI is designed to provide the user with graphics routines that can be transported without alteration to a variety of devices: screen, printer, plotter, video camera, graphics tablet, and "metafile". The VDI can perform the following tasks on these devices:

* Draw lines, polygons, circles, curves, and other graphics primitives.
* Fill or flood areas with a preset pattern or cross-hatching.
* Copy ("blit") areas of the virtual device or graphics images.
* Load type fonts, and size, justify, and rotate graphics text.

* Write and read "metafiles", or a file of an image that can be incorporated into various other VDI programs (for example, a company logo).
* Return information about virtual devices and drivers.
* Await user events and interrogate the system about them.

Devices and virtual devices
The VDI has drivers that support a number of physical graphics devices. Each physical device can, in turn, have output to it one or more "virtual devices". A virtual device is a logical description of the physical device that is handed to the device driver for display on the physical device. For example, you may open a virtual device for the screen, which creates a logical description of the screen for your program to manipulate. Every time this virtual device is changed, the device driver updates the physical device to reflect this change. Note that more than one virtual device may be created for each physical device; this allows you to create a series of "transparencies" that can be manipulated independent of each other and overlaid on the physical device.

Each virtual device can be described using either normalized device coordinates (NDC) or raster coordinates (RC). The NDC system divides a virtual device into a grid that has 32,767 points on each side. The size of points on the X scale differ from those on the Y scale, to ensure that object such as circles are drawn in correct proportion. The RC system uses the number of rasters on the physical screen to scale its objects. The number of rasters will vary, depending on the resolution to which the screen is set, as follows: high resolution, 640 horizontal and 400 vertical; medium resolution, 640 horizontal and 200 vertical; and low resolution, 320 horizontal and 200 vertical.

Note that the NDC and RC systems also differ in where they place the 0,0 point on their X/Y scales. In the NDC system, the 0,0 point is in the lower left-hand corner, whereas in the RC system, 0,0 is in the upper left-hand corner. All objects drawn on the virtual device will be oriented in the same way; for example, a rectangle drawn on an RC virtual device will be measured from the upper left-hand corner to the lower right-hand corner, whereas one on an NDC device will be measured from the lower left-hand corner to the upper right-hand corner. In general, RC are easier for the programmer to visualize and handle, but NDC are more portable.

VDI components
The VDI, like Gaul, is divided into three parts: a library of fonts, a library of device drivers, and GDOS.

The *fonts* display alphanumeric characters in various sizes, weights, and styles. The *device drivers* turn generalized VDI statements into bits that can be understood by particular physical devices. Finally, the most important element is the graphic device operating system (GDOS). The GDOS, as its name implies, coordinates the loading of fonts and drivers, and ensures that a virtual device is directed to the correct physical device. It ensures that the VDI is truly device-independent.

The GDOS also controls the writing and use of metafiles. These files are extraordinarily useful; for more information, see the entry for **metafile**.

Programming with the VDI
The VDI is "virtual" because it works not directly with physical devices, but with the logical description of a device, or a virtual device. When this logical description is altered, the VDI can either update the phyical device directly or record the changes in a metafile.

To work with the VDI, a program must first *open* a virtual device with one of the functions **v_opnwk** or **v_opnvwk**. To use these routines, you must hand them an array of 11 integers that set various aspects of the graphics environment: for example, the color and thickness of the lines to be drawn, the color of the text, and the type and style of pattern fill for polygons. These routines assign a *handle* to the virtual device, and return an array of 57 integers that give information about the newly opened virtual device.

The VDI, in its operation uses five global arrays of integers: **intin[]**, **intout[]**, **ptsin[]**, **ptsout[]**, and **contrl[]**. The last of these should be declared as having 12 members; the others should be declared as having 128. These arrays are manipulated directly by assembly-language programs; they are not used directly within C programs, but must be declared for the VDI routines to work.

Within the program, you can use routines to draw graphics primitives, receive and process information from the user, modify the virtual device's default settings, and perform many other types of useful tasks.

When finished, the functions **v_clswk** or **v_clsvwk** should be invoked to free the memory used by the virtual device, and otherwise tidy up after the program.

Note that all programs that use the graphics interface must run under the AES; this means that all programs that use the VDI must begin with **appl_init** and close with **appl_exit**.

VDI library routines
The VDI library routines are declared in the header file **vdibind.h**, and are stored in the library **libvdi.a**. These routines are, in turn, built out of the Atari Line A routines, which form graphic "primitives". The following lists the VDI routines and gives a brief description of each. For more information about a particular routine, see its entry in the Lexicon.

v_arc	draw a circular arc
v_bar	draw an outlined, filled rectangle
v_bit_image	print a bit-image file
v_cellarray	create an array of colored cells
v_circle	draw a circle
v_clear_disp_list	clear a printer's display list

v_clrwk	clear a virtual device
v_clsvwk	close the screen device
v_clswk	close a virtual device
v_contourfill	draw a filled polygon
v_curdown	move the text cursor down one row
v_curhome	move the text cursor to upper left corner
v_curleft	move the text cursor left one column
v_curright	move the text cursor right one column
v_curtext	write a string of text characters
v_curup	move the text cursor up one row
v_dspcur	move the mouse pointer to specified location
v_eeol	erase from text cursor to end of line
v_eeos	erase from text cursor to end of screen
v_ellarc	draw an elliptical arc
v_ellipse	draw an ellipse
v_ellpie	draw an elliptical pie segment
v_enter_cur	enter text mode
v_exit_cur	exit from text mode
v_fillarea	flood an enclosed area with fill pattern
v_form_adv	advance the page on a hard-copy device
v_get_pixel	find if a particular pixel has been set
v_gtext	output graphics text
v_hardcopy	dump virtual device to hard-copy device
v_hide_c	hide the mouse pointer
v_justified	output justified graphics text
v_meta_extents	update extents header of metafile
v_opnvwk	open the screen virtual device
v_opnwk	open a virtual device
v_output_window	print a portion of a virtual device
v_pieslice	draw a circular pie segment
v_pline	draw a polyline
v_pmarker	draw a polymarker
v_rbox	draw a rectangle with rounded corners
v_rfbox	draw rectangular fill area with rounded corners
v_rmcur	remove last graphics cursor from screen
v_rvoff	turn off reverse video for text text
v_rvon	turn on reverse video for text text
v_show_c	show mouse pointer
v_updwk	update workstation (flush buffers)
v_write_meta	add an item to a metafile
vex_butv	change button interrupt routine
vex_curv	change cursor movement interrupt routine
vex_motv	change mouse pointer interrupt routine
vex_timv	change timer interrupt routine
vm_filename	rename a metafile
vq_cellarray	query cell array information

vq_chcells	query no. of characters printable on device
vq_color	query/set mix for a color
vq_curaddress	query text cursor's current position
vq_extnd	perform extended inquiry
vq_key_s	query keyboard status
vq_mouse	query mouse position and button state
vq_tabstatus	query if graphics tablet is available
vqf_attributes	set fill area attributes
vqin_mode	set inquiry mode
vql_attributes	query polyline attributes
vqm_attributes	query polymarker attributes
vqp_error	query message from Polaroid Palette
vqp_films	films supported on Polaroid Palette
vqp_state	read status of Polaroid Palette driver
vqt_attributes	query graphics text attributes
vqt_extent	query length of a string
vqt_font_info	quer information about fonts
vqt_name	query name and description of font
vqt_width	query width of a character's cell
vr_recfl	draw a rectangular fill area
vr_trnfm	transform bit image format
vro_cpyfm	copy (blit) a portion of a device
vrq_choice	query choice devices, request mode
vrq_locator	query locator devices, request mode
vrq_string	query string devices, request mode
vrq_valuator	query valuator devices, request mode
vrt_cpyfm	copy (blit) a monochromatic image
vs_clip	clip an area of the virtual device
vs_color	set mix for a color
vs_curaddress	move text cursor to specified point
vs_palette	set the palette for medium resolution
vsc_form	set new mouse pointer shape
vsf_color	set fill color
vsf_interior	set fill type
vsf_perimeter	set drawing of perimeter
vsf_style	set fill style
vsf_udpat	set user-defined fill pattern
vsin_mode	set mode of logical device inquiry
vsl_color	set polyline color
vsl_ends	set polyline end types
vsl_type	set polyline's pattern
vsl_udsty	set user-defined polyline style
vsl_width	set polyline width
vsm_choice	query choice devices, sample mode
vsm_color	query color settings
vsm_height	query character cell's height

vsm_locator	query locator devices, sample mode
vsm_string	query string devices, sample mode
vsm_type	query graphics text attributes
vsm_valuator	query valuator devices, sample mode
vsp_message	suppress Polaroid Palette messages
vsp_save	save settings of driver for Polaroid Palette
vsp_state	set Polaroid Palette driver
vst_alignment	set graphics text alignmment
vst_color	set graphics text color
vst_effects	set graphics text special effects
vst_font	set graphics text font
vst_height	set graphics text height, in pixels
vst_load_fonts	load non-standard fonts
vst_point	set graphics text height, in points
vst_rotation	set angle of graphics text
vst_unload_fonts	unload non-standard fonts
vswr_mode	set writing mode

A sample VDI program

The following program is a game that demonstrates how to use a number of VDI routines. The program draws a small black rectangle, which chases the mouse pointer. If the mouse pointer is caught, the program exults briefly, then asks you if you want to play again.

```
#include <aesbind.h>
#include <gemdefs.h>
#include <osbind.h>
#include <vdibind.h>

#define BUTTON 1                 /* which button; 1 = leftmost */
#define CENTER 1                 /* indicates centering of text */
#define CENTERX 320              /* X coordinate of center of high-res screen */
#define CENTERY 200              /* Y coordinate of center of high-res screen */
#define CLICKS 1                 /* No. of clicks expected on mouse button */
#define DOWN 1                   /* mouse button is down */
#define HITIME 0                 /* high word in timer values */
#define LEFT 0                   /* set text flush left */
#define LOTIME 5                 /* low word in timer values; set to 5 ms. */
#define XOR 6                    /* code for writing in XOR mode */

/* global line A variables used by vdi; MUST be included */
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

/*
 * array used by vs_clip().  Clipping array MUST be set; otherwise,
 * low memory will be written over by graphics forms that extend
 * beyond the edge of the screen.
 */
int cliparray[] = { 1, 1, 639, 399 };
```

```
/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* arrays used by vro_cpyfm() to blit cat around screen */
int shape[] = {                        /* draw a solid black rectangle into memory */
	0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
	0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
	0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
	0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF
};

FDB cat = { shape, 16, 16, 1, 0, 1, 0, 0, 0 };       /* source form block */
FDB screen = { 0L, 0, 0, 0, 0, 0, 0, 0, 0 };         /* target form block */
int oldarray[] = {                                     /* initial position on screen */
	0, 0, 16, 16, CENTERX, CENTERY, CENTERX+16, CENTERY+16 };
int newarray[] = {                                     /* new position on screen */
	0, 0, 16, 16, CENTERX, CENTERY, CENTERX+16, CENTERY+16 };

/* throw-away declaration, to keep system from scribbling over itself */
int nowhere = 0;

main() {
	int mousex = 1;                /* mouse X coordinate */
	int mousey = 1;                /* mouse Y coordinate */
	int vdihandle;                 /* virtual device's handle */

/* OK, here we go ... */
	/* open AES process */
	appl_init();
	graf_mouse(BUSY_BEE, &nowhere);
	/* open virtual device */
	vdihandle = graf_handle(&nowhere, &nowhere, &nowhere, &nowhere);
	v_opnvwk(work_in, &vdihandle, work_out);
	/* set clipping rectangle */
	vs_clip(vdihandle, 1, cliparray);

	/* blit cat initially */
	vro_cpyfm(vdihandle, XOR, oldarray, &cat, &screen);

	for(;;) {
		/* get mouse pointer's position */
		vq_mouse(vdihandle, &nowhere, &mousex, &mousey);
```

```
		/* check cat's position vs. that of mouse */
		if (oldarray[4] < mousex)
			newarray[4] += 6;
		else if (oldarray[4] > mousex)
			newarray[4] -= 6;
		if (oldarray[5] < mousey)
			newarray[5] += 6;
		else if (oldarray[5] > mousey)
			newarray[5] -= 6;

		/* set cat's kitty corner */
		newarray[6] = newarray[4] + 16;
		newarray[7] = newarray[5] + 16;

		/* synchronize with screen, then blit cat */
		Vsync();
		vro_cpyfm(vdihandle, XOR, newarray, &cat, &screen);
		vro_cpyfm(vdihandle, XOR, oldarray, &cat, &screen);

		/* shuffle cat's new array into old array */
		oldarray[4] = newarray[4];
		oldarray[5] = newarray[5];
		oldarray[6] = newarray[6];
		oldarray[7] = newarray[7];

		/* check if cat has caught mouse */
		if ((abs(oldarray[4] - mousex) <= 16) &&
			(abs(oldarray[5] - mousey) <= 16)) {
			gotcha(vdihandle);
			playagain(vdihandle);
		}
	}
}

gotcha(vdihandle)
int vdihandle;
{
	char *text = "GOTCHA!";

	/* set text attributes */
	vst_effects(vdihandle, 32);
	vst_alignment(vdihandle, CENTER, 0, &nowhere, &nowhere);
	vst_height(vdihandle, 26, &nowhere, &nowhere, &nowhere, &nowhere);

	/* ring the bell and write "GOTCHA!" on screen in big letters */
	Cconout('\07');
	v_gtext(vdihandle, 320, 200, text);
	evnt_timer(1500, HITIME);
	return;
}
```

```
playagain(vdihandle)
int vdihandle;
{
	char *string = "[2][Play again?][Yes|No]";
	int button;

	/* reset text attributes */
	vst_effects(vdihandle, 1);
	vst_alignment(vdihandle, LEFT, 0, &nowhere, &nowhere);
	vst_height(vdihandle, 13, &nowhere, &nowhere, &nowhere, &nowhere);

	/* draw alert box */
	button = form_alert(1, string);

	/* do what user requests */
	if (button == 1) {
	/* i.e., if user want another game ... */
		/* ... clear screen again ... */
		v_clrwk(vdihandle);
		/* ... reset cat's position to center of screen ... */
		newarray[4] = oldarray[4] = CENTERX;
		newarray[5] = oldarray[5] = CENTERY;
		newarray[6] = oldarray[6] = (CENTERX + 16);
		newarray[7] = oldarray[7] = (CENTERY + 16);
		/* ... and redraw cat */
		vro_cpyfm(vdihandle, XOR, oldarray, &cat, &screen);

		/* return pointer shape to bee */
		graf_mouse(BUSY_BEE, &nowhere);
		/* wait a few moments so user can get away */
		evnt_timer(1000, HITIME);
		return;
	} else {
	/* i.e., user wants to quit */
		/* close virtual station, close application, and exit */
		v_clsvwk(vdihandle);
		appl_exit();
		exit(1);
	}
}
```

See Also

AES, libvdi.a, Line A, metafile, TOS, vdibind.h

Notes

At the time of this writing, the GDOS portion of the VDI is not included with TOS. This means that metafiles, most device drivers, most fonts, and device-independent features are not available to the programmer. At present, the VDI supports drivers only for the screen device, and acknowledges only raster coordinates. A RAM version of GDOS will be made available to programmers, although as of this writing, no release date has been set. For more information, see the article "Pursuing the Elusive GDOS", by Tim Oren, in the summer 1986 issue of *START*.

Note that both the AES and the VDI use trap 2 to access the services.

vdibind.h—Command

Declarations for VDI and AES routines

#include <vdibind.h>

vdibind.h is the header file that holds declarations and definitions for the GEM VDI routines, which are contained in the library **libvdi.a**.

For a full description of these routines, see the *Atari ST GEM Programmer's Reference*.

See Also

AES, header file, TOS, VDI

version—Command

Print/create a version string

version *file ...*

version *directory executable sourcefile ...*

version finds or creates a version string. When given an executable *file* as an argument, **version** scans it for the version string, which it prints on the standard output device.

version can also generate version numbers automatically. When given the name of a *directory* that holds source code, and the names of the *executable* (whether or not it has been created yet) and *sourcefile* (or *sourcefiles*) **version** writes a brief program in assembly language that, when assembled and linked, generates a version number for the program and writes it into the executable file.

Example

To generate a version number for an executable called **color.prg** that is compiled from the source file called **color.c**, which is found in directory **examples** on drive B:, type the following command:

```
version b:\examples color.prg color.c >version.s
```

It does not matter what you name the file into which you direct the output of **version**; however, be sure that it has the suffix **.s**, so that the compiler will know that it is an assembly-language file, and be sure to include its name on the **cc** command line when you compile the program.

See Also

as, commands, msh

vertical tab—Definition

Mark Williams C recognizes the literal character '\v' for the ASCII vertical tab character VT (octal 013). This character may be used as a character constant or in a string constant, like the other character constants: '\a', which rings the audible bell on the terminal; '\b', to backspace; '\f', to pass a formfeed command to the printer; '\r', for a carriage return; and '\t', for a tab character. The vertical tab character is whitespace; in particular, the macro **isspace** is true for '\v'.

See Also
ASCII, **character constant**

vex_butv—VDI function (**libvdi.a/vex_butv**)

Set new button interrupt routine
#include <aesbind.h>
#include <vdibind.h>
void vex_butv(*handle, address, oldaddress***)**
int *handle*; **void (****address***); void (****oldaddress***);**

vex_butv is a VDI routine that lets you set a new button interrupt routine. *handle* is the virtual device's VDI handle. *address* is the address of the new interrupt routine. Note that your routine is responsible for saving registers and resetting registers. Finally, *oldaddress*, which is set by **vex_butv**, is the address of the old interrupt routine.

See Also
TOS, **VDI, vex_curv, vex_motv, vex_timv**

vex_curv—VDI function (**libvdi.a/vex_curv**)

Set new cursor interrupt routine
#include <aesbind.h>
#include <vdibind.h>
void vex_curv(*handle, address, oldaddress***)**
int *handle*; **void (****address***); void (****oldaddress***);**

vex_curv is a VDI routine that lets you set a new cursor movement interrupt routine. *handle* is the virtual device's VDI handle. *address* points to the new interrupt routine. Note that your new routine is responsible for saving and restoring registers. Finally, *oldaddress*, which is set by **vex_curv**, points to the old interrupt routine.

See Also
TOS, **VDI, vex_butv, vex_motv, vex_timv**

vex_motv—VDI function (**libvdi.a/vex_motv**)

Set new mouse movement interrupt routine

#include <aesbind.h>
#include <vdibind.h>
void vex_motv(*handle, address, oldaddress***)**
int *handle*; **void (****address***); void (****oldaddress***);**

vex_motv is a VDI routine that sets a new interrupt routine to be invoked when the mouse pointer moves. *handle* is the virtual device's VDI handle. *address* gives the address of the new interrupt routine. Note that your routine is responsible for saving and restoring registers. *oldaddress* is set by **vex_motv**; it holds the address of the old interrupt routine.

See Also

TOS, VDI, vex_butv, vex_curv, vex_timv

vex_timv—VDI function (**libvdi.a/vex_timv**)

Set new timer interrupt routine

#include <aesbind.h>
#include <vdibind.h>
void vex_timv(*handle, address, oldaddress, &time***)**
int *handle, time*; **void (****address***); void (****oldaddress***);**

vex_timv is a VDI routine that lets you set a new timer interrupt routine. *handle* is the virtual device's VDI handle. *address* points to the address of the new interrupt routine. *oldaddress* is set by **vex_timv** upon exiting, and contains the old interrupt address. Finally, *time* is set by **vex_timv** upon exiting, and contains the interval of the interrupt call, in milliseconds. Note that your new interrupt routine is responsible for saving registers and returning to the system. The interrupt is reactivated by setting the old interrupt address.

See Also

TOS, VDI, vex_butv, vex_curv, vex_motv

Notes

This routine is called **vex_time** in some bindings.

vm_filename—VDI function (**libvdi.a/vm_filename**)

Rename a metafile

#include <aesbind.h>
#include <vdibind.h>
void vm_filename(*handle, filename***) int** *handle*; **char** **filename*;

vm_filename is a VDI routine that renames a metafile. *handle* is the virtual device's VDI handle. *filename* points to the new file name; this must be an alphabetic string that is terminated with a NUL character.

See Also
TOS, v_meta_extent, v_write_meta, VDI

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

void—Definition
In addition to the data types described in *The C Programming Language*, Mark Williams C also recognizes the data type **void**. **void** applies *only* to a function declaration, and indicates that the function does not return a value. Using **void** declarations makes programs clearer and is useful in error checking. For example, a function that prints an error message and calls **exit** to terminate a program should be declared **void** because it never returns. A function that performs a calculation and stores its result in a global variable (rather than **returning** the result), or one that returns no value, should also be declared **void** to prevent the accidental use of the function in an expression. For example,

```
void cursor_pos(x,y)
int x,y;
{
        printf("\33Y%c%c", x+' ', y+' ');
}
```

could be used to write the current position of the cursor in a screen handling program.

To add **void** to the formal definition of C, amend the list of type-specifiers in Appendix A of *The C Programming Language* to include **void**.

See Also
declarations

vq_cellarray—VDI function (**libvdi.a/vq_cellarray**)
Return information about cell arrays
#include <aesbind.h>
#include <vdibind.h>
void vq_cellarray(*handle, xyarray, rowlength, rows,*
*&cellused, &rowused, &status, cellarray***)**
int *handle, xyarray, rowlength***[4]**, *rows, cellused, rowsused, status, cellarray***[***n***]**;

vq_cellarray is a VDI routine that returns information about an established cell array.

handle is the virtual device's VDI handle. *xyarray* gives the X and Y coordinates for the rectangle in which the array is drawn. Note that these values will vary, depending on whether the device is set to normalized device coordinates (NDC) or raster coordinates (RC). On NDC devices, *xyarray[0]* and *xyarray[1]* give, respectively, the X and Y coordinates of the lower left-hand corner of the rectangle, whereas *xyarray[2]* and *xyarray[3]* give the coordinates for the upper right-handle corner. On RC devices, *xyarray[0]* and *xyarray[1]* give, respectively, the X and Y coordinates of the upper left-hand corner, whereas *xyarray[2]* and *xyarray[3]* give the X and Y coordinates of the lower right-hand corner. *rowlength* gives the horizontal length of the table to be shown, in NDCs or RCs, and *rows* is the number of rows of cells in the array.

cellused is the number of horizontal cells used in each row. *rowused* gives the number of rows in the array that were actually used. *status* gives the array's error status: zero indicates that no errors occurred, whereas a number greater than one indicates that a color value could not be found for a given cell.

Finally, *cellarray* gives the array of colors actually displayed in the used cells. *n* must be equal to the number of cells in the entire array. The color index will be set to -1 if the color requested for a given cell could not be found.

See Also
TOS, v_cellarray, VDI

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

vq_chcells—VDI function (**libvdi.a/vq_chcells**)
Find how many characters virtual device can print
#include <aesbind.h>
#include <vdibind.h>
void vq_chcells(*handle, &rows, &columns***) int** *handle, rows, columns*;

vq_chcells is a VDI routine that examines a virtual device and returns the number of rows and columns of characters that can be printed on it. *handle* is the virtual device's VDI handle. *rows* and *columns* hold, respectively, the number of rows and the number of columns of characters that can be printed on the virtual device.

Example
The following example returns the number of rows and columns available on the screen device.

```
#include <aesbind.h>
#include <vdibind.h>

/* global line A variables used by vdi; MUST be included */
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];
```

```
/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

main() {
	int nowhere = 0;
	int vdihandle;
	int rows;
	int columns;

	appl_init();
	vdihandle = graf_handle(&nowhere, &nowhere, &nowhere, &nowhere);
	v_opnvwk(work_in, &vdihandle, work_out);
	vq_chcells(vdihandle, &rows, &columns);

	printf("No. of rows: %d\n", rows);
	printf("No. of columns: %d\n", columns);

	v_clsvwk(vdihandle);
	appl_exit();
	exit(0);
}
```

See Also
TOS, VDI

vq_color—VDI function (**libvdi.a/vq_color**)
Check/set color intensity
#include <aesbind.h>
#include <vdibind.h>
void vq_color(*handle, color, flag, rgb***) int** *handle, color, flag, rgb*[3];

vq_color is a VDI routine that checks or sets the intensity of a particular color. *handle* is the virtual device's VDI handle. *color* is the code that indicates which color you wish to check or modify: for a table of color indices, see the entry for **v_opnwk**. *flag* indicates whether you want to set the color, or merely check it: zero indicates set the color, and one indicates check it. Finally, *rgb* points to an array of three integers that, respectively, set the red, green, and blue guns on the color monitor. Each should be set to a level between one and 1,000, with one being the lowest setting and 1,000 the highest.

See Also
TOS, VDI, vq_extnd

vq_curaddress—VDI function (**libvdi.a/vq_curaddress**)
Get the text cursor's current position
#include <aesbind.h>
#include <vdibind.h>
void vq_curaddress(*handle, &row, &column***) int** *handle, row, column*;

vq_curaddress is a VDI routine that returns the current position of the text cursor on the virtual device. *handle* is the virtual device's VDI handle. *row* and *column* are set by **vq_curaddress**; they give, respectively, the row and the column in which the text cursor is positioned.

See Also
TOS, VDI, vs_curaddress

vq_extnd—VDI function (**libvdi.a/vq_extnd**)
Perform extend inquire of VDI virtual device
#include <aesbind.h>
#include <vdibind.h>
void vq_extnd(*handle, type, work_out***) int** *handle, type, work_out*[57];

vq_extnd is a VDI routine that performs an extended inquire on a virtual device. *handle* is the virtual device's VDI handle. *type* is the set of values you want written into the array *work_out*: zero indicates that you want the same values returned by functions **v_opnwk** or **v_opnvwk**. See the entry for **v_opnwk** for a table of these values. Setting *type* to a non-zero value writes the extended inquiry values into *work_out*.

The following table gives the index into the *work_out* array, plus the value written there by the extended inquiry:

0	screen type: 0=not a screen; 1=separate alphabetic and graphics screens; 2=separate alphabetic and graphics controllers with common screen; 3=common alphabetic and graphics controller with separate image memories; and 4=common alphabetic and graphics controller and common image memory
1	no. of background colors available
2	which text effects are available
3	scaling possible? 0=no, 1=yes
4	no. of color planes
5	lookup table supported? 0=no, 1=yes
6	no. of 16X16-pixel raster operations done per second
7	contour fill supported? 0=no, 1=yes
8	can rotate characters? 0=no; 1=90 degrees only; 2=can rotate to arbitrary angles
9	no. of writing modes
10	highest level of input mode: 0=none; 1=request mode; 2=sample mode
11	text alignment supported? 0=no; 1=yes
12	handles multi-colored pens (e.g., plotter)? 0=no; 1=yes
13	handles multi-color ribbons (e.g., dot matrix printer)? 0=no; 1=yes
14	maximum no. of points in a polyline; -1=no maximum

15	maximum size of **intin** array: -1=no maximum
16	no. of buttons on the mouse
17	line types usable on wide lines? 0=no; 1=yes
18	drawing modes available for wide lines
19-57	reserved; contains zeroes

See Also
TOS, v_opnwk, VDI

Notes
This routine is called **vq_extend** in some bindings.

vq_key_s—VDI function (**libvdi.a/vq_key_s**)
Check control key status
#include <aesbind.h>
#include <vdibind.h>
void **vq_key_s(***handle, &status***) int** *handle, status*;

vq_key_s is a VDI routine that checks the control key status. *handle* is the virtual device's VDI handle. *status* is a bit map that, upon return, indicates the status of the control keys; zero indicates not set and one indicates set, as follows:

0	right shift key
1	left shift key
2	control key
3	alt key

See Also
TOS, VDI, vq_mouse

vq_mouse—VDI function (**libvdi.a/vq_mouse**)
Check mouse position and button state
#include <aesbind.h>
#include <vdibind.h>
void vq_mouse(*handle, &status, &xcoord, &ycoord***)**
int *handle, status, xcoord, ycoord*;

vq_mouse is a VDI routine that checks the mouse pointer's position and the status of the mouse buttons. *handle* is the virtual device's VDI handle. *status* is set by **vq_mouse**, and indicates the status of the mouse button: zero indicates not pressed, one indicates pressed. *xcoord* and *ycoord* are set by **vq_mouse** and give, respectively, the X and Y coordinates of the mouse pointer.

See Also
TOS, VDI

vq_tabstatus—VDI function (**libvdi.a/vq_tabstatus**)

Find if graphics tablet is available

#include <aesbind.h>
#include <vdibind.h>
void vq_tabstatus(*handle*) int *handle*;

vq_tabstatus is a VDI routine that checks to see if the graphics tablet is available. *handle* is the virtual device's VDI handle. **vq_tabstatus** returns the status of of the graphics tablet: zero indicates that the tablet is not available, and one indicates that it is.

See Also
TOS, VDI

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

vqf_attributes—VDI function (**libvdi.a/vqf_attributes**)

Read the area fill's current attributes

#include <aesbind.h>
#include <vdibind.h>
void vqf_attributes(*handle, attrib*) int *handle, attrib*[5];

vqf_attributes is a VDI routine that returns the attributes currently set for the area fill. *handle* is the virtual device's VDI handle. The fill area's attributes are written into the array *attrib*, as follows:

attrib[0] Fill type. For a table of fill types, see the entry for **vsf_interior**.

attrib[1] Fill color. For a table of color codes, see the entry for **v_opnwk**.

attrib[2] Fill pattern. For a table of fill patterns, see the entry for **vsf_style**.

attrib[3] Writing mode: one indicates replace mode; two, transparent mode; three, XOR (exclusive or) mode; and four, reverse transparent mode.

attrib[4] Draw border: zero indicates that a border is not drawn around a filled area, and one indicates that it will.

See Also
TOS, v_bar, VDI, vql_attributes, vqm_attributes, vqt_attributes

vqin_mode—VDI function (**libvdi.a/vqin_mode**)

Determine mode of a logical input device

#include <aesbind.h>

#include <vdibind.h>
void vqin_mode(*handle, device, &mode*) int *handle, device, mode*;

vqin_mode is a VDI routine that returns the current mode of a logical input device. *handle* is, as always, the virtual device's VDI handle. *device* is the logical input device whose mode you wish to check, as follows: one, graphic cursor unit (i.e., devices that move the mouse pointer); two, value-changing input (e.g., shift key, control key, etc.); three, selection input unit (i.e., function keys); and four, string input unit (i.e., alphabetic keys). Finally, *mode* is returned by **vqin_mode**; zero indicates request mode, and one indicates sample mode. *Request mode* waits for a particular event to occur on the device before the function returns, analogous to the AES event library; whereas *sample mode* simply polls the device and returns, without waiting for an event.

See Also
TOS, VDI, vsin_mode

vql_attributes—VDI function (**libvdi.a/vql_attributes**)
Read the polyline's current attributes
#include <aesbind.h>
#include <vdibind.h>
void vql_attributes(*handle, attrib*) int *handle, attrib***[6];**

vql_attributes is a VDI routine that returns the current attributes for the VDI polyline routine. *handle* is the virtual device's VDI handle. The polyline attributes are written into the array *attrib*, as follows:

attrib[0] Line type; see the entry for **vsl_type** for a table of line-type codes.

attrib[1] Line color; see the entry for **v_opnwk** for a table of color codes.

attrib[2] Writing mode: one indicates replace mode; two, transparent mode; three, XOR (exclusive or) mode; and four, reverse transparent mode.

attrib[3] Starting point style: zero indicates square; one, arrowhead; and two, rounded.

attrib[4] Ending point style.

attrib[5] Line width.

See Also
TOS, v_pline, VDI, vqf_attributes, vqm_attributes, vqt_attributes

vqm_attributes—VDI function (**libvdi.a/vqm_attributes**)
Read the marker's current attributes
#include <aesbind.h>
#include <vdibind.h>

void vqm_attributes(*handle, attrib*) int *handle, attrib*[5];

vqm_attributes is a VDI routine that returns the attributes currently set for the marker. *handle* is the virtual device's VDI handle. The marker's attributes are written into the array *attrib*, as follows:

attrib[0] Marker type, as follows:

1	period
2	plus sign
3	asterisk
4	square
5	diagonal cross
6	diamond
7	device-dependent

attrib[1] Marker color. For a table of color codes, see the entry for **v_opnwk**.

attrib[2] Writing mode: one indicates replace mode; two, transparent mode; three, XOR (exclusive or) mode; and four, reverse transparent mode.

attrib[3] Marker width.

attrib[4] Marker height.

See Also
TOS, **v_pmarker**, **VDI**, **vqf_attributes**, **vql_attributes**, **vqt_attributes**

vqp_error—VDI function (**libvdi.a/vqp_error**)
Inquire if an error occurred with the Polaroid Palette
#include <aesbind.h>
#include <vdibind.h>
int vqp_error(*handle*) int *handle*;

The VDI contains a driver for the Polaroid Palette, a camera that can be used to shoot slides directly from the Atari ST. **vqp_error** is a VDI routine that returns an error message or user prompt for the camera. *handle* is the virtual device's VDI handle. **vqp_error** returns one of the following error messages:

0	no error
1	open dark slide for print film
2	no port at location specified in driver
3	Polaroid Palette not found at specified port
4	video cable is disconnected
5	operating system does not allow memory allocation
6	not enough memory available to allocate buffer
7	memory not freed
8	driver file not found
9	driver file is not of the correct type
10	user should now process print film

See Also
TOS, VDI, vqp_films, vqp_state, vsp_message, vsp_save, vsp_style

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

vqp_films—VDI function (**libvdi.a/vqp_films**)
Get films supported by driver for Polaroid Palette
#include <aesbind.h>
#include <vdibind.h>
void vqp_films(*handle, names*) int *handle, names*[125];

The VDI contains a driver for the Polaroid Palette, a camera that can be used to shoot slides directly from the Atari ST. **vqp_films** is a VDI routine that returns the names of the fives types of photographic film supported by this driver. *handle* is the virtual device's VDI handle. *films* is an array that holds the names of the films supported.

See Also
TOS, VDI, vqp_error, vqp_state, vsp_message, vsp_save, vsp_style

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

vqp_state—VDI function (**libvdi.a/vqp_state**)
Read current settings of the Polaroid Palette driver
#include <aesbind.h>
#include <vdibind.h>
void vqp_state(*handle, &port, &film, &lightness, &interlace, &planes, &indices*);
int *handle, port, film, lightness, interlace, lanes, indices*[8][2];

The VDI contains a driver for the Polaroid Palette, a camera that can be used to shoot slides directly from the Atari ST. **vqp_state** returns a block of data that gives the driver's settings. *handle* is the virtual device's VDI handle. *port* is the port to which the camera is connected; zero indicates the first communications port. *film* is the number of the film for which the driver is currently set.

lightness is the intensity to which the driver is set, from -3 through three. Each number in this range is equivalent to one third of an f-stop, counting from zero. Therefore, -3 has half the intensity of zero, and three is twice as intense as zero.

interlace indicates whether the image is interlaced or not; zero indicates not interlaced, and one indicates interlaced. Note that an interlaced image requires approximately twice the memory of one that is not interlaced.

planes indicates the number of colors supported. It is set to a code, from one through four; one indicates two colors; two, four colors; three, eight colors; and four, 16 colors.

Finally, *indices* holds two-character codes for the eight color indices stored in ADE format.

See Also
TOS, VDI, vqp_error, vqp_films, vsp_message, vsp_save, vsp_style

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

vqt_attributes—VDI function (**libvdi.a/vqt_attributes**)
Read the graphic text's current attributes
#include <aesbind.h>
#include <vdibind.h>
void vqt_attributes(*handle, attrib***) int** *handle, attrib***[10];**

vqt_attributes is a VDI routine that returns the current attributes for the VDI graphics text routine. *handle* is the virtual device's VDI handle. The graphics text attributes are written into the array *attrib*, as follows:

attrib[0]	Character set.
attrib[1]	Text color. For a table of color codes, see the entry for **v_opnwk**.
attrib[2]	Rotation angle, in tenths of a degree (i.e., 0 through 3600).
attrib[3]	Horizontal alignment. For a table of alignment codes, see the entry for **vst_alignment**.

attrib[4] Vertical alignment.

attrib[5] Writing mode: one indicates replace mode; two, transparent mode; three, XOR (exclusive or) mode; and four, reverse transparent mode.

attrib[6] Character width.

attrib[7] Character height.

attrib[8] Cell width.

attrib[9] Cell height.

See Also
TOS, v_gtext, VDI, vqf_attributes, vql_attributes, vqm_attributes

vqt_extent—VDI function (**libvdi.a/vqt_extent**)
Calculate a string's length
#include <aesbind.h>
#include <vdibind.h>
void vqt_extent(*handle, text, size***) int** *handle, size***[8]; char ****text***;**

vqt_extent is a VDI routine that calculates the length of a string. This is especially useful when positioning proportionally spaced text on a virtual device.

handle is the virtual device's VDI handle. *text* points to the string whose extent you wish to calculate. *size* is an array of eight integers that give the X and Y coordinates of the box that encloses the text, as follows: **size[0]** and **size[1]** give, respectively, the X and Y coordinates of the lower left-hand corner; **size[2]** and **size[3]**, X and Y coordinates of the lower right-hand corner; **size[4]** and **size[5]**, upper right-hand corner; and **size[6]** and **size[7]**, upper left-hand corner. Note that the box extends from the top of the tallest capital letters (e.g., 'M'') to the bottom of the lowest descenders (e.g., 'j' or 'y').

See Also
TOS, v_gtext, VDI

vqt_fontinfo—VDI function (**libvdi.a/vqt_fontinfo**)
Get information about special effects for graphics text
#include <aesbind.h>
#include <vdibind.h>
void vqt_fontinfo(*handle, firstchar, lastchar, sizes, maxwidth, adjust***)**
int *handle, *firstchar, *lastchar, sizes***[5]**, **maxwidth, adjust***[3];**

vqt_fontinfo is a VDI routine that returns information about font sizes, especially about the extra space taken up by slanted and shadowed characters. This extra space is not taken into account by **vqt_extent** when it calculates the length of a string, so you may need to obtain this information from **vqt_fontinfo** when constructing text to be passed to a specialized output device.

The arguments to **vqt_fontinfo** are as follows: *handle* is the virtual device's VDI handle. *firstchar* points to the first character in the ASCII table that can be set on this device, using the font and special effects that have been set for it; *lastchar* points to the last character in the ASCII table that can be so set. These values, of course, are set by **vqt_fontinfo**. *maxwidth* points to the maximum width of a character in the current font.

sizes points to an array of five integers that are set by **vqt_fontinfo**. Each represents a dimension of the current font, as follows:

sizes[0]	bottom line to baseline
sizes[1]	descent line to baseline
sizes[2]	half line to baseline
sizes[3]	ascent line to baseline
sizes[4]	top line to baseline

These terms are defined in the entry for **vst_alignment**.

Finally, *adjust* points to an array of three integers that are set by **vqt_fontinfo**; each represents a change to the font size represented by the special effects being used, as follows:

adjust[0]	increase in character width
adjust[1]	left offset
adjust[2]	right offset

The *right offset* is the amount of space a slanted letter extends beyond the edge of its "cell", which is defined as the width of the character measured across the bottom. The *left offset* is the extra space that must be set to the left of a slanted character, so its neighbor to the left does not slant over it. The increase in character is the total of the left and right offsets; this is the value you need to figure into the value returned by **vqt_extent** to gain the true extent of a string that uses special effects.

See Also
TOS, v_gtext, VDI, vqt_extent, vqt_name, vst_alignment

vqt_name—VDI function (**libvdi.a/vqt_name**)
Get name and description of graphics text font
#include <aesbind.h>
#include <vdibind.h>
int vqt_name(*handle, font, string***) int** *handle, font*; **char** *string*[32];

vqt_name is a VDI routine that returns the name and description of a given font. *handle* is the virtual device's VDI handle. *font* is the number of the font whose name you want. Finally, *string* is where **vqt_name** writes the font name and information. The first 16 **chars** hold the name of the font, and the next 16 hold a brief description of it.

vqt_name returns the font ID that is needed to access this face with the function **vst_font**. Note that the number of fonts available on a given virtual device is returned by the functions **v_opnwk** and **v_opnvwk** in the variable *work_out[10]*.

Example
The following example prints a description of each font currently available to the screen device. Note that this example should be compiled with the option **-VGEM**, but that you do not need to run it with the **gem** command.

```
#include <aesbind.h>
#include <vdibind.h>

/* global line A variables used by vdi; MUST be included */
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

main() {
	int nowhere = 0;
	int vdihandle;
	int info[32];			/* array used by vqt_name */
	int i;				/* counter used in for() loop */
	int code;			/* font code returned by vqt_name() */

	/* open application */
	appl_init();
	vdihandle = graf_handle(&nowhere, &nowhere, &nowhere, &nowhere);
	v_opnvwk(work_in, &vdihandle, work_out);

	/* return code and description of all screen fonts */
	for (i=1; i <= work_out[10]; i++) {
		code = vqt_name(vdihandle, i, info);
		printf("Font %d: %s\n", code, info);
	}

	/* close device, exit */
	v_clsvwk(vdihandle);
	appl_exit();
	exit(0);
}
```

See Also
TOS, VDI, vst_font

vqt_width—VDI function (**libvdi.a/vqt_width**)

Get character cell width

#include <aesbind.h>
#include <vdibind.h>
int vqt_width(*handle, character, &width, &left, &right***)**
int *handle, width, left, right*; **char** *character*;

vqt_width is a VDI routine that returns the width of a given character's cell, plus information about the "white space" that surrounds the character; it does not take into account the angle at which text is written, or any special effects used. Much of the same information is returned by the AES routine **graf_handle**.

handle is the virtual device's VDI handle. *character* is the character whose size is to be checked. *width* is returned by **vqt_width**; it is the width of the character's cell. *left* and *right* are also set by **vqt_width**; they indicate the amount of white space left on, respectively, the left and the right of the character within its cell.

vqt_width returns -1 if the character requested is invalid or otherwise cannot be measured.

See Also
TOS, v_gtext, VDI

vr_recfl—VDI function (**libvdi.a/vr_recfl**)

Draw a rectangular fill area

#include <aesbind.h>
#include <vdibind.h>
void vr_recfl(*handle, xyarray***) int** *handle, xyarray***[4];**

vr_recfl is a VDI routine that draws a rectangle. Unlike its cousin **v_bar**, **vr_recfl** will draw only a rectangular chunk of the preset fill pattern; it cannot draw a perimeter. *handle* is the virtual device's VDI pattern.

xyarray sets the X and Y coordinates from which to construct the pattern; the even-numbered entries indicate the X coordinates, and the odd-numbered entries the Y coordinates. Which corner of the rectangle each pair of coordinates indicates will differ depending on whether the virtual device has been set to normalized device coordinates (NDC) or to raster coordinates (RC). On an NDC device, the first pair points to the lower left-hand corner and the second pair to the upper right-hand corner; whereas on an RC device, the first pair points to the upper left-hand corner and the second pair to the lower right-hand corner.

Note that to use this routine, the fill type must be set with **vsf_interior**, the fill style by **vsf_style**, and the fill color by **vsf_color**.

Example

This example uses the random-number routines to create a random pattern, and fills the screen with it. The random-number generator is seeded with the lower portion of the system time. Typing any key repeats the process; typing **<return>** exits. Note that because the Atari ST bumps the system time in two-second increments, you must wait at least two seconds before a new pattern can be drawn.

```
#include <aesbind.h>
#include <gemdefs.h>
#include <osbind.h>
#include <vdibind.h>

#define RETURN 0x1C0D              /* scan code returned by return key */
#define USER 4                     /* code for user-defined fill pattern */

/* global line A variables used by vdi; MUST be included */
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

/* array used by vs_clip() and vr_recfl() */
int xyarray[] = { 1, 1, 639, 399 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* array used by vsf_udpat() */
int fill[16];

/* throw-away declaration, to keep system from scribbling over itself */
int nowhere = 0;

main() {
        int vdihandle;                  /* virtual device's handle */
        int key;

/* OK, here we go ... */
        appl_init();
        vdihandle = graf_handle(&nowhere, &nowhere, &nowhere, &nowhere);
        v_opnvwk(work_in, &vdihandle, work_out);
        vs_clip(vdihandle, 1, xyarray);
        v_hide_c(vdihandle);
        vsf_interior(vdihandle, USER);

        dofill(vdihandle);

        for(;;) {
                key = evnt_keybd();
                if (key == RETURN) {
                        v_show_c(vdihandle);
                        v_clsvwk(vdihandle);
                        appl_exit();
                        exit(1);
```

```
		} else
			dofill(vdihandle);
	}
}

dofill(vdihandle)
int vdihandle;
{
	int counter;

	srand((int)Gettime());
	for (counter = 0; counter < 16; counter++)
		fill[counter] = rand();

	vsf_udpat(vdihandle, fill, 1);
	vr_recfl(vdihandle, xyarray);
}
```

See Also

TOS, v_bar, VDI

vr_trnfm—VDI function (**libvdi.a/vr_trnfm**)

Transform a raster image

#include <aesbind.h>
#include <gemdefs.h>
#include <vdibind.h>
void vr_trnfm(*handle*, *&sourcemfdb*, *&destmfdb***)**
int *handle*; **FDB** *sourcemfdb*, *destmfdb*;

vr_trnfm is a VDI routine that transforms a raster image between standard (device-independent) and device-dependent forms. *handle* is the virtual device's VDI handle. *sourcemfdb* and *destmfdb* describe the "memory form definition block for the source and destination areas. Note that these are both set to the type **FDB**, which is defined in the header file **gemdefs.h**, as follows:

```
typedef struct fdbstr
{
	long fd_addr;
	int fd_w;
	int fd_h;
	int fd_wdwidth;
	int fd_stand;
	int fd_nplanes;
	int fd_r1;
	int fd_r2;
	int fd_r3;
} FDB;
```

fd_addr points to the beginning of the raster area in RAM. If this value is set to zero, **vro_cpyfm** assumes that a virtual device is being used (e.g., the screen), and uses *handle* to address that device; it also ignores the rest of the **FDB** structure, which should be set to zeroes.

fd_w and *fd_h* give, respectively, the width and height of the area being copied to or copied from, in pixels. *fd_wdwidth* gives the width of the area being copied to/from, in 16-bit words (i.e., divided by 16), and rounded up. This information is needed internally by the VDI's raster copying routines.

fd_stand indicates whether the material is in device-dependent format or in device-independent (standard) format; zero indicates that it is in device-dependent format, and non-zero indicates standard format. Obviously, this should not be set to the same value in both *sourcemfdb* and *fd_nplanes* is the number of color planes used in the virtual device. The total number of pixels used in the image, then, is the image's height in pixels, times its width in pixels, times the number of planes.

Finally, *fd_r1* through *fd_r3* are used by the system for its own purposes; they should be set to zero.

See Also
TOS, VDI

vro_**cpyfm**—VDI function (**libvdi.a/vro_cpyfm**)
Copy raster form, opaque
#include <aesbind.h>
#include <gemdefs.h>
#include <vdibind.h>
void vro_cpyfm(*handle, logic, xyarray, &sourcemfdb, &destmfdb***)**
int *handle, logic, xyarray***[8]; FDB** *sourcemfdb, destmfdb*;

vro_cpyfm is a VDI routine that copies a portion of a virtual image, pixel by pixel, from one location to another.

handle is the virtual device's VDI handle. *logic* defines the mode in which the area being copied will be drawn. The following table lists the available modes; **S** indicates the source pixel, and **D** the destination pixel:

0 Clear destination
1 S & D
2 S & !D
3 S (replace mode)
4 !S & D (erase mode)
5 D (has no effect)
6 S ^ D (exclusive-or mode)
7 S | D (transparent mode)
8 !(S & D)
9 !(S ^ D)
10 !D
11 S | (!D)
12 !S
13 (!S) | D (reverse transparent mode)
14 !(S & D)
15 1 (black out destination area)

Note that setting *logic* to **6** (i.e., to exclusive-or mode) allows you to use **vro_cpyfm** to mimic a hardware sprite, to move images around the screen with minimal fuss.

xyarray defines the area to be copied from and the area to be copied to. *xyarray*[0] through *xyarray*[3] is the area being copied from; the first two numbers define the X and Y coordinates of one corner of the rectangle, and the second two define the corner opposite it. Note that if the virtual device is defined as using normalized device coordinates (NDC), the first corner is the lower left-hand corner and the second the upper right-hand corner; whereas if the device uses raster coordinates (RC), the first corner is the upper left-hand corner and the second is the lower right-hand corner. *xyarray*[4] through *xyarray*[7] define the destination rectangle, in the same manner as the source rectangle. Note that for predictable results, the source and destination rectangles should be of the same size.

Finally, *sourcemfdb* and *destmfdb* describe the "memory form definition block" for the source and destination areas. Note that these are both set to the type **FDB**, which is defined in the header file **gemdefs.h**, as follows:

```
typedef struct fdbstr
{
        long fd_addr;
        int fd_w;
        int fd_h;
        int fd_wdwidth;
        int fd_stand;
        int fd_nplanes;
        int fd_r1;
        int fd_r2;
        int fd_r3;
} FDB;
```

fd_addr points to the beginning of the raster area in RAM. If this value is set to zero, **vro_cpyfm** assumes that a virtual device is being used (e.g., the screen), and uses *handle* to address that device; it also ignores the rest of the **FDB** structure, which should be set to zeroes.

fd_w and *fd_h* give, respectively, the the width and height of the area being copied to or copied from, in pixels. *fd_wdwidth* gives the width of the area being copied to/from, in 16-bit words (i.e., divided by 16), and rounded up. This information is needed internally by the VDI's raster copying routines.

fd_stand indicates whether the material is in device-dependent format or in device-independent (standard) format; zero indicates that it is in device-dependent format, and non-zero indicates standard format. Obviously, this should not be set to the same value in both *sourcemfdb* and *fd_nplanes* is the number of color planes used in the virtual device. The total number of pixels used in the image, then, is the image's height in pixels, times its width in pixels, times the number of planes.

Finally, *fd_r1* through *fd_r3* are used by the system for its own purposes; they should be set to zero.

Example
The following examples allows the user to copy one portion of the screen to another. When he clicks the mouse the first time, he draws a rectangle on the screen; clicking the mouse again lets him drag the rectangle to another part of the screen. When the mouse button is lifted the second time, the contents of the rectangle are copied to where the rectangle stopped. Pressing the 'W' key changes the writing mode, and pressing **<return>** exits.

```
#include <gemdefs.h>
#include <aesbind.h>
#include <vdibind.h>
```

```
#define RETURN 0x1C0D                   /* scan code returned by return key */
#define W_KEY 0x1177                    /* scan code returned by W key */
#define DOWN 1                          /* mouse button is down */
#define CLICKS 1                        /* no. of clicks expected on mouse button */
#define BUTTON 1                        /* which button; 1 = leftmost */
#define HOLLOW 0                        /* make fill type hollow */
#define FUJI 4                          /* fill is user-defined type (default, fuji) */
#define REPLACE 1                       /* make writing mode REPLACE */
#define XOR 3                           /* make writing mode XOR */

/* global line A variables used by vdi; MUST be included */
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

/*
 * array used by vs_clip(); MUST be set, or images that extend
 * beyond the screen perimeters will write over low-level memory
 * (e.g., RAM disks, spoolers, etc.)
 */
int cliparray[] = { 1, 1, 639, 399 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* array used by v_bar() */
int xyarray[] = { 120, 100, 520, 300 };

/* arrays used by vro_cpyfm() */
int copyarray[8];

/* throw-away declarations, to keep system from scribbling over itself */
int nowhere = 0;
Rect norect = { 0, 0, 0, 0 };

main() {
/* declarations used by evnt_multi() */
        int selection;                  /* code for event that occurred */
        unsigned int which = (MU_KEYBD | MU_BUTTON);
        int buffer[11];                 /* place to write AES messages */
        unsigned key;                   /* scan code of key pressed by user */
        int mousex;                     /* mouse X coordinate */
        int mousey;                     /* mouse Y coordinate */

/* misc declarations */
        int vdihandle;                  /* virtual device's handle */
        int logic = 3;                  /* used to cycle through logic types */
        FDB holder  = { 0L, 0, 0, 0, 0, 0, 0, 0, 0 };
                                        /* used by vro_cpyfm; all zeros here */
```

```
/* OK, here we go ... */
	appl_init();
	graf_mouse(ARROW, &nowhere);
	vdihandle = graf_handle(&nowhere, &nowhere, &nowhere, &nowhere);
	v_opnvwk(work_in, &vdihandle, work_out);
	vs_clip(vdihandle, 1, cliparray);

	vsf_interior(vdihandle, FUJI);
	vsf_perimeter(vdihandle, 1);
	graf_mouse(M_OFF, &nowhere);
	v_bar(vdihandle, xyarray);
	graf_mouse(M_ON, &nowhere);

	for(;;) {
		selection = evnt_multi(which, CLICKS, BUTTON, DOWN,
			0, norect, 0, norect, buffer, 0, 0, &mousex, &mousey,
			&nowhere, &nowhere, &key, &nowhere);
		switch(selection) {

		case MU_KEYBD:
			if (key == RETURN) {
				v_clsvwk(vdihandle);
				appl_exit();
				exit(0);
			}
			if (key == W_KEY)
				logic++;
			break;

		case MU_BUTTON:
			getarray(vdihandle, mousex, mousey);
			v_bar(vdihandle, xyarray);
			vswr_mode(vdihandle, REPLACE);

			graf_mouse(M_OFF, &nowhere);
			vro_cpyfm(vdihandle, (logic%16), copyarray,
				&holder, &holder);
			graf_mouse(M_ON, &nowhere);
			break;

		default:
			break;
		}
	}
}
```

```
getarray(handle, mousex, mousey)
int handle, mousex, mousey;
{
	int width;			/* box width set by graf_rubbox */
	int height;			/* box height set by graf_rubbox */
	int newx;			/* X coordinate returned by graf_dragbox */
	int newy;			/* Y coordinate returned by graf_dragbox */

	/* set source rectangle's coordinates */
	copyarray[0] = xyarray[0] = mousex;
	copyarray[1] = xyarray[1] = mousey;
	graf_rubbox(mousex, mousey, 0, 0, &width, &height);
	copyarray[2] = xyarray[2] = (mousex + width);
	copyarray[3] = xyarray[3] = (mousey + height);

	/* Now draw a rectangle around source area */
	graf_mouse(M_OFF, &nowhere);
	vswr_mode(handle, XOR);
	vsf_interior(handle, HOLLOW);
	v_bar(handle, xyarray);
	graf_mouse(M_ON, &nowhere);

	/*
	 * wait for second button event; then set coordinates
	 * for destination rectangle.
	 */
	evnt_button(CLICKS, BUTTON, DOWN, &nowhere, &nowhere,
		&nowhere, &nowhere);
	graf_dragbox(width, height, mousex, mousey,
		0, 0, 639, 399, &newx, &newy);

	copyarray[4] = newx;
	copyarray[5] = newy;
	copyarray[6] = (newx + width);
	copyarray[7] = (newy + height);
	return;
}
```

See Also

TOS, VDI, vrt_cpyfm

vrq_choice—VDI function (**libvdi.a/vrq_choice**)

Return status of function keys when any key is pressed

#include <aesbind.h>
#include <vdibind.h>
void vrq_choice(*handle, in, &out***) int** *handle, in, out*;

vrq_choice is a VDI routine that returns the status of the function keys when any key is pressed. In VDI jaragon, it operates the select device in request mode; these terms are described more fully in the entry for **vsin_mode**.

handle is the virtual device's VDI handle. *in* is the number of the function key you want to check, one through ten. The function terminates when any key is pressed; if the key was a function key, *out* holds its number; if another key was struck, *out* holds its ASCII value.

See Also
TOS, VDI, vsm_choice

Notes
Before this function can be used, the function **vsin_mode(***handle***, 3, 1)** must be entered, which will place the valuator device into request mode.

vrq_locator—VDI function (**libvdi.a/vrq_locator**)
Find location of mouse cursor when a key is pressed
#include <aesbind.h>
#include <vdibind.h>
void vrq_locator(*handle, xcoord, ycoord, &xout, &yout, &key***)**
int *handle, xcoord, ycoord, xout, yout, key;*

vrq_locator is a VDI routine that returns the location of the mouse cursor when a mouse button is pressed. In VDI jargon, it operates the position input devices in request mode; these terms are described more fully in the entry for **vsin_mode**.

handle is the virtual device's VDI handle. *xcoord* and *ycoord* are, respectively, the X and Y coordinates of the mouse pointer's initialized position.

xout and *yout* are, respectively, the X and Y coordinates of the mouse pointer when a key is pressed. Finally, the low byte of *key* gives the ASCII code of the key that was pressed to terminate the polling of the screen. The left and right buttons on the mouse can also terminate polling; these return, respectively, 0x20 and 0x21. Note that because any key can end polling of the screen, the programmer must write a loop if he wants to terminate on a particular key.

See Also
TOS, VDI

Notes
Before this function can be used, the function **vsin_mode(***handle***, 1, 1)** must be entered, which will place the locator device into request mode.

vrq_string—VDI function (**libvdi.a/vrq_string**)
Read a string from the keyboard
#include <aesbind.h>
#include <vdibind.h>
void vrq_string(*handle, length, echo, xyarray, string***)**

int *handle, length, echo, xyarray*[2]; **char** **string*;

vrq_string is a VDI routine that reads a string from the keyboard. The string is automatically terminated with a NUL character. The systems stops accepting characters either when the user presses the **<return>** key, or when the string exceeds the maximum length set by the user. In VDI jaragon, it operates the string device in request mode; these terms are described more fully in the entry for **vsin_mode**.

handle is, as always, the virtual device's VDI handle. *length* is the maximum length of the string, in characters. *echo* indicates whether or not you want the string echoed to the screen as the user types; zero indicates no echo, whereas one indicates to echo. *xyarray* gives the X and Y coordinates of where on the screen to begin echoing the string. Finally, *string* points to where the string will be written.

See Also
TOS, **VDI**, **vsm_string**

Notes
Before this function can be used, the function **vsin_mode(***handle*, **4, 1)** must be entered, which will place the valuator device into request mode.

vrq_valuator—VDI function (**libvdi.a/vrq_valuator**)
Return status of shift and cursor keys
#include <aesbind.h>
#include <vdibind.h>
void vrq_valuator(*handle, in, &out, &key***) int** *handle, in, out, key*;

vrq_valuator is a VDI routine that returns the status of the valuator keys. In VDI jargon, it operates the valuator keys in request mode; these terms are described more fully in the entry for **vsin_mode**.

handle is the virtual device's VDI handle. *in* is the code of the valuator key whose status you wish to check. *key* is the code of the key that was pressed to terminate this function. Finally, *out* is the value of *key* plus a specific value that indicates which valuator key was pressed along with it, as follows:

Cursor up	*key* plus ten
Cursor down	*key* minus ten
Shift/cursor up	*key* plus one
Shift/cursor down	*key* minus one

See Also
TOS, **VDI**, **vsm_valuator**

Notes
Before this function can be used, the function **vsin_mode(***handle*, **2, 1)** must be entered, which will place the valuator device into request mode.

vrt_cpyfm—VDI function (**libvdi.a/vrt_cpyfm**)
Copy raster form, transparent
#include <aesbind.h>
#include <gemdefs.h>
#include <vdibind.h>
void vrt_cpyfm(*handle, mode, xyarray, &sourcemfdb, &destmfdb, color***)**
int *handle, mode, xyarray***[8]**, *color*; **FDB** *sourcemfdb, destmfdb*;

vrt_cpyfm is a VDI routine that copies a monochromatic image onto a polychromatic device, such as the screen. It resembles the blitting function, **vro_cpyfm**, but it is designed particularly for moving images around the screen.

handle is the virtual device's VDI handle. *mode* is the mode in which the image is written, as follows: one, replace mode; two, transparent mode; three, XOR (exclusive or); and four, reverse transparent. Note that these are the same codes used by the VDI routine **vswr_mode**, which is usually used to set the writing mode.

xyarray defines the area to be copied from and the area to be copied to. *xyarray*[0] through *xyarray*[3] is the area being copied from; the first two numbers define the X and Y coordinates of one corner of the rectangle, and the second two define the corner opposite it. Note that if the virtual device is defined as using normalized device coordinates (NDC), the first corner is the lower left-hand corner and the second the upper right-hand corner; whereas if the device uses raster coordinates (RC), the first corner is the upper left-hand corner and the second is the lower right-hand corner. *xyarray*[4] through *xyarray*[7] define the destination rectangle, in the same manner as the source rectangle. Note that for predictable results, the source and destination rectangles should be of the same size.

sourcemfdb and *destmfdb* describe the "memory form definition block for the source and destination areas. Note that these are both set to the type **FDB**, which is defined in the header file **gemdefs.h**, as follows:

```
typedef struct fdbstr
{
        long fd_addr;
        int fd_w;
        int fd_h;
        int fd_wdwidth;
        int fd_stand;
        int fd_nplanes;
        int fd_r1;
        int fd_r2;
        int fd_r3;
} FDB;
```

fd_addr points to the beginning of the raster area in RAM. If this value is set to zero, **vrt_cpyfm** assumes that a virtual device is being used (e.g., the screen), and uses *handle* to address that device; it also ignores the rest of the **FDB** structure, which should be set to zeroes.

fd_w and *fd_h* give, respectively, the the width and height of the area being copied to or copied from, in pixels. *fd_wdwidth* gives the width of the area being copied to/from, in 16-bit words (i.e., divided by 16), and rounded up. This information is needed internally by the VDI's raster copying routines.

fd_stand indicates whether the material is in device-dependent format or in device-independent (standard) format; zero indicates that it is in device-dependent format, and non-zero indicates standard format. Obviously, this should not be set to the same value in both *sourcemfdb* and *fd_nplanes* is the number of color planes used in the virtual device. The total number of pixels used in the image, then, is the image's height in pixels, times its width in pixels, times the number of planes.

Finally, *fd_r1* through *fd_r3* are used by the system for its own purposes; they should be set to zero.

See Also
TOS, VDI, vro_cpyfm

vs_clip—VDI function (**libvdi.a/vs_clip**)
Set the virtual device's clipping rectangle
#include <aesbind.h>
#include <vdibind.h>
void vs_clip(*handle, flag, xyarray***) int** *handle, flag, xyarray*[4];

vs_clip is a VDI routine that sets the clipping rectangle for a virtual device. The *clipping rectangle* is the portion of an image that is actually displayed on the physical device; if any portion of the image drawn on the virtual device extends beyond the clipping rectangle, it is trimmed off. If an image is not clipped, it could extend beyond the borders of the physical device; this, in turn, causes memory to be drawn over, possibly with catastrophic results.

handle is the virtual device's VDI handle. *flag* indicates whether clipping should be turned on or off: zero indicates off, one indicates on.

Finally, *xyarray* is an array of four integers that place the clipping rectangle, as follows:

xyarray[0] X coordinate of first corner
xyarray[1] Y coordinate of first corner
xyarray[2] X coordinate of opposite corner
xyarray[3] Y coordinate of opposite corner

Note that if the device is set to normalized device coordinates (NDC), the first corner is the upper left-hand corner of the image; whereas if the device is set to raster coordinates, the first corner is the lower left-hand corner.

Example
For an example of this routine, see the entry for **v_pline**.

See Also
TOS, VDI

vs_color—VDI function (**libvdi.a/vs_color**)
Set color intensity
#include <aesbind.h>
#include <vdibind.h>
void vs_color(*handle, index, rgbarray***) int** *handle, index, rgbarray***[3];**

vs_color is a VDI routine that sets the intensity of a color. Each color is set by adjusting the intensity of three electron guns, one for red pixels, another for green pixels, and a third for blue. **vs_color** allows you to adjust the intensity of each gun for given color.

handle is a virtual device's VDI handle. *index* is the code for the color being adjusted; for a table of these indices, see the entry for **v_opnwk**. Finally, *rgbarray[0]* through *rgbarray[2]* hold, respectively, the new value for the red, blue, and green guns; each value is an integer between one and 1,000.

See Also
TOS, VDI

vs_curaddress—VDI function (**libvdi.a/vs_curaddress**)
Move alphabetic cursor to specified row and column
#include <aesbind.h>
#include <vdibind.h>
void vs_curaddress(*handle, row, column***) int** *handle, row, column***;**

vs_curaddress is a VDI routine that moves the alphabetic cursor to a specified row and column on the virtual device. Note that to use this routine, the virtual device must first be placed in alphabetic mode, with the routine **v_enter_cur**. *handle* is the virtual device's VDI handle. *row* and *column* give, respectively, the row and column where you wish to position the alphabetic cursor.

See Also
TOS, VDI, vq_curaddress

vs_palette—VDI function (**libvdi.a/vs_palette**)
Select color palette on medium-resolution screen
#include <aesbind.h>
#include <vdibind.h>
void vs_palette(*handle, palette***) int** *handle, palette*;

vs_palette is a VDI routine that selects a palette for use on the medium-resolution screen. *handle* is the virtual device's VDI handle. *palette* is a pre-set color palette: zero (the default) indicates a palette of red, green, and brown; and one indicates a palette of cyan, magenta, and white.

See Also
TOS, VDI

vsc_form—VDI function (**libvdi.a/vsc_form**)
Draw a new shape for the mouse pointer
#include <aesbind.h>
#include <vdibind.h>
void vsc_form(*handle, form***) int** *handle, form***[37];**

vsc_form is a VDI routine that draws a new shape for the mouse pointer. *handle* is the virtual device's VDI handle.

form is an array of 37 integers. *form[0]* and *form[1]* give, respectively, the X and Y coordinates for the "action point", or the point on the pointer that is considered significant; in most instances, this is the upper left-hand corner. These values are set relative to the upper left-hand corner.

form[2] is reserved by the VDI, and must be set to one. *form[3]* is the color index mask, and is normally set to zero. *form[4]* is the color index cursor form, and is normally set to one. *form[5]* through *form[20]* gives the bit form of the mouse pointer's mask, or its monochromatic image. Finally, *form[21]* through *form[36]* gives the cursor form in color; bits set to one in this map are shown in the background color.

Note that once the new shape is loaded with **vsc_form**, it can be called with **graf_mouse(***handle***, 255,** *form***)**; or a similar call.

See Also
TOS, VDI

vsf_color—VDI function (**libvdi.a/vsf_color**)
Set a polygon's fill color
#include <aesbind.h>
#include <vdibind.h>
void vsf_color(*handle, color***) int** *handle, color*;

vsf_color is a VDI routine that sets a polygon's fill color. *handle* is the virtual device's VDI handle. *color* is the color to which the polygon's fill should be set; for a table of color settings, see the entry for **v_opnwk**. Note that this routine can be used only with **vsf_interior** and **vsf_style**.

See Also
TOS, v_bar, v_opnwk, VDI, vsf_interior, vsf_style

vsf_interior—VDI function (**libvdi.a/vsf_interior**)
Set a polygon's fill type
#include <aesbind.h>
#include <vdibind.h>
void vsf_interior(*handle, type***) int** *handle, type*;

vsf_interior is a VDI routine that lets you choose what type of filling will be used for a polygon. *handle* is the virtual device's VDI handle. *type* is the type of fill you choose, as follows:

0	empty (same as background)
1	solid
2	patterned
3	cross-hatched
4	user-defined type

Using the "empty" setting and having the "transparent" flag set by the routine **vswr_mode** will result in only the outline of a polygon being drawn, with what is in the background filling its interior.

Example
For an example of this routine, see the entry for **v_bar**.

See Also
TOS, v_bar, VDI, vsf_style

vsf_perimeter—VDI function (**libvdi.a/vsf_perimeter**)
Set whether to draw a perimeter around a polygon
#include <aesbind.h>

#include <vdibind.h>
void vsf_perimeter(*handle, flag***) int** *handle, flag*;

vsf_perimeter is a VDI routine that lets you choose whether or not to draw a perimeter around a polygon you are creating. The perimeter is in the color that you have set with the routine **vsf_color**, and it is always one raster wide. *handle* is the virtual device's VDI handle. *flag* indicates whether or not to draw a perimeter: zero indicates not to draw a perimeter, and one indicates to draw one.

Example
For an example of this routine, see the entry for **v_bar**.

See Also
TOS, v_bar, VDI, vsf_color

vsf_style—VDI function (**libvdi.a/vsf_style**)
Set a polygon's fill style
#include <aesbind.h>
#include <vdibind.h>
void vsf_style(*handle, style***) int** *handle, style*;

A polygon's fill *type* is set with the routine **vsf_interior**, and can be one of the following: hollow, solid, patterned, cross-hatched, or user defined. If one of the last three types is selected, then **vsf_style** can be used to selected the *style* of filling.

handle is the virtual device's VDI handle. *style* is the code number of the fill style selected. For a patterned fill, 24 styles are available, as follows:

1-8	gray tones, from lightest (1) to solid (8)
9	horizontal "brick" pattern
10	diagnonal "brick" pattern
11	inverted 'v's
12	arch
13	cross-hatched line segments
14	heavy random dots
15	light random dots
16	interwoven hollow lines
17	zig-zagged thin lines plus dots
18	horizontal and vertical lines of dots
19	black balls in checkerboard pattern
20	overlapping scale shapes
21	overlapping diagonal rectangles
22	rectangles in checkboard pattern
23	diamond pattern
24	lines in herringbone pattern

For a cross-hatched fill, 12 styles are available, as follows:

1	light, closely spaced diagonal lines
2	heavy, closely spaced diagonal lines
3	heavy, closely spaced, diagonal cross-hatched lines
4	closely spaced vertical lines
5	closely spaced horizontal lines
6	heavy, closely spaced, perpendicular cross-hatched lines
7	light, widely spaced diagonal lines
8	heavy, widely spaced diagonal lines
9	light, closely spaced, diagonal cross-hatched lines
10	widely spaced vertical lines
11	widely spaced horizontal lines
12	widely spaced perpendicular lines

The styles for a user-defined fill are set with the function **vsf_udpat**. The default user-defined fill is the "fuji" (the Atari symbol).

Example
For an example of this routine, see the entry for **v_bar**.

See Also
TOS, v_bar, VDI, vsf_interior

vsf_udpat—VDI function (**libvdi.a/vsf_udpat**)

Define a fill pattern

#include <aesbind.h>
#include <vdibind.h>
void vsf_udpat(*handle, pattern, planes***) int** *handle, pattern*[*n*], *planes*;

vsf_udpat is a VDI routine that allows a user to define a customized fill pattern. *handle* is the virtual device's VDI handle. *planes* is the number of color planes used in the pattern; the fill pattern must have a 16-integer array for each color plane. *pattern* is an array of 16 integers that defines the dot pattern, beginning in the upper left-hand corner and working through the lower right-hand corner. *n* must be set to 16 times *planes*. Note that once a pattern has been set, it must be loaded using **vsf_interior** and **vsf_style**.

Example

For an example of this function, see the entry for **vr_recfl**.

See Also

TOS, v_bar, VDI, vsf_interior, vsf_style

vsin_mode—VDI function (**libvdi.a/vsin_mode**)

Set input mode for logical input device

#include <aesbind.h>
#include <vdibind.h>
int vsin_mode(*handle, device, mode***) int** *handle, device, mode*;

vsin_mode is a VDI routine that sets the input mode for a given logical input device. This mode is used by a set of functions that poll the input devices for information about their current status.

The VDI recognizes four types of input devices: *Position input devices* control the position of the mouse cursor on the screen; these are the mouse itself or the cursor keys. *Value-changing devices* affect only the value returned by another input device; these include the shift key, the control key, and the alt key. *Selection input devices* return a selection number; these refer only to the Atari ST's function keys. Finally, *string input devices* are the alphabetic keys on the Atari ST's keyboard, by which strings are input.

handle is the virtual device's VDI handle. *device* indicates the logical device you wish to set: one indicates the position devices; two, value-changing input devices; three, the selection devices; and four, the string-input devices.

Finally, *mode* is the mode to which you want to set the device. *Request mode* tells the polling function to wait for input from a given device, e.g., for a key to be struck or a mouse button to be pressed. *Sample mode* simply polls the device and returns, without waiting for an event. One indicates request mode, and two indicates sample mode.

vsin_mode returns the mode to which the device was set.

See Also
TOS, VDI, vqin_mode, vrq_choice, vrq_locator, vrq_string, vrq_valuator, vsm_choice, vsm_locator, vsm_string, vsm_valuator

vsl_color—VDI function (**libvdi.a/vsl_color**)
Set a line's color
#include <aesbind.h>
#include <vdibind.h>
int vsl_color(*handle, color***) int** *handle, color*;

vsl_color is a VDI routine that sets the color of a line. *handle* is the virtual device's VDI handle. *color* is the color to which the line is being set. For a list of the available values, see the entry for **v_opnwk**. If the color requested is not available on the target virtual device, the line color will be set to one (black).

vsl_color returns the color to which the line was set.

See Also
TOS, VDI, v_pline

vsl_ends—VDI function (**libvdi.a/vsl_ends**)
Attach ends to a line
#include <aesbind.h>
#include <vdibind.h>
void vsl_ends(*handle, beginning, end***) int** *handle, beginning, end*;

vsl_ends is a VDI routine that attaches ends to a line. *handle* is the virtual device's VDI handle. *beginning* and *end* refer to the type of figure drawn at, respectively, the beginning and the end of the line, as follows:

0	squared end (default)
1	arrowhead
2	rounded end

Example
For an example of this routine, see the entry for **v_pline**.

See Also
TOS, VDI, v_pline

vsl_type—VDI function (**libvdi.a/vsl_type**)
Set a line's type
#include <aesbind.h>
#include <vdibind.h>

int vsl_type(*handle, type*) **int** *handle, type*;

vsl_type is a VDI routine that sets a line's type. *handle* is the virtual device's VDI handle.

type is the type to which the line is being set, as follows:

1	solid
2	long dashes
3	dots
4	dash-dot
5	dashes
6	dash-dot-dot
7	user-defined
8-*n*	device-dependent

vsl_type returns the type to which the line was set.

Example
For an example of this routine, see the entry for **v_pline**.

See Also
TOS, v_pline, VDI, vsl_udsty

vsl_udsty—VDI function (**libvdi.a/vsl_udsty**)
Set user-defined line type
#include <aesbind.h>
#include <vdibind.h>
void vsl_udsty(*handle, pattern*) **int** *handle, pattern*;

vsl_udsty is a VDI routine that lets the user design a line type to be drawn by **v_pline**. *handle* is the virtual device's VDI handle. *pattern* is a bit map for the pattern to be drawn. Setting a bit to one means that its 1/16 portion of a line unit will be drawn; setting it to zero means that its portion will be blank.

Note that once the bit pattern is set with **vsl_udsty**, it must be loaded with the function **vsl_type**.

See Also
TOS, v_pline, VDI, vsl_type

vsl_width—VDI function (**libvdi.a/vsl_width**)
Set a line's width
#include <aesbind.h>
#include <vdibind.h>
int vsl_width(*handle, width*) **int** *handle, width*;

vsl_width is a VDI routine that sets a line's width. *handle* is the virtual device's VDI handle.

width is the width of the line to be drawn; this will vary depending on whether the virtual device being drawn on is set in normalized device coordinates (NDC) or raster coordinates (RC). The value *work_out[7]* indicates how many line widths are available for you to use on that device; see the entry for **v_opnwk** for more information. If the line width you request is not available on the virtual device, the line width will be set to the next *smaller* width.

vsl_width returns the width to which the line was actually set.

Example
For an example of this routine, see the entry for **v_pline**.

See Also
TOS, VDI, v_opnwk, v_pline

vsm_choice—VDI function (**libvdi.a/vsm_choice**)
Return last function key pressed
#include <aesbind.h>
#include <vdibind.h>
int vsm_choice(*handle, &key***) int** *handle, key*;

vsm_choice is a VDI routine that returns the last function key pressed, whether or not another key is pressed. To use VDI jargon, it operates the valuator device in sample mode; these terms are explained more fully in the entry for **vsin_mode**. *handle* is the virtual device's VDI handle. *choice*, which is set by **vsm_choice**, is the number of the function key last pressed, from one to ten. If no function key was pressed, the ASCII code of the last key pressed is returned.

vsm_choice returns either zero or one; the former indicates that no key was pressed, whereas the latter indicates that a key was pressed.

See Also
TOS, VDI, vrq_choice

Notes
Before this function can be used, the function **vsin_mode(***handle*, **3, 2)** must be entered, which will place the locator device into sample mode.

vsm_color—VDI function (**libvdi.a/vsm_color**)
Set a marker's color
#include <aesbind.h>
#include <vdibind.h>
int vsm_color(*handle, color***) int** *handle, color*;

vsm_color is a VDI routine that sets a marker's color. *handle* is the virtual device's VDI handle. *color* is the color you select for the marker; for a list of the legal color codes, see the entry for **v_opnwk**. If the color you requested is not available, the marker's color will be set by default to one (black).

vsm_color returns the color to which marker is actually set.

See Also
TOS, VDI, v_pmarker

vsm_height—VDI function (**libvdi.a/vsm_height**)
Size a marker
#include <aesbind.h>
#include <vdibind.h>
int vsm_height(*handle, height***) int** *handle, height*;

vsm_height is a VDI routine that sizes a marker. *handle* is the virtual device's VDI handle.

height is new size of the image, in Y coordinate units; these are used to avoid problems with scaling. Note that not every device will support every requested size of marker. Interrogating the variable *work_out[9]*, which is a member of the array returned by the routine used to open the virtual device, will indicate the number of marker sizes available; zero indicates continuous scaling, i.e., that every size is supported. See the entry for **v_opnwk** for more information. Note that if a particular size is unavailable, the marker will be rescaled automatically to the next available *smaller* size.

vsm_height returns the height to which the marker is set.

Example
For an example of this routine, see the entry for **v_circle**.

See Also
TOS, VDI, v_opnwk, v_pmarker

vsm_locator—VDI function (**libvdi.a/vsm_locator**)
Return mouse pointer's position
#include <aesbind.h>
#include <vdibind.h>
int vsm_locator(*handle, xcoord, ycoord, &xout, &yout, &key***)**
int *handle, xcoord, ycoord, xout, yout, key*;

vsm_locator is a VDI routine that returns the mouse pointer's position whether or not a key was pressed. To use VDI jargon, it operates the position input device in sample mode; these terms are explained more fully in the entry for **vsin_mode**. Because VDI programs work by interrupts, a program does not know where the mouse pointer is at any given point; this function is handed an initial set of coordinates for the mouse pointer, then polls the screen to find where it is now. It returns and indicates whether the pointer has changed from the initializing coordinates, whether a key was pressed, or both; and it sets values for the new X and Y coordinates (if any) and for the key that was pressed (if any).

handle is, as always, the virtual device's VDI handle. *xcoord* and *ycoord* give, respectively, the X and Y coordinates of the mouse pointer's initialized position; these may be set by another function.

xout and *yout* are set by **vsm_locator**; they give, respectively, the mouse pointer's X and Y coordinates, if they are different from the initializing coordinates. *key* is also set by **vsm_locator**; its low byte gives the ASCII value of a key pressed in the interval, if any.

Finally, **vsm_locator** returns a code, from zero to three, which indicates the following: zero, the mouse pointer was not moved and no key was pressed; one, the mouse pointer was moved, but no key was pressed; two, the mouse pointer was not moved, but a key was pressed; and three, the mouse pointer moved and a key was pressed.

See Also
TOS, VDI, vrq_locator

Notes
Before this function can be used, the function **vsln_mode(***handle***, 1, 2)** must be entered, which will place the locator device into sample mode.

vsm_string—VDI function (**libvdi.a/vsm_string**)
Read a string from the keyboard
#include <aesbind.h>
#include <vdibind.h>
int vsm_string(*handle, length, echo, xyarray, string***)**
int *handle, length, echo, xyarray***[2]; char ****string***;**

vsm_string is a VDI routine that reads a string from the keyboard. The string is automatically terminated with a NUL character. Unlike **vrq_string**, it also notes if any non-alphabetic keys were struck. String entry ends either when a non-alphabetic key is struck, or when the string exceeds the maximum length set by the user.

handle is the virtual device's VDI handle. *length* is the maximum length of the string. *echo* indicates whether or not you want the characters echoed to the screen as they are input: zero indicates not to echo, and one indicates to echo. *xyarray* gives the X and Y coordinates of the position on the screen where to begin echoing the string. Finally, *string* points to the area where the string will be written; be sure to set aside at least *length* amount of space for the string, or you may write over vital memory.

vsm_string returns zero if the string was terminated by a non-alphabetic key, and a number greater than one if it was not. If you plan to have the user terminate the string with the **<return>** key, use **vrq_string** instead of the present function.

See Also
TOS, VDI, vrq_string

Notes
Before this function can be used, the function **vsin_mode(***handle***, 4, 2)** must be entered, which will place the locator device into sample mode.

vsm_type—VDI function (**libvdi.a/vsm_type**)
Set VDI marker type
#include <aesbind.h>
#include <vdibind.h>
int vsm_type(*handle, type***) int** *handle, type*;

vsm_type is a VDI routine that sets the type of marker displayed on the virtual device. *handle* is the virtual device's VDI handle. *type* is the type of marker being shown, as follows:

1	dot
2	plus sign
3	asterisk
4	square
5	diagonal cross
6	diamond
7	device-dependent

If the type of marker requested is not available on the virtual device, the default marker (an asterisk) will be used. **vsm_type** returns the type of marker to be displayed.

Example
For an example of this routine, see the entry for **v_circle.**

See Also
TOS, VDI, v_pmarker

vsm_valuator—VDI function (**libvdi.a/vsm_valuator**)
Return shift/cursor key status
#include <aesbind.h>
#include <vdibind.h>
void vsm_valuator(*handle, in, &out, &key, &status***)**

int *handle, in, out, key, status;*

vsm_valuator is a VDI routine that that returns the status of a shift key or cursor key whether or not another key is pressed. To use VDI jargon, it operates the valuator device in sample mode; these terms are explained more fully in the entry for **vsin_mode**.

handle is the virtual device's VDI handle. *in* is the code of the valuator key whose status you wish to examine. *key* is set by **vsm_valuator**; it is the code of the key pressed before this routine exits, if any. Because all functions in sample mode merely examine the status of a device and return without being triggered by a hardware event, a key may not necessarily have been pressed during this function's operation. *out* is the value of *key*, plus a value that indicates the status of one or more valuator keys, as follows:

Cursor up	*key* plus ten
Cursor down	*key* minus ten
Shift/cursor up	*key* plus one
Shift/cursor down	*key* minus one

Finally, *status* gives the status of the valuator devices, as follows:

0	no action occurred
1	value was changed
2	key was pressed

See Also
TOS, VDI, vrq_valuator

Notes
Before this function can be used, the function **vsin_mode(***handle*, **2, 2)** must be entered, which will place the locator device into sample mode.

vsp_message—VDI function (**libvdi.a/vsp_message**)
Suppress messages from Polaroid Palette device
#include <aesbind.h>
#include <vdibind.h>
void vsp_message(*handle***) int** *handle*;

vsp_message is a VDI routine that suppresses messages from the Polaroid Palette device. These messages are normally output to the screen. *handle* is the virtual device's VDI handle.

See Also
TOS, VDI, vqp_error

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

vsp_save—VDI function (**libvdi.a/vsp_save**)
Save to disk current setting of Polaroid Palette driver
#include <aesbind.h>
#include <vdibind.h>
void vsp_save(*handle***) int** *handle*;

The VDI contains a driver for the Polaroid Palette, a camera that can be used to shoot slides directly from the Atari ST. **vsp_save** is a VDI routine that writes the current settings for this driver to disk. *handle* is the virtual device's VDI handle.

See Also
TOS, VDI, vqp_error, vqp_films, vqp_state, vsp_message, vsp_state

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

vsp_state—VDI function (**libvdi.a/vsp_state**)
Set the Polaroid Palette driver
#include <aesbind.h>
#include <vdibind.h>
void vsp_state(*handle, port, film, lightness, interlace, planes, indices***)**;
int *handle, port, film, lightness, interlace, lanes, indices***[8][2]**;

The VDI contains a driver for the Polaroid Palette, a camera that can be used to shoot slides directly from the Atari ST. **vsp_state** changes the settings for this driver.

handle is the virtual device's VDI handle. *port* is the port to which the camera is connected; zero indicates the first communications port. *film* is the number of the film for which the driver is currently set.

lightness is the intensity to which the driver is set, from -3 through three. Each number in this range is equivalent to one third of an *f*-stop, counting from zero. Therefore, -3 has half the intensity of zero, and three is twice as intense as zero.

interlace indicates whether the image is interlaced or not; zero indicates not interlaced, and one indicates interlaced. Note that an interlaced image requires approximately twice the memory of one that is not interlaced.

planes indicates the number of colors supported. It is set to a code, from one through four; one indicates two colors; two, four colors; three, eight colors; and four, 16 colors.

Finally, *indices* holds two-character codes for the eight color indices stored in ADE format.

See Also
TOS, VDI, vqp_error, vqp_films, vqp_state, vsp_message, vsp_save

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

vst_alignment—VDI function (**libvdi.a/vst_alignment**)
Realign graphics text
#include <aesbind.h>
#include <vdibind.h>
void vst_alignment(*handle, horiz, vertical, sethoriz, setvert***)**
int *handle, horiz, vertical,* **sethoriz, *setvert*;

vst_alignment is a VDI routine that realigns graphics text.

Graphics text is aligned both horizontally and vertically. Horizontal alignment can be to the left (left justified), to the right (right justified), or centered. Vertical alignment can be one of the following: *baseline*, that is, aligned along the bottoms of the characters, excluding descenders (the "tails" on letters like 'j' or 'y'); *half line*, or aligned along the tops of the lower-case letters; *ascent line*, or along the tops of the upper-case letters; *bottom line*, or along the bottom of the character cell (i.e., the bottom of the white space found below the descenders); *descent line*, or along the bottom of the descenders, excluding the white space found below them; and *top line*, or along the top of the character cell (e.g., the top of the white space found above the capital letters). By default, characters are aligned to the left horizontally, and along the baseline vertically.

The following describes the arguments to **vst_alignment**: *handle* is the virtual device's VDI handle. *horiz* is the horizontal alignment you want, as follows:

0	left
1	centered
2	right

vertical is the vertical alignment you want, as follows:

0	baseline
1	half line
2	ascent line
3	bottom line
4	descent line
5	top line

sethoriz and *setvert* point, respectively, to the horizontal and vertical alignments that were actually set. You may wish to check these values, because not every alignment is available with every type face on every virtual device.

Example
For an example of this routine, see the entry for **v_gtext**.

See Also
TOS, v_gtext, VDI

vst_color—VDI function (**libvdi.a/vst_color**)
Set color for graphics text
#include <aesbind.h>
#include <vdibind.h>
int vst_color(*handle, color***) int** *handle, color*;

vst_color is a VDI routine that that sets the color for graphics text. *handle* is the virtual device's VDI handle. *color* is the color being set. See the entry for **v_opnwk** for a table of legal color settings.

If the color requested is not available on this virtual device, **vst_color** sets the color to a default of one (black). It returns the color that was actually set.

See Also
TOS, v_gtext, v_opnwk, VDI,

vst_effects—VDI function (**libvdi.a/vst_effects**)
Set special effects for graphics text
#include <aesbind.h>
#include <vdibind.h>
int vst_effects(*handle, effects***) int** *handle, effect*;

vst_effects is a VDI routine that sets special effects for graphics text. *handle* is the virtual device's VDI handle. *effect* is the set of effects that you wish to use, as follows:

0x01	thickened letters
0x02	lowered intensity
0x04	slanted letters
0x08	underlining
0x10	outlined letters
0x20	shadowed letters

For example, if you want letters that are underlined and shadowed, set *effects* to 0x28 (i.e., 0x08 plus 0x20). Not every effect will be available on every virtual device. **vst_effects** returns the settings for the effects that were actually set.

Example
For an example of this routine, see the entry for **v_gtext**.

See Also
TOS, v_gtext, VDI

vst_font—VDI function (**libvdi.a/vst_font**)
Select a new font
#include <aesbind.h>
#include <vdibind.h>
int vst_font(*handle, font***) int** *handle, font*;

vst_font is a VDI routine that selects a new font for graphics type. *handle* is the virtual device's VDI handle. *font* is the code number of the new font available. The number of fonts available on a virtual device can be determined either by examined the value returned by the font-loading routine **vst_load_fonts**, or by interrogating the *work_out* array returned by **v_opnwk** and **v_opnvwk**: *work_out[10]* contains this information. Use the routine **vqt_name** to obtain the index number and a description of each available font.

If you select a font that is not available on the virtual device you are working with, the font will be set to a default; on the screen, the default is the system font. **vst_font** returns the code of the font actually selected.

See Also
TOS, v_gtext, VDI, vqt_name, vst_load_fonts, vst_unload_fonts

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

This function is not available with every device. To see if it is available on a given virtual device, interrogate *work_out[10]* of the array returned by **v_opnwk** or **v_opnvwk**.

vst_height—VDI function (**libvdi.a/vst_height**)
Reset graphics text height, in absolute values
#include <aesbind.h>
#include <vdibind.h>
void vst_height(*handle, newheight, charwidth, charheight, cellwidth, cellheight***)**
int *handle, newheight, *charwidth, *charheight, *cellwidth, *cellheight*;

vst_height is a VDI routine that sets a new height for graphics text. Note that graphics text can be resized up to twice its original height; this limit is set to reduce the amount of jaggedness, or "aliasing", present in the characters. **vst_height** resets the characters into absolute values; these values can be either in normalized device coordinates (NDC) or raster coordinates (RC), depending on which the virtual device uses. On the high-resolution screen, the normal character height is 13 rasters, which can be increased up to 26 rasters.

The related function **vst_point** resets character height, but uses points rather than absolute values. Note that the current sizes of a character and a character cell can be obtained with the AES routine **graf_handle**. The number of text sizes supported by the virtual device is found in the variable *work_out[5]*, which is part of the array returned by the routines **v_opnwk** and **v_opnvwk**.

handle is the virtual device's VDI handle. *height* is the new height to which the characters are being set. Note that not every height is available; if the height requested is not available, the characters will be set to the next *smaller* size. *charheight* and *charwidth* are, respectively, height and width to which the characters were set; *cellheight* and *cellwidth* are, respectively, the height and width to which the character cell is set. Note that the difference in sizes between a character and its cell controls how much "white space" appears around each character.

Example
For an example of this routine, see the entry for **v_gtext**.

See Also
TOS, **graf_handle**, **v_gtext**, **VDI**, **vst_point**

vst_load_fonts—VDI function (**libvdi.a/vst_load_fonts**)
Load fonts other than the standard font
#include <aesbind.h>
#include <vdibind.h>
int vst_load_fonts(*handle, reserved***) int** *handle, font*;

vst_load_fonts is a VDI routine that loads a virtual device's non-standard fonts into memory. The new fonts must be specifically loaded for them to be used; this is done in order to save system memory that would otherwise be taken up by unused fonts. *handle* is the virtual device's VDI handle. *reserved* is reserved by GEM-DOS for future use; at present, it should be set to zero. **vst_load_fonts** returns the number of additional fonts loaded. The routine **vst_unload_fonts** should be used to free the memory given over the extra fonts once they are no longer needed.

See Also
TOS, **v_gtext**, **VDI**, **vst_unload_fonts**

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

vst_point—VDI function (**libvdi.a/vst_point**)

Reset graphics text height, in printer's points

#include <aesbind.h>
#include <vdibind.h>
void vst_point(*handle, newheight, charwidth, charheight, cellwidth, cellheight***)**
int *handle, newheight, *charwidth, *charheight, *cellwidth, *cellheight*;

vst_point is a VDI routine that sets a new height for graphics text. Note that graphics text can be resized up to twice its original height; this limit is set to reduce the amount of jaggedness, or "aliasing", present in the characters. **vst_point** resets the characters into printer's points; one points equals 1/72 of an inch. The related function **vst_height** resets character height, but uses absolute values rather than points. Note that the current sizes of a character and a character cell can be obtained with the AES routine **graf_handle**. The number of text sizes supported by the virtual device is found in the variable *work_out[5]*, which is part of the array returned by the routines **v_opnwk** and **v_opnvwk**.

handle is the virtual device's VDI handle. *height* is the new height to which the characters are being set. Note that not every height is available; if the height requested is not available, the characters will be set to the next *smaller* size. *charheight* and *charwidth* are, respectively, height and width to which the characters were set; *cellheight* and *cellwidth* are, respectively, the height and width to which the character cell is set. Note that the difference in sizes between a character and its cell controls how much "white space" appears around each character.

See Also
TOS, graf_handle, v_gtext, VDI, vst_height

vst_rotation—VDI function (**libvdi.a/vst_rotation**)

Set angle at which graphic text is drawn

#include <aesbind.h>
#include <vdibind.h>
int vst_rotation(*handle, angle***) int** *handle, angle*;

vst_rotation is a VDI routine that sets the angle at which graphics text is drawn. *handle* is the virtual device's VDI handle. *angle* is the angle at which the text is drawn, in tenths of a degree. On an imaginary clock, zero degrees is set at three o'clock, 90 degrees at noon, 180 degrees at nine o'clock, and 270 degrees at six o'clock. Not every angle is available on every device; therefore, **vst_rotation** returns the angle at which the text is actually drawn.

Example
For an example of this function, see the entry for **v_gtext**.

See Also
TOS, v_gtext, VDI

Notes
This function is not available on every virtual device. To see if it is or not, interrogate *work_out[36]* of the array returned by **v_opnwk** or **v_opnvwk**.

As of this writing, the Atari ST can rotate text only in 90-degree increments.

vst_unload_fonts—VDI function (**libvdi.a/vst_unload_fonts**)
Unload fonts
#include <aesbind.h>
#include <vdibind.h>
void vst_unload_fonts(*handle, reserved***) int** *handle, reserved*;

vst_unload_fonts is a VDI routine that unloads extra fonts used in a VDI program. This routine should be used once there is no more need for the extra fonts, to free up memory given over to the extra fonts. *handle* is the virtual device's VDI handle. *reserved* is reserved for a future application, and should be set to zero.

See Also
TOS, v_gtext, VDI, vst_load_fonts

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

vswr_mode—VDI function (**libvdi.a/vswr_mode**)
Set the writing mode
#include <aesbind.h>
#include <vdibind.h>
int vswr_mode(*handle, mode***) int** *handle, mode*;

vswr_mode is a VDI routine that sets the writing mode. *handle* is the device's VDI handle. *mode* indicates the writing mode of the device, as follows: one, replace; two, transparent; three, XOR (exclusive or); and four, reverse transparent. *Replace* mode simply replaces whatever is on the virtual device with the image being drawn. *Transparent* mode replaces all the zero (white) pixels on the device it is overlaying with ones (black), but does not affect black pixels that already exist on the screen. The effect is as if the image were drawn on a sheet of plastic that was then overlaid on the physical device. *Reverse transparent* mode is the same as transparent mode, except that it affects black pixels and ignores white ones. Finally, *XOR* mode draws an image that later can be cancelled out by reversing (or exclusive ORing it), moved elsewhere, and redrawn.

vswr_mode returns the mode set.

Example
For examples of this routine, see the entries for **v_circle** and **v_ellipse**.

See Also
TOS, VDI

Vsync—**xbios** function 37 (**osbind.h**)
Synchronize with the screen
#include <osbind.h>
#include <xbios.h>
void Vsync()

Vsync waits for the next picture return from the screen. It is used to synchronize the system's operation with that of the screen, for specialized effects.

Example
For an example of this function, see the entry for **VDI**.

See Also
TOS, xbios

wc—Command

Count words, lines, and characters in files

wc [-clw] [*file*...]

wc counts words, lines, and characters in each *file* named. If no *file* is given, **wc** uses the standard input. If more than one *file* is given, **wc** also prints a total for all of the files.

A *word* is a string of characters surrounded by white space (blanks, tabs, or newlines).

Options control the printing of various counts:

-c Print a count of character.

-l Print a count of lines.

-w Print a count of words.

The default action is to print all counts.

See Also
commands

wildcards—Definition

Wildcards are characters that, under special circumstances, can represent a range of ASCII characters. Another name for them is "metacharacters". The following is a table of wildcards, and their meanings:

? Match any one character.

* Match any number of characters, including **NULL**.

[ ] A set of characters enclosed between '[' and ']' will match any one character of the set. Sets of characters may include ranges, such as **[a-z]** for lower-case letters or **[0-9]** for numerals.

/ Remove the special meaning of a wildcard.

See Also
patterns

wind_calc—AES function (**libaes.a/wind_calc**)

Calculate a window's borders or area

#include <aesbind.h>
int wind_calc(*type, kind, input, output*)
int *type*; unsigned int *kind*; Rect *input*; Prect *output*;

wind_calc is an AES routine that calculates the borders of a window or its total area: If given the work area coordinates, it calculates the borders; if given the borders, it returns the total area of the window. *type* is the type of calculation you want performed: zero indicates calculating the borders, and one indicates calculating the area. *kind* indicates the elements contained within the window, as follows:

0x001	NAME	title name
0x002	CLOSE	"close" bar
0x004	FULL	"full" box
0x008	MOVE	"move" bar
0x010	INFO	information line
0x020	SIZE	"size" box
0x040	UPARROW	up arrow
0x080	DNARROW	down arrow
0x100	VSLIDE	vertical "slider"
0x200	LFARROW	left arrow
0x400	RTARROW	right arrow
0x800	HSLIDE	horizontal "slider"

All elements used in the window must be mentioned for **wind_calc** to return a correct value.

input contains the coordinates that are given to **wind_calc**. It is of the type **Rect**, which is defined in the header file **aesbind.h**. **Rect** consists of four elements:

x	X coordinate of rectangle
y	Y coordinate of rectangle
w	width of rectangle
h	height of rectangle

output points to the new coordinates as calculated by **wind_calc**. It is of type **Prect**, which is declared in **aesbind.h**, as follows:

***x**	pointer to X coordinate
***y**	pointer to Y coordinate
***w**	pointer to rectangle's width
***h**	pointer to rectangle's height

wind_calc returns zero if an error occurred, and a number greater than zero if one did not.

Example
For an example of this routine, see the entry for **window**.

See Also
AES, TOS, window

wind_close—AES function (**libaes.a/wind_close**)
Close a window and preserve its handle
#include <aesbind.h>
int wind_close(*handle*) int *handle*;

wind_close is an AES routine that closes a window. It preserves the window's handle, which was set by the routine **wind_create**, and all of its allocated resources, so that it can be reopened. *handle* is the handle of the window to be opened. **wind_close** returns zero if an error occurred, and a number greater than zero if one did not.

Example
For examples of how to use this routine, see the entries **evnt_multi** and **window**.

See Also
AES, TOS, wind_open, window

wind_create—AES function (**libaes.a/wind_create**)
Create a window
#include <aesbind.h>
int wind_create(*kind, area*) unsigned int *kind*; Rect *area*;

wind_create is an AES routine that creates a new window. *kind* indicates the elements of the window you wish to create, as follows:

0x001	NAME	title name
0x002	CLOSE	"close" bar
0x004	FULL	"full" box
0x008	MOVE	"move" bar
0x010	INFO	information line
0x020	SIZE	"size" box
0x040	UPARROW	up arrow
0x080	DNARROW	down arrow
0x100	VSLIDE	vertical "slider"
0x200	LFARROW	left arrow
0x400	RTARROW	right arrow
0x800	HSLIDE	horizontal "slider"

For example, if you wanted to create a window that had only a title bar and an information bar, you would set *kind* to 0x11 (i.e., **NAME|INFO**).

area gives the dimensions of the window. It is of type **Rect**, which is defined in the header file **aesbind.h**, as follows:

x	X coordinate of rectangle
y	Y coordinate of rectangle
w	width of rectangle
h	height of rectangle

wind_create returns either the handle of the window it creates, or a negative number if it cannot create a window.

Example
For examples of how to use this routine, see the entries **evnt_multi** and **window**.

See Also
AES, TOS, window

Notes
As of this writing, no more than six windows can be displayed at any given time.

wind_delete—AES function (**libaes.a/wind_delete**)
Delete a window and free its resources
#include <aesbind.h>
int wind_delete(*handle*) int *handle*;

wind_delete is an AES routine that deletes a window and frees the resources allocated to it. *handle* is the handle of the window being deleted; this is returned by the routine **wind_create**. **wind_delete** returns zero if an error occurred, and a number greater than zero if one did not.

Example
For an example of this routine, see the entry for **window**.

See Also
AES, TOS, window

wind_find—AES function (**libaes.a/wind_find**)
Determine if the mouse pointer is in a window
#include <aesbind.h>
int wind_find(*x, y*) int *x, y*;

wind_find is an AES routine that determines if the mouse pointer is positioned over a window. *x* and *y* are the mouse pointer's X and Y coordinates; they can be obtained from the AES routine **evnt_mouse**. **wind_find** returns the handle of the window that the mouse pointer is within, or zero if the pointer is not within any window.

See Also
AES, TOS, window

wind_get—AES function (**libaes.a/wind_get**)
Get information about a window
#include <aesbind.h>
int wind_get(*handle, flag, output1, output2, output3, output4***)**
int *handle, flag, *output1, *output2, *output3, *output4*;

wind_get is an AES routine that gets information about a window. *handle* is the handle of the window in question; the handle is set by the routine **wind_create**. *flag* tells **wind_get** just what information you want. Unless noted, **wind_get** will set the values for the X coordinate, Y coordinate, width, and height, as follows:

4	WF_WORKXYWH	window's working area
5	WF_CURRXYWH	window's total area
6	WF_PREVXYWH	previous window's total area
7	WF_FULLXYWH	window's greatest possible size (set **wind_create**)
8	WF_HSLIDE	*output1* set to relative position of horizontal slider (1-1,000; 1=leftmost)
9	WF_VSLIDE	*output1* set to relative position of vertical slider (1-1,000; 1=top)
10	WF_TOP	*output1* set to handle of topmost window
11	WF_FIRSTXYWH	First rectangle in rectangle list
12	WF_NEXTXYWH	Next rectangle in rectangle list
13	Reserved	
15	WF_HSLSIZE	*output1* set to size of horizontal slider relative to scroll bar; -1 is minimal (small box), 1-1,000 is relative size
16	WF_VSLSIZE	*output1* set to size of vertical slider relative to scroll bar; -1 is minimal (small box), 1-1,000 is relative size
17	WF_SCREEN	Address/length of internal menu/alert buffers: *output1*=low word of address *output2*=high word of address *output3*=low word of length *output4*=high word of length

wind_get returns zero if an error occurred, and a number greater than zero if one did not.

Example
For an example of this routine, see the entry for **window**.

See Also
AES, TOS, window

wind_open—AES function (**libaes.a/wind_open**)
Open or reopen a window
#include <aesbind.h>
int wind_open(*handle, location***) int** *handle*; **Rect** *location*;

wind_open is an AES routine that opens or reopens a window. *handle* is the window's handle, as set by **wind_create**. *location* gives the dimensions of the window to be opened. It is declared to be of type **Rect**, which is defined in the header file **aesbind.h**; **Rect** consists of four elements, as follows:

x	X coordinate of rectangle
y	Y coordinate of rectangle
w	width of rectangle
h	height of rectangle

wind_open returns zero if an error occurred, and a number greater than zero if one did not.

Example
For examples of how to use this routine, see the entries **evnt_multi** and **window**.

See Also
AES, TOS, window

wind_set—AES function (**libaes.a/wind_set**)
Set specified fields within the window
#include <aesbind.h>
int wind_set(*handle, flag, input1, input2, input3, input4***)**
int *handle, flag, input1, input2, input3, input4*;

wind_set is an AES routine that sets specific portions of a window. *handle* is the handle of the window to be altered; the handle is set by **wind_create**. The arguments *input1* through *input4* contain information you wish to insert into the window's definition. Note that not all four of these arguments are used with every task; those that are not used should be set to zero. *flag* indicates what aspect of the window you want to change, as follows:

2	WF_NAME	Point to new name for window: *input1*=low word of address *input2*=high word of address
3	WF_INFO	Point to new information line for window: *input1*=low word of address

		input2=high word of address
5	WF_CURRXYWH	Window's total area: *input1*=X coordinate *input2*=Y coordinate *input3*=width *input4*=height
8	WF_HSLIDE	*input1* set to relative position of horizontal slider (1-1,000; 1=leftmost)
9	WF_VSLIDE	*input1* set to relative position of vertical slider (1-1,000; 1=top)
10	WF_TOP	*input1* set to handle of topmost window
14	WF_NEWDESK	Address of new default GEM desktop: *input1*=low word of address *input2*=high word of address *input3*=starting object in tree

wind_set returns zero if an error occurred, and a number greater than zero if one did not.

Example
For examples of how to use this routine, see the entries **evnt_multi** and **window**.

See Also
AES, TOS, window

wind_update—AES function (**libaes.a/wind_update**)
Lock or unlock a window
#include <aesbind.h>
int wind_update(*flag*) int *flag*;

wind_update is an AES routine that locks or unlocks a window. This mechanism is provided to prevent a window's information from being updated while the screen is being redrawn, to keep information from being "dropped on the floor" by GEM. *flag* indicates what you want done, as follows:

0	END_UPDATE	The update is finished: unlock the window
1	BEG_UPDATE	Beginning an update: lock the window
2	END_MCNTRL	End mouse control through user: lock window
3	BEG_MCNTRL	Begin mouse control through user: unlock

wind_update returns zero if an error occurred, and a number greater than zero if one did not.

Example
For an example of this routine, see the entry for **window**.

See Also
AES, TOS, window

window—Definition

A **window** is an AES entity that is used to display information. It consists of a number of elements, as follows:

0x001	NAME	Title bar (across top of window)
0x002	CLOSE	Close box (upper left corner)
0x004	FULL	Full box (upper right corner)
0x008	MOVE	Move bar (across top of window)
0x010	INFO	Information bar (just below move bar)
0x020	SIZE	Size box (lower right corner)
0x040	UPARROW	Up arrow
0x080	DNARROW	Down arrow
0x100	VSLIDE	Vertical slider (right side)
0x200	LFARROW	Left arrow
0x400	RTARROW	Right arrow
0x800	HSLIDE	Horizontal slider (bottom of window)

The mnemonics used above are defined in the header file **gemdefs.h**. A window can be built with all of these elements, none of them, or any combination of them.

To create a window, use the function **wind_create**. You must tell this function what kind of window is being created (i.e., which elements compose the window), and the maximum size that the window can assume. It returns a integer *handle* for the window, which can be used to identify it to all other functions. Note that the GEM desktop is always defined as window zero; the desktop is defined as being the entire screen minus the menu bar, and this definition is handy when you wish to expand a window to fill the entire screen.

Once a window is created, its attributes must set with the function **wind_set**. For example, if the window being created has a title bar, the text to be written there must be set before the window is displayed.

Once the attributes have been set, you can open the window with the function **wind_open**. You must pass it the handle of the window being created, and the dimensions to which you want it opened.

When a user clicks one of the elements of the window, such as the full box or the close box, the AES generates a *message* which can be picked up with the routines **evnt_mesag** or **evnt_multi**. Each message is eight **ints** (16 bytes) long. See the entry for **evnt_mesag** for a list of the messages.

When a message is received, the user is free either to react appropriately, or ignore the message. For example, if the message **WM_FULLED** is received, this indicates that the user has clicked the fulled box. The size of the box can then be changed with the **wind_set** routine, and another routine then invoked to redraw the screen and remove any debris left.

When you are done with a window, it should be closed with the **wind_close** function, and then removed with the function **wind_delete**. Note that the window should be closed before deletion; otherwise, you may not be able to erase the old, left-over window from the screen.

Redrawing a window
The interior of each window is broken by the AES into a set of non-overlapping rectangles, which it records in a list. If only one window appears on the screen, then its interior is described as one rectangle; if there are two windows on the screen, however, and one overlaps the other, then the interior of the lower window is described as a set of rectangles that outline the area being encroached. Therefore, redrawing the interior of a window requires that each rectangle be redraw in turn; cycling through the rectangles and redrawing them is called "walking the rectangles". The dimensions of the first rectangle in the list can be obtained with the following call:

```
wind_get(handle, WF_FIRSTXYWH, &box.x, &box.y, &box.w, &box.h);
```

and the dimensions of the next rectangle with this call:

```
wind_get(handle, WF_NEXTXYWH, &box.x, &box.y, &box.w, &box.h);
```

AES returns zero for the width and height when it has reached the end of the rectangle list.

Example
The following example demonstrates a number of window routines. It draws two windows, one on top of the other. Each has a title bar, a full box, an exit box, and boxes for moving and changing the size of the window. Clicking the exit box closes the window; when all the windows are closed, the program ends. A redrawing function is included; it "walks the rectangles" to fill the interior of each window with a gray mask. For more information on the object that it draws, see the entry on **object**. Note that the number of windows generated can be increased by redefining the manifest constant **LIMIT**.

```
#include <aesbind.h>                    /* Contains AES bindings */
#include <gemdefs.h>                    /* Contains manifest constants */
#include <obdefs.h>                     /* Contains object definitions */
#define YES 1
#define NO 0
#define DESKTOP 0
#define KIND (NAME | CLOSER | FULLER | INFO | SIZER | MOVER)
#define LIMIT 2                         /* Maximum number of windows */
```

```
#define SPEC 0x111C1L
/*
 *      I.e.:    (1 << 16) |                [Border 1 raster thick]
 *               (BLACK << 12) |            [Fill color; BLACK = 1]
 *               (BLACK << 8) |             [Text color]
 *              ((1 << 7) |                 [Turn on replace bit]
 *               (4 << 4) |                 [Fill pattern to gray]
 *                                          [Together make one nybble] )
 *                BLACK                     [Border color]
 */

/* Rectangles used throughout program */
Rect tempbox = { 250, 50, 150, 300 };      /* Box written by wind_get, etc. */
Prect tempptr = { &tempbox.x, &tempbox.y, &tempbox.w, &tempbox.h };
Rect fullbox = { 0, 0, 0, 0 };             /* To be set to desktop dimensions */

int nowhere = 0;                            /* Unused pointers point here */

/* Strings used in window; must be global or static, or could step on memory */
char *title = " TITLE ";                   /* Window title string */
char *info = "This is a window";           /* Window information string */

/* Object used to mask interiors of windows */
OBJECT mask[] = {
/*    next/head/tail/  type / flags / state /specification/ X/ Y/  W /  H */
      -1,  -1,  -1,  G_BOX, LASTOB, NORMAL,     SPEC,      1, 1, 639, 399
};

main() {
/* Declarations for windows */
      int handle[LIMIT];                    /* Window handle */
      int n = 0;                            /* Number of handle */

/* Declarations for evnt_mesag() */
      int buffer[8];                        /* Buffer for messages; each is 8 ints long */
      int fulled = NO;                      /* Flag: has window been "fulled"? */
      int window = 1;                       /* Flag: which window is being handled? */

/* Open application */
      appl_init();                          /* Open application */
      graf_mouse(ARROW, &nowhere);          /* Make ptr an arrow */
```

```
/* Get desktop dimensions; create and open windows */
	wind_get(DESKTOP, WF_FULLXYWH,
		&fullbox.x, &fullbox.y, &fullbox.w, &fullbox.h);
	objc_draw(mask, ROOT, 0, fullbox);		/* Coat screen with gray color */
	for (n = 0; n <LIMIT; n++) {
		graf_growbox(1, 1, 1, 1, tempbox);	/* "Star wars" */
		handle[n] = wind_create(KIND, fullbox); /* Get handle */
		wind_set(handle[n], 2, title, 0, 0);	/* Set window title */
		wind_set(handle[n], 3, info, 0, 0);	/* Set info line */
		wind_open(handle[n], tempbox);		/* Open window */
	}

/* Now, wait for something to happen */
	for(;;) {
		evnt_mesag(buffer);
		switch(buffer[0]) {
		case WM_REDRAW:
			redraw(buffer[3]);
			break;

		case WM_TOPPED:
			wind_set(buffer[3], WF_TOP, &nowhere,
				&nowhere, &nowhere, &nowhere);
			break;

		case WM_FULLED:
			if (fulled == YES) {
				wind_get(buffer[3], WF_PREVXYWH, tempptr);
				graf_shrinkbox(tempbox, fullbox);
				wind_set(buffer[3], WF_CURRXYWH, tempbox);
				fulled = NO;

			} else {
				wind_get(buffer[3], WF_CURRXYWH, tempptr);
				graf_growbox(tempbox, fullbox);
				wind_set(buffer[3], WF_CURRXYWH, fullbox);
				fulled = YES;
			}
			break;

		case WM_SIZED:
		case WM_MOVED:
			tempbox.x = buffer[4];
			tempbox.y = buffer[5];
			tempbox.w = buffer[6];
			tempbox.h = buffer[7];
			wind_set(buffer[3], WF_CURRXYWH, tempbox);
			fulled = NO;
			break;
```

```
		case WM_CLOSED:
			depart(buffer[3], window);
			window++;
			break;

		default:
			break;
		}
	}
}

/* Mask out interior of window */
redraw(windhandle)
int windhandle;
{
	graf_mouse(M_OFF, &nowhere);			/* Hide mouse */
	wind_update(BEG_UPDATE);			/* Lock window */

	wind_get(windhandle, WF_FIRSTXYWH, tempptr);	/* Get 1st rectangle */
	while(tempbox.w && tempbox.h) {
		objc_draw(mask, ROOT, 0, tempbox);	/* Draw rectangle */
		wind_get(windhandle, WF_NEXTXYWH, tempptr);
							/* Get next rectangle */
	}

	wind_update(END_UPDATE);			/* Unlock window */
	graf_mouse(M_ON, &nowhere);			/* Show mouse */
	return;
}

/* Exit from the program */
depart(windhandle, flag)
int windhandle, flag;
{
	wind_get(windhandle, WF_CURRXYWH, tempptr);
	wind_close(windhandle);
	wind_delete(windhandle);
	graf_shrinkbox(1, 1, 1, 1, tempbox);

	if (flag < LIMIT)
		return;
	else {
		appl_exit();
		exit(0);
	}
}
```

See Also

AES, gem, gemdefs.h, object, TOS

write—UNIX system call (**libc.a/write**)
Write to a file
write(*fp*, *buffer*, *n*)
int *fp*; char **buffer*; int *n*;

write writes *n* bytes of data, beginning at address *buffer*, into the file *fp*. Writing begins at the current write position, as set by the last call to either **write** or **lseek**. **write** advances the position of the file pointer by the number of characters written.

Example
For an example of how to use this function, see the entry for **open**.

See Also
STDIO, UNIX routines

Diagnostics
write returns a value of -1 if an error occurred before the **write** operation commenced, such as a bad file descriptor *fp* or invalid *buffer* pointer. Otherwise, it returns the number of bytes actually written. It should be considered an error if this number is not the same as *n*.

Notes
write is a low-level call that passes data directly to TOS. It should not be intermixed with high-level calls, such as **fread**, **fwrite**, or **fopen** without care.

xbios—TOS function (**libc.a/xbios**)

Call a routine from the extended TOS BIOS

#include <osbind.h>
extern long xbios(*n, f1, f2 ... fx***);**

xbios allows you to call a routine directly in the Atari extended ROM BIOS, by triggering trap 14. *n* is the number of the routine, and *f1* through *fx* are the parameter numbers to be used with **xbios**. In most circumstances, it is unnecessary to call **xbios**, for the header file **osbind.h** defines a number of functions that use it directly. The constants and structures used by these functions are contained in the header file **xbios.h**.

The following are the **xbios** functions:

Bioskeys	restore the default keyboard table
Cursconf	set the cursor's configuration
Dosound	pass data to the sound daemon
Flopfmt	format a floppy disk
Floprd	read a floppy disk
Flopver	verify a floppy disk
Flopwr	write to a floppy disk
Getrez	read the current screen resolution
Gettime	read the current system time
Giaccess	write to the GI sound chip registers
Ikbdws	send commands to the intelligent keyboard
Initmous	initialize the mouse
Iorec	get a pointer to the serial device input record
Jdisint	disable an interrupt
Jenabint	enable an interrupt
Keytbl	create a new keyboard table
Kbdvbase	get a pointer to a set of keyboard routines
Kbrate	set the keyboard's repeat rate
Logbase	get the screen's logical base
Mfpint	initialize interrupt routine in multi-function port
Midiws	send string to musical instrument digital interface
Offgibit	turn off a bit in the sound chip's A port
Ongibit	turn on a bit in the sound chip's A port
Physbase	get the physical base of the screen
Protobt	create a prototype boot routine
Prtblk	print a dump of the screen
Puntaes	make AES go away
Random	generate a pseudo-random number
Rsconf	configure the RS-232 (serial) port
Scrdmp	print a dump of the screen
Setcolor	set a color
Setscreen	set the screen parameters

Setpallete	set the color pallette
Setprt	configure the printer port
Settime	set the system time
Supexec	run a function under supervisor mode
Vsync	synchronize with the screen refresh
Xbtimer	initialize a timer on the multi-function port

See Also
osbind.h, TOS

Notes
Note that no **xbios** function checks for bogus device numbers. Passing a bogus device number to one will crash the system.

Note that all **xbios** I/O routines, including file I/O, are unbuffered. Combining them with buffered I/O routines, such as those in the STDIO library, will lead at best to unpredictable results.

xbios.h—Header file
#include <xbios.h>

xbios.h is a header file that includes all constants and structures used by the GEM-DOS **xbios** functions. For a list of these functions, see the entry for **xbios**.

See Also
bios.h, header file, TOS, xbios

Xbtimer—**xbios** function 31 (**osbind.h**)
Initialize the MFP timer
#include <osbind.h>
#include <xbios.h>
void Xbtimer(*timer, control, data, buffer***) int** *timer, control,data*; **char** **buffer*;

Xbtimer permits you to initialize one of the 68901 chip's timers. *timer* is a value from zero through three, which corresponds to timers A through D, respectively. Timer A handles user applications; timer B handles graphics; timer C is the system timer; and timer D sets the baud rate for the RS-232 port. *control* sets the timer's control register, and *data* is a byte of data to be written into the timer's data register. *buffer* points to an interrupt handler.

See Also
TOS, xbios

XOFF—Definition
XOFF is a flow-control signal used with asynchronous communications. Usu-

ally, it consists of a **<ctrl-S>** character, and is sent by the receiving device when its asynchronous buffer is nearly full, or has reached the "high-water mark". Note that when XOFF is used to help control data transmission, binary files cannot be transmitted.

See Also
XON

XON—Definition

XON is a flow-control signal used with asynchronous communications. Usually, it consists of a **<ctrl-Q>** character, and is sent by the receiving device when its asynchronous buffer is nearly empty, or has reached the "low-water mark". Note that when XON is used to help control data transmission, binary files cannot be transmitted.

See Also
XOFF

Index

Make

Program Building Discipline

Mark Williams Company

Table of Contents

1. Introduction

make is a utility that will help you construct complex C programs. It will relieve you of much of the drudgery needed to piece together a complex program; when properly used, **make** will save you a great deal of time.

If you are new to Mark Williams C, you should first examine the Mark Williams C compiler manual, which will give you the background information you need in order to read this tutorial with understanding.

What is make?

make is a *program-building discipline* that runs under **msh**. This means that **make** governs the piecing together of a complex C program.

Unlike programs in BASIC or Pascal, a C program consists of one or more *object* modules. Each object module, in turn, is produced by compiling or assembling one or more files *source module* of code written in C or assembly language.

A simple program can consist of only one source module. For example, the program **hello** is generated by compling only one source module, called **hello.c**; all that is needed to generate **hello** is the command:

```
cc hello.c
```

If you were to alter **hello.c**, regenerating **hello** would be easy.

In contrast, generating a complex program can be difficult. C source modules may refer to *header* files that may be changed independently of the source module itself. A compilation may require that certain macros be defined on the command line, and linking may use special libraries. For example, the source code for the screen editor MicroEMACS, which is included with the Mark Williams C distribution, consists of numerous source modules and one header file. Generating MicroEMACS requires either that more than one **cc** command be used, or that all files be placed into a special directory for compilation.

Naturally, a complex program is never compiled only once. A program may need to be recompiled many times, as it is tested, debugged, and improved. A programmer faces two problems in recompiling a complex program. First, a mistake in recompilation can greatly complicate the task of debugging the program: an incorrect compilation will introduce errors that may take a long time to isolate and correct. Second, recompiling all the source modules in a complex program can take a great deal of valuable time, which is largely wasted.

One strategy for coping with this situation is to write a *script* that contains all the commands needed to generate the program. This solves the problem of errors, by ensuring that all the file names are correct and all the commands are structured correctly; however, it does not solve the problem of wasted time because using a script demands that all source modules be recompiled every time the program is regenerated.

The other strategy is to rewrite the compilation commands by hand every time the program is regenerated. This solves the second problem, of wasted time, by allowing you to recompile just the source modules that you have altered; however, this method leaves your program vulnerable to errors, because entering complicated sequences of program generation commands is tedious.

make is a *program building discipline*. It simplifies the process of building a complex program by combining the best of the two strategies for generating programs. Errors are eliminated, because you place the names of all source modules and the text of all compilation commands into one *makefile*, which is reused every time the program is regenerated; and time is saved, because **make** checks all source modules, libraries, and header files, and recompiles only those that have changed since you last generated your program. As you can see, **make** simplifies and speeds the task of generating complex programs.

Try make

The following example will show that **make** is extremely easy to use. Before you begin, however, make sure that Mark Williams C is up and running. If this has not yet been done, consult chapter I of the Mark Williams C manual, which will tell you what you need to do.

Before you begin to work the example, enter the Mark Williams Company micro-shell **msh**. If you do not know how to use **msh**, see the section on using **msh** in chapter II of the Mark Williams C user's manual before you continue.

Also, note that **make** works by examining the times when your source files and object modules were created. If you do not reset the time on your Atari ST whenever you reboot, *every time*, **make** will not work correctly.

To test **make**, create a directory called **factor**; then move the following files into it:

```
atod.c
factor.c
makefile
```

These files are included with your release of Mark Williams C.

Now, type **make**. The following will appear on your screen:

```
cc -O -c factor.c
cc -O -c atod.c
cc -O -f -o factor.prg factor.o atod.o -lm
```

You may also see some warning and strict messages from the compiler as it processes each file.

When the **msh** prompt returns, type

```
factor
```

Then type a number. **factor** will calculate factorial numbers and return them to the screen. You can leave **factor** by typing **q**.

Now, type **make** again. In a moment, **make** will quietly exit, and you will see the msh prompt again. What happened was that **make** checked the dates and times of the *object modules* and of the *source modules*, saw that the object modules were younger than the source modules, and so compiled nothing.

Now, use the MicroEMACS screen editor to open the file **factor.c** for editing. Insert the following line into the comments at the top:

```
* This comment is for test purposes only.
```

Now exit. Type **make** once again. This time, you will see the following on your screen:

```
cc -O -c factor.c
cc -O -f -o factor.prg factor.o atod.o -lm
```

make recompiled **factor.c** because you altered it, and relinked the altered object module into the finished program. It did not touch **atod.c** because it had not been changed since the last time you compiled **factor**.

As you can see, **make** greatly simplifies the construction of a large, complex C program.

This tutorial will show you how to use **make**. Chapter 2 introduces you to the essentials of **make**; chapter 3 presents more detailed information for the sophisticated user. Chapter 4 summarizes **make** and its options, and gives a table of error messages with suggestions for how to deal with them.

2. Essential make

Although **make** is a powerful program and has many features, its basic features are easy to master. This chapter will show you how to construct elementary **make** programs.

The makefile

When you invoke **make**, it looks in your present directory for a file named **makefile**. **makefile** is a text file that you create; it holds all of the information that **make** needs to build your program. For this reason, the **makefile** is called a *program specification.*

A **makefile** has three basic elements.

First, the **makefile** begins with a *target* line that names the program you wish to generate, called the *target program.* The name of the target program is followed by a colon ':', and then by the names of files out of which the target file is to be generated. For example, if you wished to build the program **feud.prg** out of the files **hatfield.c** and **mccoy.c**, you would type:

```
feud.prg:  hatfield.o mccoy.o
```

Note that you must give your source modules the **.o** suffix, which refers to their compiled form as *object modules.* If the files **hatfield.o** and **mccoy.o** do not exist, **make** knows to create them from the source modules **hatfield.c** and **mccoy.c**.

The second element is one or more *command* lines. The command line gives the actual command for compiling the program in question. The only difference between a **makefile** command line and an ordinary **cc** command to the compiler is that the **makefile** command line must begin with a space or a tab character. For example, the **makefile** to generate the program **feud.prg** described above must contain the following command line:

```
cc -o feud.prg hatfield.o mccoy.o
```

For a detailed description of the **cc** command line and its options, refer to the entry for **cc** in the Lexicon.

The **makefile** may also contain additional target and command lines that describe how to build other components of the program. These are described in chapter 3 of this tutorial.

The third element is a list of header files that your program uses. These are given so that **make** can check to see if they have been modified since your program was last generated. For example, if the program **hatfield.c** used the header file **shotgun.h** and **mccoy.c** used the header files **rifle.h** and **pistol.h**, the **makefile** to generate the program **feud** would include the following lines:

```
hatfield.o: shotgun.h
mccoy.o: rifle.h pistol.h
```

Thus, the entire **makefile** to generate the program **feud.prg** appears as follows:

```
feud.prg: hatfield.o mccoy.o
        cc -o feud.prg hatfield.o mccoy.o

hatfield.o: shotgun.h
mccoy.o: rifle.h pistol.h
```

Note, too, that a **makefile** may also contain *macro definitions* and *comments*. These will be described below.

Building a simple makefile

The program **factor.prg** is generated out of two source modules, called **factor.c** and **atod.c**. The mathematics library **libm.a** is required, but no unique header files are used. As an exercise, write the **makefile** needed to generate **factor.prg**.

One solution is the following:

```
factor.prg: factor.o atod.o
        cc -o factor.prg factor.o atod.o -lm
```

As you can see, it is not necessary to specify header files if none are used.

Comments and macros

If you wish, you can embed comments within a **makefile**. Any line that begins with a pound sign '#' is ignored by **make**, and so can hold information that describes the **makefile** to other users. For example, you may wish to include the following information in your **makefile** for **factor**:

```
# This makefile generates the program "factor".
# "factor" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# "libm", but it requires no special header files.

factor.prg: factor.o atod.o
        cc -o factor.prg factor.o atod.o -lm
```

You can also define macros within your **makefile**. A *macro* is a symbol that stands for a string of text. Usually, a macro is defined at the beginning of the makefile, using a *macro definition statement*. This statement uses the following syntax:

```
SYMBOL = string of text
```

Whenever the symbol is used in your **makefile** thereafter, it must begin with a dollar sign '$' and be enclosed within parentheses.

Macros are usually used to eliminate the chore of retyping long strings of file names. For example, with the **makefile** for the program **factor**, you may wish to use a macro to substitute for the names of the object modules out of which it is built. This is done as follows:

```
# This makefile generates the program "factor".
# "factor" consists of the source modules "factor.c" and
# "atod.c". It uses the standard mathematics library
# "libm", but it requires no special header files.

OBJ = factor.o atod.o
factor.prg: $(OBJ)
        cc -o factor.prg $(OBJ) -lm
```

Note how the macro **OBJ** is used in this **Makefile**. If a macro is used that has not been defined, **make** will substitute a NUL character for it. The use of a macro makes sense when generating large files out of a dozen or more source modules. You avoid tedious retyping of the source module names, and potential errors are avoided.

Setting the time

As noted above, **make** checks to see which source modules have been modified before it regenerates your C program. This is done to avoid the time-consuming and useless task of recompiling source modules that have not been updated.

make determines that a source module has been altered by comparing its *date* against that of the target program. For example, if the program **factor.prg** was generated on March 16, 1985, at 10:52:47 A.M., and the source module **atod.c** was then modified on March 20, 1985, at 11:19:06 A.M., **make** will know that **atod.c** needs to be recompiled because it is *younger* than **factor.prg**.

For this reason, if you wish to use **make**, you *must* reset the date and time every time you reboot your system. Some users do not do this routinely; however, unless the time is reset *every* time, **make** will be useless.

Use the command **date** to reset the date. **date** is described in the Lexicon, chapter IV of the Mark Williams C manual.

make uses two routines to handle time: **ftime**, which reads the time from TOS, and **ctime**, which translates what **ftime** has read into a format useful to **make**. For details on how these routines work, see the entries for **ftime** and **ctime** in the Lexicon.

Building a large program

As shown earlier, **make** can ease the task of generating a large program. The following is the **makefile** used to generate the screen editor MicroEMACS:

```
#
# Makefile for MicroEMACS on the Atari ST
#
CFLAGS = -O
LFLAGS = lib\libterm.a
OBJ=ansi.o basic.o buffer.o display.o file.o \
fileio.o line.o main.o random.o region.o search.o \
spawn.o tcap.o termio.o vt52.o window.o word.o
me.ttp: $(OBJ)
        cc -o me.ttp $(OBJ)
$(OBJ): ed.h
```

The first line is commentary that describes the file.

The next five lines define macros that are used on the target and command line. The first macros will be discussed in the following chapter, on "Advanced **make**". The second macro substitutes for the name of a special library that is needed to create this program. The third macro, which is three lines long, is defined as standing for the names of the source modules that produce MicroEMACS. Note that a backslash '\' must be used to tell **make** that the definition is carried over onto the next line.

The next line names the target file (**me**) and the files used to construct it, here represented by the macro **OBJ**.

Next comes the command line, which dictates the compilation to be performed. Note that the macro **LFLAG** must *follow* the the names of the files to be compiled. Note that this line *must* be preceded by a space or a tab.

The last line lists the header file **ed.h**, which is required by all of the files used to generate MicroEMACS.

Command line options

Although what **make** does is controlled largely by your **makefile**, you can also affect what **make** does through the use of command line options. These options allow you to alter **make**'s activity without having to edit your **makefile**.

Options must follow the command name on the command line and start with a hyphen '-', using the following format (note that the square brackets merely indicate that you can select any of these options, and are *not* used on the **make** command line):

```
make [ -dinpqrst ] [ -f filename ]
```

These options are described below.

-d (debug) **make** describes all of its decisions. You can use this to debug your **makefile**.

-f *filename*
(file) option tells **make** that its commands are in a file other than **makefile**. For example, the command

```
make -f smith
```

tells **make** to execute the file **smith** rather than **makefile**. If you do not use this option, **make** will search the directories named in the PATH environmental variable for a file entitled **makefile** to execute. The command line can hold more than one **-f** option.

-i (ignore errors) **make** ignores error returns from commands and continue processing. Normally, **make** exits if a command returns an error status.

-n (no execution) **make** tests dependencies and modification times but does not execute commands. This option is especially helpful when constructing or debugging a **makefile**.

-p (print) **make** prints all macro definitions and target descriptions.

-q (question) **make** checks if the target is up to date and returns an appropriate status without executing any commands. "Up to date" means that the target program is younger than all of its source modules. **make** returns 0 if the target is current, 1 if an error occurs, and 2 if the target is outdated.

-r (rules) **make** does not use the default macros and commands from **LIBPATH\mmacros** and **LIBPATH\mactions**. These files will be described in the following chapter.

-s (silent) **make** does not print each command line as it is executed.

-t (touch) **make** changes the modification time of each target to the current time. This suppresses actual regeneration. Although this option is used typically after a purely cosmetic change to a source file or after adding a definition to a header file, it must be used with great caution.

Other command line features

In addition to the options listed above, you may include other information on your command line.

First, you can define macros on the command line. A macro definition must *follow* any command line options. Arguments including spaces must be surrounded by quotation marks, as spaces are significant to msh. For example, the command line

```
make -n -f smith "OBJ=a.o b.o"
```

tells **make** to run in the *no execution* mode, reading the file **smith** instead of **makefile**, and defining the macro **OBJ** to mean **a.o b.o**.

The ability to define macros on the command line means that you can create a **makefile** using macros that are not yet defined; this greatly increases **make**'s flexibility and makes it even more helpful in creating and debugging large programs.

Another feature is the ability to change the name of the target file on the command line. As will be discussed in full in the next chapter, a **makefile** can name more than one target file. **make** normally assumes that the target is the first target file named in **makefile**. However, the command line may name one or more target files at the end of the line, after any options and any macro definitions.

To see how this works, recall the program **factor** described above. **factor** is generated out of the source modules **factor.c** and **atod.c**. With this program, the command

```
make atod.o
```

with the **makefile** outlined above would result in the following **cc** command line being produced:

```
cc -c atod.c
```

if the object module **atod.o** does not exist or is outdated. Here, **make** compiles **atod.c** to create the target specified in the **make** command line, that is, **atod.o**, but it does not create **factor**. This feature allows you to apply your **makefile** to only a portion of your program.

The use of special, or *alternative*, target files will be discussed in full in the next chapter.

3. Advanced make

This chapter describes some of the advanced features of **make**. Most programs do not require these features; however, users who create extremely complex programs will find these features to be most helpful.

Default rules

The operation of **make** is governed by a number of *default rules*. These rules were designed to ease the compilation of a typical program; however, unusual tasks may be made easier by bypassing altering the default rules.

To begin, **make** uses information from the files **LIBPATH\mmacros** and **LIBPATH\makeactions** to define default macros and regeneration commands. **make** looks for these files in the directories named in the LIBPATH environmental variable. **make** uses the commands in **mmacros** and **mactions** whenever the **makefile** specifies no explicit regeneration commands. The command line option **-r** tells **make** not to use the macros and actions defined in **mmacros** and **mactions**.

As shown in earlier examples, **make** knows by default to generate the target **atod.o** from the C source **atod.c** with the command

```
cc -O -c atod.c
```

The macro **.SUFFIXES** defines the suffixes **make** knows about by default. Its definition in **mmacros** includes both the **.o** and **.c** suffixes.

mmacros uses targets such as **.c.o** to specify the commands for recreating a **.o** object file from a **.c** source file. The entry for this target is as follows:

```
.c.o:
        $(CC) -c $<
```

Macros CC and **CFLAGS** are defined in **mmacros** as:

```
CC = cc
CFLAGS = -O
```

You can change the name of the default C compiler that **make** uses by supplying a new definition of macro **CC** or can change the default compilation flags by redefining CFLAGS.

make's files **mmacros** and **mactions** use several other pre-defined macros to increase their scope and flexibility. These are as follows:

* "$<", which is used in the above example, stands for the name of the file or files causing the action of a default rule. In the above example, **$<** stands for the file name **atod.c**.

* "$*" stands for the name of the target of a default rule with its suffix removed. If it had been used in the above example, $* would have stood for **atod**.

  "$<" and "$*" may be used *only* with default rules; these macros will not work in a **makefile.**

* "$?" stands for the names of the files that cause the action and that are younger than the target file.

* The macro "$@" stands for the target name.

  Macros "$?" and "$@" may be used in a **makefile**. For example, the following rule updates the archive **libx.a** with the objects defined by macro **$(OBJ)** that are out of date:

```
libx.a: $(OBJ)
        ar rv libx.a $?
```

mmacros also contains default commands that describe how to build additional kinds of files:

* **AS** and **ASFLAGS** call the *assembler* to assemble **.o** files out of **.s** files.

You can change the default rules of **make** by changing them in **mactions** and changing the definition of any of the macros as given in **mmacros**.

Double colon target lines

An alternative form of target line simplifies the task of maintaining archives. This form uses the double colon "::" instead of a single colon ':' to separate the name of the target from those of the files on which it depends.

A target name can appear on only one single-colon target line, whereas it can appear on several double-colon target lines. The advantage of using the double-colon target lines is that **make** will remake the target by executing the commands (or its default commands) for the *first* such target line for which the target is older than a file on which it depends. For example, for the program **factor.prg** described earlier, assume that two versions of the source files **factor.c** and **atod.c** exist: **factora.c** plus **atoda.c**, and **factorb.c** plus **atodb.c** The **makefile** would appear as follows:

```
OBJ1 = factora.o atoda.o
OBJ2 = factorb.o atodb.o
factor.prg :: $(OBJ1)
        cc -c $(OBJ1) -lm
factor.prg :: $(OBJ2)
        cc -c $(OBJ2) -lm
```

This **makefile** tells **make** to do the following: (1) check if either **factora.o** or **atoda.o** are younger than **factor.prg**; (2) if either one is, regenerate **factor.prg** using this version of these files; (3) if neither **factora.o** nor **atoda.o** are younger than **factor.prg**, then check to see if either **factorb.o** or **atodb.o** are younger than **factor**; and (4) if either of them are, then regenerate **factor.prg** using the youngest version of these files.

This technique allows you to maintain multiple versions of source files in the same directory and selectively recompile the most recently updated version without having to edit your **makefile** or otherwise trick the system.

Note that a file may not be targeted by both single-colon and double-colon target lines.

Alternative uses

make is almost always used to control the generation of complex C programs. However, **make** is also a powerful general-purpose tool, and is easily adapted to many other uses.

The targets in **makefile** may include special targets, which trigger program maintenance commands. For example, the target name **backup** might define commands to copy sources to a backup directory; typing **make backup** backs up the sources. Similar uses include removing temporary files, building archives, executing test suites, and printing hard copies. A **makefile** is a convenient place to keep all the commands used to maintain a program.

The following example shows a **makefile** that defines two special target files, **printall** and **printnew**, to be used with the source files for the program **factor.prg**.

```
# This makefile generates the program "factor".
# "factor" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# "libm.olb", but it requires no special header files.
OBJ = factor.o atod.o
SRC = factor.c atod.c
factor.prg : $(OBJ)
	cc -o factor.prg $(OBJ) -lm
# program to print all the updated source modules
# used to generate the program "factor"
printall:
	pr $(SRC) | prn:
	echo junk > prnew
printnew: $(OBJ)
	pr $? | prn:
	echo junk > printnew
```

In this instance, typing the command

```
make printall
```

forces **make** to generate the target **printall** rather than the target **factor**, which is the default as it appears first in the **makefile**. The **pr** command, with the output piped to the parallel port **prn:**, is then used to print a listing of all files defined by **SRC**. The macro **OBJ** cannot be used with these commands, because it would trigger the printing of the object files, which would not be of much use. The word **junk** is echoed into an empty file, **prnew**. This new file serves only to record the time the listing is printed. This tactic is performed in order to record the time that the listing was last generated, so that **make** will know what files have been updated when you next use **printnew**.

Typing the command

```
make printnew
```

forces **make** to generate the target **printnew** rather than the default target **factor**. **printnew** prints only the files named in the macro SRC that have changed since any files were last printed.

Special targets

A few target names have special meanings to **make**. The name of each special target begins with '.' and contains upper-case letters.

Target **.DEFAULT** defines the default commands **make** uses if it cannot find any other way to build a target. The special target **.IGNORE** in a **makefile** has the same effect as the **-i** command line option. Similarly, **.SILENT** has the same effect as the **-s** command line option.

Errors

make prints "*command* exited with status *n*" and exits if an executed *command* returns an error status. However, it ignores the error status and continues processing if the **makefile** command line begins with a hyphen '-' or if the **make** command line specifies the **-i** ignore errors option.

make reports an error status and exits if the user interrupts it. It prints "**can't open** *file*" if it cannot find the specification *file*. It prints "**Target** *file* **is not defined**" or "**Don't know how to make** *target*" if it cannot find an appropriate *file* or commands to generate *target*. Other possible errors include syntax errors in the specification file, macro definition errors, and running out of space. The error messages **make** prints are generally self-explanatory; however, a table of error messages and brief descriptions of them are given in chapter 4 of this tutorial.

Exit status

make returns a status of 0 if it succeeds and 1 if an error occurs. For the **-q** option, it returns 0 if the target is up to date and 2 if it is not.

4. Summary of make

The following sections summarize the use of **make**, and present a table of the error message that can be produced by **make**.

Usage

The usage of **make** is as follows:

```
make [-dinpqrst] [-f filename ] [f1"MACRO=definition" ] [ target ]
```

The argument *filename* refers to the file name that accompanies the **-f**, described below. The option *"MACRO=definition"* refers to the fact that macros can be defined on the command line; note that such definitions *must* be enclosed in quotation marks. The option *target* means that the name of the *target* file within the **makefile** can be changed from the default, which is the file defined in the *first* target line in the **makefile**.

The **makefile** may include (1) *target* line, which names the target file, and (2) a *command* line, which describes the command to be executed to help form a target file. A **makefile** may include more than one target line and more than one command lines for each; **make**'s default is to accept and generate the *first* target in a **makefile**, but this default may be overridden by naming another target file in the **make** command line. A **makefile** may also name *header files* to be examined for updating, and may include macro definitions and comments. See chapter 2 of this tutorial for details and examples.

Options

The options to **make** are as follows:

-d Debug option. Give verbose printout of all decisions and of the information that went into the decisions. This is useful for debugging **makefiles**.

-f *filename*
File option. This option tells **make** that *filename*, rather than **makefile**, contains the **make** specifications. If this option does not appear, **make** uses the file **makefile** in the current directory.

-i Ignore option. This tells make to ignore error returns from a command and continue processing. Normally **make** exits if a command returns error status.

-n Test option. This options instructs **make** only to test a **makefile**; it suppresses actual execution of commands.

-p Print option. Print all macro definitions and target descriptions.

-q Return a zero exit status if the target files are younger than the source modules. No commands are executed. This is the default condition.

-r Rules option. Do not use built-in rules that describe dependencies.

-s Silent option. Do not print command lines when executing them. Commands preceded by '@' are not printed, except under the **-n** option.

-t Touch option. Force the dates of targets to be the current time, to bypass actual regeneration.

Error messages

The following is a table of error messages generated by make, and a brief description of each.

; after target or macroname
  A semicolon appeared after a target name or a macro name. This is illegal.

= in or after dependency
  An equal sign '=' appeared within or followed the definition of a macro name or target file, for example "**OBJ=atod.o=factor.o**". This is illegal.

'::' not allowed for *name* not allowed for *name*'u'
  A double-colon target line was used illegally, for example, after single-colon target line.

::: or : in or after dependency list
  A triple colon is meaningless to **make**, and therefore illegal wherever it appears. A single colon may be used only in a target line (which is also called the *dependency list*), and nowhere else.

= without macro name or in token list
  An equal sign '=' can be used only to define a macro, using the following syntax: "MACRO=*definition*". An incomplete macro definition, or the appearance of an equal sign outside the context of a macro definition, will trigger this error message.

: without preceding target
  A colon appeared without a target file name, e.g., ":*filename*". This is illegal.

Bad macro name
  A bad macro name was used.

Incomplete line at end of file
  An incomplete line appeared at the end of the **makefile**. This is illegal.

Macro definition too long
  Macro definitions are limited to 200 characters.

Multiple actions for *name*
A target is defined with more than one single-colon target line. This is illegal.

Multiple detailed actions for *name*
A target is defined with more than one single-colon target line. This is illegal.

Must use '::' for *name* for *name*'u'
A double-colon target line was followed by a single-colon target line. This is illegal.

Newline after target or macroname
A newline character appears after a target name or a macro name. This is illegal.

Out of core (adddep)
System problem. Try reducing the size of your **makefile**.

Out of space
System problem. Try reducing the size of your **makefile**.

Out of space (lookup)
System problem. Try reducing the size of your **makefile**.

Syntax error
The syntax of a line is faulty.

Too many macro definitions
The number of macros you have created exceeds the capacity of your computer to process them.

Index

MicroEMACS

Interactive Screen Editor

Mark Williams Company

Table of Contents

1. Introduction

This is a tutorial for the interactive screen editor MicroEMACS.

This tutorial is written for two types of reader: the one who has never used a screen editor and needs a full introduction to the subject, and the one who has used a screen editor before but wishes to review specific topics. This tutorial is part of the documentation for MicroEMACS. MicroEMACS is also summarized in the **Lexicon**, chapter IV of the MicroEMACS manual.

What is MicroEMACS?

MicroEMACS is an interactive screen editor. An editor allows you to type text into your computer, name it, store it, and recall it later for editing. *Interactive* means that MicroEMACS will accept your editing command, execute it, display the results for you immediately, and then wait for your next command. *Screen* means that you can use nearly the entire screen of your terminal as a writing surface: you can move your cursor up, down, and around your screen to create or change text, much as you move your pen up, down, and around a piece of paper.

These features, plus the others that will be described in the course of this tutorial, make MicroEMACS a tool that is powerful, yet easy to use. You can use MicroEMACS to create or change computer programs, essays or letters, or any other type of text file.

The present version of MicroEMACS was adapted by Mark Williams Company from a public-domain program written by David G. Conroy. This tutorial is based on the descriptions in his essay *MicroEMACS: Reasonable Display Editing in Little Computers.* MicroEMACS is derived from the mainframe display editor EMACS, which was created at the Massachusetts Institute of Technology by Richard Stallman. EMACS is popular among persons who work with computers for a living; it is the parent or grandparent of a number of well-known word processors.

Structure of the tutorial

This tutorial has two parts. Part 1, which includes chapters 2 through 9, covers basic editing with MicroEMACS. Most users will find that these chapters contain all the information they need to use MicroEMACS productively. Part 2, which includes chapters 3 through 16, describes advanced editing — editing techniques that exploit the full power of MicroEMACS. These advanced techniques include the use of arguments with MicroEMACS's commands, multiple windows and buffers, and keyboard macros. The **Lexicon** contains a summary of all of MicroEMACS's commands.

Do the exercises

The following chapters include exercises that illustrate each topic being discussed. These exercises will help you understand exactly how each feature works. We recommend that you type each exercise as you come to it in the text. Even if you understand the concepts being discussed, working the exercises will reinforce the lesson and will help you grow comfortable in using MicroEMACS.

2. Basic Editing

MicroEMACS is an interactive screen editor.

Interactive means MicroEMACS accepts a command from you, executes it, displays the result on your terminal immediately, and then waits for your next command.

Screen means MicroEMACS allows you to use nearly the entire screen of your terminal as a writing surface. You can move your cursor up, down, and around the screen to enter text or make changes, much as you move your pen up, down, and around a sheet of paper.

The first half of this tutorial, chapters 2 through 9, describes basic editing with MicroEMACS. Mastering the commands described in the next few subsections will allow you to create a document, store it, and edit it thoroughly. Advanced techniques, such as assembling text from several buffers, using windows, and using arguments, will be covered in the second half.

Keystrokes—<esc>, <ctrl>

The MicroEMACS commands use **control** characters and **meta** characters.

Control characters use the *control* key, which is marked **Control** on your keyboard; meta characters use the *escape* key, which is marked **Esc**.

On your keyboard, the escape key is located in the upper left-hand corner of the keyboard. The control key is also located at the left end of your keyboard, just to the left of the 'A' character key.

To see how the *control* and *escape* keys actually work, consult any reference manual that describes the **ascii** table. To use them with MicroEMACS, however, only requires that you key them correctly.

Control works like the *shift* key: you hold it down *while* you strike the other key. Here, this will be represented with a hyphen; for example, **control X** will be shown as follows:

```
<ctrl-X>
```

The **esc** key, on the other hand, works like an ordinary character. You should strike it first, *then* strike the letter character you want. *Escape* character codes will not be represented with a hyphen; for example, **escape X** will be represented as:

```
<esc>X
```

Becoming acquainted with MicroEMACS

Now you are ready for a few simple exercises that will help you get a feel for how MicroEMACS works.

To begin, use the mouse to invoke the Mark Williams micro-shell **msh**. If you do not yet know how to use **msh**, see the section on **msh** in chapter II of the MicroEMACS manual. As soon as the prompt for **msh** has appeared in the upper left-hand corner of your screen, type

```
me sample
```

Within a few seconds, your screen will have been cleared of writing, the cursor will be positioned in the upper left-hand corner of the screen, and a command line will appear at the bottom of your screen.

Now type the following text. If you make a mistake while typing, just backspace over it and retype the text. Press the carriage return key, which is labelled **Return**, after each line:

```
There is nothing which has yet been contrived by
man, by which so much happiness is produced as by
a good tavern or inn.
```

Notice how the text appeared on the screen character by character as you typed it, much as it would appear on a piece of paper if you were using a typewriter.

Now, type **<ctrl-X><ctrl-S>**; that is, type **<ctrl-X>**, and then type **<ctrl-S>**. It does not matter whether you type capital or lower-case letters. Notice that this message has appeared at the bottom of your screen:

```
[Wrote 3 lines]
```

This command has permanently stored, or *saved*, what you typed. The text will now be preserved until you use the **rm** command to delete it.

Type the next few commands, which demonstrate some of the tasks that MicroEMACS can perform for you. These commands will be explained in full in the sections that follow; for now, it is enough for you to get a feel for how MicroEMACS works.

Type **<esc><**. Be sure that you type a less-than symbol '<', instead of a comma. Notice that the cursor has returned to the upper left-hand corner of the screen. Type **<esc>F**. The cursor has jumped forward by one word, and is now between the words **There** and **is**. Type **<ctrl-N>**. Notice that the cursor has jumped to the next line, and is now under the letter **b** of the word **by**. Type **<ctrl-A>**. The cursor has jumped to the *beginning* of the second line of your text.

Now, type **<ctrl-K>**. The second line of text has disappeared, leaving an empty space. Type **<ctrl-K>** again. The empty space where the second line of text had been has now disappeared.

Type **<esc>>**. Be sure to type a greater-than symbol '>', not a period. The cursor has jumped to the space just below the last line of text. Now type **<ctrl-Y>**. The text that you erased a moment ago has now been restored.

By now, you should be feeling more at ease with typing MicroEMACS's *control* and *escape* codes. The following sections will explain what these commands mean. For now, exit from MicroEMACS by typing **<ctrl-X><ctrl-C>**, and when the message

```
Quit [y/n]?
```

appears, type y. This will return you to **msh**.

Before you begin

There are one or two potential sources of difficulty that you should watch for as you begin to work with MicroEMACS.

Note that MicroEMACS may be overwhelmed if you attempt to edit an extremely large file. If your file proves to be too large, either when it is loaded or while you are working on it, you will see the following message:

```
File too large for available memory!
```

When this happens, you *must* exit from MicroEMACS to **msh** by typing **<ctrl-X><ctrl-C>**; if you use any other command in an attempt to save the changes you made to your text, you will corrupt your original file. Exiting to **msh** will be covered in the next few pages.

Three sample texts have been included with MicroEMACS. They are called **text1.m**, **text2.m**, and **text3.m**. Before you begin, be sure to make working copies of these texts, so that if an accident were to occur while you were working on this tutorial the master copies of the sample texts will still be preserved.

You should use the **cp** command to copy the files to the following file names: **text1.m** to **text1**, **text2.m** to **text2**, and **text3.m** to **text3**. If you do not know how to use the **cp** command, see the entry for it in the **Lexicon**.

If you are going to edit a large text, MicroEMACS may take a few seconds to load it into memory.

You will know MicroEMACS set up and ready to go when the following message appears at the bottom of your screen:

```
[Read XX lines]
```

where *XX* stands for the number of lines in your text file. If you are creating a new text file, MicroEMACS will send you this message:

```
[New file]
```

Beginning a document

You are now ready to invoke MicroEMACS and create a text file. Type the following command line, which tells MicroEMACS that you wish to edit the text called **text1**:

```
me text1
```

This text has been included with your MicroEMACS compiler; there is no need to retype it.

The computer will take a moment to set up the MicroEMACS program. As soon as it does so, the following text will appear on your screen:

```
From "Life on the Mississippi":
I know how a prize watermelon looks when it is sunning
its fat rotundity among the pumpkin vines; I know how to tell
when it is ripe without "plugging" it; I know how inviting
it looks when it is cooling itself in a tub of water under
the bed, waiting; I know how it looks when it lies on the
table in the sheltered great floor space between house and
kitchen, and the children gathered for the sacrifice and
their mouths watering; I know the crackling sound it makes
when the carving knife enters its end, and I can see the
split fly along in front of the blade as the knife cleaves
its way to the other end; I can see its halves fall apart
and display the rich red meat and the black seeds, and the
heart standing up, a luxury fit the elect; I know how a
boy looks behind a yard-long slice of that melon, and I
know how he feels; for I have been there.
```

When you type the MicroEMACS command and a file name, MicroEMACS *copies* that text file into a special area in your computer to make it available for editing. If you were creating a new text, as you did earlier with the text called **sample**, the screen would have appeared blank.

In addition to this text appearing on your screen, your cursor moved to the upper left-hand corner of the screen, and the status line appeared near the bottom of your screen as follows:

```
-- ST MicroEMACS V1.2 -- text1 -- File: text1 ------------------
```

The word to the left, MicroEMACS, is the name of the program. The next word, shown here as **V1.2**, is the version number. It may be different with your copy of Mark Williams C. The word in the center, **text1**, is the name of the *buffer* that you are using. (What a buffer is and how it is used will be covered later.) The name to the right is the name of the text file that you will be editing.

3. Moving the Cursor

Now that you have created a text file, you will want to edit it. The first step is to learn to move the cursor. Try these commands for yourself as they are described in the following pages. That way, you will quickly acquire a feel for handling MicroEMACS's commands. You can use your *arrow keys* with MicroEMACS. The arrow keys are found on the pad to the right of the alphabetic keyboard. The arrow keys move the cursor in the direction indicated (left, right, up, or down); this tutorial, however, will refer primarily to the basic cursor movement commands displayed below:

<ctrl-B>	Move back 1 space
<esc>B	Move back 1 word
<ctrl-E>	Move to end of line
<ctrl-F>	Move forward 1 space
<esc>F	Move forward 1 word
<ctrl-A>	Move to beginning of line
<ctrl-P>	Move to previous line
<ctrl-N>	Move to next line
<ctrl-V>	Move forward 1 screen
<esc>V	Move back 1 screen
<esc><	Move to beginning of text
<esc>>	Move to end of text

Moving the cursor backwards

The first set of commands move the cursor backwards. First, type the *end of text* command **<esc>>** to move the cursor to the bottom of the text. Be sure to type a greater-than symbol '>', not a period.

Type the *backspace* command **<ctrl-B>**. This is equivalent to pressing the *left arrow key*. As before, it does not matter whether the letter '**B**" is upper case or lower case. Note that the cursor is now located just to the right of the period in your last line of text. Type **<ctrl-B>** again. The cursor has moved one space to the left, and now is directly over the period.

Type **<esc>B**. The cursor has moved one *word* to the left, and is now over the letter **t** of the word **there**. Type the *beginning of line* command **<ctrl-A>**. The cursor has jumped to the beginning of the line, and is now over the letter **k** of the word **know**.

Moving the cursor forwards

Now practice moving the cursor forwards. Type the *forward* command **<ctrl-F>**. This is equivalent to pressing the *right arrow key*. Note that the cursor has moved one space to the right, and now is over the letter **n** of the word **know**. Type **<esc>F**. The cursor has moved one *word* to the right, and now is over the space between the words **know** and **how**.

Type the *end of line* command **<ctrl-E>**. The cursor has jumped to the end of the line, and once again is resting to the right of the period.

From line to line

The next two commands move the cursor up and down the screen. Type the *previous line* command **<ctrl-P>**. Note that the cursor has jumped from its position to the right of the period on the last line of your text, to being over the second letter **t** of the word **that** in the previous line.

Continue to type **<ctrl-P>** until the cursor reaches the top of the screen. This has the same effect as if you typed the *up arrow key*. Note that as you reached the first line in your text, the cursor jumped from under the letter **i** of the word **it** on the second line, to being just right of the colon on the first line of text. When you move your cursor up or down the screen, MicroEMACS will try to keep it at the same position within each line. If the line to which you are moving the cursor is not long enough to have a character at that position, MicroEMACS will move the cursor to the end of the line.

Now, practice moving the cursor back down the screen. Type the *next line* command **<ctrl-N>**. This has the same effect as pressing the *down arrow key*. Note that when the cursor jumped to the next line, it returned to under the letter **i** of the word **it**. MicroEMACS remembered the cursor's position on the line, and returned the cursor there when it jumped to a line long enough to have a character in that position.

Continue pressing **<ctrl-N>**. The cursor will move down the screen, until it reaches the bottom of your text.

Moving up and down by a screenful of text

The next two cursor movement commands allow you to roll forward or backwards by one screenful of text.

If you are editing a file with MicroEMACS that is too big to be displayed on your screen all at once, MicroEMACS will display the file in screen-sized portions (22 lines at a time). The *view* commands **<ctrl-V>** and **<esc>V** allow you to roll up or down by one screenful of text at a time.

Type **<ctrl-V>**. Note that your screen becomes empty. This is because you have rolled forward by the equivalent of one screenful of text, or 22 lines; however, because the text in this example is only 16 lines long, rolling forward 22 lines just empties the screen.

Now, type **<esc>**V. Notice that your text rolls back onto the screen, and your cursor is positioned in the upper left-hand corner of the screen, over the letter **F** of the word **From**.

Moving to beginning or end of text

The last two cursor movement commands allow you to jump immediately to the beginning or end of your text.

The *end of text* command **<esc>>** moves the cursor to the end of your text. Type **<esc>>**. Be sure to type a greater-than symbol '>'.

The *beginning of text* command **<esc><** will move the cursor back to the beginning of your text. Type **<esc><**. Be sure to type a less-than symbol '<'. Note that the cursor has jumped back to the upper left-hand corner of your screen.

These commands will move you immediately to the beginning or the end of your text, regardless of whether the text is one page long or 20.

Cursor movement strategy

When you edit a large text, you must move the cursor often. This can be very time-consuming. Three rules will help you save time by moving the cursor efficiently.

1. Plan your cursor movements so that you reach your target with a few keystrokes, if possible.

2. If you are a good typist, avoid using the **<esc>** key if possible, because using the **<esc>** key usually forces you to lift your left hand from the home position.

3. Try not to use the arrow keys if possible. Using the arrow keys moves your hands out of the home position, which slows your typing and increases the chance that you will replace them on the keyboard in the wrong position.

Try the following exercises to sharpen your command of cursor movement. Each exercise is followed by its solution. Do not look at the solution until you have at least attempted to solve the problem. The exercises should be done in order, because each one builds on the ones that came before.

1. Your cursor should be in the upper left-hand corner of the screen. If it is not, type **<esc><**. Now, move the cursor to the space just before the word **children** in line 8—you should be able to do it with ten commands.

Solution: Type **<ctrl-N>** seven times, then type **<esc>F** three times.

2. Move the cursor to the letter **n** of the word **knife** in line 10—you should be able to do it with four commands.

Solution: Type **<ctrl-N>** twice, then **<ctrl-F>** twice.

3. Move the cursor to the right of the period on line 16—you should be able to do it with two commands.

Solution: Type **<esc>>**, then type **<ctrl-B>**.

4. Move the cursor to the space after the word **fall** on line 12—you should be able to do it with six commands.

Solution: Type **<ctrl-P>** four times, then type **<esc>F** twice.

5. Move the cursor to the letter **k** of the word **kitchen** in line 8—you should be able to do it with five commands.

Solution: Type **<ctrl-A>**, then type **<ctrl-P>** four times.

6. Finally, move the cursor to under the letter **M** of the word **Mississippi** on line 1—you should be able to do it with three commands.

Solution: Type **<esc><**, type **<ctrl-E>**, then type **<esc>B**.

Saving text and quitting

If you do not wish to continue working at this time, you should *save* your text, and then *quit*.

It is good practice to save your text file every so often while you are working on it; then, if an accident occurs, such as a power failure, you will not lose all of your work. You can save your text with the *save* command **<ctrl-X><ctrl-S>**. Type **<ctrl-X><ctrl-S>**—that is, first type **<ctrl-X>**, then type **<ctrl-S>**. Note that at the bottom of your screen the following message has appeared:

```
[Wrote 16 lines]
```

The text file has again been saved to your computer's memory. Note, too, that MicroEMACS will send you messages from time to time; the messages enclosed in square brackets '[' ']' are for your information, and do not necessarily mean that something is wrong. To exit from MicroEMACS, type the *quit* command **<ctrl-X><ctrl-C>**. This will return you to **msh**.

4. Killing and Deleting

Now that you know how to move the cursor, you are ready to edit your text. To return to MicroEMACS, type the command:

```
me text1
```

Within a moment, **text1** will be restored to your screen.

By now, you probably have noticed that MicroEMACS is always ready to insert material into your text; unless you use the **<ctrl>** or **<esc>** keys, MicroEMACS will assume that whatever you type is meant to be text and will insert it onto your screen where your cursor is positioned.

The simplest way to erase text is simply to position the cursor to the right of the text you want to erase and backspace over it. MicroEMACS, however, has a set of commands that allow you to erase large amounts of text easily. These commands *kill* and *delete*; the distinction is important, and will be explained in a moment. The following display summarizes these commands:

<ctrl-D>	Delete 1 character to the right
<esc>D	Kill 1 word to the right
<del>	Delete 1 character to the left
<ctrl-H>	Delete 1 character to the left
<esc><del>	Kill 1 word to the left
<esc><ctrl-H>	Kill 1 word to the left
<ctrl-K>	Kill rest of line
<ctrl-Y>	Yank back (restore) killed text

Deleting versus killing

It is important to distinguish between killing and deleting. When text is *deleted*, it is erased completely; however, when text is *killed*, it is moved into a temporary storage area elsewhere in the computer. This storage area is erased when you move the cursor and then kill additional text. Until then, however, the killed text is saved. This aspect of killing allows you to restore text that you killed accidentally, and it also allows you to move or copy portions of text from one position to another.

Erasing text to the right

The first two commands to be presented erase text to the *right.*

Type the *delete* command **<ctrl-D>**. Note that the letter **F** of the word **From** has been erased, and the rest of the line has shifted one space to the left.

Now, type **<esc>D**. The rest of the word **From** has been erased, and the line has shifted three spaces to the left. Note that the cursor is positioned at the *space* before the string:

```
"Life
```

Type **<esc>D** again. The string **"Life** has vanished along with the *space* that preceded it, and the line has shifted *six* spaces to the left.

Note that **<ctrl-D>** *deletes* text, but **<esc>D** *kills* text.

MicroEMACS is designed so that when it erases text, it does so beginning at the *left edge* of the cursor. You should imagine that an invisible vertical bar separates the cursor from the character immediately to its left; as you enter the various kill and delete commands, this vertical bar moves to the right or the left with the cursor, and erases the characters it touches. Therefore, if you wish to erase a word but wish to keep both spaces around it, position your cursor directly *over* the first character of the word and strike **<esc>D**. If you wish to erase a word *and* the space before it, position the cursor at the space before you strike **<esc>D**, so that the invisible vertical bar sweeps away the space at which the cursor is positioned, as well as the word that follows.

Erasing text to the left

You can erase text to the *left* by using the **Delete** key. This key is located just below the backspace key on the right side of the alphabetic keyboard. Be sure to note where it is, because it is most useful. You can also erase text to the left with the command **<ctrl-H>**.

To see how to erase text to the left, first type the *end of line* command **<ctrl-E>**; this will move the cursor to the right of the colon on the first line of text. Type **<del>**. Note that the colon has vanished.

Type **<esc><del>**. The string

```
Mississippi"
```

has disappeared, and the cursor has moved to the second space following the word **the**.

Move the cursor four spaces to the left, so that it is over the letter **t** of the word **the**. Type **<esc><del>**. The word **on** has vanished, along with the space that was immediately to the right of it. As before, these commands erased text beginning immediately to the *left* of the cursor. The **<esc><del>** command can be used to erase words throughout your text.

If you wish to erase a word to the left yet preserve both spaces that are around it, position the cursor at the space immediately to the right of the word and strike **<esc><del>**. If you wish to erase a word to the left plus the space that immediately follows it, position the cursor under the first letter of the *next* word and then strike **<esc><del>**.

Note that typing **<del>** *deletes* text, but typing **<esc><del>** *kills* text.

Erasing lines of text

Finally, the following command erases a line of text: the *kill* command **<ctrl-K>**. This command erases a line of text, beginning from immediately to the left of the cursor.

To see how this works, move the cursor to the beginning of line 2. Now, strike **<ctrl-K>**. All of line 2 has vanished, and been replaced with an empty space. Strike **<ctrl-K>** again. The empty space has vanished, and the cursor is now positioned at the beginning of what used to be line 3, over the letter **I** of the word **its**.

As its name implies, the **<ctrl-K>** command *kills* the line of text.

Yanking back (restoring) text

Remember that when material is killed, MicroEMACS has temporarily stored it elsewhere. Thus, text that has been killed can be returned to the screen by using the *yank back* command **<ctrl-Y>**. Type **<ctrl-Y>**. All of line 2 has returned; the cursor, however, remains over the letter **i** of **its** in line 3.

Killing and deleting—exercises

To fix these distinctions in your mind, perform the next few exercises. Work the exercises in order, as each exercise builds on the ones that come before it, and do not look at the solution until you have at least tried to solve the problem.

Before you begin, move your cursor back to the upper left-hand corner of your screen by typing **<esc><**.

1. Erase the word **sheltered** in line 7 and the space that follows it.

Solution: To move the cursor to the correct position, type **<ctrl-N>** six times; type **<esc>F** four times; and type **<ctrl-F>** once. Then type **<esc><del>**.

2. Erase the word **children** in line 8, and the space that precedes it.

Solution: To move the cursor to the correct position, type **<ctrl-N>**, then type **<esc>F**. Type **<esc>D**.

3. Erase line 4.

Solution: To move cursor, type **<ctrl-P>** four times, then **<ctrl-A>**. Type **<ctrl-K>**.

4. Yank back line 4.

Solution: Type **<ctrl-Y>**.

Quitting

When you are finished, do not save the text. If you do so, the undamaged copy of the text that you made earlier will be replaced with the present changed copy. Rather, use the *quit* command **<ctrl-X><ctrl-C>**. Type **<ctrl-X><ctrl-C>**. On the bottom of your screen, MicroEMACS will respond:

```
Quit [y/n]?
```

Reply by typing **y** and a carriage return. If you type **n**, MicroEMACS will simply return you to where you were in the text. MicroEMACS will then return you to **msh**.

5. Block Killing and Moving Text

As noted above, text that is killed is stored temporarily within the computer. Killed text may be yanked back onto your screen, and not necessarily in the spot where it was originally killed. This feature allows you to move text from one position to another.

The following table summarizes the commands used to kill a block of text and move it:

<ctrl-K>	Kill text to end of line
<ctrl-@>	Set mark
<ctrl-W>	Kill block of text
<ctrl-Y>	Yank back text

Moving one line of text

To test these commands, invoke MicroEMACS for the text **text1** by typing the following command:

```
me text1
```

When MicroEMACS appears, the cursor will be positioned in the upper left-hand corner of the screen.

To move the first line of text, begin by typing the *kill* command **<ctrl-K>** twice. Now, press **<esc>>**, to move the cursor to the bottom of text. Finally, yank back the line by typing **<ctrl-Y>**. The line that reads

```
From "Life on the Mississippi":
```

is now at the bottom of your text.

Note that your cursor has moved to the point *after* the line you yanked back.

Multiple copying of killed text

When text is yanked back onto your screen, it is *not* deleted from within the computer. Rather, it is simply *copied* back onto the screen. This means that killed text can be reinserted into the text more than once. To see how this is done, return to the top of the text by typing **<esc><**. Then type **<ctrl-Y>**. The line you just killed now appears twice on your screen.

Note that the killed text will not be erased from its temporary storage until you move the cursor and then kill additional text. If you kill several lines or portions of lines in a row, all of the killed text will be stored in the buffer; if you are not careful, you may yank back a jumble of accumulated text.

Kill and move a block of text

If you wish to kill a block of text, you can either type the *kill* command **<ctrl-K>** repeatedly to kill the block one line at a time, or you can use the *block kill* command **<ctrl-W>**. To use this command, you must first set a *mark* on the screen, an invisible character that acts as a signal to the computer. The mark is set with the *mark* command **<ctrl-@>**.

Once the mark is set, you must move your cursor to the other end of the block of text you wish to kill, and then strike **<ctrl-W>**. The block of text will be erased, and will be ready to be yanked back elsewhere.

Try this out on **text1**. Type **<esc><** to move the cursor to the upper left-hand corner of the screen. Then type the *set mark* command **<ctrl-@>**. By the way, be sure to type a '@', not a '2'. MicroEMACS will respond with the message

```
[Mark set]
```

at the bottom of your screen. Now, move the cursor down four lines, and type **<ctrl-W>**. Note how the block of text you marked out has disappeared.

Move the cursor to the bottom of your text. Type **<ctrl-Y>**. The killed block of text has now been reinserted.

When you yank back text, be sure to position the cursor at the *exact* point where you want the text to be yanked back. This will ensure that the text will be yanked back in the proper place.

To try this out, move your cursor up four lines. Be careful that the cursor is at the *beginning* of the line. Now, type **<ctrl-Y>** again. Note that the text reappeared *above* where the cursor was positioned, and that the cursor was not moved from its position at the beginning of the line—which is not what would have happened had you positioned it in the middle or at the end of a line.

Although the text you are working with has only 16 lines, you can move much larger portions of text, using only these three commands. Remember, too, that you can use this technique to duplicate large portions of text at several positions, to save yourself considerable time in typing and reduce the number of possible typographical errors.

6. Capitalization and Other Tools

The next commands perform a number of useful tasks that will help with your editing. They are as follows:

<esc>C	Capitalize a word
<esc>L	Lowercase a word
<esc>U	Uppercase a word
<ctrl-T>	Transpose characters
<ctrl-L>	Redraw screen
<ctrl-X>F	Set word wrap

Before you begin this section, destroy the old text on your screen with the *quit* command **<ctrl-X><ctrl-C>**, and read into MicroEMACS a fresh copy of the text, as you did earlier.

Capitalization and lowercasing

MicroEMACS has several commands that can automatically capitalize words or make them all upper case or lower case.

Move the cursor to the letter **w** of the word **watermelon** on line 2. Type the *capitalize* command **<esc>C**. The word is now capitalized, and the cursor is now positioned at the space after it. Move the cursor back so that it is over the letter **m** in **Watermelon**. Press **<esc>C** again. The word changes **WaterMelon**. When you press **<esc>C**, MicroEMACS will capitalize the *first* letter the cursor meets.

MicroEMACS can also change a word to all upper case or all lower case. (There is very little need for a command that will change only the first character of an uppercase word to lower case, so it is not included.)

Type **<esc>B** to move the cursor so that it is again to the left of the word **WaterMelon**. It does not matter if the cursor is directly over the **W** or at the space to its left; as you will see, this mean that you can capitalize or lowercase a number of words in a row without having to move the cursor.

Type the *uppercase* command **<esc>U**. The word is now spelled WATERMELON, and the cursor has jumped to the space after the word.

Again, move the cursor to the left of the word **WATERMELON**. Type the *lowercase* command **<esc>L**. The word has changed back to **watermelon**. Now, move the cursor to the left of the string

```
"Life
```

on line 1. Type **<esc>L** once again. Note that the quotation mark is not affected by the command, but the letter **L** is now lower case. **<esc>L** not only shifts a word that is all upper case to lower case: it can also un-capitalize a word.

Note that the *uppercase* and *lowercase* commands will stop at the first point of punctuation they encounter after the first letter they find; this means that, for example, to change the case of a word with an apostrophe in it you must type the appropriate command twice.

Transpose characters

MicroEMACS allows you to reverse the position of two characters, or *transpose* them, with the *transpose* command **<ctrl-T>**.

Type **<ctrl-T>**. The character that is under the cursor has been transposed with the character immediately to its *left*. Type **<ctrl-T>** again. The characters have returned to their original order.

Screen redraw

Occasionally, the characters on your screen may become mixed up, due to an unforeseen complication beyond your control. The *redraw screen* command **<ctrl-L>** will redraw your screen to the way it was before it was scrambled.

Type **<ctrl-L>**. Notice how the screen flickers and the text is rewritten. Had your screen been spoiled by extraneous material, that material would have been erased and the original text rewritten.

The **<ctrl-L>** command also has another use: you can move the line on which the cursor is positioned to the center of the screen. If you have a file that contains more than one screenful of text and you wish to move a particular line to the center of the screen, position the cursor on that line and type **<ctrl-U><ctrl-L>**. Immediately, the screen will be rebuilt with the line you were interested in positioned in the center.

Word wrap

Although you have not yet had much opportunity to use it, MicroEMACS will automatically wrap around text that you are typing into your computer. Word wrapping is controlled with the *word wrap* command **<ctrl-X>F**.

To see how the word wrap command works, first move to the end of the text file by typing the *end of file* command **<esc>>**. Now type the following text; however, do *not* type any carriage returns:

```
A cucumber should be well sliced,
and dressed with pepper and vinegar,
and then thrown out, as good
for nothing.
```

When you reached the edge of your screen, a dollar sign was printed and you were allowed to continue typing. MicroEMACS accepted the characters you typed, but it placed them at a location beyond the right edge of your screen.

Now, move to the beginning of the next line and type **<ctrl-U>**. MicroEMACS will reply with the message:

```
Arg: 4
```

Type **30**. The line at the bottom of your screen now appears as follows:

```
Arg: 30
```

(The use of the *argument* command **<ctrl-U>** will be explained in full in chapter 11.) Now type the *word-wrap* command **<ctrl-X>F**. MicroEMACS will now say at the bottom of your screen:

```
[Wrap at column 30]
```

This sequence of commands has set the word-wrap function, and told it to wrap to the next line all words that extend beyond the 30th column on your screen.

To test word wrapping, type the above text again, without using the carriage return key. When you finish, it should appear as follows:

```
A cucumber should be well
sliced, and dressed with
pepper and vinegar, and then
thrown out, as good for nothing.
```

MicroEMACS automatically moved your cursor to the next line when you typed a space character after the 30th column on your screen.

The *word wrap* feature automatically moves your cursor to the beginning of the next line once you type past a preset border on your screen; when you first enter MicroEMACS, that limit is automatically set at the first column, which in effect means that that word wrap has been turned off.

When you type prose, for a report or a letter of some sort, you probably will want to set the border at the 65th column, so that the printed text will fit neatly onto a sheet of paper. If you are using MicroEMACS to type in a program, however, you probably will want to turn off word wrapping, so you do not accidentally introduce carriage returns into your code.

If you wish to fix the border at some special point on your screen but do not wish to go through the tedium of figuring out how many columns from the left it is, simply position the cursor where you want the border to be, type **<ctrl-X>F**, and then type a carriage return. When **<ctrl-X>F** is typed without being preceded by a <ctrl-U> command, it sets the word-wrap border at the point your cursor happens to be positioned. When you do this, MicroEMACS will then print a message at the bottom of your terminal tells you where the word-wrap border is now set.

If you wish to turn off the word wrap feature again, simply set the word wrap border at some very large number, such as 5,000.

7. Search and Reverse Search

When you edit a large text, you may wish to change particular words or phrases. To do this, you can roll through the text and read each line to find them; or you can have MicroEMACS find them for you. The following summarizes the MicroEMACS search commands:

<ctrl-S>	Search forward incrementally
<esc>S	Search forward with prompt
<ctrl-R>	Search backwards incrementally
<esc>R	Search backwards with prompt
<ctrl-G>	Cancel a search command

Before you continue, close the present file by typing **<ctrl-Z>**; now reinvoke the editor with the file **text1** by typing

```
me text1
```

The following sections will perform some exercises with this file.

Search forward

As you can see from the display, MicroEMACS has two ways to search forward: incrementally, and with a prompt.

An *incremental* search is one in which the search is performed as you type the characters. To see how this works, first, type the *beginning of text* command **<esc><** to move the cursor to the upper left-hand corner of your screen. Now, type the *incremental search* command **<ctrl-S>**. MicroEMACS will respond by prompting with the message

```
i-search forward:
```

at the bottom of the screen.

We will now search for the word **way**. Type the letter **w**. Note that the cursor has jumped to the first word that has a **w**, which is **watermelon**. Now, type **a**. The cursor moves forward one space, because **watermelon** also has **wa** in it. Note how the message at the bottom of the screen has also changed to reflect what you have typed.

Now type y. The cursor has jumped ahead to the word **way**. Finally, type **<esc>**. MicroEMACS will reply with the message

```
[Done]
```

which indicates that the search is finished.

If you attempt an incremental search for word that is not in the file, MicroEMACS will find as many of the letters as it can, and then give you an error message. For example, if you tried to search incrementally for the word **ways**, MicroEMACS would move the cursor to the word **way**; when you typed 's', it would tell you

```
failing i-search forward: ways
```

With the *prompt search*, however, you type in the word all at once. To see how this works, type **<esc><**, to return to the top of the file. Now, type the *prompt search* command **<esc>S**. MicroEMACS will respond by prompting with the message

```
Search [way]:
```

at the bottom of the screen. The word **ways** is shown, because that was the last word for which you searched, and so it is kept in the search buffer.

Type in the words **been there**, then press the carriage return. Notice that the cursor has jumped to the period after the word **there** in the last line of your text. MicroEMACS searched for the words **been there**, found them, and moved the cursor to them.

If the word you were searching for was not in your text, or at least was not in the portion that lies between your cursor and the end of the text, MicroEMACS would not have moved the cursor, and would have displayed the message

```
Not found
```

at the bottom of your screen.

Reverse search

The search commands, useful as they are, can only search forward through your text. To search backwards, use the reverse search commands **<ctrl-R>** and **<esc>R**. These work exactly the same as their forward-searching counterparts, except that they search toward the beginning of the file rather than toward the end.

For example, type **<esc>R**. MicroEMACS will reply with the message

```
Reverse search [been there]:
```

at the bottom of your screen. The words in square brackets are the words you entered earlier for the *search* command; MicroEMACS remembered them. If you wanted to search for **been there** again, you would just press the carriage return. For now, however, type the word **watermelon** and press the carriage return.

Notice that the cursor has jumped so that it is under the letter **w** of the word **watermelon** in line 2. When you search forward, the cursor will move to the *space after* the word you are searching for, whereas when you reverse search the cursor will be moved to the *first letter* of the word you are searching for.

Cancel a command

As you have noticed, the commands to move the cursor or to delete or kill text all execute immediately. Although this speeds your editing, it also means that if you type a command by mistake, it executes before you can stop it.

The *search* and *reverse search* commands, however, wait for you to respond to their prompts before they execute. If you type **<esc>S** or **<esc>R** by accident, MicroEMACS will interrupt your editing and wait patiently for you to initate a search that you do not want to perform. You can evade this problem, however, with the *cancel* command **<ctrl-G>**. This command tells MicroEMACS to ignore the previous command.

To see how this command works, type **<esc>R**. When the prompt appears at the bottom of your screen, type **<ctrl-G>**. Three things happen: your monitor chimes, the characters ^G appears at the bottom of your screen, and the cursor returns to where it in your text was before you first typed **<esc>R**. The **<esc>R** command has been cancelled, and you are free to continue editing.

If you cancel an *incremental search* command, **<ctrl-S>** or **<esc-S>**, the cursor will return to where it was before you began the search. For example, type **<esc><** to return to the top of the file. Now type **<ctrl-S>**, to begin an incremental search, and type **w**. When the cursor moves to the **w** in **watermelon**, type **<ctrl-G>**. The bell will ring, and your cursor will be returned to the type of the file, which is where you began the search.

8. Saving Text and Exiting

The last set of basic editing commands allow you to save your text and exit from the MicroEMACS program. They are as follows:

<ctrl-X><ctrl-S>	Save text
<ctrl-X><ctrl-W>	Write text to a new file
<ctrl-Z>	Save text and exit
<ctrl-X><ctrl-C>	Exit without saving text

You have used two of these commands already: the *save* command **<ctrl-X><ctrl-S>** and the *quit* command **<ctrl-X><ctrl-C>**, which respectively allow you to save text or to exit from MicroEMACS without saving text. (Commands that begin with **<ctrl-X>** are called *extended* commands; they are used frequently in the advanced editing to be covered in the second half of this tutorial.)

Write text to a new file

If you wish, you may copy the text you are currently editing to a text file other than the one from which you originally took the text. Do this with the *write* command **<ctrl-X><ctrl-W>**.

To test this command, type **<ctrl-X><ctrl-W>**. MicroEMACS will display the following message on the bottom of your screen:

```
Write file:
```

MicroEMACS is asking for the name of the file to which you wish to write the text. Type **twain**. MicroEMACS will reply:

```
[Wrote 21 lines]
```

The 16 lines of your text have been copied to a new file, called **twain**. Note that the status line at the bottom of your screen has changed to read as follows:

```
-- ST MicroEMACS V1.2 -- text1 -- File: twain -------------------
```

The significance of the change in file name will be discussed in the second half of this tutorial.

Before you copy text to a new file, be sure that you have not selected a file name that is already being used. If you do, whatever is being stored under that file name will be erased, and the text created with MicroEMACS will be stored in its place.

Save text and exit

Finally, the *store* command **<ctrl-Z>** will save your text *and* move you out of the MicroEMACS program. To see how this works, watch the bottom line of your terminal carefully and type **<ctrl-Z>**, and then the **msh** prompt reappears. MicroEMACS has saved your text, and now you can issue commands directly to **msh**.

9. Basic Editing—Conclusion and Summary

This concludes the presentation of MicroEMACS's basic commands. The second half of this tutorial will introduce you to the advanced features of the MicroEMACS interactive screen editor.

This section introduced the basics of using an interactive screen editor, and presented the basic MicroEMACS editing commands.

If you have mastered the commands and techniques in the first half of this tutorial, you may have no need to work any further, because you now can create a file, edit it, store it, and recall it for further editing.

The tutorial gives instructions on how to invoke MicroEMACS, how to name a text file, and the meaning of the information in the MicroEMACS command line.

An exercise text is presented, and instructions on how to type in the text and save it are given.

A number of commands can be used to move the cursor around the screen. **<ctrl-F>** and **<esc>F** move the cursor forward on the line. **<ctrl-B>** and **<esc>B** move the cursor backwards on the line. **<ctrl-P>** and **<ctrl-N>** move the cursor to the previous or next lines, respectively. **<ctrl-A>** and **<ctrl-E>** move the cursor to the beginning or the end of the line, respectively. **<esc><** and **<esc>>** move the cursor to the beginning or end of the text, respectively. **<ctrl-V>** and **<esc>V** roll the screen forwards or backwards, respectively.

You can erase text in a number of ways. **<ctrl-D>** and **<esc>D** erase text to the right. **<del>** and **<esc><del>** erase text to the left. **<ctrl-K>** erases a line of text (or a portion of a line, should the cursor be positioned in the middle of line). **<ctrl-D** and **<del>** *delete* text, whereas **<esc>D**, **<esc><del>**, and **<ctrl-K>** *kill* text. **<ctrl-Y>** yanks back killed text.

Text can be block killed and moved from one part of your text to another. To mark a block of text for killing, first type **<ctrl-@>** at one end, then move the cursor to the other end. Typing **<ctrl-W>** kills the marked block of text, and **<ctrl-Y>** yanks back killed text.

The following commands allow the user to block-kill and move text. **<ctrl-@>** sets a mark; **<ctrl-W>** deletes all text between the mark and the cursor. **<ctrl-Y>** yanks back the block-killed text wherever the cursor is positioned.

Specific commands capitalize, uppercase, and lowercase words: **<esc>C** capitalizes a word; **<esc>L** lowercases a word; and **<esc>U** uppercases a word. **<ctrl-T>** allows you to transpose characters automatically, and **<ctrl-L>** redraws a scrambled screen. The word wrap feature can be adjusted using the command **<ctrl-X>F**.

Words or parts of words can be searched for either forwards or backwards through the text: **<esc>S** searches forward, and **<esc>R** searches backwards. **<ctrl-G>** cancels these commands.

Finally, **<ctrl-X><ctrl-S>** saves text to the file named on the command line; **<ctrl-X><ctrl-C>** allows the user to exit from MicroEMACS without saving text; and **<ctrl-Z>** saves text and moves you back into **msh**.

10. Introduction to Advanced Editing

The second half of this tutorial, chapters 10 through 16, will introduce the advanced features of the MicroEMACS interactive screen editor.

The techniques described here will help you execute complex editing tasks with minimal trouble. You will be able to edit more than one text at a time, display more than one text on your screen at a time, enter a long or complicated phrase repeatedly with only one keystroke, and give commands to **msh** without having to exit from MicroEMACS.

Before beginning, however, you must prepare a new text file—you were probably a little tired of watermelon by now, anyway.

Type the following command to **msh**:

```
me text2
```

This text has been included on the floppy disks that contained there is no need to retype it. Within a moment, **text2** will appear on your screen, as follows:

```
From the "Devil's Dictionary":
A penny saved is a penny to squander.
A man is known by the company he organizes.
A bird in the hand is worth what it will bring.
Think twice before you speak to a friend in need.
He laughs best who laughs least.
Least said is soonest disavowed.
Speak of the Devil and he will hear about it.
Of two evils choose to be the least.
Strike while your employer has a big contract.
Where there's a will there's a won't.
```

11. Arguments

Most of the commands described in the first part of this tutorial can be used with *arguments*. An argument is a subcommand that tells MicroEMACS to execute a command repeatedly. With MicroEMACS, arguments are introduced by striking **<ctrl-U>**.

Arguments—default values

By itself, **<ctrl-U>** sets the argument at *four*. To illustrate this, first type the *next line* command **<ctrl-N>**. By itself, this command moves the cursor down one line, from being over the **F** in the word **From** on line 1, to being over the **A** at the beginning of line 2.

Now, type **<ctrl-U>**. MicroEMACS replies with the message:

```
Arg: 4
```

Now type **<ctrl-N>**. The cursor jumps down *four* lines, from the letter **A** in line 2 to the letter **H** of the word **He** at the beginning of line 6.

Type **<ctrl-U>**. The line at the bottom of the screen again shows that the value of the argument is 4. Type **<ctrl-U>** again. Now the line at the bottom of the screen reads:

```
Arg: 16
```

Type **<ctrl-U>** once more. The line at the bottom of the screen now reads:

```
Arg: 64
```

Each time you type **<ctrl-U>**, the value of the argument is *multiplied* by four. Type the *forward* command **<ctrl-F>**. The cursor has jumped ahead 64 characters, and is now under the period at the end of line 7.

Selecting values

Naturally, arguments do not have to be powers of four. You can set the argument to whatever number you wish, simply by typing **<ctrl-U>** and then typing in the number you want.

For example, type **<ctrl-U>**, and then type **3**. The line at the bottom of the screen now reads:

```
Arg: 3
```

Type the *delete* command **<esc><del>**. MicroEMACS has deleted three words to the left.

Arguments can be used to increase the power of any *cursor movement* command, or any *kill* or *delete* command (with the sole exception of **<ctrl-W>**, the *block kill* command).

Deleting with arguments—an exception

Killing and *deleting* were described in the first part of this tutorial. They were said to differ in that text that was killed was stored in a special area of the computer and could be yanked back, whereas text that was deleted was erased outright. However, there is one exception to this rule: any text that is deleted using an argument can also be yanked back.

Move the cursor to the upper left-hand corner of the screen by typing the *begin text* command **<esc><**. Then, type **<ctrl-U><ctrl-D>**. Note that the word **From** has disappeared. Move the cursor to the right until it is between the words **Devil's** and **Dictionary**, then type **<ctrl-Y>**. The word **From** has been moved within the line (although the spaces around it have not been moved). This function is very handy, and should greatly speed your editing.

Remember, too, that unless you move the cursor between one set of deletions and another, the computer's storage area will not be erased, and you may yank back a jumble of text.

Arguments—exercises

The next few exercises show how arguments can be used to make your editing commands more powerful and efficient. Before beginning, type **<esc><** to move the cursor to the upper left-hand corner of your screen.

1. Lowercase the word **Devil** in line 8. Use no more than three commands. (**<ctrl-U>** plus a number and command counts as one command.)

Solution: To move the cursor, type **<ctrl-U>7<ctrl-N>**, then **<ctrl-U>3 <esc>F**. Then type **<esc>L**.

2. Kill the last four lines of the text. Use no more than two commands.

Solution: To move the cursor, type **<ctrl-A>**. Then type **<ctrl-U><ctrl-K>**.

3. Make two copies of the killed lines at the top of your text. Use no more than two commands.

Solution: To move the cursor, type **<esc><**. Then type **<ctrl-U>2<ctrl-Y>**.

4. Finally, delete the last 23 characters of the second line of text. Use no more than four commands.

Solution: To move the cursor, type **<esc><**, **<ctrl-N>**, **<ctrl-E>**, and then **<ctrl-U>23<del>**.

12. Buffers and Files

Before beginning this section, replace the changed copy of the text on your screen with a fresh copy. Type the *quit* command **<ctrl-X><ctrl-C>** to exit from MicroEMACS without saving the text; once the **msh** prompt appears, return to MicroEMACS

by typing:

```
me text2
```

Now look at the status line at the bottom of your screen. It should appear as follows:

```
-- ST MicroEMACS V1.2 -- text2 -- File: text2 ------------------
```

As noted in the first half of this tutorial, the name on the left of the command line is that of the program. The name in the middle is the name of the *buffer* with which you are now working, and the name to the right is the name of the *file* from which you read the text.

Definitions

A *file* is a text that has been given a name and has been permanently stored by your computer. A *buffer* is a portion of the computer's memory that has been set aside for you to use, that may be given a name, and into which you can put text temporarily. You can put text into the buffer by typing it in from your keyboard or by *copying* it from a file.

Unlike a file, a buffer is not permanent: if your computer were to stop working (because you turned the power off, for example), a file would not be affected, but a buffer would be erased.

You must *name* your files because you work with many different files, and you must have some way to tell them apart. Likewise, MicroEMACS allows you to *name* your buffers, because MicroEMACS allows you to work with more than one buffer at a time.

File and buffer commands

MicroEMACS gives you a number of commands for handling files and buffers. The following display summarizes them.

<ctrl-X><ctrl-W>	Write text to file
<ctrl-X><ctrl-F>	Rename file
<ctrl-X><ctrl-R>	Replace buffer with named file
<ctrl-X><ctrl-V>	Switch buffer or create a new buffer
<ctrl-X>K	Delete a buffer
<ctrl-X><ctrl-B>	Display the status of each buffer

Write and rename commands

The *write* command **<ctrl-X><ctrl-W>** was introduced earlier, when the commands for saving text and exiting were discussed. To review, **<ctrl-X><ctrl-W>** changes the name of the file into which the text is saved, and then writes a copy of the text into that file.

Type **<ctrl-X><ctrl-W>**. MicroEMACS responds by printing

```
Write file:
```

on the last line of your screen.

Type **junkfile**, then a carriage return. Two things happen: First, MicroEMACS writes the message

```
[Wrote 11 lines]
```

at the bottom of your screen. Second, the name of the file shown on the status line has changed from **text2** to **junkfile**. MicroEMACS is reminding you that your text will be saved from now on to the file **junkfile**, unless you change the file name once again.

The *file rename* command **<ctrl-X><ctrl-F>** allows you rename the file to which you are saving text, *without* automatically writing the text to it. Type **<ctrl-X><ctrl-F>**. MicroEMACS will reply with the prompt:

```
Name:
```

Now type **text2**. Note that MicroEMACS does *not* send you a message that lines were written to the file; however, the name of the file shown on the status line has changed from **junkfile** back to **text2**.

Replace text in a buffer

The *replace* command **<ctrl-X><ctrl-R>** allows you to replace the text in your buffer with the text taken from another file.

Suppose, for example, that you had edited **text2** and saved it, and now wished to edit **text1**. You could exit from MicroEMACS, then re-invoke MicroEMACS for the file **text2**, but this is cumbersome. A more efficient way is to simply replace the **text2** in your buffer with **text1**.

Type **<ctrl-X><ctrl-R>**. MicroEMACS replies with the prompt:

```
Read file:
```

Type **text1**. Notice that **text2** has rolled away and been replaced with **text1**. Now, check the status line. Notice that although the name of the *buffer* is still **text2**, the name of the *file* has changed to **text1**. You can now edit **text1**; when you save the edited text, MicroEMACS will copy it back into the file **text1**—unless, of course, you choose to rename the file.

Visiting another buffer

The last command of this set, the *visit* command **<ctrl-X><ctrl-V>**, allows you to create more than one buffer at a time, to jump from one buffer to another, and move text between buffers. This powerful command has numerous features.

Before beginning, however, straighten up your buffer by replacing **text1** with **text2**. Type the *replace* command **<ctrl-X><ctrl-R>**; when MicroEMACS replies by asking

```
Read file:
```

at the bottom of your screen, type **text2**.

You should now have the file **text2** read into the buffer named **text2**.

Now, type the *visit* command **<ctrl-X><ctrl-V>**. MicroEMACS replies with the prompt

```
Visit file:
```

at the bottom of the screen. Now type **text1**. Several things happen. **text2** rolls off the screen and is replaced with **text1**; the status line changes to show that both the buffer name and the file name are now **text1**; and the message

```
[Read 16 lines]
```

appears at the bottom of the screen.

This does *not* mean that your previous buffer has been erased, as it would have been had you used the *replace* command **<ctrl-X><ctrl-R>**. **text2** is still being kept "alive" in a buffer and is available for editing; however, it is not being shown on your screen at the present moment.

Type **<ctrl-X><ctrl-V>** again, and when the prompt appears, type **text2**. **text1** scrolls off your screen and is replaced by **text2**, and the message

```
[Old buffer]
```

appears at the bottom of your screen. You have just jumped from one buffer to another.

Move text from one buffer to another

The *visit* command **<ctrl-X><ctrl-V>** not only allows you jump from one buffer to another: it allows you to *move text* from one buffer to another as well. The following example shows how you can do this.

First, kill the first line of **text2** by typing the *kill* command **<ctrl-K>** twice. This removes both the line of text *and* the space that it occupied; if you did not remove the space as well the line itself, no new line created for the text when you yank it back. Next, type **<ctrl-X><ctrl-V>**; when the prompt

```
Visit file:
```

appears at the bottom of your screen, type **text1**. When **text1** has rolled onto your screen, type the *yank back* command **<ctrl-Y>**. The line you killed in **text2** has now been moved into **text1**.

Checking buffer status

The number of buffers you can use at any one time is limited only by the size of your computer. You should create only as many buffers as you need to use immediately; this will help the computer run efficiently.

To help you keep track of your buffers, MicroEMACS has the *buffer status* command **<ctrl-X><ctrl-B>**. Type **<ctrl-X><ctrl-B>**. Note that the status line has moved up to the middle of the of the screen, and the bottom half of your screen has been replaced with the following display:

```
C   Size Buffer            File
-   ---- ------            ----
*    920 text1             text1
*    423 text2             text2
```

This display is called the *buffer status window*. The use of windows will be discussed more fully in the following section.

The letter **C** over the leftmost column stands for **Changed**. An asterisk on a line indicates that the buffer has been changed since it was last saved, whereas a space means that the buffer has not been changed. **Size** indicates the buffer's size, in number of characters; **Buffer** lists the buffer name, and **File** lists the file name.

Now, kill the second line of **text1** by typing the *kill* command **<ctrl-K>**, then type **<ctrl-X><ctrl-B>** once again. Note that size of the buffer **text1** has been reduced from 913 characters to 882, to reflect the decrease in the size of the buffer.

To make this display disappear, type the *one window* command **<ctrl-X>1**. This command will be discussed in full in the next chapter.

Renaming a buffer

One more point must be covered with the *visit* command. **msh** will not allow you to have more than one file with the same name, in order to avoid confusion; for the same reason, MicroEMACS will not allow you to have more than one *buffer* with the same name.

Ordinarily, when you visit a file that is not already in a buffer, MicroEMACS will create a new buffer and give it the same name that the file you are visiting has. However, if for some reason you already have a buffer with the same name as the file you wish to visit, MicroEMACS will stop and ask you to give a new, different name to the buffer it is creating.

For example, suppose that you wanted to visit a new *file* called **sample**, perhaps from another directory, but you already have a *buffer* named **sample**. MicroEMACS would stop and give you this prompt at the bottom of the screen:

```
Buffer name:
```

When you named this new buffer, MicroEMACS would proceed to read the file **text2** into it.

Delete a buffer

If you wish to delete a buffer, simply type the *delete buffer* command **<ctrl-X>K**. This command will allow you only to delete a buffer that is hidden, not one that is being displayed.

Type **<ctrl-X>K**. MicroEMACS will give you the prompt:

```
Kill buffer:
```

Type **text2**. Because you have changed the buffer, MicroEMACS asks:

```
Discard changes [y/n]?
```

Type y, and then type the *buffer status* command **<ctrl-X><ctrl-B>**; the buffer status window will no longer show the buffer **text2**. Note that although the prompt refers to *killing* a buffer, the buffer is in fact *deleted* and cannot be yanked back.

Summary

These buffer and file commands allow you to edit more than one text at once, and move text between buffers and files. Just how useful these commands are will be seen when you cover the next topic, *windows*.

13. Windows

Before beginning this section, it will be necessary to create a new text file. Exit from MicroEMACS by typing the *quit* command **<ctrl-X><ctrl-C>**; then reinvoke MicroEMACS for the text file **text1**

with the command:

```
me text1
```

Now, copy **text2** into a buffer by typing the **visit** command **<ctrl-X><ctrl-V>**. When the message

```
Visit file:
```

appears at the bottom of your screen, type **text2**. MicroEMACS will read **text2** into a buffer, and show the message

```
[Read 11 lines]
```

at the bottom of your screen.

Finally, copy a new text, called **text3**, into a buffer. Type **<ctrl-X><ctrl-V>** again. When MicroEMACS asks which file to visit, type **text3**. The message

```
[Read 92 lines]
```

will appear at the bottom of your screen.

The first screenful of text will appear as follows:

```
From "Gulliver's Travels":
    I said there was a society of men among us,
bred up from their youth in the art of proving
by words multiplied for the purpose, that
white is black, and black is white, according
as they are paid.  To this society all the rest
of the people are slaves.
    "For example.  If my neighbor hath a mind to my
cow, he hires a lawyer to prove that he ought to
have my cow from me.  I must then hire another to
defend my right; it being against all rules of law
that any man should be allowed to speak for himself.
Now in this case, I who am the true owner lie under
two great disadvantages.  First, my lawyer being
practiced almost from his cradle in defending
falsehood is quite out of his element when he would
be an advocate for justice, which as an office
unnatural, he always attempts with great awkwardness,
if not ill-will.  The second disadvantage is that my
lawyer must proceed with great caution, or else he
will be reprimanded by the judges, and abhorred by
his brethren, as one who would lessen the practice
-- ST MicroEMACS V1.2 -- text3 -- File: text3 -----------
```

At this point, **text3** is on your screen, and **text1** and **text2** are hidden.

You could edit first one text and then another, while remembering just how things stood with the texts that were hidden; but it would be much easier if you could display all three texts on your screen simultaneously. MicroEMACS allows you to do just that, by using *windows*.

Window commands

A *window* is a portion of your screen that is set aside and can be manipulated independently from the rest of the screen. MicroEMACS's commands for manipulating windows are summarized in the following display.

<ctrl-X>2	Create a window
<ctrl-X>1	Delete extra windows
<ctrl-X>N	Move to next window
<ctrl-X>P	Move to previous window
<ctrl-X>Z	Enlarge window
<ctrl-X><ctrl-Z>	Shrink window
<ctrl-X><ctrl-N>	Scroll down
<ctrl-X><ctrl-P>	Scroll up
<esc>!	Move within window
<ctrl-X>B	Switch buffer

Creating windows and moving between them

The best way to grasp how a window works is to create one and work with it. Type the *create a window* command **<ctrl-X>2**.

Your screen is now divided into two parts, an upper and a lower. The same text is in each part, and the command lines give **text3** for the buffer and file names. Also, note that you still have only one cursor, which is in the upper left-hand corner of the screen.

The next step is to move from one window to another.

Type the *next window* command **<ctrl-X>N**. Your cursor has now jumped to the upper left-hand corner of the *lower* window.

Type the *previous window* command **<ctrl-X>P**. Your cursor has returned to the upper left-hand corner of the top window.

Now, type **<ctrl-X>2** again. The window on the top of your screen is now divided into two windows, for a total of three on your screen. Type **<ctrl-X>2** again. The window at the top of your screen has again divided into two windows, for a total of four.

It is possible to have as many as 11 windows on your screen at one time, although each window will show only the control line and one or two lines of text. Note that neither **<ctrl-X>2** nor **<ctrl-X>1** can be used with arguments.

Now, type the *one window* command **<ctrl-X>1**. Note that all of the extra windows have been eliminated, or *closed*.

Enlarging and shrinking windows

When MicroEMACS creates a window, it divides the window in which the cursor is positioned into half. You do not have to leave the windows at the size MicroEMACS creates them, however. If you wish, you may adjust the relative size of each window on your screen, using the *enlarge window* and *shrink window* commands.

Type **<ctrl-X>2** twice. Your screen is now divided into three windows: two in the top half of your screen, and the third in the bottom half.

Now, type the *enlarge window* command **<ctrl-X>Z**. The window at the top of your screen is now one line bigger, because it has borrowed a line from the window below it. Type **<ctrl-X>Z** again. Once again, the top window has borrowed a line from the middle window.

Now, type the *next window* command **<ctrl-X>N**, to move your cursor into the middle window. Again, type the *enlarge window* command **<ctrl-X>Z**. The middle window has borrowed a line from the bottom window, and is now one line larger.

The *enlarge window* command **<ctrl-X>Z** allows you to enlarge the window your cursor is in by borrowing lines from another window, provided that you do not shrink that other window out of existence. Every window must have at least two lines in it, one command line and one line of text.

The *shrink window* command **<ctrl-X><ctrl-Z>** allows you to decrease the size of a window. Type **<ctrl-X><ctrl-Z>**. The present window is now one line smaller, and the lower window is one line larger, because the line borrowed earlier has been returned.

The *enlarge window* and *shrink window* commands can also be used with arguments introduced with **<ctrl-U>**. However, remember that MicroEMACS will not accept an argument that would shrink another window out of existence.

Displaying text within a window

Displaying text within the limited area of a window can present special problems. The *view* commands **<ctrl-V>** and **<esc>V** will roll window-sized portions of text up or down, but you may become disoriented when a window shows only four or five lines of text at a time. Therefore, three special commands are available for displaying text within a window.

Two commands allow you to move your text by one line at a time, or *scroll* it: the *scroll up* command **<ctrl-X><ctrl-N>**, and the *scroll down* command **<ctrl-X><ctrl-P>**.

Type **<ctrl-X><ctrl-N>**. Note that the line at the top of your window has vanished, a new line has appeared at the bottom of your window, and the cursor is now at the beginning of what had been the second line of your window.

Now type **<ctrl-X><ctrl-P>**. The line at the top that had vanished earlier has now returned, the cursor is at the beginning of it, and the line at the bottom of the window has vanished. These commands allow you to move forward in your text slowly, so that you do not become disoriented.

Both of these commands can be used with arguments introduced by **<ctrl-U>**.

The third special movement command is the *move within window* command <esc>!. This command moves the line your cursor is on to the top of the window.

To try this out, move the cursor down three lines by typing **<ctrl-U>3<ctrl-N>**, then type **<esc>**!. (Be sure to type an exclamation point '!', not a numeral one '1', or nothing will happen.) Note that the line to which you had moved the cursor is now the first line in the window, and three new lines have scrolled up from the bottom of the window. You will find this command to be very useful as you become more experienced at using windows.

All three special movement commands can also be used when your screen has no extra windows, although you will not need them as much.

One buffer

Now that you have been introduced to the commands for manipulating windows, you can begin to use windows to speed your editing.

To begin with, scroll up the window you are in until you reach the top line of your text. You can do this either by typing the *scroll up* command **<ctrl-X><ctrl-P>** several times, or by typing **<esc><**.

Kill the first line of text with the *kill* command **<ctrl-K>**. Note that the first line of text has vanished from all three windows. Now, type **<ctrl-Y>** to yank back the text you just killed. The line has reappeared in all three windows.

The main advantage to displaying one buffer with more than one window is that each window can display a different portion of the text. This can be quite helpful if you are editing or moving a large text.

To demonstrate this, do the following: First, move to the end of the text in your present window by typing the *end of text* command **<esc>>**. Kill the last four lines.

You could move the killed lines to the beginning of your text by typing the *beginning of text* command **<esc><**; however, it is more convenient simply to type the *next window* command **<ctrl-X>**N, which will move you to the beginning of the text as displayed in the next window. Note that MicroEMACS remembers a different cursor position for each window.

Now yank back the four killed lines by typing **<ctrl-Y>**. You can simultaneously observe that the lines have been removed from the end of your text and that they have been restored at the beginning.

Multiple buffers

Windows are especially helpful when they display more than one text. Remember that at present you are working with *three* buffers, named **text1**, **text2**, and **text3**, although your screen is displaying only **text3**. To display a different text in a window, use the *switch buffer* command **<ctrl-X>B**.

Type **<ctrl-X>B**. When MicroEMACS asks

```
Use buffer:
```

at the bottom of the screen, type **text1**. The text in your present window will be replaced with **text1**. Note that the command line in that window has changed, too, to reflect the fact that the buffer and the file names are now **text1**.

Moving and copying text among buffers

As you can see, it is now very easy to copy text among buffers. To see how this is done, first kill the first line of **text1** by typing the **<ctrl-K>** command twice. Yank back the line immediately by typing **<ctrl-Y>**. Remember, the line you killed has *not* been erased from its special storage area, and may be yanked back any number of times.

Now, move to the previous window by typing **<ctrl-X>P**, then yank back the killed line by typing **<ctrl-Y>**. This technique can also be used with the *block kill* command **<ctrl-W>** to move large amounts of text from one buffer to another. It also allows you to *copy* text from one window to another.

Checking buffer status

The *buffer status command* **<ctrl-X><ctrl-B>** can be used when you are already displaying more than one window on your screen.

When you want to remove the buffer status window, use either the *one window* command **<ctrl-X>1**, or move your cursor into the buffer status window using the *next window* command **<ctrl-X>N** and replace it with another buffer by typing the *switch buffer* command **<ctrl-X>B**.

Saving text from windows

The final step is to save the text from your windows and buffers. Close the lower two windows with the *one window* command **<ctrl-X>1**. Remember, when you close a window, the text that it displayed is still kept in a buffer that is *hidden* from your screen. For now, do *not* save any of these altered texts.

When you use the *save* command **<ctrl-X><ctrl-S>**, only the text in the window in which the cursor is position will be written to its file. If only one window is displayed on the screen, the *save* command will save only its text.

If you made changes to the text in another buffer, such as moving portions of it to another buffer, MicroEMACS will ask

```
Quit [y/n]:
```

If you answer **'n'**, MicroEMACS will *save* the contents of the buffer you are currently displaying by writing them to your disk, but it will ignore the contents of other buffers; and your cursor will be returned to its previous position in the text. If you answer **'y'**, MicroEMACS again will save the contents of the current buffer and ignore the other buffers, but you will exit from MicroEMACS and return to **msh**.

Exercises

The following exercises will help you master the use of windows and buffers. Before you begin, exit from MicroEMACS by typing the *quit* command **<ctrl-X><ctrl-C>**.

1. Invoke MicroEMACS to edit **text1**. Display **text2** in a separate window. Copy the first four lines of **text1** to **text2**. Destroy **text1**'s buffer. Exit from MicroEMACS without saving the text.

Solution: To invoke MicroEMACS for **text1** use the mouse to position the cursor over the icon labelled **ME.TTP**, and press the button twice; when the **Open Application** box appears, type in the name of the file to be edited, **text1**, and then press the carriage return key. Before **text2** can be displayed on a separate window, it must be copied into a buffer; again, type the *visit* command **<ctrl-X><ctrl-V>**. When MicroEMACS asks

```
Visit file:
```

type **text2**.

When **text2** has been copied into its buffer, type the *create window* command **<ctrl-X>2**, and read **text1** into the window with the *switch buffer* command **<ctrl-X>B**.

To copy the top four lines from **text1** to **text2**, first kill the first four lines by typing **<ctrl-U>4 <ctrl-K>**. Then yank them back immediately by typing **<ctrl-Y>**; jump to **text2**'s window by typing the *next window* command **<ctrl-X>N**, and finally yank back the four killed lines again by typing **<ctrl-Y>**.

To destroy the buffer holding the copy of **text1**, type the *one window* command **<ctrl-X>1**, then type the *delete buffer* command **<ctrl-X>K**. When MicroEMACS asks

```
Kill buffer:
```

type **text1**. When MicroEMACS asks

```
Discard changes [y/n]?
```

type **y**.

Finally, to exit without saving text, type the *quit* command **<ctrl-X><ctrl-C>**.

2. Display **text1**, **text2**, and **text3** in three equally sized windows. Scroll through each text in turn. Check the status of the buffers to see if any were altered, then exit from MicroEMACS without saving the texts.

Solution: Invoke MicroEMACS to edit **text1** by typing

```
me text1
```

Copy **text2** and **text3** into buffers by using the *visit* command **<ctrl-X><ctrl-V>** twice.

Create three windows on your screen by typing the *create a window* command **<ctrl-X>2** twice.

Move among the windows by using the *next window* command **<ctrl-X>N** and the *previous window* command **<ctrl-X>P**; then make sure the windows are of even size by using the *enlarge window* command **<ctrl-X>Z** and the *shrink window* command **<ctrl-X><ctrl-Z>**.

Read **text1** and **text2** into your extra windows by using the *switch buffer* command **<ctrl-X>B**.

To scroll through the texts, use the *scroll down* command **<ctrl-X><ctrl-N>** and the *scroll up* command **<ctrl-X><ctrl-P>**.

When you are done scrolling, check the buffers' status with the *buffer status* command **<ctrl-X><ctrl-B>**. Close the buffer status window by typing the *one window* command **<ctrl-X>1**.

Exit from MicroEMACS by typing the *quit* command **<ctrl-X><ctrl-C>**.

14. Keyboard Macros and Search and Replace

Another helpful feature of MicroEMACS is that it allows you to create a *keyboard macro.*

Before beginning this section, reinvoke MicroEMACS to edit **text3** by typing the command

```
me text3
```

The term *macro* means a number of commands or characters that are bundled together under a common name. Although MicroEMACS allows you to create only one macro at a time, this macro can consist of a common *phrase* or a common *command* or *series of commands* that you use while editing your file.

Keyboard macro commands

The keyboard macro commands are as follows:

<ctrl-X>(	Begin macro collection
<ctrl-X>)	End macro collection
<ctrl-X>E	Execute macro

To begin to create a macro, type the *begin macro* command **<ctrl-X>(**. Be sure to type an open parenthesis '(', not a numeral '9'. MicroEMACS will reply with the message

```
[Start macro]
```

Type the following phrase:

```
interactive screen editor
```

Then type the *end macro* command **<ctrl-X>)**. Be sure you type a close parenthesis ')', not a numeral '0'. MicroEMACS will reply with the message

```
[End macro]
```

Move your cursor down two lines and execute the macro by typing the *execute macro* command **<ctrl-X>E**. The phrase you typed into the macro has been inserted into your text.

Should you give these commands in the wrong order, MicroEMACS will warn you that you are making a mistake. For example, if you open a keyboard macro by typing **<ctrl-X>(**, and then attempt to open another keyboard macro by again typing **<ctrl-X>(**, MicroEMACS will say:

```
Not now
```

Should you accidentally open a keyboard macro, or enter the wrong commands into it, you can cancel the entire macro simply by typing **<ctrl-G>**.

Replacing a macro

To replace this macro with another, go through the same process. Type **<ctrl-X>(**. Then type the *buffer status* command **<ctrl-X><ctrl-B>**, and type **<ctrl-X>)**. Remove the buffer status window by typing the *one window* command **<ctrl-X>1**.

Now execute your keyboard macro by typing the **execute macro** command **<ctrl-X>E**. The *buffer status* command has executed once more.

Note that whenever you exit from MicroEMACS, your keyboard macro is erased, and must be retyped when you return.

Search and replace

MicroEMACS also gives you a power function that allows you to search for a string and replace it with a keystroke. You can do this by executing the *search and replace* command **<esc>%**.

To see how this works, first move to the top of the text file by typing **<esc><**; then type type **<esc>%**. You will see the following message at the bottom of your screen:

```
Old string:
```

As an exercise, type **lawyer**. MicroEMACS will then ask:

```
New string:
```

Type **doctor**, and press the carriage return. As you can see, MicroEMACS jumps to the first occurrence of the string **lawyer**, and prints the following message at the bottom of your screen:

```
Query replace [lawyer] -> [doctor]
```

MicroEMACS is asking if it should procede with the replacement. Typing a carriage return will display bottom of your screen the options that are available to you, as follows:

```
<SP>[,] replace, [.] rep-end, [n] dont, [!] repl rest <C-G> quit
```

The options are as follows:

Typing a space or a comma will execute the replacement, and move the cursor to the next occurrence of the old string; in this case, it will replace **lawyer** with **doctor**, and move the cursor to the next occurrence of **lawyer**.

Typing a period '.' will replace this one occurrence of the old string, and end the search and replace procedure; in this example, typing a period will replace this one occurrence of **lawyer** with **doctor** and end the procedure.

Typing the letter 'n' tells MicroEMACS *not* to replace this instance of the old string, and move to the next occurrence of the old string; in this case, typing 'n' will *not* replace **lawyer** with **doctor**, and the cursor will jump to the next place where **lawyer** occurs.

Typing an exclamation point '!' tells MicroEMACS to replace all instances of the old string with the new string, without bothering to check with you any further. In this example, typing '!' will replace all instances of **lawyer** with **doctor** without further queries from MicroEMACS.

Finally, typing **<ctrl>-G** aborts the search and replace procedure.

15. Sending Commands to msh

The last commands you need to learn are the *program interrupt* commands **<ctrl-X>!** and **<ctrl-C>**. These commands allow you to interrupt your editing, give a command directly to **msh**, and then resume editing without affecting your text in any way.

The command **<ctrl-X>!** allows you to send *one* to the operating system. To see how this command works, type **<ctrl>!**. Note that the prompt **!** has appeared at the bottom of your screen. Type **ls**. Observe that the directory's table of contents scrolls across your screen, followed by the message **[end]**. To return to your editing, simply type a carriage return. The *interrupt* command **<ctrl-C>** suspends editing indefinitely, and allows you to send an unlimited number of commands to the operating system. To see how this works, type **<ctrl-C>**. After a moment, the **msh** prompt will appear at the bottom of your screen. Type **date**. **msh** will reply by printing the time and date. To resume editing, then simply type **exit**.

If you wish, you can suspend MicroEMACS's operation, tell **msh** to invoke another copy of the MicroEMACS program, edit a file, then return to your previous editing. To see how this is done, type **<ctrl-C>**. When the prompt appears at the bottom of your screen, type

```
me text1
```

It doesn't matter that you are already editing **text1**. MicroEMACS will simply copy the **text1** file into a new buffer and let you work as if the other MicroEMACS program you just interrupted never existed.

Exit from this second MicroEMACS program by typing the *quit* command **<ctrl-X><ctrl-C>**. Then type **exit**. Your original MicroEMACS program has now been resumed. Note, however, that none of the changes you made in the secondary MicroEMACS program will be seen here.

It is not a good idea to use multiple MicroEMACS programs to edit the same program: it is too easy to become confused as to which edits were made to which version.

The only time this is advisable, is if you wish to test to see how a certain edit would affect your text: you can create a new MicroEMACS program, test the command, and then destroy the altered buffer and return to your original editing program without having to worry that you might make errors that are difficult to correct.

Compiling and debugging through MicroEMACS

MicroEMACS can be used with the compilation command **cc** to give you a reliable system for debugging new programs.

Often, when you're writing a new program, you face the situation where you try to compile, only to have the compiler produce error messages and abort the compilation. You must then invoke your editor, change the program, close the editor, and try the compilation over again. This cycle of compilation—editing—recompilation can be quite bothersome.

To remove some of the drudgery from compiling, the **cc** command has the *automatic* or MicroEMACS option, **-A**. When you compile with this option, the MicroEMACS screen editor will be invoked automatically if any errors occur. The error or errors generated during compilation will be displayed in one window, and your text in the other, with the cursor set at the number of the line that the compiler indicated had the error.

Try the following example. Use MicroEMACS to enter the following program, which you should call **error.c**:

```
main() {
        printf("Hello, world")
}
```

Note that the semicolon was left off of the **printf** statement, which is an error. Now, try compiling **error.c** with the following **cc** command:

```
cc -A error.c
```

You should see no messages from the compiler, because they are all being diverted into a buffer to be used by MicroEMACS. Then, MicroEMACS will appear automatically. In one window you should see the message:

```
3: missing ';'
```

and in the other you should see your source code for **error.c**, with the cursor set on line 3.

If you had more than one, typing **<ctrl-X>>** would move you to the next line with an error in it; typing **<ctrl-X><** would return you to the previous error. Note that with some errors, such as those for missing braces or semicolons, the compiler cannot always tell exactly which line the error occurred on, but it will almost always point to a line that is near the source of the error.

Now, correct the error. Close the file by typing **<ctrl-Z>**. **cc** will be invoked again automatically. Compilation will continue either until the program compiles without error, or until you exit from MicroEMACS by typing **<ctrl-U>** followed by **<ctrl-X><ctrl-C>**.

16. Advanced Editing—Conclusion and Summary

This concludes the tutorial for the MicroEMACS interactive screen editor. Congratulations on your diligence in working through it! MicroEMACS and its related EMACS-based screen editors are now at your command, and you have acquired a skill that will serve you well.

This section introduced the advanced editing techniques available with MicroEMACS.

Arguments can be used with most cursor movement commands and all text deletion commands to set the number of times they execute. The command **<ctrl-U>** introduces arguments. The default for **<ctrl-U>** is 4, with each subsequent entry of <ctrl-U> multiplying the argument by 4. The value of an argument can be changed by typing **<ctrl-U>**, followed by an integer.

Text is edited in a buffer, and is copied into a file when the user issues the *save* commands **<ctrl-X><ctrl-S>** or the *write* command **<ctrl-X><ctrl-W>**. **<ctrl-X><ctrl-F>** will rename the file; and **<ctrl-X><ctrl-W>** renames the file and automatically copies text to it.

MicroEMACS can handle more than one buffer at a time. **<ctrl-X><ctrl-V>** moves you from one buffer to another, and allows you to create a buffer should the buffer you requested not already exist.

<ctrl-X><ctrl-R> replaces the text in a buffer with the text drawn from a specified file. **<ctrl-X>K** deletes a buffer.

<ctrl-X><ctrl-B> displays information on the status of each buffer.

The screen can be divided into windows, which can display either the same buffer or different buffers. **<ctrl-X>2** creates a window by dividing the present window in half, whereas the command **<ctrl-X>1** erases all extra windows.

<ctrl-X>Z and **<ctrl-X><ctrl-Z>** enlarge and shrink windows, respectively. **<ctrl-X>N** moves the cursor to the next, or lower, window, whereas **<ctrl-X>P** moves the cursor to the previous window.

<ctrl-X><ctrl-N>, **<ctrl-X><ctrl-P>**, and **<esc>!** respectively scroll the window up, scroll it down, and move the line on which the cursor rests to the top of the window. **<ctrl-X>B** displays a different buffer within a window.

MicroEMACS allows you to create a keyboard macro that executes a set of commands or inserts text. **<ctrl-X>(** opens the macro, **<ctrl-X>)** closes it, and **<ctrl-X>E** executes it.

<esc>% executes a search and replace procedure.

<ctrl-C> interrupts the operation of MicroEMACS, so that the user can send commands directly to **msh**. You can resume working with MicroEMACS by typing **<ctrl-D>**.

Index