

Mark Williams C

for the
Atari ST



Mark Williams Company

© 1986 - 1988 by Mark Williams Company.

This publication conveys information that is the property of Mark Williams Company. It shall not be copied, reproduced or duplicated in whole or in part without the express written permission of Mark Williams Company. Mark Williams Company makes no warranty of any kind with respect to this material and disclaims any implied warranties of merchantability or fitness for any particular purpose.

Mark Williams C, COHERENT, csd and Fast Forward are trademarks of Mark Williams Company. Let's C is a registered trademark of Mark Williams Company. Atari, ST and TOS are trademarks of Atari Corp.

Revision 5

Printing 5 4 3 2 1

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

Contents

1. Introduction	1
What is Mark Williams C?	1
Hardware requirements.	2
Changes from release 2.0	2
How to use this manual.	5
User registration and reaction report.	6
Technical support.	7
Bibliography.	7
Atari ST information.	8
2. Installing and Running Mark Williams C	11
Back up your disks!	11
Installing Mark Williams C	12
Using Mark Williams C with single-sided floppy disks	13
Using two double-sided floppy drives	14
Introducing the Mark Williams micro-shell	14
What is msh?	14
How to enter msh	14
Editing a file	15
Setting the shell's internal variables	16
Setting the environment.	17
Directories	18
Renaming, moving, copying, and removing files	19
Redirecting input and output.	20
Redirecting to peripheral devices	21
Logical devices.	21
File-name substitutions	22
Quoted strings.	23
Joining and separating commands.	24
The profile file.	24
Embedded commands	26
The .cmd directory	26

ii Mark Williams C for the Atari ST

Device-sensitive prompts	27
if command.	27
Parentheses	27
while command	28
equal and not	28
History command.	29
Three aliases.	29
The camefrom variable	30
The is_set command	30
For more information	31
3. Compiling with Mark Williams C.	33
The phases of compilation	33
Compiling from the GEM desktop.	34
Edit errors automatically	34
Renaming executable files	36
Floating-point numbers	36
Compiling multiple source files	36
Wildcards.	37
Linking without compiling	38
Compiling without linking	38
Assembly-language files	39
Changing the size of the stack	39
Debugging with Mark Williams C.	39
csd: the C Source Debugger.	40
db: symbolic debugger	40
od: formatted dump	41
nm: print symbol tables	41
Creating smaller, faster programs.	42
PC-relative addressing.	42
Strip.	42
Compiling with a RAM disk	43
Building a RAM disk.	43
Working with a RAM disk	45
Where to go from here	46
4. Introduction to MicroEMACS	47
What is MicroEMACS?	47
Keystrokes — <ctrl>, <esc>	48
Becoming acquainted with MicroEMACS	48
Beginning a document.	49
Moving the Cursor	51
Moving the cursor forward	51
Moving the cursor backward	51
From line to line	52

Moving up and down by a screenful of text	52
Moving to beginning or end of text	53
Saving text and quitting	53
Killing and deleting	53
Deleting versus killing	54
Erasing text to the right.	54
Erasing text to the left.	55
Erasing lines of text	55
Yanking back (restoring) text.	56
Quitting.	56
Block killing and moving text	56
Moving one line of text	56
Multiple copying of killed text	57
Kill and move a block of text	57
Capitalization and other tools	58
Capitalization and lowercasing	58
Transpose characters.	59
Screen redraw	59
Return indent	60
Word wrap	60
Search and Reverse Search.	62
Search forward	62
Reverse search.	63
Cancel a command	64
Search and replace	64
Saving text and exiting	65
Write text to a new file	66
Save text and exit.	66
Advanced editing	67
Arguments.	68
Default values	68
Selecting values	68
Deleting with arguments — an exception	69
Buffers and files	69
Definitions	70
File and buffer commands	70
Write and rename commands	70
Replace text in a buffer	71
Visiting another buffer.	71
Move text from one buffer to another	72
Checking buffer status.	73
Renaming a buffer	73
Delete a buffer.	74
Windows	74
Creating windows and moving between them.	75

iv Mark Williams C for the Atari ST

Enlarging and shrinking windows	76
Displaying text within a window	77
One buffer	78
Multiple buffers	78
Moving and copying text among buffers	79
Checking buffer status	79
Saving text from windows	79
Keyboard macros	80
Keyboard macro commands	80
Replacing a macro	81
Sending commands to TOS	81
Compiling and debugging through MicroEMACS	82
The MicroEMACS help facility	83
Where to go from here	84
5. make Programming Discipline	85
How does make work?	85
Try make	86
Essential make	88
The makefile	88
Building a simple makefile	89
Comments and macros	90
Setting the time	91
Building a large program	91
Command line options	92
Other command line features	93
Advanced make	94
Default rules	94
Double-colon target lines	95
Alternative uses	96
Special targets	98
Errors	98
Exit status	98
Where to go from here	98
6. Introduction to the Resource Editor	99
How resource works	99
Planning your resource	100
Designing an interface	100
Buttons and radio buttons	100
Text input	101
Icons	101
Images	101
Menus	101
Getting started	101

The resource desktop	102
The resource menu bar	103
File operations	108
Display, copy, rename, and delete	108
Loading and saving	109
Moving and copying trees and objects	109
Trees	110
Forms	110
Editing forms	110
MENU	113
Editing menus	113
String	114
Alert	114
Image	114
Objects	116
New objects	116
Icons and images	116
Ibox	118
Button	118
Text	119
Editing objects	119
Manipulating objects	121
The Control key	121
Moving an object	122
Resizing	122
Copying	122
Deleting	123
Other functions on objects	123
Where to go from here	124
7. Resource Compiler and Decompiler	125
Using the compiler and decompiler	125
Language description	126
Tree and object descriptions	126
Trees	127
Objects	127
Resource description elements	129
Sample resource description	133
Resource description grammar	133
8. Error Messages	141
9. The Lexicon	171
example	Give an example of Mark Williams Lexicon format 172
abort	End program immediately 173

abs	Return the absolute value of an integer	173
access	Check if a file can be accessed in a given mode	174
access.h	Define manifest constants used by access()	175
acos	Calculate inverse cosine	176
address		177
AES		177
aesbind.h	Declare GEM AES routines	181
alignment		181
appl_exit	Exit from an application	182
appl_find	Get another application's handle	182
appl_init	Initiate an application	182
appl_read	Read a message from another application	183
appl_tplay	Replay AES activity	183
appl_trecord	Record user actions	183
appl_write	Send a message to another application	184
ar	The librarian/archiver	185
arena		186
argc	Argument passed to main	187
argv	Argument passed to main	187
array		188
as	Assembler for Atari ST	189
as68toas	Convert Motorola assembler	205
ASCII		206
asctime	Convert time structure to ASCII string	209
asin	Calculate inverse sine	210
assert	Check assertion at run time	210
#assert	Check assertion at compile time	211
assert.h	Define assert()	211
atan	Calculate inverse tangent	211
atan2	Calculate inverse tangent	212
atof	Convert ASCII strings to floating point	212
atoi	Convert ASCII strings to integers	213
atol	Convert ASCII strings to long integers	213
auto	Note an automatic variable	214
auto		214
aux	Logical device for serial port	216
backspace		218
basepage.h	Define TOS base page structure	218
Bconin	Receive a character	219
Bconout	Send a character to a peripheral device	220
Bconstat	Return the input status of a peripheral device	220
Bcostat	Read the output status of a peripheral device	221
BIOS		222
bios	Call an input/output routine in the TOS BIOS	222
bios.h	Declare bios constants and structures	223

Bioskeys	Reset the keyboard to its default	223
bit		223
bit map		224
Blitmode	Get/set blitter configuration	224
bombs	68000 processor exceptions	225
boot		226
break	Exit from loop or switch statement	226
buffer		227
byte		228
byte ordering		228
C keywords		230
C language		230
cabs	Complex absolute value function	234
calling conventions		234
calloc	Allocate dynamic memory	238
canon.h	Canonical conversion for the 68000	239
carriage return		239
case	Introduce entry in switch statement	240
cast		240
cat	Concatenate files	241
Cauxin	Read a character from the serial port	241
Cauxis	Check if characters are waiting at serial port	242
Cauxos	Check if serial port is ready to receive characters	243
Cauxout	Write a char to the serial port	243
cc	Compiler controller	243
cc0		249
cc1		249
cc2		249
cc3		249
Cconin	Read a character from the standard input	250
Cconis	Find if a character is waiting at standard input	250
Cconos	Check if console is ready to receive characters	251
Cconout	Write a character onto standard output	252
Cconrs	Read and edit a string from the standard input	252
Cconws	Write a string onto standard output	253
cd	Change directory	253
ceil	Set numeric ceiling	254
char	Data type	255
character constant		255
chdir	Change working directory	256
chmod	Change file protection modes	256
chmod	Change the modes of a file	257
chown	Change ownership of a file	257
clearerr	Present stream status	257
CLK_TCK		258

clock	Get number of clock ticks since system boot	258
close	Close a file	258
cmp	Compare bytes of two files	259
Cnecin	Perform modified raw input from standard input	259
commands		260
compound number		262
con	Logical device for the console	263
const	Qualify an identifier as not modifiable	263
continue	Force next iteration of a loop	263
cos	Calculate cosine	264
cosh	Calculate hyperbolic cosine	264
cp	Copy a file	265
cpp	C preprocessor	265
Cprnos	Check if printer is ready to receive characters	267
Cprnout	Send a character to the printer port	267
Crawcin	Read a raw character from standard input	268
Crawio	Perform raw I/O with the standard input	268
creat	Create/truncate a file	269
crtso.o	Default C runtime startup	269
crtso.o	C runtime startup, GEM environment	270
crtsg.o	C runtime startup, GEM environment	270
ctime	Convert system time to an ASCII string	271
ctype		271
ctype.h	Header file for data tests	273
curconf	Set the cursor's configuration	273
Curconf	Get or set the cursor's configuration	274
daemon		276
data formats		276
data types		276
date	Print/set the date and time	277
dayspermonth	Return number of days in a given month	278
db	Assembler-level symbolic debugger	278
Dcreate	Create a directory	288
Ddelete	Delete a directory	289
declarations		290
default	Default label in switch statement	291
#define	Define a variable as manifest constant	291
desk accessory		292
df	Measure free space on disk	296
Dfree	Get information on a drive's free space	297
Dgetdrv	Find current default disk drive	298
Dgetpath	Get the current directory name	298
diff	Summarize differences between two files	299
difftime	Return difference between two times	300
directory		300

do	Introduce a loop	300
Dosound	Start up the sound daemon	301
double	Data type	303
drtomw	Convert from DRI to Mark Williams format	303
Drvmap	Get a map of the logical disk drives	304
drvprs	Check if a drive is present on the machine	304
Dsetdrv	Make a drive the current drive	305
Dsetpath	Set the current directory	306
dup	Duplicate a file descriptor	307
dup2	Duplicate a file descriptor	308
echo	Repeat/expand an argument	309
ecvt	Convert floating-point numbers to strings	309
edata		310
egrep	Extended pattern search	310
#elif	Include code conditionally	312
else	Introduce a conditional statement	313
#else	Include code conditionally	313
end		314
__end		314
#endif	End conditional inclusion of code	314
entry	Undefined keyword	315
enum	Declare a type and identifiers	315
environ		316
environment		316
envp	Argument passed to main	317
EOF		317
equal	Compare two arguments	318
errno	External integer for return of error status	318
errno.h	Error numbers used by <code>errno()</code>	319
error codes		319
etext		320
evnt_button	Await a specific mouse button event	320
evnt_dclick	Get/set double-click interval	321
evnt_keybd	Await a keyboard event	321
evnt_mesag	Await a message	322
evnt_mouse	Wait for mouse to enter specified rectangle	324
evnt_multi	Await one or more specified events	325
evnt_timer	Wait for a specified length of time	328
executable file		328
execve	Execute a command from within a program	328
exit	Terminate a program	329
exit	Exit from a msh shell	329
_exit	Terminate a program	329
exp	Compute exponent	330
extern	Declare storage class	331

fabs	Compute absolute value.	332
Fattrib	Get and set file attributes	332
Fclose	Close a file.	333
fclose	Close stream	333
Fcreate	Create a file	334
fcvt	Convert floating point numbers to ASCII strings	336
Fdatetime	Get or set a file's date/time stamp	337
Fdelete	Delete a file	338
fdopen	Open a stream for standard I/O.	338
Fdup	Generate a substitute file handle	340
feof	Discover stream status	340
ferror	Discover stream status	340
fflush	Flush output stream's buffer.	341
Fforce	Force a file handle	342
fgetc	Read character from stream	342
Fgetdta	Get a disk transfer address.	343
fgets	Read line from stream.	345
fgetw	Read integer from stream	346
field		347
file		347
file	Name a file's type	347
FILE	Descriptor for a file stream.	348
file descriptor		349
fileno	Get file descriptor	349
flexible arrays		350
float	Data type	350
floor	Set a numeric floor	353
Flopfmt	Format tracks on a floppy disk	353
Floprd	Read sectors on a floppy disk	356
Flopver	Verify a floppy disk	357
Flopwr	Write sectors on a floppy disk	358
fopen	Open a stream for standard I/O.	358
Fopen	Open a file.	360
for	Control a loop.	360
form_alert	Display an alert box	361
form_center	Center an object on the screen	362
form_dial	Reserve/free screen space for dialogue.	362
form_do	Handle user input in form dialogue	363
form_error	Display a TOS error.	364
fprintf	Print formatted output onto file stream	365
fputc	Write character onto file stream.	365
fputs	Write string to file stream	366
fputw	Write an integer to a stream.	366
fraction		367
fread	Read data from file stream.	367

Fread	Read a file	367
free	Return dynamic memory to free memory pool	368
Frename	Rename a file	368
freopen	Open file stream for standard I/O	369
frexp	Separate fraction and exponent	370
fscanf	Format input from a file stream	371
fseek	Seek on file stream	372
Fseek	Move a file pointer	373
fselect	Select a file	375
Fsetdta	Set disk transfer address	378
Fsfirst	Search for first occurrence of a file	378
Fsnext	Search for next occurrence of file name	379
fstat	Find file attributes	379
ftell	Return current position of file pointer	380
function		380
fwrite	Write onto file stream	381
Fwrite	Write into a file	381
galaxy.a		382
gcvt	Convert floating point number to ASCII string	382
gem	Run a GEM program	382
gemdefs.h	GEM structures and definitions	383
gemdos	Call a routine from GEM-DOS	383
gemout.h	GEM-DOS file formats and magic numbers	385
Getbpb	Get pointer to BIOS parameter block for a disk drive	385
getc	Read character from file stream	386
getchar	Read character from standard input	387
getcol	Get a color value	387
getenv	Read environmental variable	388
Getmpb	Copy memory parameter block	388
getpal	Get the color palette settings	389
getphys	Get the base of the physical screen's display	390
getrez	Get screen's current resolution	390
Getrez	Read the current screen resolution	390
gets	Read string from standard input	391
Getshift	Get or set the status flag for shift/alt/control keys	392
Gettime	Read the current time	393
getw	Read word from file stream	394
Giaccess	Access a register on the GI sound chip	394
GMT		396
gmtime	Convert system time to calendar structure	397
goto	Unconditionally jump within a function	397
graf_dragbox	Draw a draggable box	398
graf_growbox	Draw a growing box	399
graf_handle	Get a VDI handle	400
graf_mbox	Move a box	400

graf_mkstate	Get the current mouse state	401
graf_mouse	Change the shape of the mouse pointer	401
graf_rubbox	Draw a rubber box	403
graf_shrinkbox	Draw a shrinking box	403
graf_slidebox	Track the slider within a box	404
graf_watchbox	Draw a watched box	406
handle		408
header file		408
help	Print concise description of command	408
hidemouse	Hide the mouse pointer	409
HOME		409
horizontal tab		409
htom	Redraw screen from high to medium resolution	410
hypot	Compute hypotenuse of right triangle	410
if	Execute a command conditionally	411
if	Introduce a conditional statement	411
#if	Include code conditionally	411
#ifdef	Include code conditionally	412
#ifndef	Include code conditionally	413
Ikbdws	Write a string to the intelligent keyboard device	413
INCDIR		414
#include	Copy a header file into a program	414
index	Find a character in a string	415
inherit	Pass variable to child shell	415
Initmous	Initialize the mouse	415
int	Data type	416
interrupt		416
lore	Set the I/O record	417
is_set	Check if an environmental variable is set	418
isalnum	Check if a character is a number or letter	418
isalpha	Check if a character is a letter	419
isascii	Check if a character is an ASCII character	419
isatty	Check if a device is a terminal	419
iscntrl	Check if a character is a control character	420
isdigit	Check if a character is a numeral	420
isleapyear	Indicate if a year was a leap year	420
islower	Check if a character is a lower-case letter	420
isprint	Check if a character is printable	421
ispunct	Check if a character is a punctuation mark	421
isspace	Check if a character prints white space	421
isupper	Check if a character is an upper-case letter	422
j0	Compute Bessel function	423
j1	Compute Bessel function	424
jday_to_time	Convert Julian date to system time	424
jday_to_tm	Convert Julian date to system calendar format	424

Jdisint	Disable interrupt on multi-function peripheral device . .	425
Jenabint	Enable a multi-function peripheral port interrupt. . .	425
jn	Compute Bessel function	425
Kbdvbase	Return a pointer to the keyboard vectors	427
kbrate	Reset the keyboard's repeat rate	429
Kbrate	Get or set the keyboard's repeat rate.	429
keyboard		430
Keytbl	Set the keyboard's translation table	431
Kgettime	Read time from intelligent keyboard's clock	432
kick	Force TOS to reread the disk cache	433
Ksettime	Set time in intelligent keyboard's clock	433
lc	List directory's contents in columnar format	434
lalloc	Allocate dynamic memory	434
ld	Link relocatable object files.	434
ldexp	Combine fraction and exponent	437
Lexicon		437
libaes	GEM AES bindings	439
libc		439
libm		440
LIBPATH	Directories that hold libraries	440
library		440
libvdi	GEM VDI bindings.	440
#line	Reset line numbering	441
Line A		441
linea.h	Declare Atari line A routines	445
line feed		445
lmalloc	Allocate dynamic memory	446
localtime	Convert system time to calendar structure	446
log	Compute natural logarithm	448
log10	Compute common logarithm.	449
Logbase	Read the logical screen's display base	449
long	Data type	450
longjmp	Return from a non-local goto	450
lrealloc	Reallocate dynamic memory	451
ls	List directory's contents.	451
lseek	Set read/write position	452
ltom	Redraw the screen from low to medium resolution . .	453
lvalue		453
macro		455
main	Introduce program's main function.	455
make	Program building discipline	455
malloc	Allocate dynamic memory	459
Malloc	Allocate dynamic memory	461
manifest constant		462
mantissa		462

math.h	Declare mathematics functions	462
mathematics library		462
maxmem		463
me	MicroEMACS screen editor	463
me.a		471
Mediach	Check whether disk has been changed	471
memchr	Search a region of memory for a character	472
memcmp	Compare two regions	472
memcpy	Copy one region of memory into another	473
memory allocation		473
memset	Fill an area with a character	476
menu		476
menu_bar	Show or erase the menu bar	481
menu_ichk	Write or erase a check mark next to a menu item	481
menu_ienable	Enable or disable a menu item	481
menu_register	Add a name to the desk accessory menu list	482
menu_text	Replace text of a menu item	482
menu_tnormal	Display menu title in normal or reverse video	483
metafile		483
mf	Measure space left in RAM	486
Mfpint	Initialize the MFP interrupt	486
Mfree	Free allocated memory	487
Midiws	Write a string to the MIDI port	488
mkdir	Create a directory	490
mktemp	Generate a temporary file name	490
modf	Separate integral part and fraction	490
modulus		491
mousehidden	Return how often mouse pointer has been hidden	492
msh		492
Mshrink	Shrink amount of allocated memory	500
mshversion	Print current version of msh	500
msleep	Stop executing for a specified time	500
mtoh	Redraw the screen from medium to high resolution	501
mtol	Redraw the screen from medium to low resolution	501
mtype.h	List processor code numbers	501
mv	Rename files or directories	501
mwtomw	Convert objects to 3.0 format	502
nested comments		503
newline		503
nm	Print a program's symbol table	503
not	Invert logical value of an argument	504
notmem	Check if memory is allocated	504
n.out		505
nout.h	Describe output format n.out	505
NUL		506

NULL		506
nybble		506
obdefs.h	Declare TOS objects and structures	507
objc_add	Redefine a child object within an object tree	507
objc_change	Change object's state	507
objc_delete	Delete an object from an object tree	508
objc_draw	Draw an object	508
objc_edit	Edit a text object	509
objc_find	Find if mouse pointer is over particular object	509
objc_offset	Calculate an object's absolute screen position	510
objc_order	Reorder a child object within the object tree	510
object		511
object format		520
od	Print a hexadecimal dump of a file	520
Offgibit	Clear a bit in the sound chip's A port	521
Ongibit	Turn on a bit in the sound chip's A port	521
open	Open a file	522
operator		523
osbind.h	Declare TOS functions	525
path		526
path	Build a path name for a file	526
path.h	Declare path()	527
PATH	Directories that hold executable files	527
pattern		528
peekb	Extract a byte from memory	528
peekl	Extract a long from memory	528
peekw	Extract a word from memory	529
perror	System call error messages	529
Pexec	Load or execute a process	530
Physbase	Read the physical screen's display base	531
picture	Format numbers under mask	533
pnmatch	Match string pattern	534
pointer		535
pokeb	Insert a byte into memory	536
pokel	Insert a long into memory	536
pokew	Insert a long into memory	537
port		537
portability		537
pow	Compute a power of a number	538
pr	Paginate and print files	538
precedence		539
printf	Format output	540
prn:	TOS logical device for parallel port	544
process		544
Protobt	Generate a prototype boot sector	544

Prtblk	Print a dump of the screen	545
Pterm	Terminate a process	547
Pterm0	Terminate a TOS process	547
Ptermres	Terminate a process but keep it in memory	547
pun	548
Puntaes	Disable AES	548
putc	Write character to stream	548
putchar	Write a character to standard output	549
puts	Write string to standard output	550
putw	Write word to stream	550
pwd	Print the name of the current directory	550
qsort	Sort arrays in memory	552
rand	Generate pseudo-random numbers	553
Random	Generate a 24-bit pseudo-random number	553
random access	554
ranlib	554
rational number	555
rc_copy	Copy a rectangle	555
rc_equal	Compare two rectangles	556
rc_intersect	Check if two rectangles intersect	556
rc_union	Calculate overlap between two rectangles	557
rdy	Create, save, and load rebootable RAM disk	557
rdy.a	565
read	Read from a file	566
readonly	Storage class	566
read-only memory	566
realloc	Reallocate dynamic memory	567
real number	567
record	567
register	Storage class	567
register	568
register variable	568
rescomp	Resource compiler	568
resdecom	Resource decompiler	569
resource	Invoke the resource editor	570
return	Return a value and control to calling function	571
rewind	Reset file pointer	571
rindex	Find a character in a string	571
rm	Remove files	572
rmdir	Remove directories	572
Rsconf	Configure the serial port	573
rsconf	Configure the serial port	575
rsrc_free	Free memory allocated to a set of resources	576
rsrc_gaddr	Get the address of a resource object	576
rsrc_load	Load a resource file into memory	577

rsrc_obfix	Change the form of an object's coordinates	577
rsrc_saddr	Store address of a free string or a bit image	578
runtime startup		578
rvalue		579
Rwabs	Read or write data on a disk drive	579
sbrk	Increase a program's data space	580
scanf	Accept and format input	580
Scrdmp	Print a dump of the screen	582
screen control		583
scrp_read	Read the scrap directory	584
scrp_write	Write to the scrap directory	585
set	Set a msh variable	585
setbuf	Set alternative stream buffers	586
setcol	Reset a color	586
SetColor	Set one color	586
setenv	Set an environmental variable	587
Setexc	Get or set an exception vector	588
setjmp	Perform non-local goto	589
setjmp.h	Define setjmp() and longjmp()	589
setpal	Reset the color palette	590
Setpalette	Set the screen's color palette	590
setphys	Reset physical screen's display space	591
setprt	Reset the printer port	591
Setprt	Get or set the printer's configuration	591
setrez	Reset the screen resolution	592
Setscreen	Set the video parameters	592
Settime	Set the current time	593
Sgettime	Read time from intelligent keyboard's clock	596
shLenvrn	Search for an environmental variable	596
shLfind	Search PATH for file name	598
shLread	Let an application identify the program that called it	598
shLwrite	Tell desktop which application to run next	598
shellsort	Sort arrays in memory	599
short	Data type	600
show	Display a stored screen image	600
showmouse	Redisplay the mouse pointer	601
signal.h	Define Atari ST signals	601
sin	Calculate sine	601
sinh	Calculate hyperbolic sine	601
size	Print the size of an object module	602
sizeof	Return size of a data element	602
sleep	Stop executing for a specified time	603
snap	Save a screen image	603
sort	Sort lines of text	604
sprintf	Format output	605

sqrt	Compute square root	605
srand	Seed random number generator	606
sscanf	Format input	606
stack		607
standard error		608
standard input		608
standard output		608
stat	Find file attributes	609
stat.h	Definitions and declarations used to obtain file status	610
static	Declare storage class	610
stderr		610
stdin		610
STDIO		611
stdio.h	Declarations and definitions for I/O	612
stdout		612
time	Set the operating system time	612
_stksize		613
storage class		614
strcat	Append one string to another	614
strchr	Find a character in a string	614
strcmp	Compare two strings	615
strcpy	Copy one string into another	615
strcspn	Length one string excludes characters in another	615
stream		616
strerror	Translate an error number into a string	616
string		617
strip	Strip tables from executable file	619
strlen	Measure the length of a string	619
strncat	Append one string onto another	619
strncmp	Compare two strings	620
strncpy	Copy one string into another	620
strpbrk	Find first occurrence of any character	622
strrchr	Search for rightmost occurrence of a character	622
strspn		623
strstr	Find one string within another	623
struct	Data type	624
structure		624
structure assignment		624
SUFF		625
Super	Enter privilege mode	625
Supexec	Run a function under supervisor mode	626
Sversion	Get the version number of TOS	628
swab	Swap a pair of bytes	629
switch	Test a variable against a table	629
system	Pass a command to TOS for execution	630

system variables	632
tail	Print the end of a file 636
tan	Calculate tangent. 636
tanh	Calculate hyperbolic cosine. 636
tempnam	Generate a unique name for a temporary file 637
tetd_to_tm	Convert IKBD time to system calendar format. 637
Tgetdate	Get the current date. 638
Tgettime	Get the current time. 639
Tickcal	Return system timer's calibration. 640
time	Time the execution of a command 640
time	Get current time 641
time	Print current time/time execution of a command 641
time.h	Give time-description structure 646
time_to_jday	Convert system time to Julian date 646
TIMEZONE	Time zone information 646
tm_to_jday	Convert calendar format to Julian time 648
tm_to_tetd	Convert system calendar format to IKBD time. 649
TMPDIR	649
tmpnam	Generate a unique name for a temporary file 649
toascii	Convert characters to ASCII 650
tolower	Convert characters to lower case 650
_tolower	Convert letter to lower case 651
tos	Execute GEM-DOS program. 652
TOS	652
touch	Update modification time of a file. 655
toupper	Convert characters to upper case 655
_toupper	Convert letter to upper case 655
Tsetdate	Set a new date 656
Tsettime	Set a new time 658
type checking	658
typedef	Define a new data type 658
type promotion	659
#undef	Undefine a manifest constant 660
ungetc	Return character to input stream. 660
union	Multiply declare a variable 661
uniq	Remove/count repeated lines in a sorted file 662
UNIX routines	662
unlink	Remove a file 663
unset	Discard a shell variable 664
unsetenv	Discard an environmental variable 664
unsigned	Data type 664
v_arc	Draw a circular arc 665
v_bar	Draw a rectangle. 665
v_bit_image	Print a bit image file. 668

v_cellarray	Draw a table of colored cells	669
v_circle	Draw a circle	669
v_clear_disp_list	Clear a printer's display list	672
v_clrwk	Clear the virtual workstation	672
v_clsvwk	Close the screen virtual device.	673
v_clswk	Close a virtual workstation.	673
v_contourfill	Fill an outlined area	674
v_curdown	Move text cursor down one row.	677
v_curhome	Move text cursor to the home position.	677
v_curleft	Move text cursor left one column.	677
v_curreight	Move text cursor right one column.	678
v_curtext	Write alphabetic text	678
v_curup	Move text cursor up one row	678
v_dspcur	Move mouse pointer to point on screen	679
v_eeol	Erase text from cursor to end of screen	679
v_eeos	Erase from text cursor to end of screen	679
v_ellarc	Draw an elliptical arc	680
v_ellipse	Draw an ellipse.	683
v_ellpie	Draw an elliptical pie slice	685
v_enter_cur	Enter text mode	686
v_exit_cur	Exit from text mode	688
v_fillarea	Draw a complex polygon	689
v_form_adv	Advance the page on a printer.	692
v_get_pixel	See if a given pixel is set	692
v_gtext	Draw graphics text.	692
v_hardcopy	Write the screen to a hard-copy device.	695
v_hide_c	Hide the mouse pointer.	696
v_justified	Justify graphics text.	696
v_meta_extents	Update extents header of metafile	697
v_opnvwk	Open the virtual screen device.	697
v_opnwk	Open a virtual workstation.	698
v_output_window	Dump a portion of a virtual device to a printer	702
v_pieslice	Draw a circular pie slice	702
v_pline	Draw a line	702
v_pmarker	Draw a marker	704
v_rbox	Draw a rounded rectangle	705
v_rfbox	Draw a filled, rounded rectangle	707
v_rmcur	Remove last mouse pointer from the screen	707
v_rvoff	End reverse video for alphabetic text.	708
v_rvon	Display alphabetic text in reverse video	708
v_show_c	Show the mouse cursor.	708
v_updwk	Update a virtual workstation.	709
v_write_meta	Write a metafile item	709
VDI	710
vdibind.h	Declarations for VDI routines	718

version	Print/create a version string.	718
vertical tab.		719
vex_butv.	Set new button interrupt routine.	719
vex_cuv.	Set new cursor interrupt routine.	720
vex_motv.	Set new mouse movement interrupt routine.	720
vex_timv.	Set new timer interrupt routine.	721
vm_filename.	Rename a metafile.	721
void.	Data type	722
volatile	Qualify an identifier as frequently changing	722
vq_cellarray	Return information about cell arrays.	723
vq_chcells.	Find how many characters virtual device can print.	724
vq_color.	Check/set color intensity	725
vq_curaddress.	Get the text cursor's current position	725
vq_extnd.	Perform extend inquire of VDI virtual device.	725
vq_key_s.	Check control key status	726
vq_mouse.	Check mouse position and button state	727
vq_tabstatus.	Find if graphics tablet is available	727
vqf_attributes.	Read the area fill's current attributes	727
vqin_mode.	Determine mode of a logical input device	728
vql_attributes.	Read the polyline's current attributes	728
vqm_attributes.	Read the marker's current attributes	729
vqp_error.	Inquire if an error occurred with the Polaroid Palette	730
vqp_films.	Get films supported by driver for Polaroid Palette.	730
vqp_state.	Read current settings of the Polaroid Palette driver.	731
vqt_attributes.	Read the graphic text's current attributes.	731
vqt_extent.	Calculate a string's length	732
vqt_fontinfo.	Get information about special effects for graphics text	733
vqt_name.	Get name and description of graphics text font	734
vqt_width.	Get character cell width.	735
vr_recfl.	Draw a rectangular fill area	735
vr_trnfm.	Transform a raster image.	737
vro_cpyfm.	Copy raster form, opaque.	738
vrq_choice.	Return status of function keys when any key is pressed	743
vrq_locator.	Find location of mouse cursor when a key is pressed.	743
vrq_string.	Read a string from the keyboard	744
vrq_valuator.	Return status of shift and cursor keys	745
vrt_cpyfm.	Copy raster form, transparent.	745
vs_clip.	Set the virtual device's clipping rectangle	747
vs_color.	Set color intensity	748
vs_curaddress.	Move text cursor to specified row and column	748
vs_palette.	Select color palette on medium-resolution screen	749
vsc_form.	Draw a new shape for the mouse pointer	749
vsf_color.	Set a polygon's fill color.	750
vsf_interior.	Set a polygon's fill type	750
vsf_perimeter.	Set whether to draw a perimeter around a polygon.	750

vsf_style	Set a polygon's fill style	751
vsf_udpat	Define a fill pattern	752
vsin_mode	Set input mode for logical input device	752
vsl_color	Set a line's color	753
vsl_ends	Attach ends to a line.	753
vsl_type	Set a line's type.	754
vsl_udsty	Set user-defined line type.	754
vsl_width	Set a line's width.	755
vsm_choice	Return last function key pressed	755
vsm_color	Set a polymarker's color	756
vsm_height	Set a polymarker's height	756
vsm_locator	Return mouse pointer's position	757
vsm_string	Read a string from the keyboard	757
vsm_type	Set polymarker's type	758
vsm_valuator	Return shift/cursor key status.	759
vsp_message	Suppress messages from Polaroid Palette device.	760
vsp_save	Save to disk current setting of Polaroid Palette driver	760
vsp_state	Set the Polaroid Palette driver.	760
vst_alignment	Realign graphics text	761
vst_color	Set color for graphics text	762
vst_effects	Set special effects for graphics text	762
vst_font	Select a new font.	763
vst_height	Reset graphics text height, in absolute values	763
vst_load_fonts	Load fonts other than the standard font.	764
vst_point	Reset graphics text height, in printer's points	765
vst_rotation	Set angle at which graphic text is drawn	765
vst_unload_fonts	Unload fonts.	766
vswr_mode	Set the writing mode	766
Vsync	Synchronize with the screen.	767
wc	Count words, lines, and characters in files	768
while	Introduce a loop	768
while	Execute a conditional loop	768
wildcards	769
wind_calc	Calculate a window's rectangle	769
wind_close	Close a window and preserve its handle.	770
wind_create	Create a window	770
wind_delete	Delete a window and free its resources	771
wind_find	Determine if the mouse pointer is in a window	772
wind_get	Get information about a window	772
wind_open	Open or reopen a window	773
wind_set	Set specified fields within the window	774
wind_update	Lock or unlock a window.	775
window	776
write	Write to a file.	784
xbios	Call a routine from the extended TOS BIOS	786

xbios.h	Declare xbios constants and structures	787
Xbtimer	Initialize the MFP timer	787
XOFF		788
XON		789
Permuted List of Lexicon Entries		791
Index		811

Section 1:

Introduction

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

Mark Williams C is part of the Mark Williams Company family of C compilers, which supports many different operating systems and processors. The operating systems supported include:

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

The processors supported include:

PDP-11	68000	80186
Z8001	68020	80286
Z8002	8086	

What is Mark Williams C?

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

- The Mark Williams C compiler, plus a linker, an assembler, a preprocessor, and other tools.

2 Mark Williams C for the Atari ST

- A set of commands selected from the COHERENT operating system, including the MicroEMACS screen editor and the **make** programming discipline.
- A full set of libraries, including the standard C library, mathematics library, plus libraries that implement the Atari AES, VDI, and Line A routines.
- A set of sample programs, including full source code for the MicroEMACS editor.
- The Mark Williams micro-shell **msh**, a command processor designed to control the operation of the compiler and its commands.
- A full toolkit for building and maintaining GEM resources. These include **resource**, the Mark Williams resource editor; **rescomp**, a resource compiler; and **resdecom**, a resource decompiler.

Hardware requirements

Mark Williams C runs on any Atari ST or Mega ST, with any configuration of disk drives. It is recommended that a 520 ST have at least two single-sided disk drives or one double-sided disk drive.

Changes from release 2.0

Release 3.0 differs from release 2.0 on the following points:

- Mark Williams C now allows you to create static arrays that are larger than 64 kilobytes.
- Mark Williams C now can generate modules that can be debugged with the Mark Williams C source debugger **csd**. **csd** brings full-featured C source debugging to the Atari ST. It works with programs that access the GEM AES and VDI, as well as with traditional, text-oriented programs. **csd** lets you walk through your source code and observe how it executes step by step. You set breakpoints and traps, evaluate expressions that you type in during program execution, and single-step through your program to help you find bugs.

For more information about **csd**, contact Mark Williams Company or your local software dealer.

- The symbolic debugger **db** has been improved to work through the **aux** port, and to work with GEM programs. You can plug a terminal into the **aux** port and use it to give commands to **db**; the program's output is shown on the ST monitor. This allows you to debug programs that use AES or VDI calls. **db** can be used to debug programs that are assembled by **as**, the Mark Williams assembler, as well as those compiled from C source.

db has a new switch, **-t**, which causes **stdout**, **stderr**, and **stdin** to go to the console regardless of redirection on the command line or in the shell.

db now supports symbol tables larger than 64 kilobytes.

- To support **csd**, Mark Williams C now stores debug information in the GEMDOS symbol segment instead of following the relocation stream. Utilities that access this information (**file**, **strip**, **size**, **nm**, **ld**, **db**) will no longer accept executable programs compiled by Mark Williams C versions 2.1.7 or earlier. A new utility, **mwtomw**, is included to convert executable programs from the old Mark Williams format into the new format.
- Mark Williams C now includes a full set of resource tools: **resource**, a full-featured resource editor; **rescomp**, a resource compiler; and **resdecom**, a resource disassembler.

resource is a screen-oriented resource editor. With it, you can build GEM menus, dialogues, and icons easily. For each resource it creates, **resource** generates a header file that you can use with your C program.

resdecom is a disassembler that translates a resource into a file of descriptive text. This text can be checked and edited by hand, then reassembled with the resource compiler **rescomp**. The resource compiler can also compile resources that you write by hand.

- The MicroEMACS screen editor now has an on-line help feature to assist with C programming. Its help file includes the synopsis and binding for every library function and macro included with Mark Williams C. To invoke the help feature, type **<ctrl-X>?** and type the name of the function or macro for which you need information, or move the cursor over the function or macro in your program and type **<esc>?**. In a moment, a help window will open on your screen; it will contain a synopsis of the function or macro and its binding. If you wish, you can copy information from the help window into your program. To erase the help window, type **<esc>2**.
- The compiler can now compile programs to use PC-relative addressing. PC-relative addressing is faster than the absolute addressing that Mark Williams C uses by default. This can only be used when the program has no global references that are greater than 32 kilobytes away from where they are referenced. For many programs and utilities, however, this is not a problem; for them, PC-relative addressing creates an executable that is noticeably smaller and faster than one that uses absolute addressing.

To use PC-relative addressing in your program, use the option **-VSMALL** on the **cc** command line.

For programs whose code size is small but use large amounts of *static* or global data (for example, MicroEMACS), use the option **-VCOMPAC**. This uses PC-relative addressing for the code references and absolute addressing to handle the data.

4 Mark Williams C for the Atari ST

Pointers are not affected by the **-VSMALL** or **-VCOMPAC** options. You can link modules compiled **-VSMALL** or **-VCOMPAC** with modules that are compiled into the default format of absolute addressing.

- The compiler now includes an optional peephole optimizer, for further optimization of your programs. To invoke the peephole optimizer, use the option **-VPEEP** on the **cc** command line.
- Many of the utilities have been compiled with the **-VSMALL** option, to make them smaller and faster.
- The compiler command **cc** now has several new switches:
 - VCSD** Include **csd** debug information in the object and executable.
 - VPEEP** Enable peephole optimization.
 - VSMALL** Enable PC-relative addressing for global data and function references.
 - VCOMPAC** Enable PC-relative addressing for function references.
 - VNOOPT** Turn off all optimization, to speed up compilation.
- The compiler now recognizes the ANSI type qualifiers **const** and **volatile**. It produces a warning message if **volatile** is encountered in a source file that is being compiled with the option **-VPEEP**. The old, unused keyword **entry** is no longer recognized.
- The microshell **msh** now has a built-in command **mshversion**. This makes the version of the release available to you without calling in any programs.
- **as68toas** has been changed to accept input and output file specifications. Its usage is as follows:

```
as68toas <infile> [-o <outfile>]
```
- **msh** now works with the Mega-ST internal clock and ROMs.
- The RAM-disk utility **rdy** now runs on the Mega-ST. It also allows creation of larger RAM disks.
- The symbol-table utility **nm** now supports symbol tables larger than 64 kilobytes.
- **pr** now implements two features from UNIX System III:
 - eck** Expand tab char *c* at positions *k*+1, 2**k*+1, 3**k*+1, etc., on input. *c* defaults to '\t' and *k* defaults to eight.

ick Insert tab char *c* for spaces at positions $k+1$, $2*k+1$, $3*k+1$, etc., on output. *c* defaults to '\t' and *k* to eight.

- The utility **drtomw** no longer uses the **-f** flag. It now transforms executables from the DRI object format into the new Mark Williams object format. If you convert an object library from DRI format into Mark Williams format, you should use the archiver **ar** to produce a "ranlib header".

The standard library **libc** has the following changes:

- The function **time** now works properly on a Mega-ST.
- Two new date/time routines have been added: **Ssettime** and **Sgettime**. These are similar to **Ksettime** and **Kgettime** except that they do not directly access the intelligent keyboard's clock for time resolution within one second; instead, they use the **xbios** time/date routines, which have a resolution of two seconds. These routines were added because **Kgettime** does not work as expected on Mega-STs.
- The following string functions have been added: **memchr**, **memcmp**, **memcpy**, **memset**, **strchr**, **strcspn**, **strerror**, **strpbrk**, **strrchr**, **strspn**, **strstr**, and **strtok**.

The following problems found in previous releases have been fixed:

- If *t* was a **signed long int**, $t > 16$ produced the wrong result. This has been corrected.
- If *t* and *x* were **signed long int**, $t \% x$ produced unexpected results for some combinations of values. This has been corrected.

How to use this manual

This manual is in nine sections. Section 1, which you are now reading, introduces Mark Williams C.

Section 2 shows you how to install Mark Williams C on your computer. It also introduces the microshell **msh** and its commands, introduces the MicroEMACS screen editor, and shows you how to compile simple C programs.

Section 3 introduces compiling with Mark Williams C. It describes the options to the compiler controller **cc**, and shows you how to compile using different formats. Debugging with Mark Williams C and using **rdy**, the Mark Williams RAM-disk utility, are introduced. Technical issues that involve the 68000 microprocessor and TOS are also discussed.

Section 4 is a tutorial on the MicroEMACS screen editor. It introduces most of the MicroEMACS commands and includes exercises to help sharpen your skills at editing programs.

6 Mark Williams C for the Atari ST

Section 5 is a tutorial on **make**, the Mark Williams programming discipline. **make** is one of the most useful tools available for constructing and maintaining large, intricate programs. This section describes **make**, from building relatively simple programs to using **make** to control work other than compiling C programs.

Section 6 introduces **resource**, the Mark Williams resource editor. Section 7 introduces the utilities **resdecom** and **rescomp**, which, respectively, disassemble a resource into a file of source text and compile source text into a resource. Together, these give you a powerful set of tools for creating, editing, writing, and compiling GEM resources.

Section 8 lists all of the error messages that the Mark Williams C compiler, assembler, and utilities can produce. This includes error messages from the resource utilities. Many entries have hints to help you correct or avoid the error that the message describes.

Finally, section 9 is the Lexicon. This is by far the largest part of the manual. The Lexicon contains several hundred individual entries; each describes a command, a function, defines a C technical term, or gives you other 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 cross-references to related entries.

The opening sections of this manual 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 the manual lists the Lexicon's entries sorted by category, and gives an index.

User registration and reaction report

Before you continue, fill out the User Registration Card that came with your copy of Mark Williams C. When you return this card, you become eligible for direct telephone support from the Mark Williams Company technical staff, and you will automatically receive information about all new releases and updates.

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.

Technical support

Mark Williams Company provides free technical support to all registered users of Mark Williams C. If you are experiencing difficulties with Mark Williams C, outside the area of programming errors, feel free to contact the Mark Williams Technical Support Staff. You can telephone during business hours (Central time), or write. This support is available *only* if you have returned your User Registration Card for Mark Williams C.

If you telephone Mark Williams Company, please have at hand your manual for Mark Williams C. Please collect as much information as you can concerning your difficulty before you call. If you write, be sure to include the product serial number (from the sticker on the back of this manual) and your return address.

Bibliography

The following books may be helpful in developing your skills with C. This list also contains all books that are referenced in this manual. It is by no means exhaustive; however, it should prove helpful to both beginners and experienced programmers.

American National Standards Institute: *Draft Programming Language C (October 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.

Harbison, S.; Steele, G.: *C: A Reference Manual*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.

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

Kelley, A.; Pohl, I.: *C by Dissection: The Essentials of C Programming*. Menlo Park, Ca.: The Benjamin/Cummings Publishing Company, Inc., 1987.

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

8 Mark Williams C for the Atari ST

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.

Plum, T.; Brodie, J.: *Efficient C*. Cardiff, NJ: Plum Hall, Inc., 1985.

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.

Vile, R.C., Jr.: *Programming in C with Let's C*. Glenview, IL: Scott, Foresman and Company, 1988.

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.; Fidler, 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.

Field, S.; Mandis, K.; Myers, D.: *COMPUTE!'s ST Applications Programming in C*. Greensboro, NC: COMPUTE! Publications, Inc., 1987. *Recommended*.

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.

Leemon, S.: *COMPUTE!'s Technical Reference Guide: Atari ST, Volume 1: VDI*. Greensboro, NC: COMPUTE! Publications, Inc., 1987.

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. *Recommended*.

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

Section 2:

Installing and Running Mark Williams C

This section describes how to install Mark Williams C onto your computer, and how to use it to compile simple programs.

Back up your disks!

Before you begin, you must make backup copies of your distribution disks. *Never* work directly with your distribution disks!

The distribution disks for Mark Williams C are in ten-sector format; that is, each track has ten sectors on it, instead of the nine sectors that the Atari ST uses by default. This format lets you store more files on each disk. However, you *cannot* copy one disk to another by dragging one disk icon to another, even if both disks are in ten-sector format. If you do so, only nine sectors of every ten will be copied, and the backup disk you create will be useless.

Mark Williams C includes a special utility for backing up your distribution disks: **working**. To back up your distribution disks, place distribution disk 4 into drive A, and click the icon labelled **working.tos**.

If you have two floppy disk drives on your system, **working** asks you which drive holds the destination disk; it then formats that disk into a single-sided, ten-sector format. Then, **working** asks you which drive holds the source disk; when you answer, it copies the source disk to the destination disk one track at a time.

If your system has only one floppy disk drive, **working** will prompt you when to insert the source disk and when to insert the destination disk into the drive.

You should continue to format and copy disks until you have backed up all of your distribution disks. Now, put your original distribution disks away in a safe place, and continue to work with the copies that you have just made.

Installing Mark Williams C

Mark Williams C comes on five single-sided floppy disks. It can be used with any currently available configuration of disk drives, hard disk, and RAM.

If your system has only one or two single-sided floppy disk drives, then you do not need to install Mark Williams C. It will work on your system just as it comes out of the package. All you need to do is back up your disks. Skip below to the section entitled, **Using Mark Williams C with single-sided floppy disks.**

If your system has either a hard disk or a double-sided floppy disk drive, you must install Mark Williams C onto your system. In the case of a hard disk, installation means copying the files from the distribution disks into the correct directories on your hard disk. In the case of a double-sided floppy disk drive, it means copying the source disks onto three double-sided floppy disks; doing so means that you can compile and link without having to exchange floppy disks.

To begin installation, insert distribution disk 1 into disk drive A. Double-click the icon labelled **INSTALL.PR**. In a moment, the screen clears and a new menu bar appears at the top of the screen. The title called **Desk**, as always, lets you invoke your desk accessories; the title called **Read Me** gives you information about **install**, should your memory need refreshing.

If you are installing Mark Williams C onto a double-sided disk drive, you should first format three double-sided floppy disks. To format your disks, sweep the mouse pointer over the title marked **Options**. From the menu that appears, click the entry **Format Diskette**. A dialogue appears that describes how to format floppy disks. Be sure to click the buttons for double-sided disks and for ten-sector format; the ten-sector format allows you to write more files onto each disk. After you have formatted your three disks, label one "compiler", one "commands", and the third "sources". Then exit from the **Format Diskette** dialogue by clicking the **Quit** button.

To begin installation, sweep the mouse pointer to the title called **Options**, and then double-click the entry **Begin**.

install begins by showing you what it thinks your system's configuration is. If it is wrong, click the appropriate button to correct the description.

If you have a hard disk, **install** then shows you the default drive and the default directories into which it normally installs Mark Williams C. You are not obliged to use either the default drive or the default directories; you may change either to suit your preferences. To change a directory name, simply click the appropriate entry and type your correction. The drive you select must have at least two megabytes of free space. If you do not use the default names given by **install**, you must edit the file **profile**, which is used by the micro-shell **msh**, and correct the entries for **PATH**, **LIBPATH**, and **INCDIR** to show the directories that you have selected.

Therefore, if you change the directory names, be sure to jot down the names that you choose; the section that introduces `msh`, below, gives more information on editing profile.

If you do not have a hard disk, you will be asked what portion of Mark Williams C you wish to install. We recommend that you install Mark Williams C in its entirety; however, if you wish, you can install only the compiler, the commands and utilities, the resource tools, or the source code and sample programs.

When `install` has asked all of its questions, it asks you to confirm your choices and whether you wish to begin installation. If you answer "No", `install` returns you to its desktop; otherwise, installation begins.

`install` will prompt you when it needs either a new distribution disk or, in the case of installing onto a double-sided floppy disk, when it needs a new target disk. If you have only one double-sided floppy disk and no hard disk, `install` copies files from the distribution disk into memory and then copies back onto the target floppy disk.

When installation is finished, we suggest that you copy the installed disks, and store the "original" installed disks in a safe place. This will spare you the trouble of installing Mark Williams C again, should your installed disks be spoiled by some mishap.

Using Mark Williams C with single-sided floppy disks

If your system has only one or two single-sided floppy disk drives and no hard disk, you do not need to install Mark Williams C. It will work for you exactly as you unwrap it; all you need to do is make backup copies of your disks, and you are ready to begin compiling.

To compile a program, you must use the `-Z` option to the `cc` command. `cc` controls the compiler, and the `-Z` option tells `cc` that you are using single-sided disk drives. A single-sided floppy disk is not large enough to hold the compiler, the linker, and the libraries; therefore, the `-Z` option tells `cc` to prompt you when compilation is finished, so you can remove the disk that holds the compiler, and insert the disk that holds the linker.

Another way to use Mark Williams C with single-sided drives is to keep the compiler disk in drive A and the linker disk in drive B. You can store your source files on either of these disks (although room will be limited) or on your RAM disk. If you keep sources on a RAM disk, you should back them up frequently.

If you have only one single-sided floppy disk drive, you must keep your source files on the RAM disk. By frequently changing disks, you will be able to compile and link your programs, although you will be limited in the number and size of the programs you can compile at any one time.

Using two double-sided floppy drives

The main advantage of using two double-sided floppy disk drives is that you can keep the “commands” disk in the second drive at all times, which gives you immediate access to all of **msh**’s commands and utilities. You can also use the second drive to back up your source files and compiled programs. As with one floppy drive, you will probably find it most useful to compile your source files from the RAM disk. This section also introduces **msh** and its utilities, and describes how to compile programs under Mark Williams C.

Introducing the Mark Williams micro-shell

Mark Williams C is designed to run under a micro-shell, called **msh**. **msh** allows you create commands that would be too long or too complex to enter 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*. **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 the folder named **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.PR**G and click the left button twice.

The screen clears and the current system date and time appear; then **msh** prints a percent sign ‘%’ in the upper left-hand corner. The percent sign is a *prompt*: it means that **msh** is ready to accept a command.

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

```
echo foo
```

echo is a command that repeats all of the words, or *arguments*, that follow it.

When you press the Return key, the argument **foo** appeared on the next line of the screen; then another percent sign appeared, which signals that **msh** is ready to accept another command.

Editing a file

msh includes a full-featured screen editor, called MicroEMACS. An *editor* is a program that lets you type text into your computer, store it on disk, then recall it from disk and change it. You will use an editor to type all of the programs that you compile with Mark Williams C.

MicroEMACS allows you to divide the screen into sections, called *windows*, and display and edit a different file in each one. 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. Also, MicroEMACS has a full help function for C programming. Should you need information about any macro or library function that is included with Mark Williams C, all you need to do is move the text cursor over that word and press a special combination of keys; MicroEMACS will then open a window and display information about that macro or function.

Mark Williams C includes both a compiled, binary version of MicroEMACS that is ready to use, and the full source code. We invite you to examine the code, modify it, and enhance MicroEMACS to suit your preferences.

For a list of the MicroEMACS commands, see the Lexicon entry for **me**, the MicroEMACS command. A following section of this introduction gives a full tutorial on MicroEMACS. In the meantime, however, you can begin to use MicroEMACS by learning a half-dozen or so commands.

To invoke MicroEMACS, type the command

```
me hello.c
```

at the **msh** prompt. This invokes MicroEMACS to edit a file called **hello.c**. Now, 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. Exit from the editor by typing **<ctrl-X><ctrl-C>**.

16 Mark Williams C for the Atari ST

Now, re-invoke MicroEMACS by typing

```
me hello.c
```

The text of the file you just typed is now displayed on the screen. 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>`.

If you prefer, you can also move the cursor by pressing the arrow keys.

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.

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 so that you can begin to do real work.

Now, again *save* the file by typing `<ctrl-X><ctrl-S>`, and exit from MicroEMACS by typing `<ctrl-X><ctrl-C>`.

Just as a reminder, the following table gives the MicroEMACS commands presented above:

<code><ctrl-N></code> or ↓	Move cursor to the <i>next</i> line
<code><ctrl-P></code> or ↑	Move cursor to the <i>previous</i> line
<code><ctrl-F></code> or →	Move cursor <i>forward</i> one character
<code><ctrl-B></code> or ←	Move cursor <i>backward</i> one character
<code><ctrl-D></code>	Delete a character
<code><ctrl-X><ctrl-S></code>	Save the edited file
<code><ctrl-X><ctrl-C></code>	Exit from MicroEMACS

Setting the shell's internal variables

msh allows 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 percent 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 entry 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.

18 Mark Williams C for the Atari ST

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 **A:**, which is the *root directory* on drive A, type:

```
setenv HOME=a:\
```

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 **a:**, 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.

Two dots “**..**” stand 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 sub-

directory **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
```

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 also the screen.

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* 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.

- 2> file** Redirect all material sent to the standard error into *file*.
- 2>> file** Append all material sent to the standard error onto the end of *file*. If *file* already exists, do *not* delete its contents.
- 3> file** Redirect all material normally sent to the printer into *file*.
- 3>> file** Append all material normally sent to the printer onto the end of *file*.
- >& file** Redirect both the standard output and the standard error of a command into *file*.
- >>& file** Append both the standard output of a command and the standard error onto the end of *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.

Redirecting to peripheral devices

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, for instance, 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 three *logical devices* built into it. **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:

22 Mark Williams C for the Atari ST

```
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 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 `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 *fileanyletter*. 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.)

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 reads 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.

Your **profile** will include the following set of commands. These are examples only, to demonstrate how **profile** works; your **profile** will be much larger, and may define these variables somewhat differently:

```
set drive=a:
setenv PATH=.cmd,,$drive\command
setenv SUFF=,.prg,.tos,.ttp
setenv LIBPATH=$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=.cmd,,$drive\command
```

sets the **PATH** environmental variable, which tells **msh** where to find executable files. The first directory, **.cmd**, stands for **msh**'s internal command directory. 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\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:\command** (remember that **\$drive** is interpreted to mean **a:**). 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 in the Lexicon.

Finally, the lines

```
set prompt='% '  
set history=8
```

set the prompt to a percent sign '%' and set the **history** buffer to hold the last eight commands entered. This is used with the **history** command; the history command is described below.

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

msh also uses another file, called **postfile**, that restores the desktop environment when you exit from **msh**.

Embedded commands

msh allows you to embed a command within another command; the output of the inner command is automatically passed as input to the outer command. Command substitutions are indicated by quoting the inner command with grave accents. For example, the command:

```
pr `ls *.c`
```

first invokes the list command **ls** to read the contents of the current directory, and then passes its output to the pagination command **pr**, which paginates the files named by the **ls** command and displays them on the standard output device.

One form of embedded command is included in the standard **profile**: the command

```
date `date -i`
```

resets the GEM clock from the keyboard clock after a warm boot.

The .cmd directory

The directory **.cmd** holds user-defined commands. You can create a new command and load it into **.cmd** by using the **set** command. For example, the following command to **msh** creates a new list command for you:

```
set in .cmd lc="ls -wf"
```

This tells **msh** to equate the command **lc** with the command **ls -wf**, which prints the contents of a directory in columnar format.

Device-sensitive prompts

msh contains two useful shell variables: **cwd** and **cwdisk**. **cwd** holds the name of the current directory, and **cwdisk** the name of the current physical device.

To create a prompt that always shows the current device, use the following command:

```
set prompt='$cwdisk> '
```

Your prompt will automatically change to show which physical device you are on. The following command creates a directory-sensitive prompt:

```
set prompt='$cwd> '
```

This prompt will change automatically to show the name of the directory you are in. Users familiar with the MS-DOS operating system may find this feature helpful.

if command

msh has a number of commands built into it that help you to write loops and conditional statements. The most important of these is **if**. Its syntax is as follows:

```
if word1 word2 [ word3 ]
```

If **word1** executes successfully, then **word2** is executed; otherwise, if **word3** is present, it is executed. Each of the words may be a list of commands that is enclosed within parentheses. For example, this command:

```
if (cc example.c) (cp example.c b:\sources)
```

automatically copies the source file **example.c** into the directory **b:\sources** if it compiles correctly. This sequence is helpful, especially if you are compiling your source files from a RAM disk.

Parentheses

A list of commands that is enclosed within parentheses may extend across as many lines of text as necessary. For example, the command

```
if (echo foo
    echo bar
    echo baz) (ls -l)
```

echoes the strings **foo**, **bar**, and **baz**, and prints the contents of the current directory. The command or commands in the *second* word are executed if *any* of the commands in the first word execute successfully.

while command

msh also contains a **while** command, which allows you to run a conditional loop. Its syntax is as follows:

```
while word1 word2
```

While **word1** executes successfully, **word2** is executed. Each word may be a list of commands enclosed within parentheses.

For example, the command

```
while (ls -l) (echo foo)
```

first prints the contents of the current directory, and then the string **foo**, in a perpetual loop. **while** is often used with the test commands **equal** and **not**, which are described below.

equal and not

Two built-in commands, **equal** and **not**, allow you to build conditional loops under **msh**.

equal compares two strings. It succeeds if the strings are identical, and fails if they are not. Its syntax is as follows:

```
equal argument1 argument2
```

Either argument can be a literal string, an integer, or an embedded command. For example, the following command tells you if your screen is in high resolution or not:

```
if (equal 'getrez' 2) (echo "High res") \  
    (echo "Not high res")
```

The **if** command tests whether the return value of the command **getrez** is equal to 2, which indicates that the screen is in high resolution. If the test succeeds, the command echoes the string **High res** onto the screen; if the test fails, it echoes **Not high res**.

The **not** command inverts the logical result of its argument. For example, the following command is another version of the resolution checker, shown above:

```
if (not (equal 'getrez' 2)) \  
    (echo "Not high res") (echo "High res")
```

In this example, the **if** command compares what is returned by **getrez** and two; **getrez** normally returns two if the screen is in high resolution. The result is then inverted by the **not** command, so that if the comparison fails, the **if** statement

overall would succeed and execute its first argument. In this instance, it would echo the string **Not high res** onto the screen.

History command

The *history* command allows you to repeat commands without having to type them over again.

To begin, the command **!!** re-executes your last command. Therefore, the commands

```
ls -w
!!
```

will give you two columnar listings of the contents of your current directory.

The command *lname* re-executes the last command with *name* that is in your history directory. For example, when you type the following list of commands:

```
ls -w
echo foo >stuff
rm stuff
!ls
```

the history command **!ls** reaches back and executes the command **ls -w**.

You can execute a previous command by its number relative to the current command. For example, **!-1** re-executes the previous command; it is a synonym for **!!**. To execute a command issued three commands ago, type **!-3**. This will execute **echo foo >stuff**. Remember that the number of each command changes every time a new command is executed. Remember, too, that **msh** by default saves only the last eight commands; to increase this number, use **set** to change the variable **history**.

Three aliases

msh includes a number of preset aliases. You can type these into a file of **msh** commands (or a *script*), and **msh** automatically replaces them with their proper values when you run the script.

The alias **\$*** gives the arguments to the current command. For example, if the following command is written into a file:

```
while () (echo $*)
```

typing the file's name plus any number of arguments causes those arguments to be repeated endlessly.

One use for this feature is to help control compilation. For example, the command

```
cc -V $*
```

when placed in a file, will compile all of the files listed as arguments to that file.

\$# gives the number of arguments assigned to the current command. For example, the command

```
echo $#
```

prints 1 on the screen, which is the number of arguments to that command.

Finally, the alias **\$<** represents any line received from the standard input device, up to the newline character.

The **camefrom** variable

The shell contains a built-in variable, called **camefrom**, which it maintains automatically. **camefrom** is set to either **auto**, **desktop**, **msh**, **execve**, or **NULL**, depending on where the current level of the shell thinks that it came from.

The **is_set** command

Finally, **msh** contains a built-in command called **is_set**. This returns zero if it is passed an argument to a shell variable that is already set. Its syntax is as follows:

```
is_set [ in dir ] name
```

which is much like that of the **set** command.

is_set can be combined in shell scripts with the **if** command to perform an action if a particular environmental variable is set. You will find this to be especially helpful to use with the RAM-disk utility **rdy**. As is explained in its Lexicon entry, **rdy** has two interfaces: a command-line interface, and a graphics interface. It invokes its command-line interface if the environmental variable **CMD** is set, and invokes its graphics interface if that variable is not set. You can use **is_set** to help ensure that the correct interface is used, as follows:

```
if (is_set CMD) rdy (gem rdy)
```

If **CMD** is set, then **rdy** is invoked as normal and the command-line interface is used. If **CMD** is not set, then **is_set** will return a non-zero value, the **if** condition will fail, and **rdy** will then be invoked under the **gem** command, which will prepare the environment correctly for the graphics interface.

For more information

Look in the Lexicon for more information on **msh** and its commands. **msh** itself has an entry in the Lexicon; also see the entry for **commands**, which lists all of the commands available with **msh**. See the entry for **environment** for a list of the important environmental variables, each of which has its own entry within the Lexicon. Also check the index at the end of this volume if you are searching for information on a particular topic. You should find it helpful.

Section 3:

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 involves several steps; however, Mark Williams C simplifies it with the **cc** command, which controls all the actions of the compiler.

The phases of compilation

Mark Williams C is not just one program, but a number of different programs that work together. 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 translates 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 it into machine code. The code generation is table driven, with entries for each operator and addressing mode.
- cc2** The optimizer/object generator. This phase optimizes the generated code and writes the object module.
- cc3** Mark Williams C also includes a fifth phase, called **cc3**, which can be run after the object generator, **cc2**. **cc3** generates a file of assembly language instead of a relocatable object module. 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 the **-S** option, Mark Williams C creates 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 link the module with other program modules and with the library functions.

As the final step in its execution, **cc** calls the linker **ld** to produce an executable program.

Compiling from the GEM desktop

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. Move the following files plus your source code into the same folder:

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

Also move all of the header files, which have the suffix **.h**.

2. 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.

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.

Edit errors automatically

The first option, and one that you'll use most often, is the MicroEMACS option **-A**. Often when you're writing a new program, you try to compile it, only to have the compiler tell you that you've made a mistake. You must then invoke your editor, change the program, exit from the editor, and start compiling the program again.

To make this process easier, `cc` command has the *automatic* (or MicroEMACS) option, `-A`. If Mark Williams C detects any errors in your program, it will automatically invoke the MicroEMACS screen editor. MicroEMACS will display all error messages in one window and your source code in another, with the cursor set at the number of the line where the first error occurred.

Try the following example. Use MicroEMACS to create a program called `error.c`. To invoke MicroEMACS, type the command

```
me error.c
```

at the `msh` prompt. Then type the following code:

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

Note that the semicolon was left off of the `printf` statement. Type `<ctrl-X><ctrl-S>` to save the file to disk, and `<ctrl-X><ctrl-C>` to exit from MicroEMACS. Now, try compiling `error.c` with the following `cc` command:

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

You will 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 the upper window you will see the message:

```
4: missing ';'

```

and in the lower window you will see your source code for `error.c`, with the cursor set on line 4. If you had more than one error, 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.

With some errors, such as those for missing braces or semicolons, the compiler cannot always tell exactly which line the error occurred on; it will point to a line that is near the source of the error.

Now, use `<ctrl-E>` to move the cursor to the end of line 3, and type a semicolon to correct the error. Type `<ctrl-X><ctrl-S>` to save the file to disk, and then type `<ctrl-X><ctrl-C>` to exit from MicroEMACS. `cc` will recompile the program automatically, to produce a normal working executable file.

`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>`.

Renaming executable files

When Mark Williams C compiles a source file, by default it names the executable program after the source file. For example, when you compiled **error.c**, Mark Williams C automatically named the executable file **error.prg**.

If you wish, you can give the executable file a different name. Use the **-o** (output) option, followed by the desired name. For example, should you wish the executable file to have the name **example.prg**, use the command:

```
cc -o example.prg error.c
```

This command will compile the source file **error.c** and generate an executable file called **example.prg**. The suffix **.prg** tells TOS that the file is executable.

Floating-point numbers

Often, you will need to use floating-point numbers in your programs. If you are unsure what a floating-point number is, see the Lexicon entry for **float**.

The routines that print floating-point numbers are large, and most C programs do not need to print floating-point numbers; therefore, the code to perform floating-point arithmetic is not included in a program by default. You must ask Mark Williams C to include these routines with your program by using the **-f** option with the **cc** command.

For example, if the program **example.c** used floating-point numbers, you would compile it with the following command line:

```
cc -f example.c
```

If your program prints floating-point numbers or reads them from an input device, and it is *not* compiled with the **-f** option, it will print the following error message when it is run:

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

Compiling multiple source files

Many programs are built from more than one file of C source code. For example, the program **factor**, which is provided with Mark Williams C, is built from the C source files **factor.c** and **atod.c**. To produce the executable program **factor**, both source files must be compiled; the linker **ld** then joins them to form an executable file.

To compile a program that uses more than one source file, type all of the source files onto the **cc** command line. For example, to compile **factor** type the following:

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

This command compiles both C source files to create the program **factor**.

When the **cc** command line includes several file name arguments, by default it uses the *first* to name 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**.

The argument **-lm** tells **cc** to include routines from the mathematics library when the object modules are linked. This option must come *after* the names of all of the source files, or the program will not be linked correctly.

Wildcards

A *wildcard* character is one that represents a variety of characters. The two most commonly used wildcards are the asterisk ***** and the question mark **?**. The asterisk can represent any string of characters of any length (including no character at all), whereas the question mark can represent any one character.

For example, if the current directory held the following files:

```
a.c
ab.c
abc.c
abcd.c
```

typing **ls a?.c** would print:

```
ab.c
```

whereas typing **ls a*.c** would print all four files.

The **cc** command lets you use wildcards in your command line to save you time and effort. For example, you can compile all of the C source files in the current directory simply by typing:

```
cc *.c
```

This command compiles all of the files with the suffix **.c** and links the resulting object modules.

In another example, if the program **example** were built from the source files **example1.c**, **example2.c**, and **example3.c**, you could compile them *with the* following command:

```
cc example?.c
```

Linking without compiling

When you are writing a program that consists of several source files, you will need 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, if you modify the **factor** program by changing the source file **factor.c**, you can recompile **factor.c** and relink the entire program with the following command:

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

The first two arguments are the C source file **factor.c** and the *object module* **atod.o**. **cc** recognizes that **atod.o** is an object module and simply passes it to the linker **ld** 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 will need to compile a source file but not link the resulting object module to the other object modules. You will do this, for example, to compile a module that you wish to insert into a library. Use the **-c** option to tell **cc** not to link the compiled program. This option is used most often to create relocatable object modules that can be archived into a library for later use.

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 -f factor.o atod.o -lm
```

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 access 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 `example.c` and the assembly language source file `example.s`, simply use the `cc` command as usual:

```
cc example1.c example2.s
```

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

The Lexicon entry for the `ROS` function `Stacsize` includes an example that demonstrates how to combine routines written in assembly language with routines written in C.

Changing the size of the stack

The *stack* is the segment of memory that holds function arguments, local variables, and function return addresses. Mark Williams C by default sets the size of the stack to two kilobytes (2,048 bytes). This is enough stack space for most programs; however, some programs, such as the example program on page 26 of the first edition of *The C Programming Language*, require more than two kilobytes of stack. A program that uses more than its allotted amount of stack will cause a *stack overflow*; this may force you to reboot your computer.

The size of the stack cannot be altered while a program is running. 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.

Debugging with Mark Williams C

Mark Williams C comes with several utilities that help you debug your programs. These include `dbx`, which is a powerful symbolic debugger; `nm`, which prints symbol tables from programs for analysis; and `od`, which will print a formatted dump of a file. It also supports `cdd`, the Mark Williams C Source Debugger.

csd: the C Source Debugger

Mark Williams C creates executable files that can be debugged with **csd**, the Mark Williams C Source Debugger. **csd** lets you step through your source code one expression at a time, set break points, enter new expressions for evaluation, and compare the execution of your source code with its screen output.

To connect the C source code with the compiled executable, **csd** uses a special, enlarged debug table. To include **csd**'s debug table when you compile a program, include the option **-VCSD** on your **cc** command line. If you do not compile the program with this option, it cannot be debugged by **csd**.

A program that is compiled with the **-VCSD** option will run as quickly as one that is compiled without it; however, it will be somewhat larger due to the extra debug information that it holds. If you do not wish to recompile the program once you have finished debugging it, you can use the command **strip** to remove the debug table from the executable. This will not affect the program's performance in any way, and it will make the executable file noticeably smaller.

csd has proved invaluable to programmers of the IBM PC. It now makes source-level debugging available on the Atari ST. For more information about **csd**, contact Mark Williams Company or your local software dealer.

db: symbolic debugger

Mark Williams C includes the symbolic debugger **db** to assist you with debugging your programs. Unlike **csd**, **db** works on the level of assembly language. **db** can be used to debug programs that are assembled by **as**, the Mark Williams assembler, as well as debug programs that are compiled by Mark Williams C.

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 breakpoint 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 to **db**, and instructions on how to use it to debug your programs, see the entry for **db** in the Lexicon.

With release 3.0, **db** can work through the **aux** port, so you can debug programs that use AES and VDI calls. This feature allows you to plug a terminal into the **aux** port and send commands to **db** from it; the action of the program is then displayed on the ST screen. To use this feature, invoke **db** with the option **-A**.

od: formatted dump

od prints a formatted dump of a file. 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 hexadecimal. Normally, you give **od** a file name as an argument; to display a hexadecimal dump of the file **tempfile**, type:

```
od tempfile
```

od can also display files in octal, decimal, or characters, and in bytes or words, whichever you prefer. See the Lexicon entry for **od** for more information.

nm: print symbol tables

nm prints out the symbol table from an object module or library. It is designed to work with libraries created 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 **nm** for the library **libc.a**, use the **cd** command to move to the directory where you have stored **libc.a** and then type:

```
nm libc.a
```

For more information on **nm**, see its entry in the Lexicon.

Creating smaller, faster programs

Mark Williams C creates executables that are small and fast. However, Mark Williams C includes a number of features that let you increase the speed and decrease the size of your programs.

PC-relative addressing

By default, Mark Williams C uses absolute addressing in the programs it compiles. This allows a program to address the full scope of memory, and to build objects that are extremely large.

The Atari ST, however, can use another type of addressing, called *PC-relative addressing*. PC stands for *program counter*. In this mode of addressing, a 16-bit offset is added to or subtracted from the value in the program counter register.

The advantage of PC-relative addressing is that on the M68000, it often is faster than absolute addressing. Also, programs that use PC-relative addressing are smaller than those that use absolute addressing because it uses only 16 bits, instead of a full, 32-bit address. The disadvantage is that a program can jump only 32-kilobytes from the address in the program counter. Therefore, a program that uses extremely large objects or that has global references that are more than 32 kilobytes apart cannot use PC-relative addressing. However, if your program is small, you may wish to consider PC-relative addressing for its advantage in size and speed.

The `cc` command has two options for generating PC-relative addressing: `-VSMALL` and `-VCOMPAC`. `-VSMALL` is for programs whose code and data both can use PC-relative addressing. `-VCOMPAC` is for programs that have small amounts of code and large amounts of static or global data (e.g., a text editor); it uses PC-relative addressing for code and absolute addressing for data.

Both options produce executables that have full, 32-bit pointers. Thus, a module compiled with the `-VSMALL` or `-VCOMPAC` option can be linked with modules compiled with the default of absolute addressing. If a function is used repeatedly within your program, you may wish to compile it to use PC-relative addressing to speed up the program.

Strip

Once a program is compiled and linked, it often does not need the debug and relocation tables that the compiler builds into it. These tables are needed only if further debugging will be performed on the program, such as with `csd`, the Mark Williams C Source Debugger. These tables can be removed from an executable program with no loss in performance, and with a considerable savings in space.

The utility **strip**, which is included with Mark Williams C, removes the symbol and debug tables from a linked executable program. To use it under **msh**, simply type

```
strip filename
```

where **filename** is the name of the executable you wish to alter. **strip** can be used with the following options:

```
-d      Keep debug table
-r      Keep relocation table
-s      Keep symbol table
```

One note of caution: **strip** should not be used on an object module (that is, a file whose suffix is **.o**), because if you do it cannot be linked.

Compiling with a RAM disk

Mark Williams C includes a utility, called **rdy**, which lets you build a rebootable RAM disk on your Atari ST. The term “rebootable” means that the RAM disk and its contents will not be affected by a warm boot on your computer. You can use a RAM disk to hold the temporary files that Mark Williams C builds; this will noticeably accelerate compilation on your system.

For a full description of **rdy**, see the Lexicon. The following describes in brief how to install and use a RAM disk.

rdy works by creating a *prototype* RAM disk, which it stores in a file. This prototype contains information concerning the size of the RAM disk in kilobytes, its device name (e.g., whether it is disk E or G), and other necessary information. Then **rdy** will *load* the prototype RAM disk into memory.

rdy can copy a RAM disk plus all of its contents into a file; this allows you to back up a RAM disk easily. You can create a RAM disk, load utilities into it, then back up the RAM disk. Thus, whenever you need to recreate a RAM disk, you can use your backup copy and spare yourself the trouble of reloading your utilities. **rdy** can also remove a RAM disk from memory.

Building a RAM disk

To begin building a RAM disk, insert the copy of distribution disk 1 into drive A, and then click the icon labelled **rdy.prg**. In a moment, the screen will clear and a new menu bar will appear at the top of the screen. The title at the left of the menu bar, called **Desk**, gives you access to all desk accessories. The title at the right, **Read Me**, describes how **rdy** works. You can use this feature to refresh your memory while using **rdy**. The title in the center, **Options**, lets you *command* **rdy** to perform a task.

If you sweep the mouse pointer over the **Options** title, a menu drops down; this menu has six entries. The first entry, **Create a RAM disk**, creates a prototype RAM disk and writes it into a file. The second, **Load a RAM disk**, loads a RAM disk into memory. The third, **Back up a RAM disk**, writes a RAM disk and all of its contents into a file that you can later reload into memory. The fourth, **Remove a RAM disk**, removes a RAM disk from memory. The fifth, **Get data on a RAM disk**, will display information either about a RAM disk that is currently in memory, or about a RAM disk file that you created earlier, whichever you prefer. The last entry, **Quit**, lets you exit from **rdy**.

To begin, click the first entry, **Create a RAM disk**. A series of dialogues will ask you to describe the RAM disk that you want to build.

The first dialogue box asks you how much RAM your system has. Click the appropriate button.

The next dialogue asks the size of the RAM disk you wish to create. Again, click the appropriate button. Your RAM disk should be large enough to hold a significant number of files, but not so big that it stops you from loading any program that you use frequently. A good rule of thumb is to use a RAM disk that takes up approximately one quarter to one half of the RAM on your machine.

The next dialogue asks what drive the RAM disk should be. You should not use a drive that is already taken up by another device, such as a logical partition on your hard disk or another RAM disk. If you do so, **rdy** will not be able to load your RAM disk.

Then it asks if you want this RAM disk to be your system's boot disk. At present, answer **No** to this question. For more information on using the RAM disk as your boot disk, see the Lexicon entry for **rdy**.

rdy then asks the name of the file in which to store the prototype RAM disk.

When you have answered these questions, **rdy** displays the configuration of the new RAM disk and asks you if it is correct. If you answer "No", you will return to the **rdy** desktop; otherwise, the new prototype RAM disk will be written.

Finally, **rdy** asks if you wish to load the new RAM disk. If you answer "No", **rdy** returns you to its desktop. Answer "Yes", which tells **rdy** to load the new file. As it installs a new RAM disk, **rdy** warm boots your system. Do not be alarmed when the screen clears and you are returned to the GEM desktop: this indicates that the RAM disk has been loaded successfully.

The next step is to install your new disk on the GEM desktop. To do so, first single-click the icon for one of your existing storage devices; then move the mouse pointer to the **Options** title on the menu bar, and double-click the entry **Install Disk Drive**. Change the name of the drive from its old setting to the name of your RAM disk and then type in the name that you want to appear under the icon (e.g., "RAM DISK"). Then click the button labelled **Install**. The desktop will

return with the new icon displayed.

Working with a RAM disk

A RAM disk speeds up your work by reducing the time the compiler needs to read a file. For example, the compiler writes temporary files to pass information between its phases; writing the temporary files onto the RAM disk eliminates the time taken by writing these files onto a disk and reading them back. Test compilations have shown that this change alone will cut the time you need to compile and link a large program by more than half.

To take full advantage of your RAM disk, you will need to tell the Mark Williams microshell **msh** that it exists and how you want it to be used. To do so, you must edit the file **profile**, which **msh** reads when invoked. The **profile** file is described in the Lexicon, under the entry for **msh**. In brief, the line that begins **PATH=** lists all the directories where programs should look for executable files. Your RAM disk should go near the beginning of that list. For example, this line may read as follows:

```
PATH=.cmd,,a:\bin, b:\bin
```

If your RAM disk is named as drive E, change the **PATH** description to the following:

```
PATH=.cmd,e:\,,a:\bin,b:\bin
```

This tells **msh** that the RAM disk should be searched for executable files *before* either of the floppy disk drives; naturally, a RAM disk can be searched much more quickly than a floppy disk drive, which will save you time.

We suggest that you not attempt to alter the **profile** until *after* you have installed Mark Williams C and have read the chapter in the manual that introduces **msh**. If you alter the **profile** too radically without knowing how it works, you may confuse **msh** and create difficulties for yourself.

Finally, use the command **mkdir** to create the directories **tmp** and **bin** on your RAM disk. For example, if your RAM disk is device E on your system, type the following command:

```
mkdir e:\tmp e:\bin
```

You will find that **rdy** is both versatile and powerful. See the Lexicon for more information on **rdy** and its features. We urge you to experiment with it.

One warning: if you have a 520ST and you want to debug programs with **csd**, the Mark Williams C Source Debugger, you may need to remove your RAM disk first. This is because **csd** requires large amount of memory for its buffers, and a 520ST may not have enough memory to hold **csd**, plus the program it is debugging, the source files, and a RAM disk.

Where to go from here

For more information on compiling, see the Lexicon entry for `cc`. This entry summarizes all of `cc`'s options, and presents many that are not discussed here. For more information on the assembler `as`, see its entry in the Lexicon as well.

The following section introduces the MicroEMACS screen editor. If you have worked the exercises in this part of the book, you have already used MicroEMACS a little; this tutorial, however, will show you how to use all of its advanced features to input text quickly and easily.

Then comes an introduction to `make`, the Mark Williams programming discipline. If you are building programs that use multiple files of source code, you will find `make` to be an invaluable tool.

Section 4:

Introduction to MicroEMACS

This section introduces MicroEMACS, the interactive screen editor for Mark Williams C. It 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.

What is MicroEMACS?

MicroEMACS is an interactive screen editor. An *editor* lets you type text into your computer, name it, store it, and recall it later for editing. *Interactive* means that MicroEMACS will accept an editing command, execute it, display the results for you immediately, 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 or any type of text file.

The TOS 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.

For a summary of MicroEMACS and its commands, see the entry for *me* in the *Lexicon*.

Keystrokes – <ctrl>, <esc>

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**.

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, pressing the control key and the letter 'X' key simultaneously 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 section 2 of this manual. If you do not have a hard disk, insert the disk that holds MicroEMACS into drive A as soon as the **msh** prompt appears. Then 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, just backspace over it and retype the text. Press the carriage return or enter key after each line:

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

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 4 lines]

This command has permanently stored, or *saved*, what you typed into a file named **sample**.

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, try them 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 on the left parenthesis. Type `<ctrl-N>`. Notice that the cursor has jumped to the next line, and is now just to the right of the left brace '`{`'. Type `<ctrl-A>`. The cursor has jumped to the *beginning* of the second line of your text. Type `<ctrl-N>` again, and the cursor is at the beginning of the third line of the program, the `printf` statement.

Now, type `<ctrl-K>`. The third line of text has disappeared, leaving an empty space. Type `<ctrl-K>` again. The empty space where the third 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** and then `<return>`. This will return you to **msh**.

Beginning a document

Now, edit the file called **example1.c**. First, use the `cd` to move to directory `\src`, which is where this file was stored when you installed Mark Williams C. If you stored the sample programs in a different directory, then use the `cd` command to transfer to that directory. Now, type the following command:

me example1.c

50 Mark Williams C for the Atari ST

In a moment, the following text will appear on your screen:

```
/*
 * This is a simple C program that computes the results
 * of three different rates of inflation over the
 * span of ten years. Use this text file to learn
 * how to use MicroEMACS commands
 * to make creating and editing text files quick,
 * efficient and easy.
 */
#include <stdio.h>
main()
{
    int i;                /* count ten years */
    float w1, w2, w3;     /* three inflated quantities */
    char *msg = " %2d\t%f %f %f\n"; /* printf string */
    i = 0;
    w1 = 1.0;
    w2 = 1.0;
    w3 = 1.0;
    for (i = 1; i <= 10; i++) {
        w1 *= 1.07;       /* apply inflation */
        w2 *= 1.08;
        w3 *= 1.10;
        printf (msg, i, w1, w2, w3);
    }
}
```

When you type the MicroEMACS command and a file name, MicroEMACS *copies* that file into memory. Your cursor also moved to the upper left-hand corner of the screen. At the bottom of the screen appears the *status line*, as follows:

```
-- ST MicroEMACS v1.2-- example1.c  File: example1.c -----
```

The word to the left, MicroEMACS, is the name of the editor. The word in the center, *example1.c*, 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.

Moving the Cursor

Now that you have read a text file into memory, 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 paragraphs. 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:

Moving the cursor forward

This first set of commands moves the cursor forward.

<code><ctrl-F></code>	Move forward one space
<code><esc>F</code>	Move forward one word
<code><ctrl-E></code>	Move to end of line

To see how these commands work, do the following: Type the *forward* command `<ctrl-F>`. This is equivalent to pressing `<+>`. As before, it does not matter whether the letter 'F' is upper case or lower case. The cursor has moved one space to the right, and now is over the character '*' in the first line.

Type `<esc>F`. The cursor has moved one *word* to the right, and is now over the space after the word *this*. MicroEMACS considers only alphanumeric characters when it moves from word to word. Therefore, the cursor moved from under the * to the space after the word *this*, rather than to the space after the *. Now type the *end of line* command `<ctrl-E>`. The cursor has jumped to the end of the line and is now just to the right of the e of the word *three*.

Moving the cursor backward

The following summarizes the commands for moving the cursor backwards.

<code><ctrl-B></code>	Move back one space
<code><esc>B</code>	Move back one word
<code><ctrl-A></code>	Move to beginning of line

To see how these work, first type the *backward* command `<ctrl-B>`. This is equivalent to pressing `<->`. As you can see, the cursor has moved one space to the left, and now is over the letter e of the word *three*. Type `<esc>B`. The cursor has moved one *word* to the left and now is over the t in *three*. Type `<esc>B` again, and the cursor will be positioned on the o of the word *of*.

Type the *beginning of line* command <ctrl-A>. The cursor jumps to the beginning of the line, and once again is resting over the '/' character in the first line.

From line to line

<ctrl-P>	Move to previous line
<ctrl-N>	Move to next line

These two commands move the cursor up and down the screen. Type the *next line* command <ctrl-N>. The cursor jumps to the space before the '*' in the next line. Type the *end of line* command <ctrl-E>, and the cursor moves to the end of the second line to the right of the period.

Continue to type <ctrl-N> until the cursor reaches the bottom of the screen. This is the same as if you typed <↓>. As you reached the first line in your text, the cursor jumped from its position at the right of the period on the second line to just right of the brace on the last line of the file. 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 up the screen. Type the *previous line* command <ctrl-P>. This has the same effect as pressing <↑>. When the cursor jumped to the previous line, it retained its position at the end of the line. MicroEMACS remembers the cursor's position on the line, and returns the cursor there when it jumps to a line long enough to have a character in that position.

Continue pressing <ctrl-P>. The cursor will move up the screen until it reaches the top 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.

<ctrl-V>	Move forward one screen
<esc>V	Move back one screen

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 one screenful of text at a time.

Type <ctrl-V>. Your screen now contains only the last three lines of the file. This is because you have rolled forward by the equivalent of one screenful of text, or 22 lines.

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 character `'/'` in the first line.

Moving to beginning or end of text

Finally, these two cursor movement commands allow you to jump immediately to the beginning or end of your text.

<code><esc><</code>	Move to beginning of text
<code><esc>></code>	Move to end of 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 `'<'`. 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 pages long.

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>`. If you had modified this file, the following message would appear:

```
[Wrote 23 lines]
```

The text file would have been saved to your computer's disk. 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`.

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 example1.c
```

Within a moment, **example1.c** 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, also has a set of commands that allow you to erase text easily. These commands, *kill* and *delete*, perform differently; the distinction is important, and will be explained in a moment.

Deleting versus killing

When text is *deleted*, it is erased completely; however, when text is *killed*, it is copied into a temporary storage area in memory. This storage area is overwritten 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.

MicroEMACS is designed so that when it erases text, it does so beginning at the *left edge* of the cursor. This left edge is called the *current position*.

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 right

The first two commands to be presented erase text to the *right*.

<ctrl-D>	Delete one character to the right
<esc>D	Kill one word to the right

<ctrl-D> deletes one *character* to the right of the current position. **<esc>D** deletes one *word* to the right of the current position.

To try these commands, type the *delete* command **<ctrl-D>**. The character *'* in the first line has been erased, and the rest of the line has shifted one space to the left.

Now, type `<esc>D`. The `*` character and the word `This` have been erased, and the line has shifted six spaces to the left. The cursor is positioned at the *space* before the word `is`. Type `<esc>D` again. The word `is` has vanished along with the *space* that preceded it, and the line has shifted *four* spaces to the left.

`<ctrl-D>` *deletes* text, but `<esc>D` *kills* text.

Erasing text to the left

You can erase text to the *left* with the following commands:

<code></code>	Delete one character to the left
<code><ctrl-H></code>	Delete one character to the left
<code><esc></code>	Kill one word to the left
<code><esc><ctrl-H></code>	Kill one word to the left

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 word `three` on the first line of text. Then, type ``. The second `e` of the word `three` has vanished.

Type `<esc>`. The rest of the word `three` has disappeared, and the cursor has moved to the second space following the word `of`.

Move the cursor four spaces to the left, so that it is over the letter `o` of the word `of`. Type `<esc>`. The word `results` 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>` 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 type `<esc>`. 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 type `<esc>`.

Typing `` *deletes* text, but typing `<esc>` *kills* text.

Erasing lines of text

Finally, the following command erases a line of text:

<code><ctrl-K></code>	Kill from cursor to end of line
-----------------------------	---------------------------------

This command erases the line 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, in the space before `* Use`.

As its name implies, the `<ctrl-K>` command *kills* the line of text.

Yanking back (restoring) text

The following command allows you restore material that you have killed:

`<ctrl-Y>` Yank back (restore) killed text

Remember that when material is killed, MicroEMACS has temporarily stored it elsewhere. You can return this material to the screen by using the *yank back* command `<ctrl-Y>`. Type `<ctrl-Y>`. All of line 2 has returned; the cursor, however, remains at the beginning of line 3.

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 now return you to `msh`.

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.

Moving one line of text

You can kill and move one line of text with the following commands:

`<ctrl-K>` Kill text to end of line
`<ctrl-Y>` Yank back text

To test these commands, invoke MicroEMACS for the text `example1.c` by typing the following command:

`me example1.c`

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

```
/* This is a simple C program that computes the results
is now at the bottom of your text.
```

Your cursor has moved to the point on your screen that is *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 as both the first and last line of the file.

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 and move more than one line of text at a time, use the following commands:

<code><ctrl-@></code>	Set mark
<code><ctrl-W></code>	Kill 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 `example1.c`. 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 `'@'`, not `'2'`. MicroEMACS will respond with the message

```
[Mark set]
```

at the bottom of your screen. Now, move the cursor down six 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 six lines. Be careful that the cursor is at the *beginning* of the line. Now, type <ctrl-Y> again. The text reappeared *above* where the cursor was positioned, and the cursor has 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 23 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.

Capitalization and other tools

The next commands perform a number of useful tasks that will help with your editing. 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 program, as you did earlier.

Capitalization and lowercasing

The following MicroEMACS commands can automatically capitalize a word (that is, make the first letter of a word upper case), or make an entire word upper case or lower case.

<esc>C	Capitalize a word
<esc>L	Lowercase an entire word
<esc>U	Uppercase an entire word

To try these commands, do the following: First, move the cursor to the letter d of the word *different* on line 2. Type the *capitalize* command <esc>C. The word is now capitalized, and the cursor is now positioned at the space after the word. Move the cursor forward so that it is over the letter t in *rates*. Press <esc>C again. The word changes to *raTes*. 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 upper-case 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 **Different**. It does not matter if the cursor is directly over the **D** or at the space to its left; as you will see, this means 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 **DIFFERENT**, and the cursor has jumped to the space after the word.

Again, move the cursor to the left of the word **DIFFERENT**. Type the *lowercase* command `<esc>L`. The word has changed back to **different**. Now, move the cursor to the space at the beginning of line 3 by typing `<ctrl-N>` then `<ctrl-A>`. Type `<esc>L` once again. The character “*****” is not affected by the command, but the letter **U** 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.

The *uppercase* and *lowercase* commands stop at the first punctuation mark they meet *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*. In this example,

```
* use this
```

in line 3 now appears:

```
* us ethis
```

The space and the letter **e** have been transposed. 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 have that particular line in 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.

Return indent

`<ctrl-J>` Return and indent

You may often be faced with a situation in which, for the sake of programming style, you need many lines of indented text. After every line, you must return, then tab the correct number of times, then type your text. *Block indents* can be a time-consuming typing chore. The MicroEMACS `<ctrl-J>` command makes this task easier. When you type a file that has many lines of indented text, such as a C program, you can save many keystrokes by using the `<ctrl-J>` command. `<ctrl-J>` moves the cursor to the next line on the screen, and positions the cursor at the previous line's level of indentation.

To see how this works, first move the cursor to the line that reads

```
w3 *= 1.10:
```

Press `<ctrl-E>`, to move the cursor to the end of the line. Now, type `<ctrl-J>`.

As you can see, a new line opens up and the cursor is indented the same amount as the previous line. Type

```
/* Here is an example of auto-indentation */
```

This line of text begins directly under the previous line.

Word wrap

`<ctrl-X>F` Set 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 exit from MicroEMACS by typing `<ctrl-X><ctrl-C>`; then reinvoke MicroEMACS by typing

```
me cucumber
```

When MicroEMACS re-appears, 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 a few sections.) 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.

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 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 leave word wrap off, so you do not accidentally introduce carriage returns into your code.

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.

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 that 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 to one.

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. Before you continue, close the present file by typing `<ctrl-X> <ctrl-C>`; now, reinvoke the editor to edit the file **example1.c**, as you did before. The following sections will perform some exercises with this file.

Search forward

<code><ctrl-S></code>	Search forward incrementally
<code><esc>S</code>	Search forward with prompt

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 pointer `*msg`. Type the letters `*msg` one at a time, starting with `*`. The cursor has jumped to the first place that a `*` was found: at the second character of the first line. The cursor moves forward in the text file and the message at the bottom of the screen changes to reflect what you have typed.

Now type `m`. The cursor has jumped ahead to the letter `s` in `*msg`. Type `s`. The cursor has jumped ahead to the letter `g` in `*msg`. Finally, type `g`. The cursor is over the space after the token `*msg`. Finally, type `<esc>` to end the string. MicroEMACS will reply with the message

[Done]

which indicates that the search is finished.

If you attempt an incremental search for a 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 ***msgs**, MicroEMACS would move the cursor to the phrase ***msg**; when you typed 's', it would tell you

failing i-search forward: ***msgs**

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 [***msgs**]:

at the bottom of the screen. The word ***msgs** 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 **editing text**, then press the carriage return. Notice that the cursor has jumped to the period after the word **text** in the next to last line of your text. MicroEMACS searched for the words **editing text**, 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

<ctrl-R>	Search backwards incrementally
<esc>R	Search backwards with prompt

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 [**editing text**]:

at the bottom of your screen. The words in square brackets are the words you en-

tered earlier for the *search* command; MicroEMACS remembered them. If you wanted to search for **editing text** again, you would just press the carriage return. For now, however, type the word **program** and press the carriage return.

Notice that the cursor has jumped so that it is under the letter **p** of the word **program** in line 1. 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

<ctrl-G> Cancel a search 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 for you to initiate 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** appear at the bottom of your screen, and the cursor returns to where it 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 **m**. When the cursor moves to the **m** in **simple**, type **<ctrl-G>**. The bell will ring, and your cursor will be returned to the top of the file, which is where you began the search.

Search and replace

<esc>% Search and replace

MicroEMACS also gives you a powerful 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, move to the top of the text file by typing `<esc><`; then type `<esc>%`. You will see the following message at the bottom of your screen:

Old string:

As an exercise, type `msg`. MicroEMACS will then ask:

New string:

Type `message`, and press the carriage return. As you can see, MicroEMACS jumps to the first occurrence of the string `msg`, and prints the following message at the bottom of your screen:

Query replace: `[msg] -> [message]`

MicroEMACS is asking if it should proceed with the replacement. Type a carriage return: this displays the options that are available to you at the bottom of your screen:

`<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 `msg` with `message`, and move the cursor to the next occurrence of `msg`.

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 `msg` with `message` 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 `msg` with `message`, and the cursor will jump to the next place where `msg` occurs.

Typing an exclamation point `!` tells MicroEMACS to replace all instances of the old string with the new string, without checking with you any further. In this example, typing `!` will replace all instances of `msg` with `message` without further queries from MicroEMACS.

Finally, typing `<ctrl-G>` aborts the search and replace procedure.

Saving text and exiting

This set of basic editing commands allows you to save your text and exit from the MicroEMACS program. They are as follows:

<code><ctrl-X> <ctrl-S></code>	Save text
<code><ctrl-X> <ctrl-W></code>	Write text to a new file
<code><ctrl-Z></code>	Save text and exit
<code><ctrl-X> <ctrl-C></code>	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

<code><ctrl-X> <ctrl-W></code>	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 **sample**. MicroEMACS will reply:

[Wrote 23 lines]

The 23 lines of your text have been copied to a new file called **sample**. The status line at the bottom of your screen has changed to read as follows:

```
-- ST MicroEMACS V1.2 -- example1.c -- File: sample -----
```

The significance of the change in file name will be discussed in the second half of this tutorial.

Before you copy text into a new file, be sure that you have not selected a file name that is already being used. If you do, whatever is 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 editor. To see how this works, watch the bottom line of your terminal carefully and type `<ctrl-Z>`. The MicroEMACS has saved your text, and now you can issue commands directly to **msh**.

Advanced editing

The second half of this tutorial introduces the advanced features of MicroEMACS.

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 TOS without having to exit from MicroEMACS.

Before beginning, however, you must prepare a new text file. Type the following command to **msh**:

```
me example2.c
```

In a moment, **example2.c** will appear on your screen, as follows:

```
/* Use this program to get better acquainted
 * with the MicroEMACS interactive screen editor.
 * You can use this text to learn some of the
 * more advanced editing features of MicroEMACS.
 */

#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)
            fputc(ch, stdout);
    }

    else
        printf("Cannot open %s.\n", filename);
    fclose(fp);
}
```

Arguments

Most of the commands already described in this tutorial can be used with *arguments*. An argument is a subcommand that tells MicroEMACS to execute a command a given number of times. With MicroEMACS, arguments are introduced by typing `<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 *'* at the beginning of line 1, to being over the *space* 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 beginning of line 2 to the letter *m* of the word *main* 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 four. 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 over the *i* of the word *file* in the *printf* statement in line 11.

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>D`. MicroEMACS has deleted three words to the right.

Arguments can be used to increase the power of any *cursor movement* command, or any *kill* or *delete* command. The sole exception is `<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> 5 <ctrl-D>`. The word **Use** has disappeared. Move the cursor to the right until it is between the words **better** and **acquainted**, then type `<ctrl-Y>`. The word **Use** 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.

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; then return to MicroEMACS to edit the file **example2.c**, as you did earlier.

Now, look at the status line at the bottom of your screen. It should appear as follows:

```
-- ST MicroEMACS V1.2 -- example2.c -- File: example2.c -----
```

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, which 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. These include the following:

<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 **<return>**. Two things happen: First, MicroEMACS writes the message

[Wrote 21 lines]

at the bottom of your screen. Second, the name of the file shown on the status line has changed from **example2.c** to **junkfile**. MicroEMACS is reminding you that your text is now being saved into the file **junkfile**.

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 :

Type **example2.c** and **<return>**. 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 **example2.c**.

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 **example2.c** and saved it, and now wished to edit **example1.c**. You could exit from MicroEMACS, then re-invoke MicroEMACS for the file **example2.c**, but this is cumbersome. A more efficient way is to simply replace the **example2.c** in your buffer with **example1.c**.

Type **<ctrl-X><ctrl-R>**. MicroEMACS replies with the prompt:

Read file:

Type **example1.c**. Notice that **example2.c** has rolled away and been replaced with **example1.c**. Now, check the status line. Notice that although the name of the *buffer* is still **example2.c**, the name of the *file* has changed to **example1.c**. You can now edit **example1.c**; when you save the edited text, MicroEMACS will copy it back into the file **example1.c** — unless, of course, you again 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 **example1.c** with **example2.c**. Type the *replace* command **<ctrl-X><ctrl-R>**; when MicroEMACS replies by asking

Read file:

at the bottom of your screen, type **example2.c**.

You should now have the file **example2.c** read into the buffer named **example2.c**. Now, type the *visit* command **<ctrl-X><ctrl-V>**. MicroEMACS replies with the prompt

Visit file:

at the bottom of the screen. Now type **example1.c**. Several things happen. **example2.c** rolls off the screen and is replaced with **example1.c**; the status line changes to show that both the buffer name and the file name are now **example1.c**; and the message

[Read 23 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>**. **example2.c** 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 **example2.c**. **example1.c** scrolls off your screen and is replaced by **example2.c**, 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 to 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 **example2.c** 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 would be 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 **example1.c**. When **example1.c** has rolled onto your screen, type the *yank back* command **<ctrl-Y>**. The line you killed in **example2.c** has now been moved into **example1.c**.

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>`. The status line has moved up to the middle of the screen, and the bottom half of your screen has been replaced with the following display:

C	Size	Lines	Buffer	File
-	----	-----	-----	-----
*	655	24	example1.c	example1.c
*	403	20	example2.c	example2.c

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 `example1.c` by typing the *kill* command `<ctrl-K>`. Then type `<ctrl-X><ctrl-B>` once again. The size of the buffer `example1.c` has been reduced from 657 characters to 595 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 section.

Renaming a buffer

One more point must be covered with the *visit* command. TOS will not allow you to have more than one file with the same name. 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 as the file you are visiting. 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 named `sample`, but you already had a *buffer* named `sample`. MicroEMACS would stop and give you this prompt at the bottom of the screen:

Buffer name:

You would type in a name for this new buffer. This name could not duplicate the name of any existing buffer. MicroEMACS would then read the file **sample** into the newly named buffer.

Delete a buffer

If you wish to delete a buffer, simply type the *delete buffer* command **<ctrl-X>K**. This command will allow you to delete only a buffer that is hidden, not one that is being displayed.

Type **<ctrl-X>K**. MicroEMACS will give you the prompt:

Kill buffer:

Type **example2.c**. Because you have changed the buffer, MicroEMACS asks:

Discard changes [y/n]?

Type **y**. Then type the *buffer status* command **<ctrl-X><ctrl-B>**; the buffer status window will no longer show the buffer **example2.c**. Although the prompt refers to *killing* a buffer, the buffer is in fact *deleted* and cannot be yanked back.

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 reinvolve MicroEMACS for the text file **example1.c** as you did earlier.

Now, copy **example2.c** 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 **example2.c**. MicroEMACS will read **example2.c** into a buffer, and show the message

[Read 21 lines]

at the bottom of your screen.

Finally, copy a new text, called **example3.c**, into a buffer. Type **<ctrl-X><ctrl-V>** again. When MicroEMACS asks which file to visit, type **example3.c**. The message

[Read 123 lines]

will appear at the bottom of your screen.

The first screenful of text will appear as follows:

```
/*
 * Factor prints out the prime factorization of numbers.
 * If there are any arguments, then it factors these.  If
 * there are no arguments, then it reads stdin until
 * either EOF or the number zero or a non-numeric
 * non-white-space character.  Since factor does all of
 * its calculations in double format, the largest number
 * which can be handled is quite large.
 */
#include <stdio.h>
#include <math.h>
#include <ctype.h>

#define NUL '\0'
#define ERROR 0x10 /* largest input base */
#define MAXNUM 200 /* max number of chars in number */

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

-- ST MicroEMACS V1.2 -- example3.c -- File: example3.c -----

At this point, example3.c is on your screen, and example1.c and example2.c
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*.

Creating windows and moving between them

A *window* is a portion of your screen that is set aside and can be manipulated independently from the rest of the screen. The following commands let you create windows and move between them:

<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

The best way to grasp how a window works is to create one and work with it. To begin, 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 `example3.c` 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. Neither `<ctrl-X>2` nor `<ctrl-X>1` can be used with arguments.

Now, type the *one window* command `<ctrl-X>1`. 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:

<code><ctrl-X>Z</code>	Enlarge window
<code><ctrl-X><ctrl-Z></code>	Shrink window

To see how these work, first 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: 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:

<code><ctrl-X><ctrl-N></code>	Scroll down
<code><ctrl-X><ctrl-P></code>	Scroll up
<code><esc>!</code>	Move within 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>`. 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.) 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>`. 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>>`, then typing the *previous line* command `<ctrl-P>` four times. Now, 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. 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 **example1.c**, **example2.c**, and **example3.c**, although your screen is displaying only **example3.c**. 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 `example1.c`. The text in your present window will be replaced with `example1.c`. The command line in that window has changed, too, to reflect the fact that the buffer and the file names are now `example1.c`.

Moving and copying text among buffers

It is now very easy to copy text among buffers. To see how this is done, first kill the first line of `example1.c` 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.

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 positioned 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`. Exit from MicroEMACS by typing the *quit* command `<ctrl-X><ctrl-C>`.

Keyboard macros

Another helpful feature of MicroEMACS is that it allows you to create a *keyboard macro*.

Before beginning this section, reinvoke MicroEMACS to edit **example3.c** as you did earlier.

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:

MAXNUM

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.

Whenever you exit from MicroEMACS, your keyboard macro is erased, and must be retyped when you return.

Sending commands to TOS

The only remaining 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 TOS, and then resume editing without affecting your text in any way.

The command `<ctrl-X>!` allows you to send *one* command line (one command, or several commands plus separators) to the operating system. To see how this command works, type `<ctrl>!`. 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 example1.c
```

It doesn't matter that you are already editing `example1.c`. MicroEMACS will simply copy the `example1.c` 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. However, 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.

Now type `<ctrl-X><ctrl-C>` to exit.

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 in which you try to compile, but the compiler produces error messages and aborts 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!\n")  
}
```

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 error, 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. 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 by typing a semicolon at the end of line 2. Close the file by typing `<ctrl-Z>`. `cc` will be invoked again automatically.

`cc` will continue to compile your program either until the program compiles without error, or until you exit from MicroEMACS by typing `<ctrl-U>` followed by `<ctrl-X><ctrl-C>`.

The MicroEMACS help facility

MicroEMACS has a built-in help function. With it, you can ask for information either for a word that you type in, or for a word over which the cursor is positioned. The MicroEMACS help file contains the bindings for all library functions and macros included with Mark Williams C.

For example, consider that you are preparing a C program and want more information about the function `fopen`. Type `<ctrl-X>?`. At the bottom of the screen will appear the prompt

Topic:

Type `fopen`. MicroEMACS will search its help file, find its entry for `fopen`, then open a window and print the following:

```
fopen - Open a stream for standard I/O
#include <stdio.h>
FILE *fopen (name, type) char *name, *type;
```

If you wish, you can kill the information in the help window and copy it into your program to ensure that you prepare the function call correctly.

Consider, however, that you are checking a program written earlier, and you wish to check the call to `fopen`. Simply move the cursor until it is positioned over one of the letters in `fopen`, then type `<esc>?`. MicroEMACS will open its help window, and show the same information it did above.

To erase the help window, type `<esc>2`.

Where to go from here

For a complete summary of MicroEMACS's commands, see the entry for **me** in the Lexicon.

The next section introduces **make**, a utility is helpful in building and maintaining large programs. After that come sections that introduce the Mark Williams resource tools: **resource**, the Mark Williams resource editor; **rescomp**, the resource compiler; and **resdecom**, the resource decompiler.

Section 5: make Programming Discipline

make is a utility that relieves you of the drudgery of building a complex C program.

How does make work?

To understand how **make** works, it is first necessary to understand how a C program is built: how Mark Williams C takes you from the C source code that you write to the executable program that you can run on your computer.

The file of C source code that you write is called a *source module*. When Mark Williams C compiles a source module, it uses the C code in the source module, plus the code in the header files that the code calls to produce an *object module*. This object module is *not* executable by itself. To create an *executable file*, the object module generated from your source module must be handed to a linker, which links the code in the object module with the appropriate library routines that the object module calls, and adds the appropriate C runtime startup routine.

For example, consider the following C program, called **hello.c**:

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

When Mark Williams C compiles the file that contains C code shown above, it generates an object module called **hello.o**. This object module is not executable because it does not contain the code to execute the function **printf**; that code is contained in a library. To create an executable program, you must hand **hello.o** to the linker **ld**, which copies the code for **printf** from a library and into your

program, adds the appropriate C runtime startup routine, and writes the executable file called **hello.prg**. This third file, **hello.prg**, is what you can execute on your computer.

The term *dependency* describes the relationship of executable file to object module to source module. The executable program *depends* on the object module, the library, and the C runtime startup. The object module, in turn, depends on the source module and its header files (if any).

A program like **hello.prg** has a simple set of dependencies: the executable file is built from one object module, which in turn is compiled from one source module. If you changed the source module **hello.c**, creating an updated version of **hello.prg** would be easy: you would simply compile **hello.c** to create **hello.o**, which you would link with the library and the runtime startup to create **hello.prg**. Mark Williams C, in fact, does this for you automatically: all you need to do is type

```
cc hello.c
```

and Mark Williams C takes care of everything.

On the other hand, the dependencies of a large program can be very complex. For example, the executable file for the MicroEMACS screen editor is built from several dozen object modules, each of which is compiled from a source module plus one or more header files. Updating a program as large as MicroEMACS, even when you change only one source module, can be quite difficult. To rebuild its executable file by hand, you must remember the names of all of the source modules used, compile them, and link them into the executable file. Needless to say, it is very inefficient to recompile several dozen object modules to create an executable when you have changed only one of them.

make automatically rebuilds large programs for you. You prepare a file, called a **makefile**, that describes your program's chain of dependencies. **make** then reads your **makefile**, checks to see which source modules have been updated, recompiles only the ones that have been changed, and then relinks all of the object modules to create a new executable file. **make** both saves you time, because it recompiles only the source modules that have changed, and spares you the drudgery of rebuilding your large program by hand.

Try make

The following example shows how easy it is to use **make**.

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 section 1 of this manual.

To begin, **make** examines the time and date that TOS has stamped on each source file and object module. When you edit a source module, TOS marks it with the time at which you edited it. Thus, if a source module has a time that is *later* than that of its corresponding object module, then **make** knows that the source module was changed since the object module was last compiled and it will compile a new object module from the altered source module. If you do not reset the time on your system whenever you reboot, *every time*, some files will not have the correct date and time and **make** cannot work correctly.

To see how **make** works, try compiling a program called **factor**. It is built from the following files:

```
atod.c
factor.c
makefile
```

All three are included with your copy of Mark Williams C.

Use the **cd** command to shift into directory **src**.

Now, type **make**. **make** will begin by reading **makefile**, which describes all of **factor**'s dependencies. It will then use the **makefile** description to create **factor**. The following will appear on your screen:

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

Each of these messages describes an action that **make** has performed. The first shows that **make** is compiling **factor.c**, the second shows that it is compiling **atod.c**, and the third shows that it is linking the compiled object modules **atod.o** and **factor.o** to create the executable file **factor.prg**.

When **make** has finished, the TOS prompt will return. To see how your newly compiled program works, type

```
factor 100
```

factor will calculate the prime factors of its argument **100**, and print them on the screen.

To see what happens if you try to re-make your file, type **make** again. **make** will run quietly for a moment, and then exit. **make** checked the dates and times of the object modules and their corresponding source modules and saw that the object modules had a time later than that of the source modules. Because no source module changed, there was no need to recompile an object module or relink the executable file, so **make** quietly exited.

To see what happens when one of the source modules changes, try the following. Use the MicroEMACS screen editor to open the file **factor.c** for editing. Insert the following line into the comments at the top, immediately following the /*:

```
* This comment is for test purposes only.
```

Now exit. Type **make** once again. This time, you will see the following on your screen:

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

Because you altered the source module **factor.c**, its time was later than that of its corresponding object module, **factor.o**. When **make** compared the times of **factor.c** and **factor.o**, it noted that **factor.c** had been altered. It then recompiled **factor.c** and relinked **factor.o** and **atod.o** to re-create the executable file **factor.prg**. **make** did not touch the source module **atod.c** because **atod.c** had not been changed since the last time it was compiled.

As you can see, **make** greatly simplifies the construction of a C program that uses more than one source module.

Essential make

Although **make** is a powerful program, its basic features are easy to master. This section will show you how to construct elementary **make** scripts.

The makefile

When you invoke **make**, it searches the directories named in the environmental variable **PATH** for a file called **makefile**. As noted earlier, the **makefile** is a text file that describes a C program's dependencies. It also describes the type of program you wish to build, and the commands for building it.

A **makefile** has three basic parts.

First, the **makefile** describes the executable file's dependencies. That is, it lists the object modules needed to create the executable file. The name of the executable file is always followed by a colon ':' and then by the names of files from which the target file is generated.

For example, if the program **feud.prg** is built from the object modules **hatfield.o** and **mccoy.o**, you would type:

```
feud.prg:    hatfield.o mccoy.o
```

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**.

Second, the **makefile** holds one or more *command* lines. The command line gives the command to compile the program in question. The only difference between a **makefile** command line and an ordinary **cc** command is that a **makefile** command line *must* begin with a space or a tab character.

For example, the **makefile** to generate the program **feud.prg** must contain the following command line:

```
cc -o feud.prg hatfield.o mccoey.o
```

For a detailed description of the **cc** command and its options, refer to the entry for **cc** in the Lexicon.

Third, the **makefile** lists all of the header files that your program uses. These are given so that **make** can check if they were modified since your program was last compiled. For example, if the program **hatfield.c** used the header file **shotgun.h** and **mccoey.c** used the header files **rifle.h** and **pistol.h**, the **makefile** to generate **feud.prg** would include the following lines:

```
hatfield.o: shotgun.h
mccoey.o: rifle.h pistol.h
```

Thus, the entire **makefile** to generate the program **feud.prg** is as follows:

```
feud.prg: hatfield.o mccoey.o
    cc -o feud.prg hatfield.o mccoey.o

hatfield.o: shotgun.h
mccoey.o: rifle.h pistol.h
```

A **makefile** may also contain *macro definitions* and *comments*. These are described below.

Building a simple makefile

The program **factor.prg** is built from two source modules, **factor.c** and **atod.c**. No header files are used. The **makefile** contains the following two lines:

```
factor.prg: factor.o atod.o
    cc -f -o factor.prg factor.o atod.o -lm
```

The first line describes the dependency for the executable file **factor.prg** by naming the two object modules needed to build it. The second line gives the command needed to build **factor.prg**. The option **-lm** at the end of the command line tells **cc** that this program needs the mathematics library **libm** when the program is linked. No header file dependencies are described because these programs use no header files.

Comments and macros

You can embed comments within a **makefile**. A *comment* is a line of text that is ignored; this lets you “document” the file, so that whoever reads it will now know what it is for. **make** ignores all lines that begin with a pound sign ‘#’. 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.
# "-f" lets you use printf for floating-point numbers.
```

```
factor: factor.o atod.o
        cc -f -o factor.prg factor.o atod.o -lm
```

Anyone who reads this file will know immediately what it is for by looking at the comments.

make also lets you define macros within your **makefile**. A *macro* is a symbol that represents 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
```

Thereafter, when you use the symbol in your **makefile**, it must begin with a dollar sign ‘\$’ and be enclosed within parentheses.

Macros 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.
# "-f" lets you use printf for floating-point numbers.
```

```
OBJ = factor.o atod.o
factor: $(OBJ)
        cc -o factor.prg $(OBJ) -lm
```

The macro **OBJ** is used in this **makefile**. If you use a macro that has not been

defined, **make** substitutes an empty string for it. The use of a macro makes sense when generating large files out of a dozen or more source modules. You avoid retyping 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 wasteful recompiling of 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 object module **factor.o** was generated on March 16, 1987, 10:52:47 A.M., and the source module **factor.c** was modified on March 20, 1987, at 11:19:06 A.M., **make** will know that **factor.c** needs to be recompiled because it is *younger* than **factor.o**.

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 not work correctly.

Use the command **date** to reset the date. **date** is described 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) $(LFLAGS)

$(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 section. 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. 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.ttp**) 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. The macro **LFLAG** must *follow* the the names of the files to be compiled. 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 **make** is controlled by your **makefile**, you can also control **make** by using command line options. These allow you to alter **make**’s activity without having to edit your **makefile**.

Options must follow the command name on the command line and begin with a hyphen, ‘-’, using the following format. The square brackets merely indicate that you can select any of these options; do *not* type the brackets when you use the **make** command:

```
make [ -dinprst ] [ -f filename ]
```

Each option is 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 use the file **smith** rather than **makefile**. If you do not use this option, **make** searches the directories named in the environmental variable **PATH**, and then the current directory for a file entitled **makefile** to execute.

-i (ignore errors) **make** ignores error returns from commands and continues 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.
- r (rules) **make** does not use the default macros and commands from **\$LIBPATH\mmacros** and **\$LIBPATH\mactions**. These files will be described below.
- s (silent) **make** does not print each command line as it is executed.
- t (touch) **make** changes the modification time of each executable file and object module to the current time. This suppresses recreation of the executable file, and recompilation of the object modules. Although this option is used typically after a purely cosmetic change to a source module 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 "CSD=-VCSD"
```

tells **make** to run in the *no execution* mode, reading the file **smith** instead of **makefile**, and defining the macro **CSD** to mean **-VCSD**.

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. In the above example, you can define a command line as follows:

```
cc $(CSD) example.c
```

When you define the macro **CSD** on the command line, then the program is compiled using the **-VCSD** option, which creates an executable that can be debugged with **csd**, the Mark Williams C Source Debugger. If the macro is not set, however, then it is simply skipped when the command line is executed, and the program is compiled in the usual manner.

Another command-line feature is the ability to change the name of the *target file* on the command line. Normally, the target file is the executable file that you wish to create, although, as will be seen, it does not have to be. As will be discussed below, 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**. The command

```
make atod.o
```

with the **makefile** outlined above would produce the following **cc** command line:

```
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 is discussed below.

Advanced make

This section describes some of **make**'s advanced features. For most of your work, you will not need these features; however, if you create an extremely complex program, you will find them most helpful.

Default rules

The operation of **make** is governed by a set of *default rules*. These rules were designed to simplify the compilation of a typical program; however, unusual tasks may require that you bypass or alter the default rules.

To begin, **make** uses information from the files **mmacros** and **mactions** to define default macros and compilation commands. **make** looks for these files in the directories named in the environmental variable **LIBPATH**. **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 object module **atod.o** from the source module **atod.c** with the command

```
cc -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.

make's files **mmacros** and **mactions** use pre-defined macros to increase their scope and flexibility. These are as follows:

- \$<** This stands for the name of the file or files that cause the action of a default rule. For example, if you altered the file **atod.c** and then invoked **make** to rebuild the executable file **factor.prg**, **\$<** would then stand for **atod.c**.
- \$*** This 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 **\$*** work *only* with default rules; these macros will not work in a **makefile**.
- \$?** This stands for the names of the files that cause the action and that are younger than the target file.
- \$@** This stands for the target name.

You can use the macros **\$?** and **\$@** 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 a default command that describes how to build additional kinds of files:

- **AS** and **ASFLAGS** call the *assembler* to assemble **.o** files out of source modules written in assembly language rather than C.

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 modules **factor.c** and **atod.c** exist: **factora.c** plus **atoda.c**, and **factorb.c** plus **atodb.c**. The **makefile** would appear as follows:


```
OBJ1 = factor.a 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 **factor.a** or **atoda.o** is younger than **factor.prg**. (2) If either one is, regenerate **factor.prg** using this version of these files. (3) If neither **factor.a** nor **atoda.o** is younger than **factor.prg**, then check to see if either **factorb.o** or **atodb.o** is younger than **factor.prg**. (4) If either of them is, 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.

You cannot target a file in both a single-colon and a double-colon target line.

Alternative uses

make is a program that helps you construct complex things from a number of simpler things.

make usually is used to build complex C programs: the executable file is made from object modules, which are made from source modules and header files. However, **make** can be used to create any type of file that is constructed from one or more source modules. For example, an accountant can use **make** to generate monthly reports from daily inventories: all the accountant has to do is prepare a **makefile** that describes the dependencies (that is, the name of the monthly report they wish to create and the names of the daily inventories from which it is created), and the command required to generate the monthly report. Thereafter, to recreate the report, all the accountant has to do to generate a monthly report is type **make**.

In another example, the **makefile** can trigger program maintenance commands. For example, the target name **backup** might define commands to copy source modules to another directory; typing **make backup** saves a copy of the source modules. Similar uses include removing temporary files, building archives, executing test suites, and printing listings. 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.prg".
# "factor.prg" 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
SRC = factor.c atod.c

factor: $(OBJ)
    cc -o factor $(OBJ) -lm

# program to print all the updated source modules
# used to generate the program "factor.prg"

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.prg**, 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.

The target name **.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** 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 a later section of this manual.

Exit status

make returns a status of zero if it succeeds and **-1** if an error occurs.

Where to go from here

make is summarized in the Lexicon. Look there for more information about how to use it with C programs.

The next two sections introduce the Mark Williams resource tools: **resource**, the resource editor; **rescomp**, the resource compiler; and **resdecomp**, the resource de-compiler.

Section 6:

Introduction to the Resource Editor

This section introduces **resource**, the Mark Williams Resource Editor. **resource** simplifies the creation of GEM icons, menus, dialogue boxes, forms, and alerts for your program.

It is difficult to design an effective interface for a computer program. You must decide which elements to include and how they fit together. Implementing your design can be even more difficult. Drawing the objects on graph paper, counting character cells, and keeping track of spatial relationships are only the beginning. You must also keep track of the genealogical relationships for all the elements within the object tree and set pointers for each object correctly.

resource streamlines this editing process. You simply position objects on the screen and edit images, strings, forms, and menus as you go. **resource** keeps track of all tree relationships, leaving you free to concentrate on creating the interface for your application.

How resource works

resource encodes objects that you display and manipulate on the editor's desktop. **resource** sets the X and Y coordinates, the width, and the height of each object, as well as its relative position within its object tree. It lets you name each object and writes a header file that contains those names so that you can reference them in your program.

In addition to the C header file, **resource** produces two other files that contain information that your application program will use to reproduce the interface you have created. One file, with the suffix **.rsc**, is the resource file called by your application program. The other is a "name and type" definition file with the suffix **.rsd**. The definition file is used only by **resource** and by the resource decompiler

resdecom. It is not used by the application programmer.

Planning your resource

Most programs present the user with information in the form of text, and expect the user to type commands from the keyboard. These interfaces are easy to write and manipulate, but may be difficult for the user to learn.

The move away from text-based interface began with the invention of the *menu*. In its crudest form, the menu is simply a list of choices, each of which is labelled with one character. This list is printed on the screen, and the user indicates his preference by typing the character that corresponds to his selection.

With a graphics interface, the user can use *windows*, *icons*, and *menus* to pass information to the program. An item is selected by maneuvering a pointer to it, usually by moving a mouse, roller-ball, or joystick. In this way, the user can see graphic representations of the parts of the program.

Designing an interface

When you design a graphics interface, you must help the user interact with your program in the most straightforward way possible. The program itself will usually dictate the graphics tools to use.

For example, your program may require that the user answer the question, "Do you really want to quit?" There are only two possible answers: yes or no. Under GEM, you can gather this information easily by creating an *alert box*. An alert box holds a string that poses a question, such as "Do you really want to quit?" It would also contain two *buttons*, one labeled "Quit" and one labeled "Continue". The user clicks the appropriate button to indicate his choice.

The following sections describe some of the situations in which a given resource element is useful.

Buttons and radio buttons

Buttons allow the user to select from a number of alternatives. *Radio buttons* together form a bank of buttons of which only one can be selected. If a second button is selected, then the first button is un-selected. The name "radio button" comes from the bank of buttons on an automobile radio: when a button is punched, the button that had been punched pops out.

Text input

Text input is accepted from the keyboard. A program normally uses text to accept information for which there are too many alternatives to encode in buttons or menus, such as the name of a file or the user's name.

Icons

An icon usually is used to represent an object in memory or a part of the computer system itself. For example, the GEM desktop uses a drawing of a garbage can to indicate the file-deletion utility. The garbage can is an unmistakable symbol; dragging something in the garbage can means that you are throwing it away.

Images

Images are used solely for decoration. Often they are used to help distinguish objects from one another. For example, if a program has five objects, each with four buttons, using images can help the user know instantly just which of the five objects he is dealing with.

Menus

A menu lists one or more alternatives from which the user can choose. A program may have several menus available; the title of each is displayed in the menu bar. To select an alternative, the user sweeps the mouse pointer over the appropriate menu title, which invokes menu; then he selects an alternative from the menu.

As a rule, menus are used in two situations. First, they are used at the beginning of a program to set the basic conditions of operation. For example, a game may use a menu to ask the user if he wants to play, examine the copyright notice, or quit.

Second, menus are used to let the user invoke certain alternatives at any point in the program. For example, a game may allow the user to turn off its sound effects at any point in the game, and the easiest way to allow the user to access this feature at his whim is to make it available through a menu.

Getting started

resource is designed to work in medium or high resolution. Many of its dialogues contain large amounts of information and will not work correctly in low resolution.

The editor also has the following limitations:

- The structure of a resource file limits it to 64 kilobytes.
- No text string can exceed 65 bytes.
- The colors of an object are limited to white, black, red, and green, and the thickness of its border to four rasters (inside and out).

The files required to use the editor are **resource.prg** and **resource.rsc**. Both should be copied into the same directory, and it should be one of the directories named in the environmental variable **PATH**.

To run **resource** from **msh**, the Mark Williams micro-shell, type:

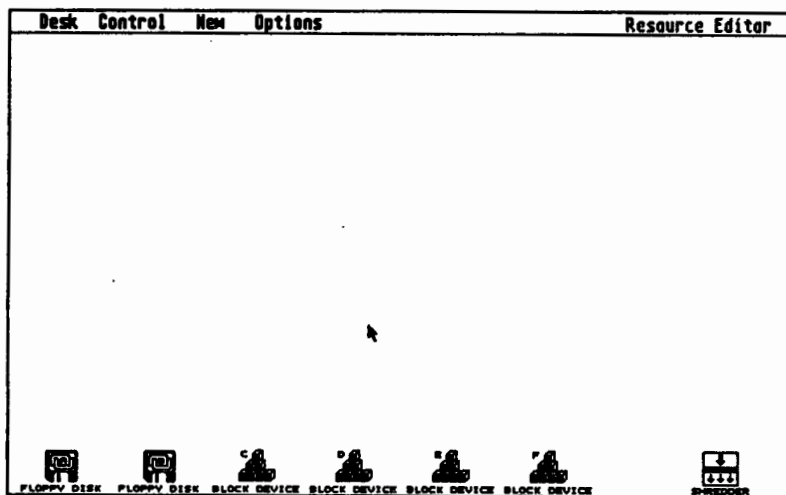
```
gem resource
```

at the prompt.

To invoke the Resource Editor from the GEM desktop, double-click the icon labelled **resource.prg**.

The resource desktop

The **resource** desktop resembles the GEM desktop. When you first invoke **resource**, the following desktop appears:

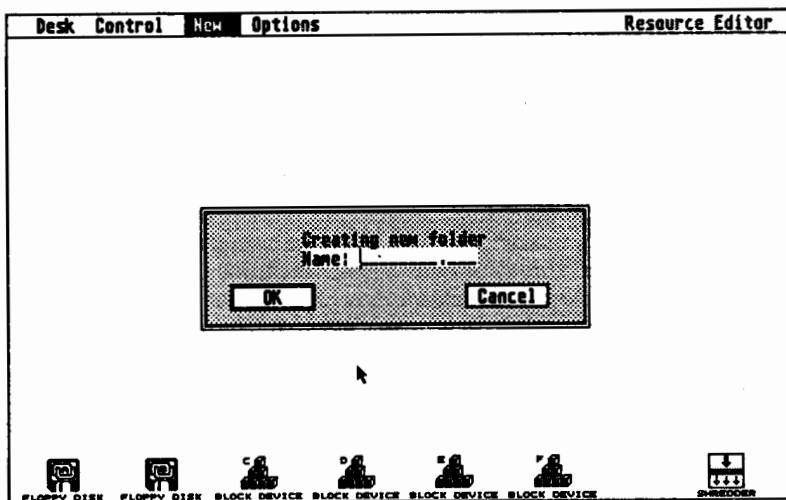


A menu bar extends across the top of the screen, and two types of icons are shown: In the above example, the first six icons are for file systems and the last represents the *Shredder*. Each file-system icon represents a RAM disk, a floppy disk, or a logical segment on a hard disk.

To select a file on one of the file systems, you must open a file-system window. To do so, double-click the appropriate icon for the file system you wish to access. The root directory for that file system will be displayed in a window. The file window displays two types of files: *folders* (also called *subdirectories*) and *resource sets*. To display the contents of a folder in the file window, double-click its icon. **resource** displays only folders and resource sets; it will not display other types of files.

You can also open a folder as a separate window. To do so, drag the folder out of the file window onto a free area of the desktop. **resource** creates a new file window and displays the contents of the folder in it.

You can create new folders by selecting the *File* entry in the **New** menu. Drag the folder icon that appears in the *File* partsbox to the file window or icon that represents the directory you want to contain the new folder. A *name dialogue* will appear, which prompts you to name the new folder:



The first action expected by a name dialogue is that you name the new folder.

The resource menu bar

resource's main menu appears across the top of the desktop. The following describes the items in it.

Desk Menu

This menu shows the desk accessories. When you sweep the mouse pointer under **Desk**, the dropped box contains a menu of desk accessories.

About the Resource Editor

Click this item to display a brief description of the product and the version number. Click the OK button once to return to the main menu.

Desk accessories

If you have desk accessories loaded in memory when you invoke **resource**, their names will appear here.

Control Menu

These menu items let you open and close windows, check file statistics, and exit **resource**. Sweeping the mouse pointer under this main menu item displays the following menu entries:

Open Clicking this item displays all of the folders and resource files in a selected file system or folder, or opens a new resource set. You can also open a file system or folder by double-clicking its icon.

Show information

This menu item may be used in three ways. If you single-click a file-system icon, then single-click **Show information**, a box will appear that shows the space statistics for that file system. When used in the same way on a resource stored on disk, this option displays the sizes and modification dates of the two files in the resource. You can also rename files from within their information box. The most important use of this feature is for a resource that has been loaded into memory; there, it displays the block counts and the size of the resource to be renamed.

Close Click this menu item to close the top window.

Close window

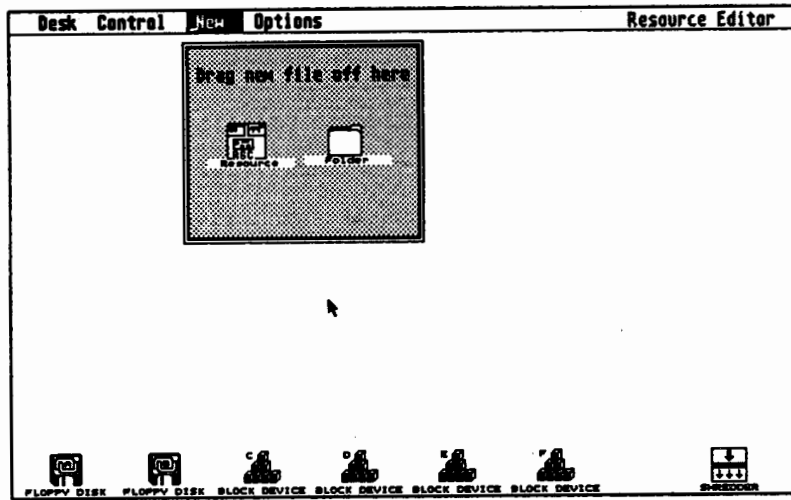
Click this item to close the top window entirely, even if clicking its close box, in the upper left corner of the window, would have taken it up to the next level.

Quit When you click this item, **resource** exits. You will return to the shell or the GEM desktop. *When you exit from resource, all information that has not been saved is thrown away.*

New Menu

The menu items under **New** create new folders and resources, and create new trees and objects.

File Use this to open a new folder or resource file. Clicking the *File* option opens the *File* partsbox. The *File* partsbox looks like this:



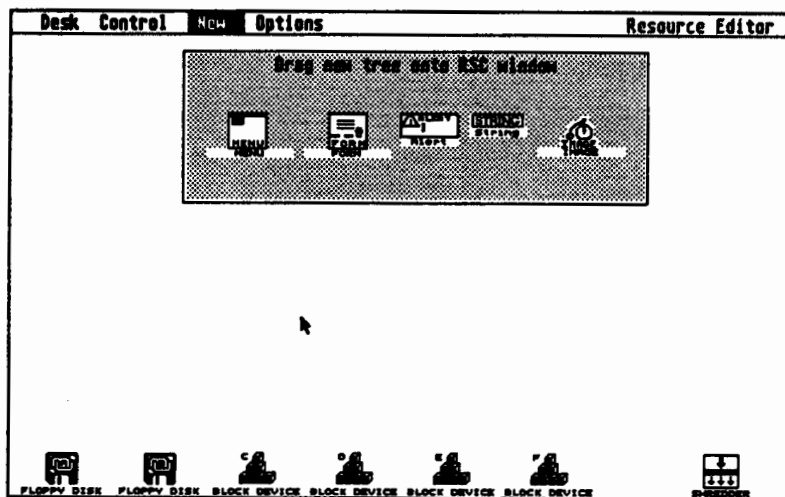
To open a new folder or resource, drag the appropriate icon onto an open area of the desktop.

Tree A resource consists of a number of discrete groups of information called *trees*. The Resource Editor recognizes five types of trees, although forms and menus, and alerts and strings are stored in the same way in the resource file.

Clicking the *Tree* entry under *New* opens the tree partsbox, where icons for the following tree types are displayed:

- form
- menu
- free string
- alert
- free image

The *Tree* partsbox looks like this:



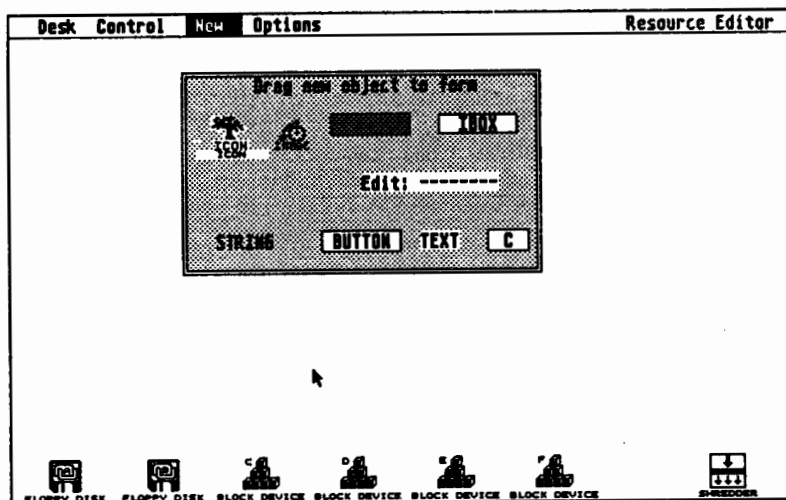
These trees are described in detail later in this section.

Object

An *object* is an AES data form that encodes an element to be displayed on the screen. When you click *Object*, the partsbox opens, displaying the following object types:

- icon
- image
- ibox
- editable string
- button
- text
- c (character)

The *Object* partsbox appears as follows:



Objects are linked together to form a tree.

For detailed information on these objects, see their separate headings later in this section.

Options Menu

The options menu allows you to set three options that modify the ways in which objects are manipulated. These options toggle with a click of the mouse pointer; when they are on, a tick mark appears in front of the selected menu entry.

Auto Snap

When you select Auto Snap, all moving and sizing operations are adjusted to the points of a character-sized grid within the parent object. The snap grid makes it easy to accurately align objects with other snapped objects. It also means that all snapped objects, except images and icons, are the same size and in the same place on medium- and high-resolution screens.

Auto Size

When Auto Size is enabled, the width of a **STRING** or **TEXT** object is automatically recalculated whenever the string is changed. When it is disabled, the width will only be changed if it must be enlarged to accommodate a longer text string.

Compatibility

Use compatibility mode when you run **resource** under high resolution and you want to design forms that can be used on medium or high resolution without having to modify them. The height of the ghost outline of icons or images is doubled when they are dragged. This shows you the height these objects will be on a medium resolution display so that you can make size allowances for them as you build your resource.

File operations

When you create a resource, the editor generates two files: the *resource file* and the *definition file*. The resource file, which has the suffix **.rsc**, is the resource file that your application program calls through the function **rsrc_load**. The definition file, which has the suffix **.rsd**, contains the names and types of the items you created for this resource. A resource icon in a file window represents this pair of files, which together form a resource set. Whenever you change a resource, both files are affected.

resource also produces a *header or include file* for each resource. The header file contains definitions that can be used by a C program to access the objects within a resource. A header file is named after its resource, and always has the suffix **.h**.

If something should happen to one of the files in a resource set, the icon that represents the resource set changes. Incomplete resource sets are marked to indicate which member of the set is missing. The letter 'N' appears in the upper right corner of the resource icon if the definition file is missing. The letter 'D' on the resource icon indicates that the resource file is missing. If you have a color monitor, the icon of an incomplete resource is drawn in red; the letters 'N' and 'D' tell you which file is missing from the set.

Display, copy, rename, and delete

The operations to *display*, *copy*, *rename*, or *delete* resource files work in much the same way as the same operations on the GEM desktop, with the following exceptions:

- Deleting a resource set deletes both of its files.
- Folders cannot be copied.
- Non-empty folders cannot be deleted.

resource will ask for verification of delete operations. Copies do not require verification.

Loading and saving

The desktop's backdrop represents the structures that **resource** is holding in memory. To load a resource in memory so that it can be edited, drag its icon from the file window onto a free area of the desktop.

To create a new resource, select the *File* entry from the **New** menu, and drag the resource icon from the *File* partsbox to a convenient clear area on the desktop. To save a resource, drag its icon from the desktop either into a file window or onto a file-system icon.

If you drag a resource onto the icon of the drive from which it was loaded, it will overwrite the old version unless you have renamed the resource. If the new resource set will overwrite an existing resource set of the same name, other than the original resource set from which it was loaded, you will be prompted to confirm that you want the files to be overwritten.

Be sure that you copy a new or edited resource set to a file system before you quit **resource**. It does not automatically save new or modified files before exiting. If you click **Quit** before you save your resource, you will lose any changes or additions you may have made to it.

Moving and copying trees and objects

Double clicking a resource icon on the desktop opens a *resource window*, which displays the trees that the resource contains. Several such windows can be opened for the same resource. Trees may be deleted, copied, or dragged to another resource. To a copy tree, move the mouse pointer over its icon and press either a shift key or the right mouse button along with the left mouse button.

To create a new tree, select *Tree* from the **New** menu and drag a tree icon from the partsbox into the resource window. A new tree is always added to the end of its group of trees. The groups correspond to the three fundamental types in the resource file: *forms* and *menus*, *alerts* and *free strings*, and *free images*. Unlike file icons, tree icons can be dragged only one at a time.

Adding a tree or clicking its icon invokes the *name dialogue* for that tree. The name dialogue allows you to rename a tree, and in some cases change its type and other global values. Selecting the edit button or double clicking, rather than single clicking, the tree item allows you to edit the tree's contents.

Trees

A resource consists of a number of discrete blocks of information called *trees*. **resource** recognizes five types of trees, although forms and menus, and alerts and strings are stored in the same way within a resource. When you select *Tree* on the New menu, **resource** displays the following selection of trees in the *Tree* partsbox:

- MENU
- FORM
- Alert
- String
- Image

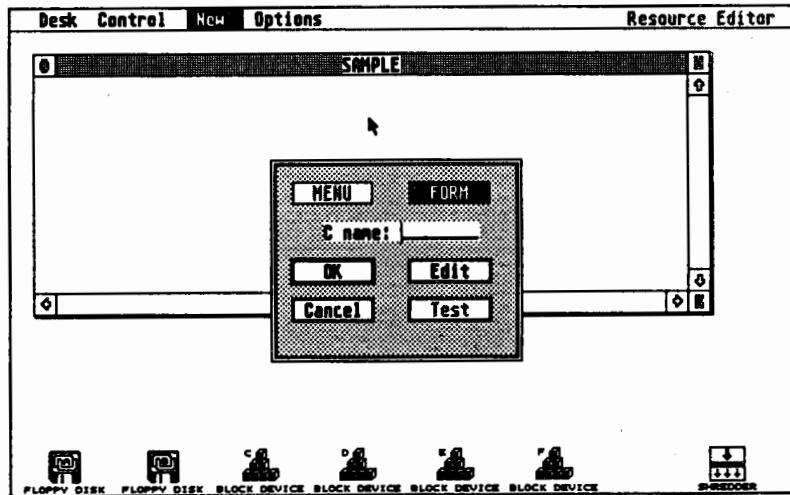
The different types of trees are described as follows:

Forms

A form is a hierarchy of rectangles, strings, and icons that represent a screen display. Each component of a form is called an *object*. Objects are arranged in a linked tree structure. The hierarchical structure of a tree, in turn, reflects how the objects you see on the screen are nested. See the Lexicon article on *object* for more information on relationships within object trees.

Editing forms

To create a new form, drag the *FORM* icon from the *Tree* partsbox to the resource window. When you release the mouse button, the following name dialogue will appear:



The name dialogue for forms allows you to change the C name of the form. The following options also appear on the name dialogue for forms.

menu

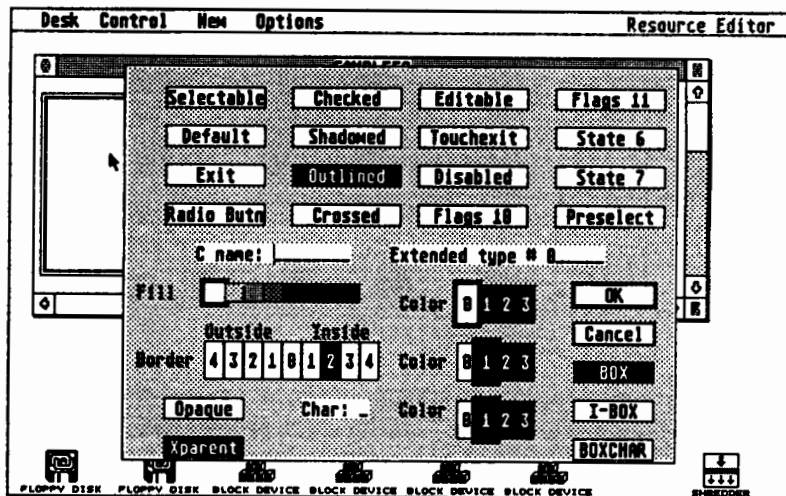
If the form can be treated as a menu, the button **MENU** will be enabled on the name dialogue. Usually this is the case only when you have previously changed type from **MENU** to **FORM** to edit the menu in non-standard ways (see *MENU*, below).

test

If the form is to be used as a dialogue, this option allows you to test some of its functions before you write the program. This button is only enabled if the form has at least one exit button. When you click **Test**, a copy of the form will be displayed as a dialogue. This allows you to confirm that the buttons and editable fields are working properly. This option is enabled only if the form has one or more *exit buttons*. An exit button is one for which the **exit** flag is set. When an exit button is selected, the return value from the AES function **form.do** is displayed in an alert box, together with the C name of the button, if any. You may then either resume the dialogue or terminate the test mode by clicking either the **Continue** or **Exit** on the test alert box.

edit

When you click **edit**, you can view or edit the form's structure. An *edit dialogue* appears on the screen. It displays the choices you can make about your form's appearance and contents. If your form is already open on the screen, double-clicking anywhere on the form itself will also invoke the edit dialogue. The edit dialogue for forms looks like this:



When **edit** is selected or the form's icon is double clicked, the resource window is replaced by a view of the form itself. The form display window scrolls over an area that is half again the screen size in each direction, which allows you to edit forms which are quite large. This area must contain exactly one root object inside; all other objects in the form must be nested within the root object. This outer object is almost always a **BOX** or **IBOX** type. **resource** will not allow an object to be added outside the root object. You can, however, discard and replace the root object. **resource** will not allow you to save forms that do not have a root object.

You can only display one form, menu, or alert at a time in any window. However, more than one such display either from the same or from different resources may be on the screen simultaneously and objects may be moved between them. For details about object trees and their organization, see the Lexicon entry for **object**.

MENU

A menu is much like a form, but is structured somewhat differently. To be processed correctly by **resource**, a menu must have a standard structure. This restriction may force you to edit your menu as a form if you want your menu to have some non-standard features. By clicking the menu icon, you can invoke the name dialogue for that menu, and choose the **FORM** or **MENU** buttons to change a tree from a menu to a form and back again.

A menu is a graphics form that is used extensively in programs that run under GEM. It is a specialized form of AES object that uses the structure **OBJECT** described in the header file **obdefs.h**. Because the structure of a form is already defined as an **OBJECT**, all menus must contain certain elements.

Each menu's object tree must be built in a special way. By design, the first (leftmost) title must be called **Desk**; it triggers the drop-down menu that names the available GEM desk accessories.

Editing menus

The name dialogue for menu trees is the same as that for forms. The menu-editing routines assume a menu of a standard format to make standard menus easy to edit. If you wish to put non-standard elements into your menus, you must change the tree type (at least temporarily) to *Form* and use the Form edit routines. It is convenient to use **HIDE** on those drop boxes you are not changing in this case. If, after you edit a menu as a form, the *Menu* button is enabled on the name dialogue, you may change it back to menu mode. If not, it is probably too changed to be a menu, and you will have to treat it as a form.

When you edit a menu, the resource window contents are replaced by the menu bar. Click a title to access the associated drop box. Click twice to edit the title itself.

When you test a menu, **resource**'s menu bar along the top of the screen is replaced with the menu you created. Selecting menu items now brings up an alert telling you the index selected and the name, if any.

All the objects that are manipulated while editing menus are strings, although those in the title bar are automatically typed as **TITLE** objects. **resource** automatically sizes and positions strings within the title bar and drop boxes when they are changed. There is no need to put extra spaces on the end of menu entries to equalize their length. These adjustments are automatic.

An entry that consists of a repeated non-alphanumeric character and is flagged **DISABLED** is regarded as a separator bar. Separator bars will be automatically stretched or contracted to the width of the drop box. This is determined by the longest text string in the drop box.

New title-bar entries are created by dragging a **STRING** type object into the title bar. New entries in the drop box under the title bar are added by dragging **STRINGs** into the drop-box display. These strings need only rough positioning. The new order is determined by the relationship between the center of the ghost object and the centers of the items that are already on the menu. When title strings are added or deleted, the corresponding drop boxes are created and destroyed. If you move a title string within the menu's title bar, it takes its drop box with it; however, if you copy or move a title string to another tree, it does not take its drop box with it.

Both single and double clicking a menu entry displays its edit dialogue.

For consistent appearance, a menu entry should always begin with two spaces, and a title entry with one space.

String

This is simply a text string and may consist of anything displayable, including the Atari ST graphics characters.

The free string edit is the simplest of all. The name dialogue contains the string.

You can turn a free string into a **STRING** object by dragging its icon into a form window. Conversely, when you drag a **STRING** object from a resource window, it becomes a free string.

Alert

This is a string in the format recognized by the GEM routine **form_alert**. It is stored as a free string. In C, it can be useful to treat these strings as format strings for **sprintf**, allowing variable data to be inserted before an actual **form_alert** call is made.

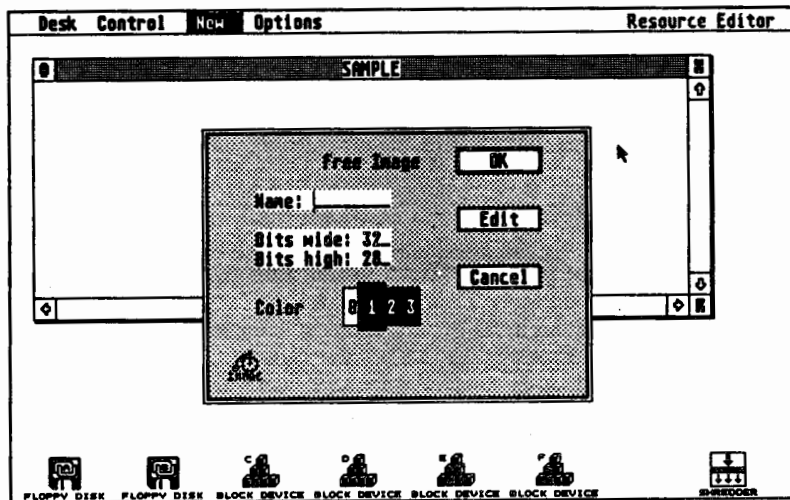
The icon displayed in an alert is adjusted from the name dialogue. You can edit the string and buttons within the alert by clicking the *Edit* button or by double-clicking the alert's icon.

The alert edit window behaves in many ways like the menu edit. As with the menu edit, you need only to position new and moved strings and buttons roughly; **resource** will reorganize the display to accommodate your revisions.

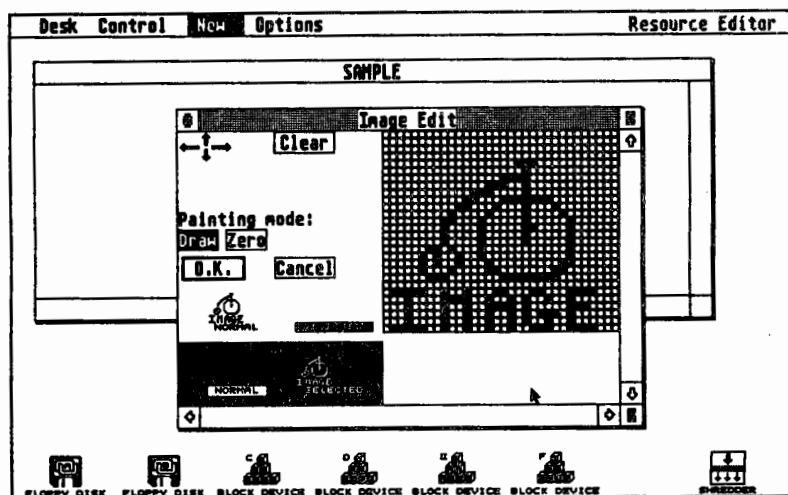
Image

A *free image* is a monochrome bit-image block with the structure **BITBLK**, which is defined in the file **obdefs.h**. Generally, to display a free image, a program writes its address into an **OBJECT** structure.

Editing a free image tree is similar to editing an image object. The name dialogue allows you to change its size and color. This name dialogue is different from the name dialogues for forms and menus. It looks like this:



Clicking the *edit* button or double-clicking the tree icon opens the icon/image edit dialogue. This second level edit dialogue appears as follows:



Objects

An *object* is any graphics element: a box, a string, a button, a menu title, etc. Objects are assembled into *trees*. The root object is located absolutely on the screen; the other objects are located relative to the root object. This is done so that all of the objects in the tree can be relocated easily, without having to compute a new absolute location for each when a tree is dragged from one location to another.

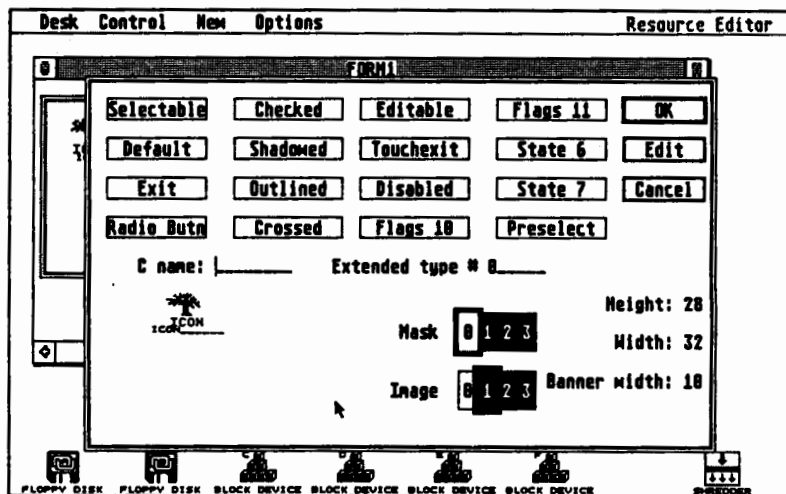
See the Lexicon article on **object** for a fuller discussion of objects and object trees.

New objects

To create a new object, single-click *Object* under *New* on the main menu bar. Drag the type of object you want from the **Object** partsbox and place it where you want it. For objects of type **FTEXT** or **BOXTEXT**, use the **TEXT** prototype and change its type by double-clicking that item to choose the editing dialogue. The editing dialogue will prompt you to change the type.

Icons and images

The first level icon edit dialogue looks like this:

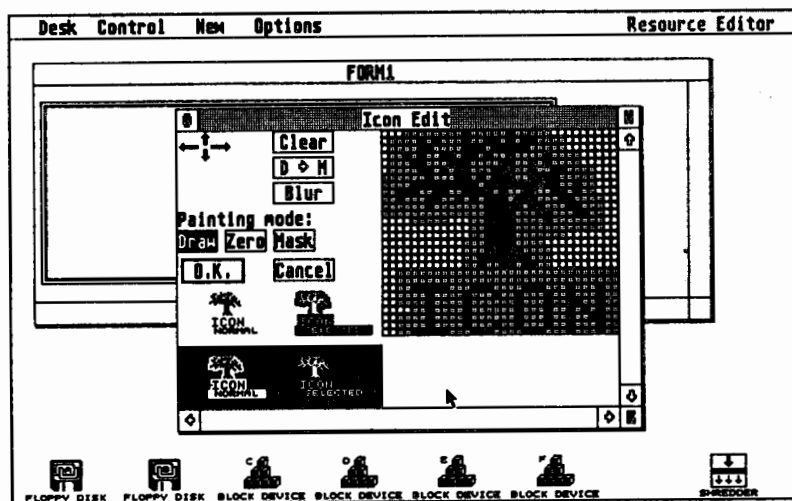


On the icon dialogue, the icon character and banner fields may be dragged to reposition them relative to the icon. The banner as displayed on the dialogue is left justified, whereas it is centered when the icon is displayed normally. The banner width field is in characters, and is limited to 20. Changes in the banner width are

not reflected on the dialogue until you terminate the dialogue and edit again.

Because the character and banner field are mouse sensitive for dragging, you cannot use the mouse to place the text cursor on them. Instead, use the down-arrow key to move to the banner field and use the keyboard to edit the banner.

For icons and images, a second level of editing is available by clicking on the *Edit* button on the first edit dialogue. The image edit dialogue appears as a window, and while it is active resource ignores all menu selections and clicks in other windows. The second level of image and icon editing looks like this:



To understand how the icon editor functions, it is necessary to understand how GEM treats icons.

An icon consists of two bit-images: one for a *mask* and the other for *data*. In a bit image, there is a one-to-one correspondence between bits and pixels: if a bit's value is one, the pixel is turned on, whereas if its value is zero, the pixel is turned off. When GEM draws an icon, it first draws the mask in the background color; it then draws the data image in the foreground color.

When an icon is selected, GEM simply reverses the foreground and background colors. Almost invariably, the foreground color is black and the background white. The color indices are written into the top byte of the field *ib_char*. Each pixel has, therefore, four possible settings, two of which have the same effect. The enlarged icon display on the right of the icon editor window gives a different grey level for each state.

<i>Mask</i>	<i>Data</i>	<i>Effect</i>	<i>Edit Display Color</i>
0	0	Transparent	White
0	1	Foreground	Dark grey
1	0	Background	Light grey
1	1	Foreground	Black

Clicking a display cell cycles the state around these four values. To set multiple cells to the same state, select the state on the buttons *Draw*, *Mask*, or *Zero*, and while holding the mouse button down wipe the mouse pointer across the relevant cells.

The other facilities on the icon edit window are as follows:

Clear Zero both data and mask.

D to M Copy the data image to the mask image.

Blur Sets all pixels in the mask to one if they are adjacent to an already set bit. The quick way to produce a simple mask for most icons is to do *D to M* and the *Blur* a few times to give a halo of mask around the image.

Arrows Clicking the arrows in the top left of the display shifts the image and mask one pixel in the indicated direction.

Cancel Abandon changes.

OK Process changes.

Editing an image is similar, except that no mask exists; therefore, *D to M* and *Blur* are not available.

You can resize an icon or image in their respective object dialogues. It is possible to make an image so large that the normal-sized view to the left of the image edit grid cannot display the entire image. The different state views may overlap in this case, and the image shown bears little similarity to what you have in the grid.

Ibox

An **IBOX** is a box that is invisible on the screen. You can use an **IBOX** to group elements together. For example, if you wanted to use two discrete sets of radio buttons on a form, you would put each group in its own **IBOX**.

Button

A **BUTTON** is a box with text in it. Text is limited to 20 characters.

Text

The *Template* field on the text dialogue is displayed and edited in an unusual way. The template field displays both template and validation characters. The validation characters appear in pink if you have a color monitor, in grey on monochrome. To insert validation characters, hold down the *Alternate* key while typing. The *Caps Lock* key has no effect on this field.

When you first create a template, you must type at least as many characters into the initial text object as there are validation characters. If you want an editable field in a dialogue to be initially empty, your program must put a null character into the first field position.

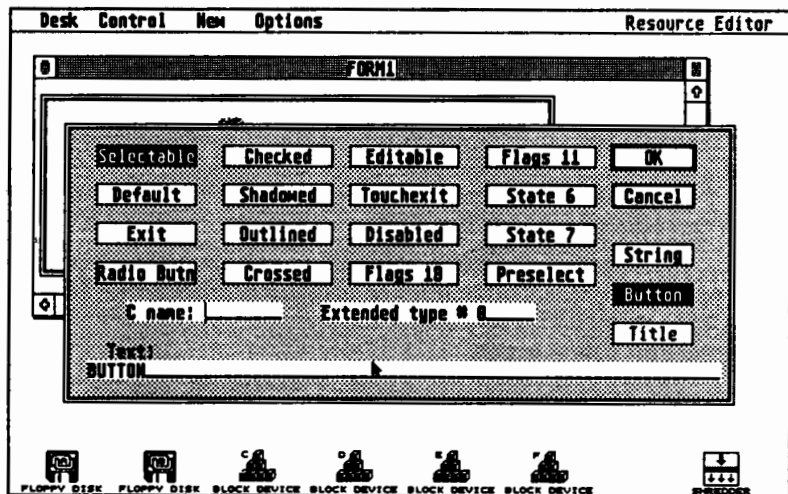
When you create a new editable field by changing a **TEXT** item to **FTEXT** or **FBOXTEXT**, remember to set the **EDITABLE** flag.

Editing objects

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_states; /* 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
```

When you double-click an object, a box called an *edit dialogue* will appear. The contents of the dialogue you see depends on the type of object you are editing. A sample object edit dialogue will appear on your screen similar to this:



Every dialogue has a group of 16 buttons in its upper left corner. These buttons allow you to set the `ob_flags` and `ob_states` fields in the **OBJECT** structure. The following describes these buttons and fields:

Selectable

Making the object you are editing *Selectable* means that clicking that object from within an application program sets the **SELECTED** bit in `ob_state`.

Default If no other object is selected, then this object is used by default.

Exit Clicking an exit object exits that part of the application program.

Radio Butn

Radio buttons are a set of buttons only one of which can be selected at any given time. For example, in a form where there are three radio buttons labeled **red**, **blue**, and **green**, only one of these buttons may be selected; if one is selected and then another of the group is clicked, then the first button is "unselected".

If you wish to have more than one set of radio buttons on a form, you must enclose each set within an **IBOX**.

Checked Selecting *Checked* when editing an object puts a tick mark in front of the object on the screen.

Shadowed

Tell GEM to draw a shadow around an object.

Outlined Tell GEM to draw a border around an object.

Crossed If you select *Crossed*, the object has an 'X' drawn over it. This works on rectangles only.

Editable Such an object can be edited by the user. This is used almost exclusively with string objects.

Touchexit

Click once to end the dialogue.

Disabled Draw in shading rather than solid. Any item that has been disabled cannot be selected, even if it is also marked as being **SELECTABLE**.

C name Give an object a name by which a C program can refer to it.

Extended type

The *Extended type #* field allows you to put any desired byte in the upper half of the object-type word. These numbers are ignored by GEM, so you can use them in your program as you see fit. This is useful when you want your program to process a group of objects in similar ways.

Preselect The *Preselect* button sets the **SELECTED** bit in **ob_state**.

Flags 10, Flags 11

Bits 10 and 11 of the **flags** member are not used by GEM. If you wish, you may store values in these bits for your own use.

State 6, State 7

Along with the flags used by GEM, these buttons give you access to two unused flag bits each in the members **ob_state** and **ob_flags**, to use as you see fit.

Manipulating objects

Much of the following applies not only to forms, but also to menus and alerts. It is described here because forms are the most complete and flexible trees. The operations for menus and alerts are subsets of those forms.

The Control key

When a parent object is entirely covered by its children, it may still be accessed by use of the control key. The control key works on all mouse operations within form windows, and causes the operations to reference the parent of the object on which the mouse rests instead of the object itself.

Moving an object

To move an object, place the mouse pointer on it and hold down the left mouse button. The object will disappear, and its “ghost”, a simple outline, will replace the object under the mouse pointer. Continue to hold the left mouse button, and drag the object to its new position.

You can move objects within the form, or to other form windows. In some cases, objects can be moved to other windows: strings and buttons can be moved to alerts, strings to menus, and even strings and images to resource windows (thus becoming free strings or free images).

You can also move objects onto the open desktop. You may then copy these objects from the desktop into various windows. This is convenient if a common element is to be used in several different forms.

When an object is moved within a form and the move causes it to have a new parent object, a dialogue box will appear and ask you to verify the move. When an object is moved to cover other objects, you will be given the option of having it adopt those objects as children to preserve their visual hierarchy.

Resizing

An object may be resized by putting the mouse pointer just inside its bottom right corner and *stretching* it in the same manner that windows are resized. The object may not be stretched outside the boundaries of its parent, nor may it be made too small for any contents (text, bit image data, or children).

Copying

An object may be copied by being dragged with either a shift key or the right hand mouse button held down. Holding down the right mouse button or shift key initiates the copy operation; holding down the left mouse button and moving the mouse drags the object.

If you use the right mouse button to initiate the copy operation, you must hold down both mouse buttons at the same time. The usual dragging “ghost” outline appears, but, unlike a simple drag, the icon of the object you are copying remains on the desktop. The object is not copied unless the ghost has been moved. An object can be copied anywhere it can be moved. Copies do not retain any of the names that may have been assigned to the original.

Deleting

To delete an object and all its children, position the mouse pointer on the object, hold down the left mouse button, and drag the object to the **Shredder**. When the mouse pointer is positioned over the **Shredder**, release the left mouse button.

Other functions on objects

There are a number of miscellaneous operations that may be performed on objects. To use these operations, click once on the object in question. The object will be inverted on the screen and a menu will appear just below the mouse pointer. You may click one of the options or cancel by clicking anywhere off the pop-up menu. Only functions meaningful for that particular object will be offered.

- | | |
|----------------|---|
| Edit | Edit the dialogue, as if it were double clicked. |
| Hide | Make an object and any of its children invisible by setting its HIDETREE flag. This is useful for getting at objects which may be underneath it. |
| Unhide | Make any hidden children of this object visible by clearing their HIDETREE flags. <i>Remember to unhide any objects you have hidden before you save the resource.</i> |
| Flatten | This removes an object but not its children. The children are transferred to the parent of the deleted object, but retain their relative screen position. |
| Snap | This command adjusts the position and size of an object to the nearest character grid position. <i>Snap</i> makes it easy to align objects. All snapped objects, except images and icons, are the same size and in the same place on medium- and high-resolution screens. |
| Sort... | Order the object's children according to their screen position. A dialogue allows the order to be chosen from a number of options. |
| Retype | You can change the type of certain objects without altering their appearance. With Retype , you can change between TEXT and STRING , BUTTON and BOXTEXT , and between ICON and IMAGE . However, you may lose some information when you change an object to a simpler type. |

Where to go from here

The following section introduces the other tools in the Mark Williams resource toolkit: **rescomp**, the resource compiler , and **resdecom**, the resource decompiler.

resdecom can decompile a resource that you create with **resource**, and create a file of resource-description language that you can edit by hand. **rescomp** can recompile a decompiled resource, or compile a resource that you write by hand.

Section 7:

Resource Compiler and Decompiler

Mark Williams C comes with a tool to help you create and maintain applications interfaces. These tools include a *resource editor* (which is described in the preceding section), a *resource compiler*, and a *resource decompiler*.

The resource compiler and decompiler let you create or change a resource file from a textual description. It is easy to track changes between versions of your resource by comparing decompiled resource files.

Using the compiler and decompiler

To create a GEM resource file from a resource description, use the resource compiler **rescomp.prg**. Its command line is as follows:

```
rescomp [-v][-s][-o rscfile][-d rsdfile][-h header] src
```

The **-v**, or verbose, option tells **rescomp** to report statistics to you as it compiles. The option **-o** allows you to name the three files that the compiler creates. If you do not use this option, **rescomp** names the resource after your description file.

When **rescomp** creates these files, it gives the resource file the suffix **.rsc**, the compiled resource description the suffix **.rsd**, and the C header file the suffix **.h**. **rescomp** creates these three files from the description file, that is, from *resfile*. If no file extension *[.ext]* is given the infile on the command line, the compiler looks for a file with the extension **.rdl**. For example:

```
rescomp -o sample example
```

Here, **rescomp** looks for the file **example.rdl**, and compiles a resource from the resource descriptions in that file. The resulting resource files are named **sample.rsc**, **sample.rsd**, and **sample.h**.

To decompile an existing resource set into a resource description file, use **resdecom.prg**. Its command line is as follows:

```
resdecom [-m][-o outfile[.ext]][-d resdef[.ext]] resfile[.ext]
```

The option **-d defile[.ext]** allows you to specify the name of the definition file.

resdecom looks for two files with the suffixes **.rsc** and **.rsd**. From them, it creates a resource description file, which it names after the resource files, and adds the extension **.rdl**. If you want your decompiled description file to have a name different from the *resfile*, or an extension other than **.rdl**, use the **-o** option.

For example, the command

```
resdecom -o foo.des bar
```

tells the compiler to look for the resource sources **bar.rsc** and **bar.rsd**, and to create the resource description **foo.des** from them. Likewise, the command

```
resdecom bar
```

tells **resdecom** to look for the files **bar.rsc** and **bar.rsd** and create a description file named **bar.rdl**.

Language description

When you use resource editor **resource** to build a resource, you must begin by establishing an object tree. Then, you must order objects within the tree. The resource compiler expects you to describe resources in the same way: from the root object out. You must describe a root object, then tell the compiler where to place its child objects in relation to it. You must also indicate the level of child objects as you want them nested on the screen.

Tree and object descriptions

Like any compiler, **rescomp** expects you to communicate with it in a specific way: this is its *language*. Like any language, words must be in a particular order so that they have a meaning when linked together: this is its *syntax*. The compiler language and its syntax are described in the following paragraphs. At the end of this section is the full description for the resource description language.

Each description you give **rescomp** must be terminated with a period. Strings and single characters within icons are always bracketed by quotation marks, **' '**.

Trees

The resource compiler recognizes four types of tree:

```
menu
form
image
string
```

Trees are always described in the following form:

```
tree type C-NAME .
```

For example, you would describe a **menu** to the compiler by typing:

```
menu MENU-NAME .
```

into a resource description file.

Tree descriptions of free images and free strings are somewhat more complex. A free image is described as follows:

```
image C-NAME level n size n data (0xnnn)
      color n options (...) .
```

image is the tree name. **C NAME** is the name by which you reference this free image in a C program. For explanations of **level**, **size**, **data**, **color**, and **options**, see their separate headings later in this section.

A free string is described to the compiler as follows:

```
string C-NAME "quoted string" .
```

Objects

The resource compiler expects an object to be described in the following way:

```
form object_spec size offset options ext .
```

Depending on the type of object you want to describe, the **object_spec** part of this description is one of the following:

box	name_and_level	box_spec
ibox	name_and_level	box_spec
boxcharacter	name_and_level	box_spec
button	name_and_level	string_spec
string	name_and_level	string_spec
title	name_and_level	string_spec
boxtext	name_and_level	text_spec
boxedit	name_and_level	text_spec
text	name_and_level	text_spec
edit	name_and_level	text_spec
icon	name_and_level	icon_spec
image	name_and_level	image_spec

name_and_level always consists of an optional C name plus the keyword **level**, as described below.

box_spec contains any of the following optional characteristics:

- border
- boxcolor
- textcolor
- bordercolor
- fill
- trans (transparent)
- boxcharacter

These are described under separate headings, below.

string_spec consists of a quoted string. Strings are always enclosed by quotation marks.

text_spec includes one or all of the following:

- text
- template
- validation
- font
- textbox

These are described under separate headings, below.

icon_spec requires one or all of the following:

banner
iconchar
iconcolor
iconsize
icondata
iconmask

These are described under separate headings, below.

image_spec contains any one of the following:

boxcolor
iconsize
icondata

These are described under separate headings, below.

Resource description elements

extended

This description element is optional; **rescomp** expects a description of **extended** to be in the form:

extended *n*

where *n* is an unsigned integer. **extended** allows you to put any desired byte in the upper half of the object type word. These numbers are ignored by GEM, so you can use them in your program as you see fit. This is useful when you want your program to process a group of objects in similar ways.

bordercolor

This description element is optional; **rescomp** expects a description of **bordercolor** to be in the form:

bordercolor *colors*

where *colors* is one of the following:

white	whitel
black	blackl
red	redl
green	greenl
blue	bluel
yellow	yellowl
cyan	cyanl
magenta	magental

fill This description element is optional. The compiler expects a description of **fill** to be in the following form:

`fill n`

where *n* is an unsigned integer from zero through three. These values represent the colors used to fill boxes; respectively, white, black, red, and green.

transparent

This description element is optional. When used, the compiler expects a description of **transparent** to be in the following form:

`transparent`

Use the keyword to describe a box object through which you wish other objects to show.

template

This resource description element is optional. When used, **rescomp** expects its description in the following form:

`template text`

where *text* is the text of the template used in an editable string or box.

validation

This description element is optional. When used, **rescomp** expects its description to be in the following form:

`validation text`

where *text* is the validation character used in an editable string or box.

justify

This is an optional description element, used to describe a **string** object. When used, **rescomp** expects **justify** to be in the following form:

`justify justification`

where **justification** is one of **left**, **right**, or **center**.

data This element is used to describe icons and images. **rescomp** expects **data** to be described as follows:

`data bitdata`

where *bitdata* is a list of hexadecimal numbers that describe the data elements of the bit image you are creating.

mask This element is used to describe icons and images. **rescomp** expects that it be described as follows:

mask *bitdata*

where *bitdata* is a list of hexadecimal numbers which describe the mask elements of the bit image you are creating.

level *n*

This field describes the relationship of the object with the other objects within the tree. A root object is always level 1, as are its siblings. Children of the root object are level 2; their children are level 3, and so on.

size [*w,h*]

The size of an object is always described in characters, with the exception of images and icons; their size is described in pixels. Putting something in this field is optional; the default is the size of the parent object. If there is no parent object, the default size is the size of the screen.

border [+][-][*n*]

border contains the information **rescomp** needs to draw a border around certain objects. A border can be up to four pixels thick, inside or outside. A number from one to four indicates the thickness of the border in pixels. A preceding '-' indicates inside thickness, whereas '+' indicates outside thickness. This field is optional; the default border thickness is '+1'.

pattern *n*

The pattern field contains the word pattern, followed by the number representing the interior pattern for the object. The patterns range from no color to all color, with shades of color made with a dot or slash pattern in between. The numbers for these patterns range from one to eight. This field is optional; the default pattern in a box is one.

interior *n*

This field refers to the color of the pattern you have chosen for the object. There are four colors, numbered from zero to three, representing white, black, red, and green, respectively. This field is optional; the default is zero.

textcolor *n* This is the field that sets the color of the text that will appear within the object. As with **interior**, the colors are numbered zero to three, representing white, black, red, and green, respectively. The **textcolor** field is also optional, the default being zero.

text

Text is the description of a string that you want to appear in an object. Strings are always enclosed within quotation marks. Single characters on icons are also quoted.

offset [*n,n*]

The **offset** field contains the character coordinates for the placement of the object on the screen. The root object, as in the form box example, is started at 0,0. For children of the object, the coordinates represent the relative position of the children to the root.

A resource file encodes an object's coordinates in the form of character coordinates; these coordinates are changed into pixel coordinates when the resource file is loaded and the resolution of the screen is known.

options(...)

The options for an object are always found in parentheses at the end of the object description, using the following form:

option (*option name*)

The options you can select for your object are words that encode the status of the object for **ob_state** in the **OBJECT** structure, and set the flags for **ob_flags** in the same structure.

With the resource editor, you make these selections with the buttons in the top left of the edit dialogue. For the resource description file, however, you must type in the options you want your object to have.

The following table lists the options as they appear in the edit dialogue in the Resource Editor, and the corresponding names that you must give **res-comp** for those same states and flags.

<i>C definition</i>	<i>Resource definition</i>
SELECTABLE	selectable
DEFAULT	default
EXIT	exit
EDITABLE	editable
RBUTTON	radiobutton
TOUCHEXIT	touchexit
HIDETREE	hidden
SELECTED	selected
CROSSED	crossed
CHECKED	checked
DISABLED	disabled
OUTLINED	outlined
SHADOWED	shadowed

C NAME

This is the C name for the tree or object you create. **rescomp** will accept a resource description without this field, but without a C name, it is difficult to access a tree from within an application.

Sample resource description

The following is an annotated example of a resource description of a simple menu.

Create a menu with the C name **TOPMNU**:

```
menu TOPMNU.
```

Add to the menu bar the title "Desk":

```
title DESKMNU " Desk ".
```

Under the title "Desk", add the selectable entry "About this program...". This consists of the keyword **entry**, followed by the C name **DESKABOU**, the string for the title entry, and the option "selectable":

```
entry DESKABOU " About this program..." options \  
(selectable).
```

Add a separator bar to the dropped box. It is described using the keyword **entry**, the C name **DESKSEP**, and a string of disabled dashes:

```
entry DESKSEP "-----" options \  
(disabled).
```

Add the entries for the desk accessories, using the keyword **entry**, followed by the name of the string and the text of the string itself:

```
entry DESKACC1 " Desk Accessory 1 " .  
entry STRN_004 " Desk Accessory 2 " .  
entry STRN_005 " Desk Accessory 3 " .  
entry STRN_006 " Desk Accessory 4 " .  
entry STRN_007 " Desk Accessory 5 " .  
entry DESKACC6 " Desk Accessory 6 " .
```

Resource description grammar

The following lists the Backus-Naur form for the resource compiler. *Keywords appear in bold.*

```
resfile      : resource  
              ;  
resource     : tree
```

```
resource tree
tree      : formtree
           : menutree
           : freeimage
           : freestring
formtree  : formspec objlist
formspec  : tree c_name '.'
objlist   : object
           : objlist object
object     : form object_spec size offset options ext '.'
object_spec : box name_and_level box_spec
             : ibox name_and_level box_spec
             : boxcharacter name_and_level box_spec
             : button name_and_level string_spec
             : string name_and_level string_spec
             : title name_and_level string_spec
             : boxtext name_and_level text_spec
             : boxedit name_and_level text_spec
             : text name_and_level text_spec
             : edit name_and_level text_spec
             : icon name_and_level icon_spec
             : image name_and_level image_spec
name_and_level : opt_cname level_spec
level_spec     : level unsigned_integer
size           : /* null */
               : size coords
offset         : /* null */
               : offset coords
ext            : /* null */
               : extended unsigned_integer
box_spec       : border boxcolor textcolor bordercolor \
               : fill trans boxcharacter
;
```

```

border      : /* null */
              | border int
              :
textcolor   : /* null */
              | textcolor colors
              :
bordercolor : /* null */
              | bordercolor colors
              :
fill        : /* null */
              | fill unsigned_integer
              :
trans       : /* null */
              | transparent
              :
boxcharacter : /* null */
              | character numorchr
              :
menutree    : menuspec menulist
              :
menuspec    : menu c_name '.'
              :
menulist    : /* null */
              | menulist menu
              :
menu        : menutitle entrylist
              :
menutitle   : title opt_cname string_spec options '.'
              :
entrylist   : /* null */
              | entrylist entry
              :
entry       : entry opt_cname string_spec options '.'
              :
freestring  : string c_name string_spec '.'
              :
freeimage   : image c_name image_spec '.'
              :
text_spec   : ted_text template validation font textbox
              :
ted_text    : text
              :
template    : /* null */
              | template text
              :

```


validation	: /* null */ validation text
font	: /* null */ font unsigned_integer
textbox	: border boxcolor textcolor bordercolor justify
justify	: /* null */ justify justification
image_spec	: boxcolor iconsize icondata
icon_spec	: banner iconchar iconcolor iconsize icondata \ iconmask
banner	: /* null */ banner text bitoffset
iconchar	: /* null */ character iconc bitoffset
iconc	: /* null */ numorchr
iconcolor	: boxcolor maskcolor
boxcolor	: /* null */ color colors
maskcolor	: /* null */ maskcolor colors
iconsize	: /* null */ size coords iconoff
iconoff	: /* null */ offset coords
icondata	: data bitdata
iconmask	: mask bitdata
bitoffset	: offset coords

```

bitsize      : size coords
               ;
text          : quoted_string length
               ;
length        : /* null */
               | length unsigned_integer
               ;
coords        : [ ord , ord ]
               ;
ord           : unsigned_integer fine
               ;
fine          : /* null */
               | + unsigned_integer
               | - unsigned_integer
               ;
bitdata       : { byte_list }
               ;
byte_list     : /* null */
               | number
               | byte_list , number
               ;
number        : int
               | BITCONSTANT
               | HEXCONSTANT
               ;
int           : unsigned_integer
               | + unsigned_integer
               | - unsigned_integer
               ;
colors        : unsigned_integer
               | color
               ;
color         : white
               | black
               | red
               | green
               | blue
               | yellow
               | cyan
               | magenta
               | whitel
               | blackl
               | redl
               | greenl
               | bluel

```

	yellow1
	cyan1
	magenta1
justification	: left
	: right
	: center
options	: /* null */
	: options (option_list)
option_list	: option
	: option_list , option
option	: flag
	: state
flag	: selectable
	: default
	: exit
	: editable
	: radiobutton
	: last
	: touchexit
	: hide
	: flags9
	: flags10
	: flags11
state	: selected
	: crossed
	: checked
	: disabled
	: outlined
	: shadowed
	: state6
	: state7
numorchr	: unsigned_integer
	: SQUOTEDCHR
opt_cname	: /* NIL */
	: c_name
c_name	: ID_ALPHA

```
string_spec      ;  
                  : quoted_string  
                  ;
```

Section 8:

Error Messages

This chapter lists all of the error messages that can be produced by the compiler, the assembler, the linker, **make**, **resource**, **rescomp**, and **resdecom**.

The messages are in alphabetical order, and each is marked to indicate which program generated it (e.g., **cc0**, **ccp**). Each message from the compiler indicates whether it is a *fatal*, *error*, *warning*, or *strict* 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, and may indicate problems in the compiler.

An error message points to a condition in the source code that Mark Williams C cannot resolve. This almost always occurs when the program does something illegal, e.g., has unbalanced braces.

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 portable.

. (**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.

n string adjusting object *C-name* to contain children (**rescomp**, warning)

All children of an object must fall inside the bounds of the object. The object in question, at line number *n*, is being adjusted so that it covers all of its children.

; after target or macroname (**make**, error)

A semicolon appeared after a target name or a macro name.

ambiguous reference to "*string*" (**cc0**, error)

string is defined as a member of more than one **struct** or **union**, 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.

string argument mismatch (**cpp**, error)

The argument *string* does not match the type declared in the function's prototype. Either the function prototype or the argument should be changed.

array bound must be a constant (**cc0**, error)

An array's size can be declared only with a constant; you cannot declare an array's size by using a variable. For example, it is correct to say **foo[5]**, but illegal to say

```
bar = 5;  
foo[bar];
```

array bound must be positive (**cc0**, error)

An array must be declared to have a positive number of elements. The array flagged here was declared to have a negative size, e.g., **foo[-5]**.

array bound too large (**cc0**, error)

The array is too large to be compiled with 16-bit index arithmetic. You should devise a way to divide the array into compilable portions.

array row has 0 length (**cc0**, error)

This message can be triggered by either of two problems. The first problem is declaring an array to have a length of zero; e.g., **foo[0]**. The second problem is failing to declare the size of a dimension *other than the first* in a multi-dimensional array. C allows you to declare an indefinite number of array elements of *n* bytes each, but you cannot declare *n* array elements of an indefinite length. For example, it is correct say **foo[][5]** but illegal to say **foo[5][]**.

#assert failure (cpp, error)

The condition being tested in a **#assert** statement has failed.

associative expression too complex (cc1, fatal)

An expression that uses associative binary operators (e.g., '+') has too many operators; for example, **i=i1+i2+i3+ ... +i30**; You should simplify the expression.

at beginning of macro (cpp, error)

Macro replacement lists may contain tokens that are separated by **##**, but **##** cannot appear at the beginning or the end of the list. The tokens on either side of the **##** are pasted together into one token.

at end of macro (cpp, error)

Macro replacement lists may contain tokens that are separated by **##**, but **##** cannot appear at the beginning or the end of the list. The tokens on either side of the **##** are pasted together into one token.

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 call to size_tree (rescomp, panic)

A null pointer was passed to the tree fixup function.

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 either to be negative or to be larger than the object that holds it. For example, **char foo:9** or **char foo:-1** will trigger this error.

bad filler field width (cc0, error)

A filler field width was declared either to be negative or to be larger than the object that holds it. For example, **char foo:9** or **char foo:-1** will trigger this error.

bad flexible array declaration (cc0, error)

A flexible array is missing an array boundary; e.g., **foo[5][]**. C permits you to declare an indefinite number of array elements of *n* bytes each, but you cannot declare an array to have *n* elements of an indefinite number of bytes each.

Bad macro name (make, error)

A bad macro name was used; for example, a macro name included a control character.

Bad number input. (**resource**, warning)

You tried to give an object a value that is out of range.

To clear this message, press the left mouse button or any key.

name: bad name for include file (**rescomp**, fatal)

You have asked the compiler to create a file with an invalid name.

name: bad name for RDL source file (**rescomp**, fatal)

You have asked the compiler to look for a file with an invalid name.

name: bad name for resource file (**rescomp**, fatal)

You have asked the compiler to create a file with an invalid name.

Bad RSC file format. (**resource**, warning)

resource does not recognize the format of this resource file. The file may be damaged or truncated.

To clear this message, press the left mouse button or any key.

bad tree in size_level (**rescomp**, panic)

An improperly formed tree was passed to the tree-sizing function.

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. Check to make sure that you have not accidentally declared a function as a variable; e.g., typing **char *foo**; when you meant **char *foo()**;

cannot add pointers (**cc0**, error)

The program attempted to add two pointers. **ints** or **longs** 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 allocate enough memory for *string* (**resdecom**, fatal)

The resource decompiler cannot allocate enough memory to hold the decompiled resource.

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 apply unary '&' to an alien function (**cc0**, error)

The unary **&** operator cannot be used with any function that has been

- declared to be of type **alien**. **alien** functions cannot be called by pointers.
- Cannot be saved while it contains an empty tree. . . (**resource**, warning)
 - Every tree in a resource must contain data before the resource can be saved.
- cannot cast double to pointer (**cc0**, error)
 - The program attempted to cast a **double** to a pointer. This is illegal.
- cannot cast pointer to double (**cc0**, error)
 - The program attempted to cast a pointer to a **double**. This is illegal.
- cannot cast structure or union (**cc0**, error)
 - The program attempted to cast a **struct** or a **union**. This is illegal.
- cannot cast to structure or union (**cc0**, error)
 - The program attempted to cast a variable to a **union** or **struct**. This is illegal.
- cannot create *string* (**resdecom**, fatal)
 - You have run out of memory.
- string*: cannot create (**as**, error)
 - The assembler 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 not full, and that it is working correctly.
- 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 not full and that it is working correctly.
- 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)
 - For example, the declaration **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 in this way.

string: cannot open (**cpp**, **cc0**, fatal)

The compiler cannot open the file *string* of source code that it was asked to read. **cpp** may not have been told the correct directory in which this file is to be found; check that the file is located correctly, and that the **-I** options, if any, are correct.

cannot open *string* (**resdecom**, fatal)

The file you are trying to decompile cannot be opened.

cannot open definition file *string* (**resdecom**, warning)

The definition file for the resource you are trying to decompile cannot be opened by the resource decompiler.

cannot open include file *string* (**cpp**, **cc0**, fatal)

The program asked for file *string*, which 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 in **cc2** cannot reopen a file with which it has worked. Make sure that your mass storage device is working correctly and that it is not full.

Cannot test this dialog since it has no exits. (**resource**, alert)

At least one object in a dialogue must have the **exit** flag set before the dialogue can be tested.

can't allocate memory for identifier (**rescomp**, fatal)

This message indicates that you are out of memory.

can't allocate memory for string token (**rescomp**, fatal)

This message indicates that you are out of memory.

can't create C header file *string* (**rescomp**, fatal)

There is not enough room on the disk to create the file, or you have used an invalid file name.

- can't create definition file *string* (**rescomp**, fatal)
There is not enough room on the disk to create the file, or you have used an invalid file name.
- Can't create new folder. (**resource**, warning)
There may not be enough memory left on disk to hold a new folder, or the disk may be write protected. Check that your disk drive is working properly.
You can clear this message by pressing any key or the left mouse button.
- can't create output file *string* (**rescomp**, fatal)
The compiler could not create a GEM resource file. There is not enough room on the disk to create the file, or you have used an invalid file name.
- Can't delete this file because it is read only (**resource**, warning)
A read-only file cannot be deleted.
- Can't delete this folder because it contains files. (**resource**, warning)
You cannot delete a folder if it contains a file. You must first delete the files in the folder before you delete the folder itself.
To clear this message, press the left mouse button or any key.
- can't expand bit image pool (**rescomp**, fatal)
You are out of memory.
- can't expand bitblk pool (**rescomp**, fatal)
You are out of memory.
- can't expand iconblk pool (**rescomp**, fatal)
You are out of memory.
- can't expand object pool (**rescomp**, fatal)
You are out of memory.
- can't expand string pool (**rescomp**, fatal)
You are out of memory.
- can't expand tedinfo pool (**rescomp**, fatal)
You are out of memory.
- can't open for input (**rescomp**, fatal)
The specified input file (*.rdl*) cannot be opened. Check that it exists where you think it does, and check that you specified the proper path name for it.
- can't open *libstring.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 .RSD file. (**resource**, warning)

resource cannot open a resource file. Make sure that the file exists where you think it does, and that you have used the correct path name for it.

To clear this message, press the left mouse button or any key.

can't open temp file (**ld**, fatal)

The linker **ld** cannot open a temporary file. Make sure that your mass storage device is working correctly, and that the environmental variable **TMPDIR** is set correctly.

Can't open the header file. (**resource**, warning)

resource cannot open a resource's header file. The file may not exist in the current directory, or you may have used the wrong path name for the resource.

To clear the message, press the left mouse button or any key.

Can't open this file. (**resource**, warning)

A file cannot be opened. Make sure that the requested file exists in the current directory.

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. See the Lexicon entry for **case**.

character constant overflows long (**cc0**, error)

The character constant is too large to fit into a **long**. It should be redefined.

character constant promoted to long (**cc0**, warning)

A character constant has been promoted to a **long**.

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 does not have one, e.g., a function definition, array initialization, or **switch** statement.

conditional stack overflow (cpp, fatal)

A series of **#if** expressions is nested so deeply that it overflowed the allotted stack space. You should simplify this code.

constant expression required (cc0, error)

The expression used with a **#if** statement cannot be evaluated to a numeric constant. It probably uses a variable in a statement rather than a constant.

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. For example, **while(1)** will trigger this message.

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 **for** or **while** loop.

corrupt tree in finish_tree (rescomp, panic)

An improperly formed tree was passed to the tree-fixup function.

n string data size mismatch in icon object *C-name* (rescomp, warning)

Different amounts of data were provided for icon data and mask in line number *n*. The compiler zero-pads missing bits; the picture you see may not be what you expected.

#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. See the Lexicon entry for **default**.

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 this code is executed. Although the program can be parsed, this statement may create trouble if executed.

Duplicate file name. (**resource**, warning)

A file name duplicates that of a file that already exists in the current folder. If you proceed, the file that bears the name will be overwritten by the file you are creating.

To clear this message, press the left mouse button or any key.

Duplicate name. (**resource**, warning)

This is a tree manipulation error. You have given the same name to two trees within the same resource.

To clear this message, press the left mouse button or any key.

duplicated case constant (**cc0**, error)

A **case** value can appear only once in a **switch** statement. See the Lexicon entries for **case** and **switch**.

#elif used without **#if** or **#ifdef** (**cpp**, error)

An **#elif** control line must be preceded by an **#if**, **#ifdef**, or **#ifndef** control line.

#elif used after **#else** (**cpp**, error)

An **#elif** control line cannot be preceded by an **#else** control line.

#else used without **#if** or **#ifdef** (**cpp**, error)

An **#else** control line must be preceded by an **#if**, **#ifdef**, or **#ifndef** control line.

empty switch (**cc0**, warning)

A **switch** statement has no **case** labels and no **default** labels. See the Lexicon entry for **switch**.

#endif used without **#if** or **#ifdef** (**cpp**, error)

An **#endif** control line must be preceded by an **#if**, **#ifdef**, or **#ifndef** control line.

EOF in comment (cpp, fatal)

Your source file appears to end in mid-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 `*/`. Also, be sure that you did not accidentally embed a `<ctrl-Z>` in the line.

EOF in macro *string* invocation (cpp, error)

Your source file appears to end in a macro call. The source file may have been truncated, or you may have accidentally embedded a `<ctrl-Z>` in the line.

EOF in midline (cpp, warning)

Check to see that your source file has not been truncated accidentally. Also, make sure that you did not accidentally embed a `<ctrl-Z>` in the line.

EOF in string (cpp, error)

Your file appears to end in the middle of a quoted string literal. Check to see that your source file has not been truncated accidentally. Also, check that you did not accidentally embed a `<ctrl-Z>` in the line.

#error: *string* (cpp, fatal)

An `#error` control line has been expanded, printing the remaining tokens on the line and terminating the program.

error in `#define` syntax (cpp, error)

The syntax of a `#define` statement is incorrect. See the Lexicon entry for `#define` for more information.

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 find one.

error in `#include` syntax (cpp, error)

An `#include` directive must be followed by a string enclosed by either quotation marks (`"`) or angle brackets (`<>`). Anything else is illegal.

Expected 'level' (rescomp, error)

The compiler could not find the keyword `level`.

exponent overflow in floating point constant (cc0, warning)

The exponent in a floating point constant has overflowed. The compiler has set the constant to the maximum allowable value, with the expected sign.

exponent underflow in floating point constant (**cc0**, warning)

The exponent in a floating point constant has underflowed. The compiler has set the constant to zero, with the expected sign.

expression too complex (**cc1**, fatal)

The code generator cannot generate code for an expression. You should 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;`.

file name conflict (*srcname*, *rsname*, *defname*, *hdrname*) (**rescomp**, fatal)

You gave the same name to two or more files.

function cannot return a function (**cc0**, error)

The function is declared to return another function, which is illegal. A function, however, can return a *pointer* to a function, e.g., `int (*signal(n, a))()`.

function cannot return an array (**cc0**, error)

A function is declared to return an array, which is illegal. A function, however, can return a pointer to a structure or array.

functions cannot be parameters (**cc0**, error)

The program uses a function as a parameter, e.g., `int q(); x(q);`. This is illegal.

Icon/image size change will cause loss of significant looking data. Do it anyway? (**resource**, alert)

A change that you requested in the size of an icon or image will cause it to lose data that appear to be significant. Do you wish to proceed?

identifier *string* has too many arguments (**cpp**, error)

Too many actual parameters have been provided.

identifier "*string*" is being redeclared (**cc0**, error)

The program declares variable *string* to be of two different types. This often is due to an implicit declaration, which occurs when a function is used before it is explicitly declared. Check for name conflicts.

identifier "*string*" is not a label (**cc0**, error)

The program attempts to **goto** a nonexistent label.

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)

A legal character constant consists of a backslash '\' followed by a, b, f, n, r, t, v, x, or 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 cpp character (*n* decimal) (cpp, error)

The character noted cannot be processed by cpp. It may be a control character or a non-ASCII character.

illegal integer constant suffix (cc0, error)

Integer constants may be suffixed with u, U, l, or L to indicate **unsigned**, **long**, or **unsigned long**.

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 &-ed. 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 defined (**c++**, error)

The construction **defined(token)** or **defined token** is legal only in **#if**, **#elif**, or **#assert** expressions.

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** improperly. 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.

Improper banner (**rescomp**, error)

A bad value was supplied for **banner** in the source.

Improper border (**rescomp**, error)

A bad value was supplied for **border** in the source.

Improper border color (**rescomp**, error)

A bad value was supplied for **bordercolor** in the source.

Improper character (**rescomp**, error)

A bad value was supplied for **character** in the source.

Improper color(**rescomp**, error)

A bad value was supplied for **color** in the source.

Improper data (**rescomp**, error)

A bad value was supplied for **data** in the source.

Improper extension (**rescomp**, error)

A bad value was supplied for **extension** in the source.

Improper fill (**rescomp**, error)

A bad value was supplied for **fill** in the source.

Improper font (**rescomp**, error)

A bad value was supplied for **font** in the source.

Improper justification (**rescomp**, error)

A bad value was supplied for **justification** in the source.

Improper length (**rescomp**, error)

A bad value was supplied for **length** in the source.

Improper level (**rescomp**, error)

A bad value was supplied for **level** in the source.

Improper mask (**rescomp**, error)

A bad value was supplied for **mask** in the source.

Improper mask color (**rescomp**, error)

A bad value was supplied for **mask color** in the source.

Improper menu name (**rescomp**, error)

A bad value was supplied for **menu name** in the source.

Improper object type (**rescomp**, error)

A bad value was supplied for **objecttype** in the source.

Improper offset (**rescomp**, error)

A bad value was supplied for **offset** in the source.

Improper size (**rescomp**, error)

A bad value was supplied for **size** in the source.

Improper template (**rescomp**, error)

A bad value was supplied for **template** in the source.

Improper text color (**rescomp**, error)

A bad value was supplied for **textcolor** in the source.

Improper tree name (**rescomp**, error)

A bad value was supplied for **treename** in the source.

Improper validation (**rescomp**, error)

A bad value was supplied for **validation** in the source.

string in **#if** (**cpp**, error)

A syntax error occurred in a **#if** declaration. *string* describes the error in detail.

= in or after dependency (**make**, error)

An equal sign '=' appeared within or followed the definition of a macro name or target file; for example, **OBJ=atod.o=factor.o** will produce this error.

inappropriate signed (**cc0**, error)

The **signed** modifier may only be applied to **char**, **short**, **int**, or **long** types.

include stack overflow (**cpp**, fatal)

A set of **#include** statements is nested so deeply that the allotted stack space cannot hold them. Examines the files for a loop. You should try to fold some of the header files into one, instead of having them call each

other.

Incomplete line at end of file (**make**, error)

An incomplete line appeared at the end of the **makefile**.

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 (e.g., a **long** or **fBint**) as a pointer; you must first cast it to a pointer.

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.

Input and output names are both *string* (**resdecom**, fatal)

You used the same name for both input and output files.

I/O error during delete. (**resource**, warning)

resource could not delete a file or folder due to a problem with the disk drive. Make sure that the disk is not write-protected, and that the drive is working correctly.

To clear this message, press the left mouse button or any key.

integer pointer comparison (**cc0**, strict)

The program compares an integer or **long** with a pointer without casting one to the type of the other. Although this is legal, the comparison may not work on machines with non-integer pointers, e.g., Z8001 or LARGE-model i8086, or on machines with pointers larger than ints, e.g., the 68000.

insufficient memory for relocation rewrite (**ld**, fatal)

The linker **ld** cannot allocate enough memory to build its relocation tables. You should free up some memory within your system; if you have a RAM disk, you may need to make it smaller or unload it altogether.

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. For example,

```
char *foo;  
long bar;  
foo = bar;
```

Although this is permitted, it is often an error if the integer has less precision than the pointer does. Make sure that you properly declare all functions that returns pointers.

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)

The assembler has detected a situation that “should not occur”. Please send a copy of the source code that triggered this error to Mark Williams Company. For immediate help during business hours, contact Mark Williams Company.

Invalid or missing file name. (resource, warning)

This error message comes in the form of a *message balloon*. The folder icon is surrounded by a thick, rounded rectangle linked to an error box. When this type of message is presented, pressing any key, or the left mouse button, clears it.

invalid token *token* (rescomp, error)

You have specified a token that the compiler does not recognize.

“*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. See the Lexicon entry on **typedef** for more information.

"string" is not an "enum" tag (cc0, error)

An **enum** with tag *string* is referenced before any such **enum** has been declared. See the Lexicon entry for **enum** for more information.

class "string" [number] is not used (cc0, strict)

Your program declares variable *string* or *number* but does not use it.

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 the expression *string* should be a pointer, but is not.

lvalue required (cc0, error)

The left-hand value of a declaration is missing or incorrect. See the Lexicon entries for **lvalue** and **rvalue**.

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 definition too long (make, error)

Macro definitions are limited to 200 characters.

macro expansion buffer overflow in string (cpp, fatal)

A macro call has expanded into more characters than **cpp** can handle. Try to shorten the macro, or break it up.

macro string redefined (cpp, error)

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 may have inadvertently created an infinite loop. Try 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 program references a structure member that has not been declared.

Memory is in short supply. Please remove anything you don't need.

(**resource**, alert)

If you have something in memory that you do not need, throw it away.

Memory shortage is now getting critical. (**resource**, alert)

If you copy or create many more items, the editor will crash. Remove all unnecessary matter from memory.

Menu items must be in dropped box or title bar. (**resource**, alert)

You have positioned an object outside the menu bar or dropped box. This causes the object you are trying to position to "snap" to an acceptable position nearby.

To clear this message, select a button on the form or press <return>.

must have at least one title in a menu tree (**rescomp**, fatal)

No title object was specified in a menu tree.

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**, error)

The **if**, **while**, **for**, and **switch** keywords must be followed by parenthesized expressions.

missing ")" (**cc0**, error)

A right parenthesis ')' is missing anywhere after a left parenthesis '('.

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`. Use of this feature is strongly discouraged, and it will disappear when the draft ANSI standard for the C language is adopted in full.

missing "," (**cc0**, error)

A comma is missing from an enumeration member list.

missing ":" (**cc0**, error)

A colon ':' is missing after a **case** label, after a default label, or after the '?' in a '?-:' construction.

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 ']' (**as**, error)

The assembler expected to find a right bracket in the present expression, but did not.

missing "]" (**cc0**, error)

A right bracket ']' is missing from an array declaration, or from an array reference; for example, `foo[5`.

missing "{" (**cc0**, error)

A left brace '{' is missing after a **struct tag**, **union tag**, or **enum tag** in a definition.

missing "}" (**cc0**, error)

A right brace '}' is missing from a **struct**, **union**, or definition, from an initialization, or from a compound 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 actions for *name* (**make**, error)

A target is defined with more than one single-colon target line.

multiple classes (**cc0**, error)

An element has been assigned to more than one storage class, e.g., **extern register**.

Multiple detailed actions for *name* (**make**, error)

A target is defined with more than one single-colon target line.

multiple **#else**'s (**cpp**, error)

An **#if**, **#ifdef**, or **#ifndef** expression can be followed by no more than one **#else** expression.

multiple types (**cc0**, error)

An element has been assigned more than one data type, e.g., **int float**.

Must use '::' for *name* (**make**, error)

A double-colon target line was followed by a single-colon target line.

Name conflicts on resource file. (**resource**, alert)

You gave the resource the name of an existing file. The existing file will be overwritten if you proceed.

To clear this message, press the left mouse button or any key.

name *C-name* is not unique (**rescomp**, error)

You have given the same name to two objects.

n string name *C-name* truncated to *len* characters (**rescomp**, warning)

The name you assigned at line number *n* to an object was too long. Maximum identifier length for objects is eight characters. This restriction is imposed by the resource editor and the **.rdl** file format.

nested comment (**cpp**, warning)

The comment introducer sequence **/*** has been detected within a comment. Comments do not nest.

new line in *string* literal (**cpp**, error)

A newline character appears in the middle of a string. If you wish to embed a newline within a string, use the character constant **'\n'**. If you wish to continue the string on a new line, insert a backslash **'\'** before the new line.

Newline after target or macroname (**make**, error)

A newline character appears after a target name or a macro name.

newline in macro argument (**cpp**, warning)

A macro argument contains a newline character. This may create trouble when the program is run.

New object items must be dragged into form windows. (**resource**, alert)

You attempted to create an object item incorrectly. Follow the directions.

New resource sets must be dragged onto empty desktop. (**resource**, alert)

You attempted to create a resource set incorrectly. Follow the directions.

New tree items must be dragged into RSC windows. (**resource**, alert)

You attempted to create a tree item incorrectly. Follow the directions.

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 '\ '.

'::' not allowed for *name* (**make**, error)

A double-colon target line was used illegally; for example, after single-colon target line.

no tree in finish_tree (**rescomp**, panic)

A null pointer was passed to the tree fixup function.

null pointer in addentry (**rescomp**, panic)

A null pointer was passed to the menu entry function.

null title in nextmenu (**rescomp**, panic)

No title was specified before an **entry** keyword.

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 is in error.

Object must be within outer box of form or menu. (**resource**, alert)

You attempted to drag an object onto the **resource** desktop. Try dragging the object's icon again.

name object not allowed in *number* window. (**resource**, alert)

You attempted to move an object into the wrong window.

Object now covers other object or objects. Adopt them as children? (**resource**, alert)

You have positioned an object on top of one or more other objects. You may wish to rearrange the tree so that the uppermost object becomes the parent to those objects beneath it.

Objects moved to desktop must be wholly on it. (**resource**, alert)

You didn't drag an object all the way onto the desktop. Try again.

only one default label allowed (**cc0**, error)

The program uses more than one **default** label in a **switch** expression. See the Lexicon entries for **default** and **switch** for more information.

Options require parens (**rescomp**, error)

The option specification syntax includes parentheses. One or both were missing.

::: or : in or after dependency list (**make**, error)

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.

Out of core (adddep) (**make**, error)

This results from a system problem. Try reducing the size of your **makefile**.

Out of range number input. (**resource**, warning)

You attempted to use a numeric value that is out of range.

To clear this message, press the left mouse button or any key.

Out of space (**make**, error)

System problem. Try reducing the size of your **makefile**.

Out of space (lookup) (**make**, error)

System problem. Try reducing the size of your **makefile**.

out of space (**ld**, fatal)

malloc could not allocate adequate space in memory for the linker **ld** to work.

out of space (**cpp**, **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 space for bit data (**rescomp**, fatal)

The amount of bit data specified for an icon or image exceeds the limit allowed by GEM.

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 er-

ror; 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.

parameter *string* is not addressable (cc0, error)

The parameter has a stack frame offset greater than 32,767. Perhaps you should pass a pointer instead of a structure.

parameter must follow # (cpp, error)

Macro replacement lists may contain **#** followed by a macro parameter name. The macro argument is converted to a string literal.

Period missing in entry (rescomp, error)

Each object or tree specification must end with a period.

Period missing in form (rescomp, error)

Each object or tree specification must end with a period.

Period missing in image (rescomp, error)

Each object or tree specification must end with a period.

Period missing in menu (rescomp, error)

Each object or tree specification must end with a period.

Period missing in string (rescomp, error)

Each object or tree specification must end with a period.

Period missing in title (rescomp, error)

Each object or tree specification must end with a period.

Period missing in tree (rescomp, error)

Each object or tree specification must end with a period.

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.

Read error. (**resource**, warning)

An error appears to have occurred as **resource** was reading a file from disk. Check that the disk drive is working correctly; then try again.

To clear this message, press the left mouse button or any key.

read error on definition file *string* (**resdecom**, warning)

The definition file (*file.rdl*) cannot be read.

Read error on header in *string* (**resdecom**, fatal)

The header file you have specified cannot be read.

Read error on resource data in *string* (**resdecom**, fatal)

The resource file you have specified cannot be read.

string redefined (**cpp**, error)

cpp macros should not be redefined. You should check to see that you are not **#include**ing two different versions of a file somehow, or attempting to use the same macro name for two different purposes.

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.

RSC too big. (**resource**, warning)

You have attempted to build or load a resource that is too large to be held in memory. Try removing some items from memory to free space for the resource.

To clear this message, press the left mouse button or any key.

s (**as**, error)

Segment error. The program attempted to initialize something in a segment that contains only uninitialized data.

size of *string* overflows *size_t* (**cc0**, strict)

A string was so large that it overran an internal compiler limit. You should try to break the string in question into several small strings.

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 **long**s.

sizeof truncated to unsigned (cc0, warning)

An object's **sizeof** value has lost precision when truncated to a **size_t** integer.

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*.

Sorry, can't copy folders. (resource, warning)

resource can copy only files, not folders.

To clear this message, press the left mouse button or any key.

Sorry program must terminate immediately for lack of space. (resource, alert)

You have run out of memory.

storage class not allowed in cast (cc0, error)

The program **casts** an item as a **register**, **static**, or other storage class.

string initializer not terminated by NUL (cc0, warning)

An array of **chars** that was initialized by a *string* is too small in dimension to hold the terminating NUL character. For example, **char foo[3] = "ABC"**.

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 **switch**es.

Syntax error (make, error)

The syntax of a line is faulty.

This file type cannot be dragged onto the desktop. (**resource**, warning)
Only resource files can be dragged onto the desktop.

To clear this message, press the left mouse button or any key.

This move will cause the object tree to be reconstructed. (**resource**, alert)
Moving this object will force **resource** to restructure the object tree being built. This may have a significant effect upon the program that uses this resource.

This tree is already open. (**resource**, warning)
A window for this tree is already open on the desktop.

To clear this message, press the left mouse button or any key.

To create folders drag icon from menu to relevant file window. (**resource**, alert)
You attempted to create a folder incorrectly. Follow these directions.

To load resource file drag it onto open desktop. (**resource**, alert)
Dragging the resource file onto the desktop loads the file into memory.

To clear this message, select a button on the form or press the **<return>** key.

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 the allowed 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 (**epp**, fatal)
The program uses more than the allowed ten **#include** directories.

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

Too many macro definitions (**make**, error)
The number of macros you have created exceeds the capacity of your computer to process them.

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

Too many windows. (**resource**, alert)

You have attempted to open more windows than the AES allows. Close one of your windows before you attempt to open another.

To clear this message, click one of the buttons on the form or press <return>.

trailing “,” in initialization list (**cc0**, warning)

An initialization statement ends with a comma, which is legal.

Tree objects can only be dragged to resource windows ... (**resource**, alert)

The only exception is that strings and images can be dragged to form windows.

Trees must have names. (**resource**, warning)

Every tree must have a C name. You must name the tree before you can proceed.

To clear this message, press the left mouse button or any key.

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 of function “*string*” adjusted to *string* (**cc0**, warning)

This warning is given when the type of a numeric constant is widened to **unsigned**, **long**, or **unsigned long** to preserve the constant's value. The type of the constant may be explicitly specified with the **u** or **L** constant suffixes.

type of parameter “*string*” adjusted to *string* (**cc0**, warning)

The program uses a parameter that the C language says must be adjusted to a wider type, e.g., **char** to **int** or **float** to **double**.

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.

Unable to create *string*. (**resource**, alert)

resource, for any number of reasons, cannot create a file using the name that you requested. Check that the name is legal, and that your mass-storage devices are operating properly.

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 accidentally. Check your disk drive to see that it is working correctly. Also, make sure that you did not accidentally embed a **<ctrl-Z>** in the line.

n string unknown object *index* type *type value* (**rescomp**, warning)

A generated object has an unrecognized type at line number *n*. This usually indicates an internal error.

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.

Warning: errors in .RSD file. (**resource**, warning)

resource has read a resource file that it does not recognize. It may be a resource that was built by another, incompatible resource editor, or it may have been damaged or truncated in some way.

To clear this message, press the left mouse button or any key.

= without macro name or in token list (**make**, error)

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 (**make**, error)

A colon appeared without a target file name, e.g., *:string*.

Write error. (**resource**, warning)

resource could not write a file. Check that the disk is not write protected, and that the drive is working properly.

To clear this message, press the left mouse button or any key.

Write error on *string*. (**resource**, alert)

A write error occurred as **resource** tried to write a file. If you are writing onto a floppy disk, make sure that it is not write protected.

write error on output object file (**cc2**, fatal)

cc2 could not write the relocatable object module. Most likely, your mass storage device has run out of room. Check to see that your disk drive or hard disk has enough room to hold the object module, and that it is working correctly.

You can't reassemble shredded documents. (resource, alert)

A shredded document is gone forever.

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.

Section 9:

The Lexicon

The rest of this manual consists of the Lexicon. The Lexicon consists of several hundred 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.

Internally, the Lexicon has a *tree structure*. The “root” entry is the one for **Lexicon**. It, in turn, refers to a series of **Overview** entries. Each Overview entry introduces a group of entries; for example, the Overview entry for **string** introduces all of the string functions and macros, lists them, and gives a lengthy example of how to use them.

Each entry cross-references other entries. These cross-references point up the documentation tree, to an overview article and, ultimately, to the entry for **Lexicon** itself; down the tree to subordinate entries; and across to entries on related subjects. For example, the entry for **getchar** cross-references **STDIO**, which is its Overview article, plus **putchar** and **getc**, which are related entries of interest to the user. The Lexicon is designed so that you can trace from any one entry to any other, simply by following the chain of cross-references up and down the documentation tree. Other entries refer to *The Art of Computer Programming* and the first edition of *The C Programming Language*.

For more information on how to use the Lexicon and how it is organized, see the entry in the Lexicon on **Lexicon**.

example — Example

Give an example of Mark Williams Lexicon format

```
#include <example.h>
```

```
char *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 lines in **boldface** describe how to use the function being described. The first line, **#include <example.h>**, indicates that this function requires the imaginary header file **example.h**. The second line gives the syntax of the function. **char *example** means that the imaginary function **example** returns a pointer to a **char**. *foo* and *bar* are **example's** arguments: *foo* must be declared to be an **int**, and *bar* must be declared to be a **long**.

Example

The following program gives an example of an example.

```
main()
{
    printf("Many entries include examples\n");
}
```

See Also

Lexicon, all other related topics and functions

Notes

If a Lexicon entry uses a technical term that you do not understand, look it up in the Lexicon. In this way, you will gain a secure understanding of how to use Mark Williams C.

A

abort — General function (libc)

End program immediately

void abort()

abort terminates a process and prints a message on the screen. It is normally invoked in situations that “should not happen”. **abort** terminates the program by calling **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)

Return the absolute value of an integer

int abs(n) int n;

abs returns the absolute value of integer *n*. The *absolute value* of a number is its distance from zero. This is *n* if *n* ≥ 0, and *-n* otherwise.

Example

This example prompts for a number, and returns its absolute value.

```
#include <ctype.h>
#include <stdio.h>

main()
{
    extern char *gets();
    extern int atoi();
    char string[64];
    int counter;
    int input;

    printf("Enter an integer: ");
    gets(string);

    for(counter=0; counter<strlen(string); counter++)
    {
        input = *(string+counter);
        if ((isascii(input)) == 0)
        {
            fprintf(stderr,
                "%s is not ASCII\n", string);
            exit(1);
        }
    }
}
```

```
        if ((isdigit(input)) == 0)
            if (input != '-' && counter != 0)
            {
                fprintf(stderr,
                    "%s is not a number\n", string);
                exit(1);
            }
        }
        input = atoi(string);
        printf("abs(%d) is %d.\n", input, abs(input));
    }
}
```

See Also

fabs, **floor**, **int**

Notes

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

access — General function (libc)

Check if a file can be accessed in a given mode

#include <access.h>

int **access**(*filename*, *mode*) **char** **filename*; **int** *mode*;

access checks whether a file can be accessed in the mode you wish. *filename* is the full path name of the file you wish to check. *mode* is the mode in which you wish to access *filename*, as follows:

1	AEXEC	execute the file
2	AWRITE	write into the file
4	AREAD	read the file

The header file **access.h** defines the manifest constants that are commonly used with **access**.

access returns zero if *filename* can be accessed in the requested mode, and a number greater than zero if it cannot.

Example

The following example checks if a file can be accessed in a particular manner.

```
#include <access.h>
#include <path.h>
#include <stdio.h>

main(argc, argv)
int argc; char *argv[];
{
    char *env, *pathname;
    extern char *getenv(), *path();
    int mode;
    extern int access();
```

```

if (argc != 3)
{
    printf("Usage: access filename mode\n");
    exit(0);
}

switch(*argv[2])
{
    case 'e':
    case 'E':
        mode = AEXEC;
        break;
    case 'w':
    case 'W':
        mode = AWRITE;
        break;
    case 'r':
    case 'R':
        mode = AREAD;
        break;
    default:
    {
        printf("modes: e=execute, w=write, r=read\n");
        exit(0);
    }
}

env = getenv("PATH");
if ((pathname = path(env,argv[1],mode)) != NULL)
{
    printf("PATH = %s\n", env);
    printf("pathname = %s\n", pathname);

    if (access(pathname, mode) == 0)
        printf("%s accessible in mode %s\n",
            pathname, argv[2]);
    else
        printf("%s not accessible in mode %d\n",
            pathname, mode);
}
else
    printf("file %s of mode %d not found in path\n",
        argv[1], mode);
}

```

*See Also***access.h**, **path***Notes*

The only meaningful test that **access** can perform on the Atari ST is to check if a file is writable.

access.h — Header file

Define manifest constants used by **access()**

#include <access.h>

access.h is a header file that defines the manifest constants used with the function **access**.

See Also

access, header file

acos — Mathematics function (libm)

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.

On the i8086, a physical address is a 20-bit number. The i8086 builds an address by left-shifting a 16-bit segment address by four bits, and then adding it to a 16-bit offset address. The segment address points to a particular chunk of memory. The i8086 uses four segment registers, each of which governs a different portion of a program, as follows:

CS	address of the code segment
DS	address of the data segment
ES	address of the “extra” segment
SS	address of the stack segment

SMALL-model programs use only the offset address; hence, their pointers are only 16 bits long, equivalent to an **int**. **LARGE**-model programs use both segment and offset addresses. Their addresses are 20 bits long, which must be stored in a 32-bit pointer, equivalent to a **long**.

On the 68000, an address is simply a 24-bit integer that is stored as a 32-bit integer. The upper eight bits are ignored; this is not true with the more advanced microprocessors in this family, such as the 68020. The 68000 uses no segmentation; memory is organized as a “flat address space”, with no restrictions set on the size of code or data.

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***data formats**, **peekb**, **peekl**, **peekw**, **pointer**, **pokeb**, **pokel**, **pokew****AES** — Technical information

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

The 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 an “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 mouse pointer crosses into the menu bar at the top of the screen.

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

The following names each AES routine and 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_ichack	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_offset	return location of object on the screen
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
scrip_read	find name of scrap directory
scrip_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	tell desktop which application to run next
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 AES 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

The basic skeleton of an AES or VDI application is as follows:

```
/* application-specific initialization */
appl_init();
/* used with VDI only */
v_opnvwk(work_in, &vdi_handle, work_out);

for (;;) {
    /* event loop body */
}

/* application-specific cleanup */
v_clswnk(vdi_handle); /* used with VDI only */
appl_exit();
exit(status);
```

Every process must be declared to the AES through the function **appl_init**. This routine assigns a *handle* to the process, with which it is recognized and manipulated by the kernel. It also notifies the AES that this program is a GEM application.

The function **appl_exit** frees up AES structures allocated to the process, and ensures that the process terminates gracefully. The cleanup phase of an application is very important. Programs that depend upon the desktop to close windows or perform other housekeeping tasks limit their usefulness unnecessarily.

If the program is intended to be a desk accessory, replace the call to **appl_init** with the following:

```
menu_register(appl_init(), " Menu name");
```

This tells the AES that the program is a desk accessory, and registers its name with the menu of desk accessories (the one that always appears leftmost on the menu bar). See **desk accessory** for more information on how to build and com-

pile an accessory.

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**, **libvdi**, **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

Declare GEM AES routines

aesbind.h is the header file that declares the GEM AES routines contained in the library **libaes.a**, and lists the parameters for each.

See Also

AES, **header file**, **TOS**

a form to - see form-10

alignment — Definition

Alignment refers to the fact that some microprocessors require the address of a data entity to be *aligned* to a numeric boundary in memory so that *address modulo number* equals zero. For example, the 68000 and the PDP-11 require that an integer 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 ensures that data types are aligned properly under foreseeable conditions. You should, however, beware of copying structures and of casting a pointer to **char** to a pointer to a **struct**, for these could trigger alignment problems.

Processors react differently to an alignment problem. On the VAX or the i8086, it causes a program to run more slowly, whereas on the 68000 it causes a bus error.

See Also

data types, declarations

appl_exit — AES function (libaes)

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 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)

Get another application's handle

```
#include <aesbind.h>
```

```
int appl_find(name) char name[9];
```

appl_find is an AES routine that fetches the handle of another application.

name is the name of the application to find, minus any suffix and all in upper-case letters. It is always eight characters long. If the name of the application is less than eight characters long, you must use space characters to pad *name* to eight characters. For example, if the name of the application is **example.prg**, then *name* must point to the string

```
"EXAMPLE "
```

appl_find returns the handle if it is found, and -1 if an error occurred. This requires that the application be started with **shel_write**.

See Also

AES, TOS

appl_init — AES function (libaes)

Initiate an application

```
#include <aesbind.h>
```

```
int appl_init()
```

appLinit is an AES routine that declares an application. It registers the application with AES, and initializes all resources used by the application. It returns the application's handle if all went well, or -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, **appLexit**, TOS

appL_read — AES function (libaes)

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 helps to read a message sent by the function **appL_write**. The first 16 bytes of such a message are read by either of the functions **evnt_mesag** or **evnt_multi**; **appL_read** can read the portion of the message that extends beyond the first 16 bytes, should the message be longer than 16 bytes.

handle is the AES handle of the application that wrote the message, and *length* is the number of bytes to read. The third word of the message received by **evnt_mesag** or **evnt_multi** gives the number of extra bytes to be read, i.e., the value to which this variable should be set. *buffer* is the place into which the message is written. It returns zero if an error occurred, or a number greater than zero if one did not.

See Also

AES, **appL_write**, **evnt_mesag**, TOS

appL_tplay — AES function (libaes)

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 quickly the actions should be replayed. **appL_tplay** always returns one.

See Also

AES, **appL_trecord**, TOS

appL_trecord — AES function (libaes)

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 actions with the AES. 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)

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. *length* is the length of the message, in bytes. *buffer* points to the buffer into which your message is written.

Standard messages are 16 bytes (eight words) long. The first word identifies the type of message being written, and the second gives the identifier of the application to which the message is being sent. The identifier can be found with the function **appl_find**; see its entry for more information on its use. **appl_write** returns zero if an error occurred, and a number greater than zero if one did not. The third word gives the number of bytes in the message beyond the standard 16. Thus, if you are sending a standard 16-byte message, this value should be zero.

The target application can read the first 16 bytes of a message through the functions `evnt_mesag` or `evnt_multi`. Any additional bytes in the message should be read with the function `appl_read`.

See Also

AES, `appl_find`, `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

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**, **lmalloc**, **lcalloc**, **lrealloc**, 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**, **lcalloc**, **lmalloc**, **lrealloc**, **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; by convention, it is the second argument passed to **main**. By convention, **argv[0]** always points to the name of the command itself.

Example

This example demonstrates both **argc** and **argv[]**, to recreate 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, crt0.o, crtstd.o, crtsg.o, main, Pexec
The C Programming Language, page 110

array — Definition

An **array** is a concatenation of data elements, all of which are of the same type or structure. All the elements of an array are stored consecutively in memory, and each element within the array can be addressed by the array name plus a subscript.

For example, the array **int foo[3]** has three elements, each of which is an **int**. The three **int** are stored consecutively in memory, and each can be addressed by the array name **foo** plus a subscript that indicates its place within the array, as follows: **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.

Page 83 of *The C Programming Language* forbids the initialization of automatic arrays. Mark Williams C lifts this restriction. You can initialize automatic arrays and structures, provided that you know the size of the array, or of any array contained within a structure. An automatic array is initialized in the same manner as aggregate, but initialization is performed on entry to the routine at run time, instead of at compile time. Note that because this feature is not part of the standard C language, its use will limit the portability of your program.

Example

The following program initializes an automatic array, and prints its contents.

```
main()
{
    int foo[3] = { 1, 2, 3 };
    printf("Here's foo's contents: %d %d %d\n",
          foo[0], foo[1], foo[2]);
}
```

See Also

declarations, flexible array, struct

The C Programming Language, pages 25, 83, 210

as — Command

Assembler for Atari ST

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 some features seen with more elaborate assemblers.

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:

- 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.
- o** Write the assembled executable into *outfile*. The default is **l.out**.
- v** Give verbose error messages.
- 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:

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

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 aN). 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 **-g**, 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 hidden value that identifies it internally; however, all hidden values are converted to absolute constants 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
.l	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

MOVM 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.

movm	\$n,ea	4880
movm	ea,\$n	4C80
movm.l	\$n,ea	48C0
movm.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
dblc	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**, calling conventions, 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. If you use the **-v** (verbose) option, **as** issues longer, more descriptive error messages.

as68toas — Command

Convert Motorola assembler to Mark Williams assembler

as68toas [*infile.asm*] [**-o** *outfile.s*]

as68toas converts files of 68000 assembly language from the Motorola dialect into the Mark Williams dialect. It accepts Motorola-style instructions from the standard input device and translates them into Mark Williams-style instructions, which it prints on the standard output device. If it cannot handle a given instruction, it prints an error message on the standard error device.

The option **-o** lets you name an output file into which **as68toas** writes the translated assembly language program. If you give **as68toas** a file name *without* the **-o** option, **as68toas** reads that file for its input. Thus, to convert the file **example.asm**, which is written in Motorola-style assembly language, into a file of Mark Williams-style assembly language called **example.s**, simply type either:

```
as68toas -o example.s example.asm
```

or

```
as68toas example.asm -o example.s
```

If **-o** is not followed by a file name, **as68toas** reports an error. If more than one infile or outfile is named, only the last one is used. Files of Mark Williams-style assembly language *must* have the suffix **.s**. Otherwise, they will not be accepted by the assembler **as**.

If you wish to see in detail the differences between Motorola-style assembly language and that used by Mark Williams, just type the command **as68toas**, and then type instructions from your keyboard. **as68toas** will print the modified instruction on your screen.

*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 sequences 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 (sharp)
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)

Convert time structure to ASCII string

```
#include <time.h>
```

```
char *asctime(tmp) tm *tmp;
```

asctime takes the data found in *tmp*, and turns it into an ASCII string. *tmp* is of the type **tm**, 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**, 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 *timestruct;

    /* read system time, print using ctime */
    time(&timenumber);
    printf("%s", ctime(&timenumber));

    /* use gmtime to fill tm, print with asctime */
    timestruct = gmtime(&timenumber);
    printf("%s", asctime(timestruct));

    /* use localtime to fill tm, print with asctime */
    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 (overview)

Notes

asctime returns a pointer to a statically allocated data area that is overwritten by successive calls.

asin — Mathematics function (libm)

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(expression)

assert checks the value of the given *expression*. If the *expression* is false (zero), **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

For an example of this function, see the entry for **index**.

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.

Notes

assert is a macro whose body is an if expression; therefore, it cannot by definition return a value. Using **assert** in a value context, such as

```
foo = assert(a < b); /* WRONG */
```

will generate an error message when you attempt to compile your program.

#assert — Preprocessor instruction

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 *expression*

cpp evaluates the expression. If it is false (zero), **cpp** prints the diagnostic message

#assert failure

and compilation ceases. The condition being tested must be an expression that uses constants of the form acceptable to **cpp**'s **#if** command. You should use **#assert** 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

assert.h — Header file

Define **assert()**

#include <**assert.h**>

assert.h is the header file that defines the macro **assert**.

See Also

assert, header file

atan — Mathematics function (libm)

Calculate inverse tangent

#include <**math.h**>

double **atan**(*arg*) **double** *arg*;

atan calculates the inverse tangent of *arg*, which 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)

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)

Convert ASCII strings to floating point

double atof(string) char * string;

atof converts *string* into the binary representation of a double-precision floating point number. *string* must be the ASCII representation of a floating-point number. It can contain a leading sign, any number of decimal digits, and a decimal point. It can be terminated with an exponent, which consists of the letter 'e' or 'E' followed by an optional leading sign and any number of decimal digits. For example,

123e-2

is a string that can be converted by **atof**.

atof ignores leading blanks and tabs; it stops scanning when it encounters any unrecognized character.

Example

For an example of this function, see the entry for **acos**.

See Also

atoi, **atol**, **float**, **long**, **printf**, **scanf**

Notes

atof does not check to see if the value represented by *string* fits into an IEEE double. It returns zero if you hand it a string that it cannot interpret.

atoi — General function (libc)

Convert ASCII strings to integers

int atoi(string) char * string;

atoi converts *string* into 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

atoi does not check to see if the number represented by *string* fits into an int. It returns zero if you hand it a string that it cannot interpret.

atol — General function (libc)

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 — C keyword

Note an automatic variable

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 function exits. The word **auto** is a C keyword, and may not be used to name any function, macro, or variable.

See Also

C keywords, C language, extern, 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 boot device. 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 can be used to perform routine tasks, such as setting the system time or installing a RAM disk.

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 **ctrl**, **intn**, 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 *ctrl[4];
        linea0();
        INTIN = intin;
        INTOUT = intout;
        PTSIN = ptsin;
        PTSOUT = ptsout;
        CTRL = ctrl;
    }

/*
 * 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 — Operating system device

Logical device for serial port

TOS gives names to its logical devices. Mark Williams C uses these names to access these devices via TOS. **aux:** is the logical device for the the serial port auxiliary device.

Example

The following example opens the auxiliary port and sends it the string **hello, world**.

```
#include <stdio.h>
FILE *fp, *fopen();
if ((fp = fopen("aux:", "w")) != NULL) {
    printf("aux: enabled\n");
    fprintf(fp, "hello, world.\n");
}
else printf("aux: cannot open.\n");
}
```

See Also

con:, prn:, Rsconf, STDIO

Notes

aux: may be spelled **aux:** or **AUX:**.

B

backspace — Character constant

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.

Example

The following example prints a string, then backspaces over it and prints another message.

```
main()
{
    printf("BLINK!\b\b\b\b\bhello, world\n");
}
```

See Also

ASCII, character constant

basepage.h — Header file

Define TOS base page structure

#include <basepage.h>

basepage.h is a header file that defines the TOS 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**, **Bcstat**, 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 to 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 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 zero 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**, **Bconstat**, **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 — Definition

BIOS is an acronym for *basic input/output system*. In most machines, the BIOS consists of a group of routines carried in the read-only memory (ROM). These routines contain basic instructions for accessing the various aspects of the hardware.

See Also

bios, STDIO

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 for it. 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. The first column gives its function number, the second its name, and the third a brief description:

2	Bconin	receive a character
3	Bconout	output a character
1	Bconstat	return input status of device
8	Bcostat	return output status of device
10	Drvmap	return map of logical drives
7	Getbpb	return pointer to BIOS parameter block
0	Getmpb	copy memory parameter block
11	Getshift	get/set status for shift/alt/control keys
9	Mediach	check if medium has been changed
4	Rwabs	read/write a disk drive
5	Setexc	set an exception vector
6	Tickcal	return system timer's calibration

See Also

osbind.h, TOS

Notes

No **bios** function checks for incorrect device numbers. Passing an invalid device number to a routine may crash the system.

bios and **xbios** traps can be nested to a level of three deep. This occurs either when an interrupt-level routine calls a **bios** or **xbios** function while a **bios** or **xbios** function is executing, or when a **bios** or **xbios** function itself traps to the **bios** or **xbios**. A dangerous situation may occur if a **bios** or **xbios** routine in a routine that is executed by an interrupt handler or can be invoked asynchronously; in these situations, the level of nesting can quickly exceed the limit of three.

All **bios** functions are unbuffered. Combining them with buffered routines, such as those in the **STDIO**, **gemdos**, or **GEM AES** libraries, will lead at best to unpredictable results.

bios.h — Header file

Declare bios constants and structures

```
#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**

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. A bit can have a value of either zero or one. Bits can be concatenated to form bytes.

A bit can be used either as a placeholder to construct a number with an absolute

value, or as a flag whose value has 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 other than 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. These generate code rather than calls to routines. Bit fields are generally less efficient than masking because they always generate masking and shifting.

Blitmode — xbios function 64 (osbind.h)

Get/set blitter configuration

int Blitmode(flag) int flag;

Blitmode gets or sets the configuration of the blitter chip. The bits of *flag* encode the new setting for the blitter chip, as follows:

- 0 zero, set blit mode to software; one, set to hardware
- 1-14 reserved, may be anything
- 15 reserved, must be zero

If *flag* is set to -1, the blitter mode is not reset and **Blitmode** returns an **int** whose bits encode the current configuration of the blitter, as follows:

- 0 zero, in software; one, in hardware
- 1 zero, no blit chip installed; one, blit chip installed
- 2-14 reserved, may be anything
- 15 always zero

Example

This example returns the current blitter mode.

```

#include <osbind.h>
main()
{
    int setting = Blitmode(-1);
    /* check if blitter chip is present */
    if (setting & 0x02)
    {
        printf("A blitter chip is present.\n");
        /* check if mode is hardware or software */
        if (setting & 0x01)
            printf("The blit mode is in hardware.\n");
        else
            printf("The blit mode is in software.\n");
    }
    else
        printf("No blitter chip is present.\n");
}

```

See Also

TOS, xbios

Notes

The reserved bits will be used in future models of the Atari ST.

If you attempt to set the blitter mode on a machine that does not have the blitter chip, the mode will always be set to zero (in software). This function works only on machines that have the Atari blitter ROMs. On machines with earlier ROMs, the function returns 0x40 (the xbios function number).

bombs — Technical information

68000 processor exceptions

When a program goes seriously wrong on the Atari ST, TOS takes the following 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, as follows:
 - 2 Bus error
 - 3 Address error
 - 4 Illegal instruction
 - 5 Zero divide
 - 6 CHK instruction
 - 7 TRAPV instruction
 - 8 Privilege violation
 - 9 Trace
 - 10 Line A emulator

- 11 Line F emulator
- 12 Reserved
- 13 Reserved
- 14 Reserved (000, 008), format error (010)

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 the operating system, organizes and tests memory, and initializes peripheral devices.

Some operating systems use the term *warm boot* to refer to a second-stage bootstrapping procedure. An operating system may execute a warm boot to restore portions of the operating system that may have been overlaid by user code during the operation of a program, or to reinitialize the system without going through the entire boot procedure.

See Also

exit

Notes

TOS does not warm boot on program termination.

break — C keyword

Exit from loop or switch statement

break is a C statement that causes an immediate exit from a **switch** sequence, or from a **while**, **for**, or **do** loop.

Example

For an example of this instruction, see the entry for **VDI**.

See Also

C keywords, C language

The C Programming Language, page 56

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 read from or to be written to a file stream. 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 associated with that file. **exit** calls **close**.

Combining unbuffered and buffered I/O functions on the same file or device within one program will produce results that are at best unpredictable.

On the Atari ST, all **STDIO** routines use buffering by default. **stdin** and **stdout** are buffered, but **stderr** is not. Buffering can be turned off with the function **setbuf**. All Atari BIOS functions that perform I/O are not buffered.

Example

The following example demonstrates what does and does not happen when you use **fflush** with the output buffer.

```
#include <stdio.h>
main()
{
    extern char *malloc();    /* declare malloc & what it returns */
    char *buffer;

    /* use malloc() to create a 120-char buffer */
    if ((buffer = malloc(120)) == NULL)
    {
        /* if malloc() fails, bail out */
        fprintf(stderr, "malloc failed\n");
        exit(1);
    }

    printf("Type your name: ");
    fflush(stdout);          /* flush stdout buffer */
    gets(buffer); /* copy string into malloc'd buffer */
    printf("Your name is %s\n", buffer);
}
```

See Also

arena, array, Cconrs, Cconws, close, exit, 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 or a small integer quantity. Note that for C, the term “byte” has no meaning. C defines data types as being multiples of the data type **char**, and what a **char** is 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 — Technical information

Byte ordering is the order in which a given machine stores successive bytes of a multibyte data item. Note that different machines order bytes differently.

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 the 68000 or the Z8000, the program gives the following results:

```
12 34 56 78
1234 5678
12345678
```

As you can see, the order of bytes and words from low to high memory is the same as is represented on the screen.

When run on a PDP-11, however, the program gives these results:

```
34 12 78 56
1234 5678
12345678
```

As you can see, the PDP-11 inverts the order of words in memory.

Finally, when the program is run on the i8086 you see these results:

```
78 56 34 12
5678 1234
12345678
```

The i8086 inverts both words and long words.

See Also

C language, canon.h, data formats

C

C keywords — Overview

A **keyword** is a word that is reserved within C, and may not be used to name variables, functions, or macros. Mark Williams C recognizes the following keywords:

alien	extern	signed
auto	float	sizeof
break	for	static
case	goto	struct
char	if	switch
const	int	typedef
continue	long	union
default	readonly	unsigned
do	register	void
double	return	volatile
else	short	while
enum		

In conformity with the proposed ANSI standard, the keyword **entry** is no longer recognized. The keywords **const** and **volatile** are now recognized, but not implemented. Mark Williams C recognizes the keywords **readonly** and **alien**, but these are not implemented on the 68000.

See Also

C language

C language — Overview

The following summarizes how Mark Williams C implements the C language.

Identifiers:

Characters allowed: A-Z, a-z, _, 0-9

Case sensitive.

Number of significant characters in a variable name:

at compile time: 128

at link time: 16

C appends '_' to end of external identifiers

Reserved identifiers (keywords):

alien	extern	signed
auto	float	sizeof
break	for	static
case	goto	struct
char	if	switch
continue	int	typedef
const	long	union
default	readonly	unsigned
do	register	void
double	return	volatile
else	short	while
enum		

In conformity with the proposed ANSI standard, the keyword **entry** is no longer recognized. The keywords **const** and **volatile** are now recognized, but not implemented. The compiler will produce a warning message if the keyword **volatile** is used with the peephole optimizer. Mark Williams C reserves the keywords **readonly** and **alien**, but these are not implemented on the 68000.

Data formats (in bits):

char	8
unsigned char	8
double	64
float	32
int	16
unsigned int	16
long	32
unsigned long	32
pointer	32
short	16
unsigned short	16

float format:

DECVAX floating point format:

- 1 sign bit
- 8-bit exponent
- 24-bit normalized fraction with hidden bit

DECVAX double format:

Same as **float**, but with 56 bits of fraction

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 auto array size: 32 kilobytes
Maximum static array size: no formal limit

Preprocessor instructions:

#assert	#if
#define	#ifdef
#else	#ifndef
#elif	#include
#endif	#line
#file	#undef

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 **ints**, **longs**, or pointers in d0
Return values for **doubles** in d0 and d1
Pointers to returned structures in a0, 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 must clear 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 and function structure return
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.
- Multiplication and division by constant powers of two are changed to shifts when the results are the same.
- Sequences that can be resolved at compile time are identified and resolved.
- Peephole optimization remembers register contents.
- 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, calling conventions, data formats, data types, declarations, keywords, Lexicon, memory allocation

cabs — Mathematics function (libm)

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 — Technical information

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 beginning of the function’s name. Assembly-language programmers must append ‘_’ to the beginning of the name of each C-callable function.

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 (returned as double)
double	in d0 and d1
pointer	in d0.L

Functions that return **struct** or **union** actually return a pointer to the **struct** or **union** in register a0. 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:

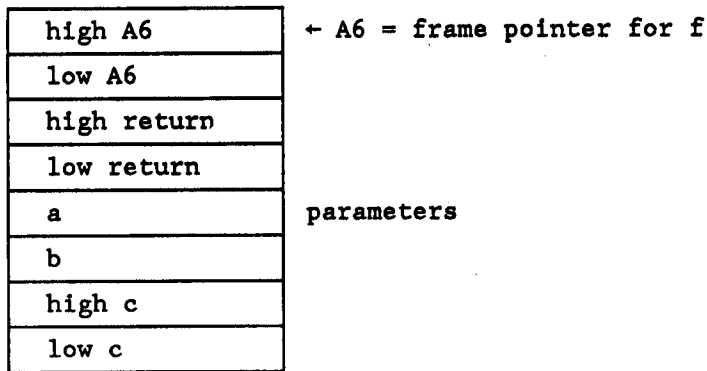
```
        .shri
        .globl f_
f_:      link      a6, $0
        move      10(a6), d0
        muls      8(a6), d0
        ext.l     d0
        add.l     12(a6), d0
        unlk      a6
        rts
        .globl main_
main_:   link      a6, $-8
        moveq     $1, d0
        move.b    d0, -2(a6)
        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:



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 is translated into assembly language by compiling it with the -S option, the result is as follows:

```

.comm today_, 6
.shri
.globl mkda_

```

```

mkda_:
    link    a6, $-6          DECLARED 3 PARAMS
MOVE ARGS { move    8(a6), -6(a6)      NEG OFFSETS ARE DECLARED (A6)
JUNK LOCAL { move    10(a6), -4(a6)   NEG OFFSETS ARE DECLARED (A6)
STRUCT { move    12(a6), -2(a6)   POS OFFSETS ARE STATIC (A6)
SET ADDRESS { lea    -6(a6), a1
              lea    8(a6), a0
              movea.l a0, a2
MOVE LOCAL { move    (a1)+, (a2)+
STRUCT TO { move.l  (a1)+, (a2)+
STACK {      unlk    a6
          rts
          .globl main_

main_:
    link    a6, $0          NO DECLARATIONS
PUSH CALLING { moveq  $85, d0
ARGS AND STACK { move    d0, -(a7)
                moveq  $20, d0
                move    d0, -(a7)
                moveq  $3, d0
                move    d0, -(a7)
                jsr    mkda_
                addq   $6, a7
                lea    6(a0), a1
MOVE STRUCTURE { movea.l  $today_+6, a0
FROM STACK TO { move    -(a1), -(a0)
GENERAL STACK { move.l  -(a1), -(a0)
                unlk    a6
                rts

```

See Also

C language, memory allocation

calloc — General function (libc)

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. When this memory is no longer needed, you can return it to the free pool by using the function **free**.

Example

This example attempts to **calloc** a small portion of memory; it then reallocates it to demonstrate **realloc**.

```
#include <stdio.h>

main()
{
    register char *ptr, *ptr2;
    extern char *calloc(), *realloc();
    unsigned count, size;

    count = 4;
    size = sizeof(char *);

    if ((ptr = calloc(count, size)) != NULL)
        printf("%u blocks of size %u calloced\n",
            count, size);
    else
        printf("Insuff. memory for %u blocks of size %u\n",
            count, size);

    if ((ptr2 = realloc(ptr, (count*size) + 1)) != NULL)
        printf("1 block of size %u reallocated\n",
            (count*size)+1);
}
```

See Also

arena, free, lalloc, lmalloc, lrealloc, malloc, notmem, realloc

Diagnostics

calloc returns **NULL** if insufficient memory is available.

Notes

The related function **lalloc** 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 — Character constant

Mark Williams C recognizes the literal character **'\r'** for the ASCII carriage return character CR (octal 015). This character "tosses the carriage", i.e., it returns the cursor to the beginning of the line. The newline character **'\n'** drops the cursor down to the next line. With the routines in **libc** and **libm**, **'\n'** is a synonym for **'\n'** plus **'\r'**. TOS routines, such as **Cconws**, need both characters explicitly.

See Also

ASCII, character constant

Notes

Note that files that contain the carriage return character must be opened in binary mode. The default mode for opening a file recognizes only alphanumeric characters, plus `<space>`, the tab character, and `'\n'`. See the entry for **fopen** for more information on how to open a file in binary mode.

case — C keyword

Introduce entry in switch statement

case is a prefix that is used to introduce the individual entries in a **switch** statement. For example,

```
while ((int = getchar()) != EOF)
    switch (foo) {
        case 'q':
        case 'Q':
            exit(0);
        case ' ':
            n++;
        default:
            break;
    }
```

case introduces the three possibilities recognized by the **switch** statement: a space, 'q', and 'Q'. The statements that follow a **case** statement behave as if they were enclosed within braces. Note that if a **case** statement is not specifically concluded with **exit**, **break**, **return**, or a similar statement, the **switch** statement will continue to search its list for variables that satisfy its condition.

See Also

break, C keywords, C language, switch

The C Programming Language, page 55

cast — Definition

The **cast** operation is when you “coerce” a variable from one data type to another.

There are two reasons to cast a variable. The first is to convert a variable's data into a form acceptable to a given function. For example, the function **hypot** takes two **doubles**; if the variables **leg_x** and **leg_y** are **ints**, then you would pass them to **hypot** as follows:

```
hypot((double)leg_x, (double)leg_y);
```

If you do not do this, **hypot** will still grab a **double**'s worth of memory: the two bytes of your **int**, plus two bytes of whatever happens to be sitting in memory.

The other reason to cast a variable is when you cast one type of pointer to another. For example,

```
char *foo;  
int *bar;  
bar = (int *)foo;
```

Although **foo** and **bar** are of the same length, you would cast **foo** in this instance to stop the C compiler from complaining about a type mismatch.

See Also

data formats, data types

cat — Command

Concatenate files

cat [*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.

<ctrl-S> stops the printing of text, and <ctrl-Q> resumes printing.

See Also

commands, msh

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 = Ccawcin()) == 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

Programs that use 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

Programs that use this function must be compiled with the **-VGEM** option, and run either from the GEM desktop or with the **tos** command.

cc — Command

Compiler controller

```
cc [options] file ...
```

cc is the program that controls compilation. 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 the file or files that hold 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.

File names

cc assumes that each *file* name that ends in **.c** or **.h** is a C program and passes it to the C compiler for compilation.

cc assumes that each *file* argument that ends in **.s** is in Mark Williams assembly language and processes it with the assembler **as**.

cc also passes all files with the suffixes **.o** or **.a** unchanged to the linker **ld**.

How cc works

cc normally works as follows: First, it compiles or assembles the source files, naming the resulting object files by replacing the **.c** or **.s** suffixes with the suffix **.o**. Then, it links the object files with the C runtime startup 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 existed before you began to compile, then the object file or files are not deleted.

Setting the environment

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. 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. |
| PATH | This sets where cc finds the executable files it uses to compile and link your program. |
| 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**.

Options

The following lists all of **cc**'s command-line options. **cc** passes some options through to the linker **ld** unchanged, and correctly interprets to it the options **-o** and **-u**.

Note that a number of the options are esoteric and normally are not used when compiling a C program. The following are the most commonly used options:

- A** invoke editor when errors occur
 - c** compile only; do not link
 - f** include floating-point **printf**
 - lname** pass library **libname.a** to linker
 - o name** call output file **name**
 - V** print details of compiler's actions
 - VASM** generate assembly-language output
- 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>**.
- c** Compile option. Suppress linking and the removal of the object files.
- Dname[=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 **cpp** and write its output onto the standard output.
- f** Floating point option. Include library routines that perform floating-point arithmetic. Because the floating-point routines require approximately five kilobytes of memory, 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 with -f option for floating point
- will be printed and the program will exit.
- Idirectory**
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 **-Idirectory** 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 **-Idirectory** option, and then in the system's default directories. Multiple **-Idirectory** options are executed in their order of appearance.

-K Keep option. Do not erase the intermediate files generated during compilation. Temporary files will be written into the current directory.

-l name

library option. Pass the name of a library to the linker. **cc** expands **-lname** into **libname.a** and searches **LIBPATH**.

-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 **-Nrctsg.o** to change the name of the run-time start-up module.

-NOVstring

No variant option. Turn off a variant option that is turned on by default. See the table of variant options, below, for more information.

-o name

Output option. Rename the executable file from the default *file.prg* to *name*.

-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. The option **-VLINES** can be used with **-S** to generate line numbers as comments in the assembly-language output.

-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 step-by-step description of each action it takes.

Vstring

Variant option. Variants that are marked **on** are turned on by default. To turn them off, use the appropriate form of the option **-NOVstring**. For example, to turn off the option **-VSTRICT**, use the option **-NOVSTRICT**. Most options are turned off by default. To turn them on, enter their names as given below. For example, to turn on the option **-VPEEP**, which turns on the peephole optimizer, simply include it in the **cc** command line. 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. It can be used with the **-VLINES** option, described below, to generate a line-numbered file of assembly language. Default is **off**.
- VCOMPAC** Similar to **-VSMALL**, except that PC-relative addressing is used only for code reference.
- VCSD** Generate debugging information for **csd**, the Mark Williams C Source Debugger.
- 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**.
- VLINES** Generate line number information. Can be used with the option described above to generate assembly language output that uses line numbers. Default is **off**.
- VMOASM** This switch is for an unsupported feature. It is similar to the **-S** switch, except that it produces Motorola-style assembly language. **as**, the Mark Williams assembler, does not recognize this syntax. Also, the output of this feature may not be a valid source for the Motorola 68000 assembler. Although this is not a supported feature, please contact Mark Williams Company if you discover any problems with it.

-VNOOPT Turn off optimization. Default is **off**, i.e., optimization is on.

-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 or use routines that have any of the aforementioned names. Default is **off**.

-VPEEP Peephole optimization. Perform additional optimization on executable. This should not be used when device registers are accessed repeatedly, because the peephole optimizer attempts to reduce memory accesses when values are known to be in registers already.

-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**.

-VSMALL Enable PC-relative addressing for global data and function references. This can only be used when the program has no global references that are more than 32 kilobytes away from where they are referenced. The linker will detect if a span is not reached and report an error. The size of pointers does not change, and there is no problem mixing modules compiled with and without **-VSMALL**. The advantage of using **-VSMALL** is that the code generated is smaller and tends to be faster.

-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 **on**.

-VSPVAL Strict: pointer value truncated. Default is **off**.

-VSRTVC Strict: risky types in truth contexts. Default is **off**.

-VSTAT Give statistics on optimization.

-VSTRICT Turn on all strict checking. Default is **on**.

-VSUREG Strict: note unused registers. Default is **off**.

-VSUVAR Strict: note unused variables. Default is **on**.

-V3GRAPH

Translate ANSI trigraphs. Default is **off**.

-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 *parser*. It parses C programs using the method of recursive descent and translates the program into a logical-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. Mark Williams C 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 output phase of Mark Williams C that writes a file of assembly language rather than a relocatable object module. This phase is optional; it allows you 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 called **foo.s**, 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**

Notes

<ctrl-C> aborts a program if typed in response to **Cconin**.

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 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) intc;

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**

Notes

<ctrl-C> aborts a program if used with **Cconout**.

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 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 Cconws.

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. 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**. By definition, a root directory has no parent.

See Also

commands, msh, pwd

ceil — Mathematics function (libm)

Set numeric ceiling

```
#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 *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(ceil(x));
        display(floor(x));
        display(fabs(x));
        display(sqrt(x));
    }
}
```

See Also

abs, fabs, floor, frexp

char — C keyword

Data type

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. **sizeof(char)** returns one by definition, 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.

Note that under Mark Williams C, a **char** by default is signed; this conforms with the description of a character on page 183 of *The C Programming Language*.

See Also

byte, C keywords, C language, data formats, declarations, unsigned

character constant — Overview

A **character constant** is a constant of the form '*X*', where *X* is any printable character 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. For example, on the IBM PC and compatible machines, the character constant '*A*' is equivalent to the ASCII value of the letter '*A*', or 0x41. This gives you a portable way to manipulate the machine values of characters.

Selected non-printable characters can also be represented as character constants by using the following escape sequences:

\0	NUL
\NNN	octal number
\a	bell
\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab
\xNN	hexadecimal number
\0xNN	hexadecimal number

See Also

ASCII, backspace, carriage return, horizontal tab, newline, vertical tab
The C Programming Language, page 35

Notes

The draft ANSI standard describes the form “\xNNN” as a one-character constant. Note, too, that use of this form may not be portable to all compilers. Because it departs from the Kernighan and Ritchie standard for C, it will generate a warning message if the compiler option **-VSBOOK** is used.

chdir — UNIX system call (libc)

Change working directory

chdir(*directory*)

char * *directory*;

The function **chdir** changes the working directory to the directory pointed to by *directory*. This change is in effect until the program exits or calls **chdir** again.

By convention, the working directory has the name ‘.’.

Diagnostics

chdir returns zero if successful. It returns -1 if an error occurred, e.g., that *directory* does not exist, is not a directory, or is not searchable.

See Also

chmod, **directory**

chmod — UNIX system call (libc)

Change file protection modes

#include <stat.h>

chmod(*file*, *mode*)

char * *file*; **int** *mode*;

chmod sets the mode of *file* to *mode*. *mode* is constructed from the following values, which are defined in the header file **stat.h**:

S_IRO	0x01	Read-only
S_IRHID	0x02	Hidden from search
S_IRSYS	0x04	System, hidden from search
S_IJVOL	0x08	Volume label in first 11 bytes
S_IJDIR	0x10	Directory
S_IJWAC	0x20	Written to and closed

chmod returns -1 if an error occurs.

See Also

chdir, **directory**

Notes

At present, **chmod** is included for compatibility with the UNIX system libraries. It performs no work, and always returns zero.

chmod — Command

Change the modes of a file

chmod *+modes file*

chmod *-modes file*

The command **chmod** changes the modes of a file. *file* is the file whose modes are being changed. *modes* may be one or more of the following:

- s** System file (hidden from normal directory searches)
- h** Hidden file (" ")
- m** Backed-up (shows up as 'm' in **ls -l**)
- w** Write allowed (shows up as 'w' in **ls -l**)

Preceding *modes* with '+' adds the modes to a file, whereas preceding it with '-' deletes them. For example, the command

```
chmod +h example
```

adds the "hidden" mode to the file **example**. The file will be hidden from normal system searches.

Typing the command

```
ls -l example
```

will show the letter 'h' among the file's modes. See **ls** for more information about how that command presents a file's modes.

See Also

commands

chown — UNIX system call (libc)

Change ownership of a file

chown(*file, uid, gid*)

char * *file* ;

short *uid, gid* ;

chown changes the owner of *file* to user id *uid* and group id *gid*.

To change only the user id without changing the group id, **stat** should be used to determine the value of *gid* to pass to **chown**.

chown is included for compatibility with the UNIX operating system. It performs no work, and is always zero.

See Also

UNIX routines

clearerr — STDIO macro (stdio.h)

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.

Example

For an example of this function, see the entry for **ferror**.

See Also

ferror, **STDIO**

CLK_TCK — Manifest constant

CLK_TCK is a manifest constant that is set in the header file **time.h**. The draft ANSI standard defines it as being equivalent to the rate at which the system clock ticks. On the Atari ST, this is equivalent to 5 milliseconds.

See Also

manifest constants, **time**, **time.h**

clock — Time function (libc)

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 (overview)**, **time.h**

close — UNIX system call (libc)

Close a file

```
int close(fd) int fd;
```

close closes the file 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 only a limited number of files open at any given time, 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 is a command that is included with Mark Williams C. It 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 EOF on one file but not on the other, it prints the message "EOF on file". 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);
        Cwio(c);
        if(c == 0x0D)                 /* If a <RETURN> */
            Cwio(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 however, appears not to be correct.

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 Motorola to Mark Williams assembler
cat	concatenate files
cc	the compiler driver
cd	change directory
chmod	change "mode" of a file
cmp	compare two files
cp	copy a file
cpp	the C preprocessor
curconf	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
drvprs	check if drive is present
echo	repeat/expand an argument
egrep	find embedded strings
equal	test if two values are equal
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
if	execute a command conditionally
inherit	pass variable to child shell
is_set	test if an environmental variable is set
kbrate	get/set the keyboard's repeat rate
kick	force TOS to reread the floppy disk cache
lc	print directory contents in columns
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
mousehidden	print number of times mouse pointer has been hidden
msh	the Mark Williams micro-shell
mshversion	print current version of msh
msleep	suspend processing for <i>n</i> milliseconds
mtoh	redraw screen, moving from medium to high resolution
mtol	redraw screen, moving from medium to low resolution
mv	rename a file
mwtomw	convert old Mark Williams object files to 3.0 format
nm	print symbol tables
not	invert the logical value of its argument
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
rescomp	the Mark Williams resource compiler
resdecom	the Mark Williams resource disassembler
resource	the Mark Williams resource editor
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 <i>n</i> 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
time	print current time; time execution of a program
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
while	set a conditional loop

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 **.tpp**, 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 con-

structed of a sign bit; an exponent; and a *fraction*, or base upon which the exponent operates.

See Also

data formats, double, float, fraction

con — Operating system device

Logical device for the console

TOS gives names to its logical devices. Mark Williams C uses these names to allow its 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:, STDIO

Notes

con: may be spelled **con:** or **CON:**.

const — C keyword

Qualify an identifier as not modifiable

The type qualifier **const** marks an object as being unmodifiable. An object declared as being **const** cannot be used on the left side of an assignment (an *lvalue*), or have its value modified in any way. Because of these restrictions, an implementation may place objects declared to be **const** into a read-only region of storage.

See Also

C keywords, volatile

Notes

Mark Williams C does recognize this keyword, but its semantics are not implemented in release 3.0. Thus, storage declared with the **const** qualifier will *not* be treated as unmodifiable by the compiler, and no warnings will be generated.

continue — C keyword

Force next iteration of a loop

continue forces the next iteration of a **for**, **while**, or **do** loop. For example,

```
while ((foo = getchar()) != EOF) {
    if ((foo < 'a') && (foo > 'z'))
        continue;
    ...          /* do something */
}
```

forces the **while** loop to throw away everything except lower-case alphabetic characters.

See Also

C keywords, **C language**, **for**, **while**
The C Programming Language, page 62

cos — Mathematics function (libm)

Calculate cosine

#include <math.h>

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)

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 program prompts you for a number; it then uses **cosh**, **sinh**, and **tanh** to generate, respectively, the hyperbolic cosine, sine, and tangent of your number.

```
#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*.

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. **cpp** can also be run by itself.

cpp reads each input *file*; it processes directives, and writes its product on **stdout**.

If the **-E** option is not used, **cpp** also writes into its output statements of the form

#n filename, so that the parser **cc0** will be able to connect its error messages and debugger output with the original line numbers in your source files.

Options

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 to preprocess assembly-language files 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 tells **cpp** to search a specific directory for the files you have named in your **#include** directives, in addition to the directories that it searches by default. 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**. Note that you can have more than one **-I** option on your **cc** command line.

-o *file*

Write output into *file*. If this option is missing, **cpp** writes its output onto **stdout**, 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**.

Directives

cpp processes the following directives:

#assert	#ifdef
#define	#ifndef
#elif	#include
#else	#line
#endif	#undef
#if	

Each of the directives has its own entry in the Lexicon. Note that no directive can be indented on the line; if it is not set flush with left margin on the screen, **cpr** will ignore it.

See Also

cc

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.\n\r";
    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 = Crawl(0xFF)) != 0x1A) {
        Crawl(c);
        if (c == 0x0D)
            Crawl(0x0A);
    }
}
```

See Also

gemdos, **TOS**

creat — UNIX system call (libc)

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 UNIX and related operating systems.

Example

For an example of how to use this routine, see the entry for **open**.

See Also

fopen, **fdopen**, **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 — Runtime startup

Default C runtime startup

crts0.o is the runtime startup routine for C programs compiled into Mark Williams 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 another program that handles 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 — Runtime startup

C runtime startup, GEM environment

crtsd.o is the runtime startup routine for a C programs that is designed to be used as a GEM desktop accessory.

crtsd.o can be specified 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, crtso.o, crtsg.o, runtime startup

crtsg.o — 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 **crtso.o**:

1. **ARGV, ARGV, 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**, **crtso.o**, **crtsd.o**, runtime startup

ctime — Time function (libc)

Convert system time to an ASCII string

```
#include <time.h>
```

```
char *ctime(time_t 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**, which is 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 another example of this function, see the entry for **asctime**.

```
#include <time.h>
```

```
main()
```

```
{
```

```
    time_t t;
    time(&t);
    printf(ctime(&t));
```

```
}
```

See Also

time (overview), **time_t**, **time.h**

Notes

ctime returns a pointer to a statically allocated data area that is overwritten by successive calls.

ctype — Overview

```
#include <ctype.h>
```


The **ctype** macros and functions test a character's *type*, and can transform some characters into others. They are:

isalnum	test if alphanumeric character
isalpha	test if alphabetic character
isascii	test if ASCII character
isctrl	test if a control character
isdigit	test if a numeric digit
islower	test if lower-case character
isprint	test if printable character
ispunct	test if punctuation mark
isspace	test if a tab, space, or return
isupper	test if upper-case character
_tolower	change to lower-case character
_toupper	change to upper-case character

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**, **isctrl**, **isdigit**, **islower**, **isprint**, **ispunct**, and **isspace**, and the function **toupper**. It prints information about the type of characters it contains, and converts its name to upper-case characters.

```
#include <ctype.h>
#include <stdio.h>

main()
{
    FILE *fp;
    char fname[20];
    int ch,i;
    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: ");
    fflush(stdout);
    gets(fname);

    for(i=0; fname[i] != '\0'; i++)
        fname[i] = islower(fname[i]) ? toupper(fname[i])
        : fname[i];
}
```

```

if ((fp = fopen(fname, "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++;
        } else {
            printf("%s is not ASCII.\n", fname);
            exit(1);
        }
    }

    printf("%s has the following:\n", fname);
    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", fname);
}

```

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 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.

All arguments to **cursconf** can be C-style constants.

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

D

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.

See Also
process

data formats — Technical information

Mark Williams Company has written C compilers for a number of different computers. Each has a unique architecture and defines data formats in its own way.

The following table gives the sizes, in chars, of the data types as they are defined by various microprocessors.

Type	i8086 SMALL	i8086 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, which conform to all restrictions set by the microprocessor. Byte ordering is set by the microprocessor; see the Lexicon entry on **byte ordering** for more information.

See Also

byte ordering, C language, data types, declarations, double, float, memory allocation

data types — Technical information

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

Note that the terms **unsigned short int** and **unsigned short** are synonymous, as are the terms **unsigned long int** and **unsigned long**. 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] [[[cc]ymmdd]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:

<i>cc</i>	century (AD; default, "19")
<i>yy</i>	year (00-99)
<i>mm</i>	month (01-12)
<i>dd</i>	day (01-31)
<i>hh</i>	hour (00-23)
<i>mm</i>	minute (00-59)
<i>ss</i>	seconds (00-59)

Note that the century and seconds fields are optional. 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 command

```
date `date -i`
```

resets the keyboard clock and GEM-DOS times with the output of the system clock. Embedding this command in your **msh profile** will ensure that files are always date-stamped correctly.

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)

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 (overview), time.h

db — Command

Assembler-level symbolic debugger

db [-afkort] [mapfile] [datafile]

db is an assembly language-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. You can use it to debug assembly-language programs that have been assembled by **as**, the Mark Williams assembler, as well as those that have been compiled with the Mark Williams C compiler.

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. 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*:

- a Accept commands from the **aux** port. This feature allows you to plug a terminal into the Atari's **aux** port and give commands to **db** from it. The program's output is displayed on the Atari's monitor. This allows you to easily debug programs that use AES or VDI calls.
- f Map *program* as a straight array of bytes (file).

- g GEM option: turn on the mouse pointer for programs that use the GEM interface.
- k The *kernel* option. This allows a user to debug all of the Atari ST's memory. The default *symbolfile* in *tos.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.
- t Force *stdin*, *stdout*, and *stderr* to the console (keyboard and screen), regardless of redirection on the command line or in the shell.

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 set and 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), 0xFFFFC(a6)
sprintf_+0x4A movea.l   0xFFFFC(a6), a0
sprintf_+0x4E move.l    (a0), d0
sprintf_+0x50 movea.l   d0, a4
sprintf_+0x52 addq.l    $0x4, 0xFFFFC(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, $0xFFFF8
```

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 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, 0x3BF0)
```

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

Notes

Because version 3.0 changes the object format, the edition of *ld* shipped with version 3.0 does not work with objects compiled with Mark Williams C version 2.1.7 or earlier. To convert such objects to a format that *ld* recognizes, use the command *mwtoomw*.

db now supports symbol tables larger than 64 kilobytes.

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");
    Pterm(0);
}
```

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]);
    Pterm(0);
}
```

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

default — C keyword

Default label in switch statement

default is a prefix used in **switch** statement. If none of the **case** labels match the parameter in the **switch** statement, then the **default** label is used. Note that a **switch** is not required to have a **default** case, but it is good programming practice to use one.

See Also

C keywords, C language, case, switch

The C Programming Language, page 55

#define — Preprocessor instruction

Define a variable as manifest constant

#define *constant value*

#define tells the C preprocessor **cpp** to define *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.

The judicious use of **#define** instructions allows you to write code that is more easily understood, maintained, and enhanced. With them, you can modify a major parameter throughout a program by changing one line of code. They also allow you to use a variable name that suggests the function of the parameter it represents; for example, the name **MAXARGS** clearly refers to the maximum number of arguments, whereas the numeral **9** could refer to nearly anything.

See Also

cpp, manifest constant

Notes

The '#' of this instruction must appear in the *first*, or leftmost, column on a line or it will be ignored.

The present release of Mark Williams C implements the ANSI standard for the C preprocessor. Note that according to the ANSI standard, a macro expansion always occupies no more than one line, no matter how many lines the definition or the actual parameters to the macro span. If you have defined macros that span more than one line, you must either redefine them to occupy one line, or somehow embed the newline character within the macro itself; otherwise, the macro will not expand correctly.

desk accessory — Technical information

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 **rsrc_load**, remember to use **rsrc_free** before you give up control, if possible.

- Do not use `evnt_timer` calls: use `evnt_multi` instead.

Example

The following example, called `desk.c`, demonstrates how to write a desk accessory. It is based on a public-domain program written by Jan Gray in 1986. To compile it, use the following command:

```
cc -o desk.acc -VGEMACC desk.c
```

It displays a digital clock or a calendar in a window in the upper-right hand corner of the desktop.

```
#include <gemdefs.h>
#include <osbind.h>
#include <time.h>

typedef struct { int x, y, w, h; } Rectangle;
#define elements(r) r.x, r.y, r.w, r.h
#define pointers(r) &r.x, &r.y, &r.w, &r.h

char clock_s[] = "hh:mm TZT";
char calend_s[] = "ddd mmm dd yyyy";

/* Faked timezone environment for desk accessory */
char *getenv() { return "EST:300:EDT:1.1.4"; }

main()
{
    register int clock_id;      /* Clock menu identifier */
    register int calend_id;    /* Calendar menu identifier */
    register int clock_w;      /* Clock window handle */
    register int calend_w;     /* Calendar window handle */
    register int w;            /* Temporary window handle */

    Rectangle clock_r;         /* Clock window rectangle */
    Rectangle calend_r;        /* Calendar window rectangle */
    Rectangle r;               /* Temporary rectangle */
    int mb[8];                 /* Message buffer */
    int ret;                   /* Dummy return buffer */

    /* Register menu title */
    ret = appl_init();
    clock_id = menu_register(ret, " Clock");
    calend_id = menu_register(ret, " Calendar");

    /* Size window titles for templates */
    graf_handle(pointers(r));
    clock_r.w = r.w + r.x * sizeof(clock_s);
    clock_r.h = r.h;
    calend_r.w = r.w + r.x * sizeof(calend_s);
    calend_r.h = r.h;

    /* Position window at upper right corner */
    wind_get(0, WF_FULLXYWH, pointers(r));
    clock_r.x = r.w - clock_r.w; clock_r.y = r.y;
    calend_r.x = r.w - calend_r.w; calend_r.y = r.y;
}
```

```
/* Initialize window handles as closed */
clock_w = calend_w = -1;
for (;;) {
/* Await message or timer event */
if (MU_MESAG & evt_multi(
    MU_MESAG | MU_TIMER,
    0, 0, 0,
    0, 0, 0, 0, 0,
    0, 0, 0, 0, 0,
    mb, 30000, 0, /* 30 second timer interval */
    &ret, &ret, &ret, &ret, &ret, &ret)) {
    switch (mb[0]) {
/* Accessory menu line selected */
case AC_OPEN:
    if (mb[4] == clock_id) {
        w = clock_w;
        r = clock_r;
    } else if (mb[4] == calend_id) {
        w = calend_w;
        r = calend_r;
    } else
        break;
    if (w > 0) {
        wind_set(w, WF_TOP, 0, 0, 0, 0);
        break;
    }

    w = wind_create(NAME|CLOSER|MOVER, elements(r));
    if (w > 0) {
        if (mb[4] == clock_id) {
            clock_w = w;
            wind_set(w, WF_NAME, clock_s, 0, 0);
        } else {
            calend_w = w;
            wind_set(w, WF_NAME, calend_s, 0, 0);
        }
        wind_open(w, elements(r));
    }
    break;
/* Screen manager restart */
case AC_CLOSE:
    if (mb[3] == clock_id)
        clock_w = -1;
    else if (mb[3] == calend_id)
        calend_w = -1;
    break;
}
```

```
/* Window close box selected */
case WM_CLOSED:
    w = mb[3];
    if (w == clock_w)
        clock_w = -1;
    else if (w == calend_w)
        calend_w = -1;
    else
        break;
    wind_close(w);
    wind_delete(w);
    break;

/* Window dragged to new position */
case WM_MOVED:
    w = mb[3];
    r = *(Rectangle*)(mb+4);
    if (w == clock_w)
        clock_r = r;
    else if (w == calend_w)
        calend_r = r;
    else
        break;
    wind_set(w, WF_CURRXYWH, elements(r));
    break;

case WM_NEWTOP:
case WM_TOPPED: /* Window clicked to top */
    w = mb[3];
    if (w != clock_w && w != calend_w)
        break;
    wind_set(w, WF_TOP, 0, 0, 0, 0);
    break;
}

/* Update time on each event if window is open */
if (clock_w > 0 || calend_w > 0) {
    register struct tm *tp;
    register char *p;
    time_t tt;

    time(&tt);
    tp = localtime(&tt);
    p = asctime(tp);
```



```
calend_s[0] = *p++;
calend_s[1] = *p++;
calend_s[2] = *p++;
calend_s[3] = *p++;
calend_s[4] = *p++;
calend_s[5] = *p++;
calend_s[6] = *p++;
calend_s[7] = *p++;
calend_s[8] = *p++;
calend_s[9] = *p++;
calend_s[10] = *p++;

clock_s[0] = *p++;
clock_s[1] = *p++;
clock_s[2] = *p++;
clock_s[3] = *p++;
clock_s[4] = *p++;
p += 3;
clock_s[5] = *p++;

calend_s[11] = *p++;
calend_s[12] = *p++;
calend_s[13] = *p++;
calend_s[14] = *p++;
p = tp->tm_isdst <= 0 ? tzname[0] : tzname[1];

clock_s[6] = *p++;
clock_s[7] = *p++;
clock_s[8] = *p++;

if (clock_w >= 0)
    wind_set(clock_w, WF_NAME, clock_s, 0, 0);
if (calend_w >= 0)
    wind_set(calend_w, WF_NAME, calend_s, 0, 0);
    }
}
```

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 currently using.

The option **-a** prints the amount of space left on all devices.

See Also

commands, **mf**, **msh**

Dfree — gemdos function 54 (osbind.h)

Get information on a drive's free space

```
#include <osbind.h>
```

```
void Dfree(fs, drive) long fs[4]; int drive;
```

Dfree retrieves information about free space on a disk drive.

fs is an array of four **unsigned longs** into which **Dfree** writes, respectively, the number of free allocation units (also called "clusters") on a disk; the total number of allocation units on the disk; the size of a sector, in bytes; and the size of each allocation unit, in sectors. If you prefer, you can pass *fs* as a pointer to a structure of four **longs**.

drive is the number of the disk drive you wish to check, 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 %ld bytes free in %ld sectors\n",
           dd+'A', fb, fs);
    printf("from total of %ld bytes in %ld sectors (cluster size %ld)\n",
           tb, ts, disk.di_spau*disk.di_ssize);
}
```

See Also

gemdos, TOS

Dgetdrv – gemdos function 25 (osbind.h)

Find current default disk 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];
    char *buf;

    buf = pathbuf;
    *buf++ = (drv=Dgetdrv())+'A';
    *buf++ = ':';
    Dgetpath(buf, drv+1);
    printf("Current path is %s\n", pathbuf );
}
```

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).

Notes

diff cannot handle files with more than 32,000 lines. Handing **diff** a file that exceeds that limit will cause it to fail, with unpredictable side effects.

difftime — Time function (libc)

Return difference between two times

#include <time.h>

double difftime(newtime, oldtime) time_t newtime, oldtime;

difftime calculates the difference in seconds between *newtime* and *oldtime*.

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 (overview), **time.h**

directory — Definition

A **directory** is a table 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**

do — C keyword

Introduce a loop

do is a C control statement that introduces a loop. Unlike **for** and **while** loops, the condition in a **do** loop is evaluated *after* the operation is performed. **do** always works in tandem with **while**; for example

```
do {  
    puts("Next entry? ");  
    fflush(stdout);  
} while(getchar() != EOF);
```

prints a prompt on the screen and waits for the user to reply. The **do** loop is convenient in this instance because the prompt must appear at least once on the screen before the user replies.

See Also

break, C keywords, C language, **continue**, **while**
The C Programming Language, page 59

Dosound — xbios function 32 (osbind.h)

Start up the sound daemon

```
#include <osbind.h>
```

```
long 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 set how long a tone is sustained.

When *buffer* points to a list of commands, **Dosound** returns the old pointer if the sound daemon was active, and NULL if it was not. If *buffer* is set to -1L, **Dosound** returns zero if the sound daemon is idle, and a number greater than zero if it is not. This will be helpful in constructing musical resources.

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 — C keyword

Data type

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**. If you wish your code to be portable, do *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 described in the entry for **float**. Mark Williams C always uses DECVAX format.

See Also

C keywords, C language, data formats, declarations, float, portability

The C Programming Language, page 34

drtomw — Command

Convert from DRI to Mark Williams format

drtomw 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.

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 have 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 executable 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 executable file, **drtomw** reports this and aborts the conversion.

See Also

as, **as68toas**, **commands**, **mwtomw**

Notes

The edition of **drtomw** that is shipped with version 3.0 transforms DRI objects into the new Mark Williams object format.

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**

drvprs — Command

Check if a drive is present on the machine

drvprs [-q] *drive*

The command **drvprs** checks to see if *drive* is present on the machine, where *drive* is the name of a floppy drive, a logical drive on a hard disk, or a RAM disk.

The option **-q** suppresses the message that this command normally returns.

drvprs returns a status value so that it can be used in a **msh** script conditional.

Notes

When a program exits, GEMDOS always resets the current drive and the current directory to what they were before the program was run.

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)

Duplicate a file descriptor
int 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

For an example of this function, see the entry for **system**.

See Also

fopen, fdopen, 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)

Duplicate a file descriptor

int dup2(*fd, newfd*) int *fd, newfd*;

dup2 duplicates the file descriptor *fd*. Unlike its cousin **dup**, **dup2** allows you 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* less than six.

Example

For an example of this function, see the entry for **system**.

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.

E

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)

Convert floating-point numbers to strings

char *ecvt(*d*, *prec*, *dp*, *signp*) **double** *d*; **int** *prec*, **dp*, **signp*;

ecvt converts *d* into a NUL-terminated ASCII string of numerals with the precision of *prec*. Its operation resembles that of **printf**'s %e operator. **ecvt** 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.

Example

The following program demonstrates **ecvt**, **fcvt**, and **gcvt**.

```
char *ecvt(), *fcvt(), *gcvt();
main()
{
    char buf[64];
    double d;
    int i, j;
    char *s, *strcpy();

    d = 1234.56789;
    s = ecvt(d, 5, &i, &j);
    printf("ecvt=\"%s\" i=%d j=%d\n", s, i, j);
    /* prints ecvt="12346" i=4 j=0 */
}
```

```
strcpy(s, fcvt(d, 5, &i, &j));
printf("fcvt=\"%s\" i=%d j=%d\n", s, i, j);
/* prints fcvt="123456789" i=4 j=0 */

s = gcvt(d, 5, buf);
printf("gcvt=\"%s\"\n", s);
/* prints gcvt="1234.56789" */
}
```

See Also

fcvt, frexp, gcvt, ldexp, modf, printf

Notes

ecvt performs conversions within static string buffers that are overwritten by each execution.

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 it 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*.

Wildcards

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 literal characters must be quoted by enclosing *pattern* within double quotation marks.

Options

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. This is useful when you are constructing a batch file.
- n When a line is printed, also print its number within the file.

- s Suppress all output, just return exit 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 zero for success, one for no matches, and two 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 earlier pattern-searching commands, on almost any length of file.

#elif — Preprocessor instruction

Include code conditionally

#elif (*expression*)

#elif is an instruction interpreted by the C preprocessor **cpp**. It can be used within a conditional expression begun with the instructions **#if**, **#ifdef**, or **#ifndef**.

This instruction tells **cpp** that if the condition named in the preceding **#if** or **#ifdef** expression is false and if the current condition succeeds, then include the following lines of code up to the next **#else** or **#endif** instruction.

An **#elif** command can also be coupled with **#else** commands to create several levels of conditions. Note that a conditional expression can have any number of **#elif** statements. For example:

```
#if (condition1)
    int foo = 1;
#elif (condition2)
    int foo = 2;
#elif (condition3)
    int foo = 3;
#else
    int foo = 4;
#endif
```

See Also

cc, **cpp**, **#else**, **#endif**, **#if**, **#ifdef**

Notes

Note that all preprocessor commands must be set flush with the left margin, or they will not be executed by `cpp`.

else — C keyword

Introduce a conditional statement

else is the flip side of an **if** statement: if the condition described in the **if** statement fails, then the statements introduced by the **else** statement are executed. For example,

```
if (getchar() == EOF)
    exit(0);
else
    dosomething();
```

exits if the user types **EOF**, but does something if the user types anything else.

See Also

C keywords, **C language**, **if**

The C Programming Language, pages 51, 53

#else — Preprocessor instruction

Include code conditionally

#else

#else is an instruction interpreted by the C preprocessor `cpp`. It can be used within a conditional expression initiated by the instructions **#if**, **#ifdef**, or **#ifndef**.

This instruction tells `cpp` that if the conditions named in the preceding **#if** and **#elif** instructions are false, then include the following lines of code up to the next **#endif** instruction. Note that a conditional expression can include any number of **#elif** statements, but can have only one **#else** statement. For example:

```
#if (condition1)
    int foo = 1;
#elif (condition2)
    int foo = 2;
#elif (condition3)
    int foo = 3;
#else
    int foo = 4;
#endif
```

See Also

cc, **cpp**, **#elif**, **#endif**, **#if**, **#ifdef**, **#ifndef**

Notes

Note that all preprocessor commands must be set flush with the left margin, or they will not be executed by **cpp**.

end — Linker-defined symbol

```
extern int end[];
```

end is the location after the uninitialized data segment; it is defined by the linker when it 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**

__end — External data

```
extern char * __end;
```

__end is an external variable that points to the end of your program's data space. It is set by the C runtime startup, and can be incremented by the function **sbrk**.

See Also

malloc, **maxmem**, **sbrk**

#endif — Preprocessor instruction

End conditional inclusion of code

#endif

#endif is an instruction interpreted by the C preprocessor **cpp**. It ends a conditional expression. For example:

```
#if (condition1)
    int foo = 1;
#elif (condition2)
    int foo = 2;
#elif (condition3)
    int foo = 3;
#else
    int foo = 4;
#endif
```

See Also

cc, **cpp**, **#elif**, **#else**, **#if**, **#ifdef**, **#ifndef**

Notes

Note that all preprocessor commands must be set flush with the left margin, or they will not be executed by **cpp**.

entry — C keyword

Undefined keyword

entry is a C key word that is reserved for future use.

See Also

C keywords, C language

Notes

The draft ANSI standard for the C language eliminates **entry** from the table of keywords.

enum — C keyword

Declare a type and identifiers

An **enum** declaration is a data type whose syntax resembles those of the **struct** and **union** declarations. It lets you enumerate the legal value for a given variable. For example,

```
enum opinion {yes, maybe, no} GUESS;
```

declares type **opinion** can have one of three values: **yes**, **no**, and **maybe**. It also declares the variable **GUESS** to be of type **opinion**.

As with a **struct** or **union** declaration, the tag (**opinion** in this example) is optional; if present, it may be used in subsequent declarations. For example, the statement

```
register enum opinion *op;
```

declares a register pointer to an object of type **opinion**.

All enumerated identifiers must be distinct from all other identifiers in the program. The identifiers act as constants and be used wherever constants are appropriate.

Mark Williams C assigns values to the identifiers from left to right, normally beginning with zero and increasing by one. In the above example, the values of **yes**, **no**, and **maybe** are set, respectively, to one, two, and three. 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 one from that value; for example,

```
enum opinion {yes=50, no, maybe} guess;
```

sets the values of the identifiers **yes**, **no**, and **maybe** to 50, 51, and 52, respectively.

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

C keywords, C language, 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

environment, envp

environment — Definition

The **environment** is a set of information that you wish to pass to all programs run on your system. It consists of one or more **environmental variables** that you set; for example, when you set the environmental variable **PATH**, you tell TOS that you wish to pass this information to all programs on your system, including TOS itself.

By changing the environment, you can change the way a command works without rewriting any commands that you may have embedded in batch files, scripts, or **makefiles**.

Mark Williams C's compiler controller `cc` uses the environment extensively to find its subordinate programs and files. For example, the environmental variable `INCDIR` tells `cc` where to find its header files. Embedding the command sets `INCDIR` to indicate the directory `include` on drive A. `cc` will pass this information to the C preprocessor `cpp.prg`, which will then look in that directory for the files that are called with an `#include` statement.

See Also

envp — Definition

Argument passed to main

```
char *envp[];
```

`envp` is an abbreviation for **environmental parameter**. 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`.

See the Lexicon entry for `argv` for more information how `envp` and `argv` work together to pass information into a program.

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

The following example echoes characters you type at the keyboard until you type EOF.

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

See Also

manifest constant, stdio.h

equal — Command

Compare two arguments

equal *argument1 argument2*

equal is a test command that is built into **msh**. It tests if *argument1* is equal to *argument2*; it returns zero if they are equal, and a value greater than zero if they are not. The arguments may be absolute values or values returned by another command.

Example

The following command prints the string **High res** on the screen if the monitor is in high resolution, and prints **Not high res** if it is not.

```
if (equal 'getrez' 2) (echo "High res") (echo "Not high res")
```

Note that the command **getrez** returns two if the monitor is in high resolution.

See Also

commands, if, is_set, msh, not, while

errno — UNIX data

External integer for return of error status

extern int errno;

errno is an external integer that Mark Williams C sets to the negative value of any error status returned by TOS to the functions that emulate UNIX system calls. The function **perror** or the array **sys_errlist** can be used to translate the **errno** into text.

The number stored in **errno** is not the TOS error, but is often the absolute value of it. Because all TOS errors are negative, and are defined as such in **errno.h**, the value stored in **errno** must be negated before it can be used to determine what error occurred.

Mathematical functions also use **errno** to indicate classifications of errors on return. **errno** 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.

Example

For an example of using **errno** in a mathematics program, see the entry for **acos**.

See Also

errno.h, error codes, mathematics library, perror, UNIX routines

errno.h — Header file

Error numbers used by errno()

#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 — Technical information

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
AEGBSF	-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)

Await a specific mouse button event

```
#include <aesbind.h>
```

```
int evnt_button(clicks, button, state, xptr, yptr, bptr, kptr)
```

```
int clicks, button, state, *xptr, *yptr, *bptr, *kptr;
```

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.

xptr points to an integer that holds the X coordinate of the mouse pointer. *yptr* points to an integer that holds the Y coordinate of the mouse pointer. *bptr* points to an integer that indicates the button state when the event occurred. Finally, *kptr* points to an integer that represents the states of the control, alt, and shift keys OR'd together, as follows:

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

evnt_button can wait for only one button event. 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)

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)

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 an **int**: the high byte contains the scan code of the key pressed, and the low byte contains the ASCII code, if any, modified by the **<ctrl>** or **<shift>** keys.

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

alertf(n, p) int n; char *p;
{
    static char buffer[512];
    sprintf(buffer, "%r", &p);
    return form_alert(n, buffer);
}

main()
{
    unsigned key;
    appl_init();

    for(;;) {
        key = evnt_keybd();

        switch(key) {
            case RETURN:
                appl_exit();
                exit(0);

            default:
                alertf(1, "[1] [The scan code is: %x] [OK]", key);
                continue;
        }
    }
}

```

See Also

AES, keyboard, TOS

evnt_mesag — AES function (libaes)

Await a message

```
#include <aesbind.h>
```

```
int evnt_mesag(buffer) int buffer[8];
```

evnt_mesag is the AES routine that awaits a message. *buffer* is an array of eight integers that holds the message.

GEM uses 12 predefined messages to pass information among its applications. Each is eight ints (or “words”) long, and each has the following structure:

Word 0	type of message
Word 1	handle of application that sent the message
Word 2	no. of bytes in message beyond 16
Words 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's 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: <table><tr><td>0</td><td>Page up</td></tr><tr><td>1</td><td>Page down</td></tr><tr><td>2</td><td>Row up</td></tr><tr><td>3</td><td>Row down</td></tr><tr><td>4</td><td>Page left</td></tr><tr><td>5</td><td>Page right</td></tr><tr><td>6</td><td>Column left</td></tr><tr><td>7</td><td>Column right</td></tr></table>	0	Page up	1	Page down	2	Row up	3	Row down	4	Page left	5	Page right	6	Column left	7	Column right
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 lowest position, and 1,000 the highest.																
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 line in the desk menu that was clicked to open the application. Word 4 gives the desk accessory's menu item identifier, as set by the function <code>menu_register</code> .																
AC_CLOSE	(desk accessory closed) Word 3 gives the desk accessory's menu item identifier, as set by the function <code>menu_register</code> .																

evnt_mesag always returns one.

Example

For an example of this routine, see the entry for **window**.

See Also

AES, **TOS**, **window**

Notes

The information included with the message **AC_CLOSE** does not appear to conform to the description given in DRI documentation. The description given above is the result of our experience in working with **evnt_mesag**. Users who wish to perform sophisticated tasks, such as passing messages between applications, should be on guard that the correct information is being passed.

evnt_mouse — AES function (libaes)

Wait for mouse to enter specified rectangle

```
#include <aesbind.h>
```

```
int evnt_mouse(inout, x, y, w, h, xptr, yptr, bptr, kptr)
```

```
int inout, x, y, w, h, *xptr, *yptr, *bptr, *kptr;
```

evnt_mouse is the AES routine that waits for the mouse pointer to enter or leave a specified rectangular space 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.

The arguments *x*, *y*, *w*, and *h* hold, respectively, the X coordinate of the target rectangle, its Y coordinate, its width, and its height; all are in rasters.

xptr points to an integer that holds the X coordinate of the mouse pointer. *yptr* points to an integer that holds the Y coordinate of the mouse pointer. *bptr* points to an integer that indicates the button state when the event occurred: zero indicates up and one indicates down. Finally, *kptr* points to an integer that represents the states of the control, alt, and shift keys OR'd together, as follows:

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)

Await one or more specified events

```
#include <aesbind.h>
```

```
int evnt_multi(events, clicks, button, state, mlinout, x1, y1, w1, h1,
               m2inout, x2, y2, w2, h2, buffer, lowtime, hightime, xptr, yptr, bptr,
               kptr, key, times)
```

```
int events, clicks, button, state, mlinout, x1, y1, w1, h1;
int m2inout, x2, y2, w2, h2, buffer[8], lowtime, hightime;
int *key, *times, *xptr, *yptr, *bptr, *kptr;
```

evnt_multi is an AES routine that awaits any one of a set of AES 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:

0x01	leftmost button
0x02	second button from left
0x04	third button from left

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 of two mouse events. A “mouse event” occurs when the mouse pointer either enters into or exits from a defined rectangular space on the screen. *mlinout* indicates whether the process is waiting for the mouse pointer to enter (zero) or exit (one) the first mouse rectangle. Note that the screen manager constantly polls 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. The arguments *x1*, *y1*, *w1*, and *h1* define, respectively, the X point of the rectangle to be watched, its Y point, its width, and its height.

m2inout plus *x2*, *y2*, *w2*, and *h2* define the second mouse event. They are defined in exactly the same manner as the arguments for the first mouse event.

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.

xptr points to an integer that holds the X coordinate of the mouse pointer when an awaited event occurs. *yptr* points to an integer that holds the Y coordinate of the mouse pointer. *bptr* points to an integer that indicates the button state when the event occurred. Finally, *kptr* points to an integer that represents the states of the control, alt, and shift keys OR'd together, as follows:

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 (that is, if the user presses a key on the keyboard), *key* points to the code of the key pressed. See the Lexicon 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>

struct ( int x, y, w, h; ) Rectangle;

/* place for unused pointers to point at */
int nowhere[11];

main()
{
    /* declarations for window */
    int handle;
    char *title = " TITLE ";

    /* declarations for evnt_multi() */
    unsigned int which = (MU_KEYBD | MU_M1 | MU_M2);

    /* no. of times mouse button enters state */
    int times = 0;
    appl_init();
```

```

/* get dimensions of desktop window */
wind_get(0, WF_WORKXYWH, &Rectangle.x, &Rectangle.y,
         &Rectangle.w, &Rectangle.h);

/* alter size of Rectangle */
Rectangle.x = Rectangle.w/3;
Rectangle.y = Rectangle.h/3;
Rectangle.w /= 3;
Rectangle.h /= 3;

/* create window */
handle = wind_create(NAME, Rectangle);

/* set window, open */
wind_set(handle, WF_NAME, title, 0, 0);
graf_growbox(0, 0, 0, 0, Rectangle);
wind_open(handle, Rectangle);

for(;;) {
    switch(evnt_multi(which, 1, 1, 1, 0,
                     Rectangle, 1, Rectangle, nowhere,
                     nowhere[0], nowhere[0], nowhere,
                     nowhere, nowhere, nowhere, nowhere,
                     &times)) {

        case MU_KEYBD:
            wind_close(handle);
            graf_shrinkbox(0, 0, 0, 0, Rectangle);
            appl_exit();
            exit(0);

        case MU_M1:
            graf_mouse(ARROW, nowhere);
            which = (MU_KEYBD | MU_M2);
            continue;

        case MU_M2:
            graf_mouse(BUSY_BEE, nowhere);
            which = (MU_KEYBD | MU_M1);
            continue;

        default:
            continue;
    }
}

```

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.

evnt_timer — AES function (libaes)

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.

Example

For an example of this function, see the entry for **VDI**.

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 GDOS routine. For more information on GDOS, 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 both been *compiled*, where it is rendered into machine language, and *linked*, where the compiled program has received all operating system-specific information and library functions.

See Also

file

execve — UNIX system call (libc)

Execute a command from within a program

int 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; it 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, UNIX routines

exit — General function (libc)

Terminate a program

void exit(status) int status;

exit terminates a program gracefully. It flushes all buffers, closes each open file, and then returns *status* to the calling program. By convention, an exit status of zero indicates success, whereas an exit status greater than zero indicates failure. Some systems, such as the Series III under ISIS, throw away the status. On TOS, it is returned to the parent program as the result of **Pexec**.

Example

For an example of this function, see the entry for **fopen**.

See Also

_exit, runtime startup, system, UNIX routines

The C Programming Language, page 154

exit — Command

Exit from a **msh** shell

exit [status]

exit terminates the lowest level of the shell **msh**. If you are working in a sub-shell, such as MicroEMACS invokes with its command **<ctrl-X>!**, **exit** returns you to the program that invoked the sub-shell; otherwise, **exit** terminates **msh** and returns you to the GEM desktop. **msh** executes **exit** directly. The optional argument *status* is an integer which is returned as the exit status.

See Also

commands, msh

_exit — UNIX system call (libc)

Terminate a program

int _exit(status) int status;

_exit terminates a program directly. It returns *status* to the calling program, and exits.

Unlike the library function **exit**, **_exit** does not perform extra termination cleanup, such as flushing buffered files and closing open files.

_exit should be used only in situations where you do *not* want buffers flushed or files closed; for example, when your program detects an irreparable error condition, and you want to “bail out” to keep your data files from being corrupted.

_exit should also be used with programs that do not use **STDIO**. Unlike **exit**, **_exit** does not use **STDIO**. This will help you create programs that are extremely

small when compiled.

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)

Compute exponent

#include <math.h>

double exp(z) double z;

exp returns the exponential of *z*, or e^z .

Example

The following program prompts you for a number, then prints the value for it as returned by **exp**, **pow**, **log**, and **log10**.

```
#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 — C keyword

Declare storage class

extern indicates that a C element belongs to the *external* storage class. Both variables and functions may be declared to be **extern**. Use of this keyword tells the C compiler that the variable or function is defined outside of the present file of source code. All functions and variables 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, the following declares the array **tzname**:

```
extern char tzname[2][32];
```

When a function calls a function that is defined in another source file or in a library, it should declare the function to be **extern**. 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 (such as on the 68000 or i8086 LARGE model), a **long**, or a **double**.

For example, the function **malloc** appears in a library and returns a pointer; therefore, it should be declared as follows:

```
extern char *malloc();
```

If you do not do so, Mark Williams C will assume that **malloc** returns an **int**, and generate the error message

integer pointer pun

when you attempt to use **malloc** in your program.

See Also

auto, **C keywords**, **C language**, **pun**, **register**, **static**, **storage class**

The C Programming Language, pages 28, 72, 204

F

fabs — Mathematics function (libm)

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 that 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 *attrtable[] = {
    "read only",
    "hidden",
    "system file",
    "volume label",
    "subdirectory",
    "written to and closed"
};
```

```

main(argc, argv) int argc; char **argv; {
    int 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");
        Pterm(0);
    }
    point = 1;
    for (i=0 ; i<6 ; i++) {
        if (point & attribs)
            printf(" (%s)", attrtable[i]);
        point <<= 1;
    }
    printf("\n");
}

```

See Also

gemdos, TOS

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

fclose — STDIO function (libc)

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 if an error occurs.

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. If a file could be created, **Fcreate** returns a handle with which it can be accessed through TOS. If a file could not be created, **Fcreate** returns an error code. Note that all TOS error codes are negative.

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=Ffirst(argv[1], 0xF7)) != 0) {
    Fclose(inhand);
    fprintf(stderr, "\nError getting stats on input file %s",
            argv[1]);
    errno = -status;
    perror("Ffirst 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);
    Pterm(0);
}
```

See Also

gemdos, **TOS**

fcvt — General function (libc)

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. Its operation resembles that of the **%f** operator to **printf**. It converts *d* into a NUL-terminated string of decimal digits with a precision (i.e., the number of characters to the right of the decimal point) of *prec*. 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* to point to an integer that indicates the sign of *d*: zero if positive, and nonzero if negative. **fcvt** rounds the result to the FORTRAN F-format.

Example

For an example of this function, see the entry for `ecvt`.

See Also

`ecvt`, `frexp`, `gcvt`, `ldexp`, `modf`, `printf`

Notes

`fcvt` performs conversions within static string buffers that are overwritten by each execution.

Fdatetime — gemdos function 87 (osbind.h)

Get or set a file's date/time stamp

```
#include <osbind.h>
```

```
long Fdatetime(info, handle, getset)
```

```
int handle, getset, info[2];
```

Fdatetime 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 will either has the time/date stamp written into it, or 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:

<i>info</i> [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)
<i>info</i> [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).

Fdatetime returns an error status, or zero.

Example

The following example demonstrates **Fdatetime**.

```
#include <osbind.h>
```

```
#include <errno.h>
```

```
#include <time.h>
```

```
main(argc, argv)
```

```
int argc; char *argv[]; {
```

```
    int fd;
```

```
    rtetd_t rtd;
```

```
    utetd_t utd;
```

```
    time_t t;
```

```
    tm_t *tp;
```

```
    /* Backwards time, date */
```

```
    /* Forwards date, time */
```

```
    /* COHERENT time */
```

```
    /* Time fields */
```

```

    if (argc < 2) {
        printf("Usage: Fdatetime <filename>\n");
        exit(1);
    }

    if ((fd = Fopen(argv[1], 0)) < 0) {
        errno = -fd;
        perror(argv[1]);
        exit(1);
    }

    Fdatetime(&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 environmental parameter **TIMEZONE**. Therefore, during the summer months, the time returned by this routine may be one hour behind the time returned by the **date** command.

Note, too, that **Fdatetime** overwrites the time/date buffer, even when you are setting the time and date on a file.

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**

fdopen — STDIO function (libc)

Open a stream for standard I/O

```
#include <stdio.h>
```

FILE *fdopen(*fd*, *type*) int *fd*; char **type*;

fdopen allocates and returns a **FILE** structure, or *stream*, for the file descriptor *fd*, as obtained from **open**, **creat**, or **dup**. *type* is the manner in which you want *fd* to be opened, as follows:

r	read a file
w	write into a file
a	append onto a file

Example

The following example obtains a file descriptor with **open**, and then uses **fdopen** to build a pointer to the **FILE** structure.

```
#include <ctype.h>
#include <stdio.h>

main(argc, argv)
int argc; char *argv[];
{
    extern FILE *fdopen();
    FILE *fp;
    int fd;
    int holder;

    if (--argc != 1)
        adios("Usage: example filename");

    if ((fd = open(argv[1], 0)) == -1)
        adios("open failed.");
    if ((fp = fdopen(fd, "r")) == NULL)
        adios("fdopen failed.");

    while ((holder = fgetc(fp)) != EOF)
    {
        if ((holder > '\177') && (holder < ' '))
            switch(holder)
            {
                case '\t':
                case '\n':
                    break;
                default:
                    fprintf(stderr, "Seeing char %d\n", holder);
                    exit(1);
            }
        fputc(holder, stdout);
    }

    adios(message)
    char *message;
    {
        fprintf(stderr, "%s\n", message);
        exit(1);
    }
}
```

See Also

creat, dup, fopen, open, STDIO

Diagnostics

fdopen returns **NULL** if it cannot allocate a **FILE** structure. Currently, only 20 **FILE** structures can be allocated per program, including **stdin**, **stdout**, and **stderr**.

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 zero 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>

int feof(fp) FILE *fp;

feof is a macro that tests the status of the argument stream *fp*. It returns a number other than zero if *fp* has reached the end of file, and zero if it has not. 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>

int ferror(fp) FILE *fp;

ferror is a macro that tests the status of the file stream *fp*. It returns a number other than zero if an error has occurred on *fp*. Any error condition that is dis-

covered 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.

Example

This example reads a word from one file and writes it into another.

```
#include <stdio.h>

main()
{
    FILE *fpin, *fpout;
    int word;
    char infile[20], outfile[20];

    printf("Name data file you wish to copy:\n");
    gets(infile);
    printf("Name new file:\n");
    gets(outfile);

    if ((fpin = fopen(infile, "rb")) != NULL) {
        if ((fpout = fopen(outfile, "wb")) != NULL) {
            while ((word = fgetw(fpin)) != EOF)
                fputw(word, fpout);

            if (ferror(fpin)) {
                clearerr(fpin);
                printf("Read error\n");
                exit(0);
            }
        }
        else
            printf("Cannot open output file\n");
    }
    else
        printf("Cannot open output file\n");
    fclose(fpin);
    fclose(fpout);
}
```

See Also

STDIO

fflush — STDIO function (libc)

Flush output stream's buffer

```
#include <stdio.h>
```

```
int fflush(f) FILE *f;
```

fflush flushes any buffered output data associated with the file stream *f*. The file stream stays open after **fflush** is called. `fclose` calls **fflush**, so there is no need for you to call it when normally closing a file or buffer.

Example

For an example of this routine, see the entry for **v_gtext**.

See Also

buffer, **gets**, **STDIO**, **write**

Diagnostics

fflush returns **EOF** if it cannot flush the contents of the buffers; otherwise it returns a meaningless value.

Note, also, that all **STDIO** routines are buffered. **fflush** should be used to flush the output buffer if you follow a **STDIO** routine with an unbuffered routine, such as **Cconin**.

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**)

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 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, Ffirst, Fsnx, gemdos, TOS

fgets — **STDIO** function (libc)

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 either *n*-1 characters have been read, or a newline or EOF is encountered. It retains the newline, if any, and appends a NUL character at the end 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 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)

Read integer from stream

#include <stdio.h>

int fgetw(*fp*) **FILE** **fp*;

fgetw is a function that reads an integer from the stream *fp*.

Example

This example copies one binary file into another. It demonstrates the functions **fgetw** and **fputw**.

```
#include <stdio.h>
```

```
main()
```

```
{
```

```
    FILE *fpin, *fpout;
```

```
    int word;
```

```
    char infile[20], outfile[20];
```

```
    printf("Name data file to copy:\n");
```

```
    gets(infile);
```

```
    printf("Name new file:\n");
```

```
    gets(outfile);
```

```
    if ((fpin = fopen(infile, "rb")) != NULL)
```

```
    {
```

```
        if ((fpout = fopen(outfile, "wb")) != NULL)
```

```
        {
```

```
            while ((word = fgetw(fpin)) != EOF)
```

```
                fputw(word, fpout);
```

```
            if (!ferror(fpin))
```

```
                printf("Read error\n");
```

```
        } else
```

```
            printf("Cannot open output file\n");
```

```
    } else
```

```
        printf("Cannot open output file\n");
```

```
    fclose(fpin);
```

```
    fclose(fpout);
```

```
}
```

See Also

fputw, **STDIO**

Notes

fgetw returns EOF on errors. A call to **feof** or **ferror** may be necessary to distinguish this value from a genuine end-of-file signal.

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 — Definition

A **file** is a mass of bits that has been given a name and is stored on a nonvolatile medium. These bits may form ASCII characters or machine-executable data. 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.

To manipulate the contents of a file, you must first open it. This can be done with the UNIX-compatible routine **open**, or with the function **fopen**. You can then read the file, write material to it, or append material onto it with the low-level UNIX-system calls **read** and **write**, or with the functions **fread** and **fwrite**. See the entries on **UNIX system calls** and **STDIO** for more information on manipulating material within a file.

See Also

close, **executable file**, **fopen**, **fclose**, **FILE**, **open**

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

Descriptor for a file stream

#include <stdio.h>

FILE describes a *file stream* which can be either a file on disk 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**, **fdopen**, and related functions.

The **FILE** structure is as follows:

```
typedef struct      FILE
{
    unsigned char *_cp,
                  *_dp,
                  *_bp;
    int    _cc;
    int    (*_gt)(),
          (*_pt)();
    int    _ff;
    char   _fd;
    int    _uc;
} FILE;
```

_cp points to the current character in the file. **_dp** points to the start of the data within the buffer. **_bp** points to the file buffer. **_cc** is the size of the file, in characters. **_gt** and **_pt** point, respectively, to the function **getc** and **putc**. **_ff** is a bit map that holds the various file flags, as follows:

_FINUSE	0x01	unused
_FSTBUF	0x02	used by macro setbuf
_FUNGOT	0x04	used by ungetc
_FEOF	0x08	tested by macro feof
_FERR	0x10	tested by macro ferror
_FASCII	0x20	file is in ASCII mode
_FWRITE	0x40	file is opened for writing
_FDONTC	0x80	don't close file

_fd is the file descriptor, which is used by low-level routines like **open**; it is also used by **reopen**. Finally, **_uc** is the character that has been “ungotten” by **ungetc**, should it be used.

See Also

fopen, **freopen**, **stdio.h**, **stream**

file descriptor — Definition

A **file descriptor** is an integer between 1 and 20 that indexes an area in **_psbase**, which, in turn, points to the operating system's internal file descriptors. 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** stream, 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".

See Also

file, **FILE**, **UNIX** routines

fileno — **STDIO** function (**libc**)

Get file descriptor

```
#include <stdio.h>
```

```
int fileno(fp) FILE *fp;
```

fileno returns the file descriptor associated with the file stream *fp*. The file descriptor is the integer returned by **open** or **creat**; it is used by routines such as **fopen** used to create a **FILE** stream.

Example

This example reads a file descriptor and prints it on the screen.

```
#include <stdio.h>

main(argc,argv)
int argc; char *argv[];
{
    FILE *fp;
    int fd;

    if (argc !=2)
    {
        printf("Usage: fd_from_fp filename\n");
        exit(0);
    }

    if ((fp = fopen(argv[1], "rw")) == NULL)
    {
        printf("Cannot open input file\n");
        exit(0);
    }

    fd = fileno(fp);
    printf("The file descriptor for %s is %d\n",
        argv[1], fd);
}
```

See Also

FILE, file descriptor, STDIO

flexible arrays — Definition

Flexible arrays are arrays whose length is not declared explicitly. Each has exactly one empty '[' array-bound declaration. If the array is multidimensional, the flexible dimension of the array must be the *first* array bound in the declaration; for example:

```
int example1[][20]; /* RIGHT */
int example2[20][]; /* WRONG */
```

Note that the C language allows you to declare an indefinite number of array elements of a set length, but not a set number of array elements of an indefinite length.

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 — C keyword

Data type

Floating point numbers are a subset of the real numbers. Each has a built-in radix point (or “decimal point”) that shifts, or “floats”, as the value of the number changes. It consists of the following: 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. 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 throughout.

The following describes DECVAX, IEEE, and BCD formats, for your information.

DECVAX Format

The 32 bits in a **float** consist of one sign bit, an eight-bit exponent, and a 24-bit fraction, as follows:

Sign	Exponent	Fraction			
s	eeeeeee e	fffffff	fffffff	fffffff	fffffff
	Byte 4	Byte 3	Byte 2	Byte 1	

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 most significant bit in the fraction is always set to one and is not stored. It is usually called the "hidden bit".

The format for **doubles** simply adds another 32 fraction bits to the end of the **float** representation, as follows:

Sign	Exponent	Fraction							
s	eeeeeee e	fffffff	fffffff	fffffff	fffffff	fffffff	fffffff	fffffff	fffffff
	Byte 8	Byte 7	Byte 6	Byte 5					
		fffffff	fffffff	fffffff	fffffff	fffffff	fffffff	fffffff	fffffff
		Byte 4	Byte 3	Byte 2	Byte 1				

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 several 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.

An IEEE double, unlike DECVAX format, increases the number of exponent bits. It consists of a sign bit, an 11-bit exponent, and a 53-bit fraction, as follows:

Sign	Exponent	Fraction			
s	eeeeeee eeee	ffff	fffffff	fffffff	fffffff
	Byte 8	Byte 7	Byte 6	Byte 5	
		fffffff	fffffff	fffffff	fffffff
		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, an exponent, and a chain of four-bit numbers, each of which is defined to hold the digits zero through nine.

A BCD float has a sign bit, seven bits of exponent, and six four-bit digits, as follows:

Sign	Exponent	Fraction			
s	eeeeeee	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 digits, as follows:

Sign	Exponent	Fraction			
s	eeeeeee eeee	dddd	dddd	dddd	dddd
	Byte 8	Byte 7	Byte 6	Byte 5	
		dddd	dddd	dddd	dddd
		Byte 4	Byte 3	Byte 2	Byte 1

Passing the hexadecimal numbers A through F in a digit yields unpredictable results.

The following rules apply when 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

C keywords, C language, data formats, declarations, double

The Art of Computer Programming, vol. 2, page 180ff

The C Programming Language, page 34

floor — Mathematics function (libm)

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, intbuf, device, sectors, track, side,
interleave, magic, new)**

char *buffer;

int *intbuf, device, sectors, track, side, interleave, new;

long magic;

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 512-byte sectors per track. The SF314 supports single- and double-sided media, whereas the SF354 supports only single-sided media.

buffer points to a word-aligned area of memory that is large enough to hold the image of an entire track. This is the number of bytes per sector (512), times the number of sectors (ten), plus the overhead information. This comes to approximately eight kilobytes for a nine-sector track, or nine kilobytes for a ten-sector track. All data in this area are overwritten during the format and verify operation. In the case of a format failure, a zero-terminated list of bad sectors is returned to this array as an array of ints.

device is the number of the floppy disk drive. Zero indicates drive A and one indicates drive B. Any other value yields unpredictable results.

sectors is the number of sectors to be formatted onto each track, one through ten. The standard format uses nine tracks.

track is the number of the track that you wish to format, from zero through 79. Any attempt to format beyond track 79 may damage the drive.

side is the side of the floppy disk on which you wish to write, i.e., zero or one. Any attempt to format side 1 with an SF354 drive will fail and may hang the system.

interleave gives the sector interleave factor. With Mega STs, this can be -1, which specifies that the pointer *intbuf* contains the sector numbers in the order in which they appear within the track. This will not work on earlier versions of TOS, and will yield unpredictable results if used.

magic is a magic number that TOS uses to verify the operation. It must be set to 0x87654321L.

new is the value with which to fill the newly formatted sectors. It must not be zero, and the high nybble of neither byte can be 0xF. This is a word value, and the recommended value is 0xE5E5, which sets up a good test pattern in the format.

Flopfmt returns zero if the track was formatted correctly, and nonzero if an error occurred. If bad sectors are discovered, their numbers are written into the area pointed to by *buffer*, in consecutive words terminated by zero. Bad sectors do not cause **Flopfmt** to return an error, so the bad sector list should always be checked. If bad sectors are found, you can either reformat the track, use the bad-sector information to mark the bad sectors in the FAT so that GEMDOS will not use them, or reject the floppy disk altogether.

Flopfmt forces the "changed" status (used by the functions **Mediach** and **Rwabs**) to "definitely changed."

Example

This example formats a single-sided floppy disk and initializes the first two tracks. It demonstrates **Flopfmt**, **Flopwr**, and **Protobt**.

```
#include <stdio.h>
#include <osbind.h>

#define BLANK (0xE5E5)          /* Standard sector format value */
#define MAGIC (0x87654321L)    /* Mandatory 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 = Crawlcn()) != 'y' && reply != 'Y') {
    printf("No. Floppy in drive B not formatted.\n");
    Pterm0();
}

printf("Yes\n");
printf("Press any key when ready...");
fflush(stdout);
Crawlcn();
printf("\n");
bf = (short *) malloc(BUFSIZE);

for (track=0; track<80; track++) {
    printf("now formatting track %d:", track);
    fflush(stdout);
    status = Flopfmt(bf, NULL, 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);
Crawlcn();
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 bpb 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_nsid[0],bb.bp_nsid[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.

Flopwr 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 be 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)

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 "rwb", 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
r+ read and write ASCII; no truncation, error if not found
r+b read and write binary data

w+ write and read ASCII; truncate if found, create if not found
w+b write and read binary data
a+ append and read ASCII; no truncation, create if not found
a+b append and read binary data

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.

Note that files opened in ASCII mode, which is the default, will return only printable characters and the newline character '\n'. Text files that use the return character '\r' must be opened in binary mode.

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, fdopen, freopen, STDIO

The C Programming Language, page 151, 167

Diagnostics

fopen returns **NULL** 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 that encodes the mode in which the file is opened, as follows: zero, read only; one, write only; and two, read or write. If the file can be opened, **Fopen** returns a handle by which the file can be accessed through **TOS**.

If the file cannot be opened, then **Fopen** returns a negative number. Note that this is a **long** negative; some devices, such as **con:** or **prn:**, may return a handle that is negative when examined as a word but positive when examined as a **long**.

Example

For examples of how to use this macro, see the entries for **Fseek** and **Fcreate**.

See Also

gemdos, TOS

for — C keyword

Control a loop

for(initialization; endcondition; modification)

for is a C keyword that introduces a loop. It takes three arguments, which are separated by semicolons ';'. *initialization* is executed before the loop begins. *endcondition* describes the condition which will end the loop. *modification* is the statement that modifies *variable* to control the number of iterations of the loop. For example,

```
for (i=0; i<10; i++)
```

first sets the variable *i* to zero; then it declares that the loop will continue as long as *i* remains less than ten; and finally, increments *i* by one after every iteration of

the loop. This ensures that the loop will iterate exactly ten times (from `i = 0` through `i = 9`). The statement

```
for(;;)
```

will loop until its execution is interrupted by a **break**, **goto**, or **return** statement.

See Also

break, C keywords, C language, **continue**, **while**

The C Programming Language, page 56

form_alert — AES function (libaes)

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. 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. An alert box can hold no more than five lines of text, each no longer than 31 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|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/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

The following example demonstrates **form_alert**.

```
#include <aesbind.h>

main()
{
    alertf(2, "[1][Alert Box][I dig it]");
}

alertf(n, p) int n; char *p;
{
    static char buffer[512];
    sprintf(buffer, "%r", &p);
    return form_alert(n, buffer);
}
```

See Also

AES, cc, gem, TOS

form_center — AES function (libaes)

Center an object on the screen

```
#include <aesbind.h>
#include <obdefs.h>
int form_center(picture, xptr, yptr, wptr, hptr)
OBJECT *picture; int *xptr, *yptr, *wptr, *hptr;
```

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**. See the Lexicon entry on **object** for more information.

The arguments *xptr*, *yptr*, *wptr*, and *hptr* point, respectively, to ints that hold the object's X coordinate, its Y coordinate, its width, and its height.

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)

Reserve/free screen space for dialogue

```
#include <aesbind.h>
int form_dial(flag, openx, openy, openw, openh, endx, endy, endw, endh)
int flag, openx, openy, openw, openh, endx, endy, endw, endh;
```

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, as follows:

0	FMD_START	reserve screen space
1	FMD_GROW	draw "growing box"
2	FMD_SHRINK	draw "shrinking box"
3	FMD_FINISH	free memory, send redraw message

The variables *openx*, *openy*, *openw*, and *openh* give, respectively, the X position, the Y position, the width, and the height of the rectangle from which the dialogue box grows. These values are used only with flags **FM_GROW** or **FM_SHRINK**. *endx*, *endy*, *endh*, and *endw* give, respectively, the X position, the Y position, the width, and the height of the dialogue box itself. All values should be given in rasters.

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

Note that you are responsible for saving and restoring the portion of the screen that is obscured by the **form_dial** dialogue box. **form_dial** will generate a redraw message when invoked with flag **FMD_FINISH**; your program must capture this message and process it properly.

Example

For an example of this routine, see the entry for **object**.

See Also

AES, **form_do**, **object**, **TOS**

Notes

The call **form_dial(FMD_FINISH ...)**; can be used to force the screen manager to redraw any portion of the screen.

form_do — AES function (libaes)

Handle user input in form dialogue

#include <aesbind.h>

int form_do(tree, object) OBJECT *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 closed the dialogue.

Example

For an example of this routine, see the entry for **object**.

aform_do -

WORKS THE SAME AS form_do BUT WILL SENSE ALT KEY COMBINATIONS AND SEARCH FOR THE EXIT BUTTON WHEN THAT IS THE CASE

See Also

AES, **form_dial**, **object**, TOS

form_error — AES function (libaes)

Display a TOS error

```
#include <aesbind.h>
```

```
int form_error(error) int error;
```

form_error is an AES routine that displays a preset 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 |

The above numbers correspond to error codes under MS-DOS; note, however, that they are *not* the same as GEMDOS or TOS errors. 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)

Print formatted output onto file stream

int fprintf(*fp*, *format*, [*arg1*, ..., *argN*])

FILE *fp; **char *format**;

[**data type**] *arg1*, ..., *argN*;

fprintf prints formatted strings onto the file stream *fp*. It uses the *format* string to specify an output format for *arg1* through *argN*.

See **printf** for a description of **fprintf**'s formatting codes.

Example

For an example of this routine, see the entry for **fscanf**.

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**.

At present, **fprintf** does not return a meaningful value.

fputc — **STDIO** function (libc)

Write character onto file stream

#include <stdio.h>

int fputc(*c*, *fp*) **char c**; **FILE *fp**;

fputc writes the character *c* onto the file stream *fp*. It returns *c* if *c* was written successfully.

Example

The following example uses **fputc** to write the contents of one file into another.

```
#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

fputc returns EOF when a write error occurs, e.g., when a disk runs out of space.

fputs — STDIO function (libc)

Write string to file stream

#include <stdio.h>

fputs(string, fp) char *string; FILE *fp;

fputs writes *string* onto the file stream *fp*. Unlike its cousin **puts**, it does not append a newline character to the end of *string*.

Example

For an example of this function, see the entry for **freopen**.

See Also

STDIO

The C Programming Language, page 155

fputw — STDIO function (libc)

Write an integer to a stream

#include <stdio.h>

```
int fputw(word, fp) int word; FILE *fp;
```

fputw writes *word* onto the file stream *fp*, and returns the value written.

Example

For an example of this function, see the entry for **fgetw**.

See Also

fgetw, **STDIO**

Diagnostics

fputw returns EOF when an error occurs. A call to **ferror** or **feof** may be needed to distinguish this value from a valid end-of-file signal.

fraction — Definition

A **fraction**, in the context of C programming, is the fractional portion of a floating point number. The term “mantissa” is often used as a synonym for it.

See Also

data formats, double, float, frexp

fread — STDIO function (libc)

Read data from file stream

```
#include <stdio.h>
```

```
int fread(buffer, size, n, fp)
```

```
char *buffer; unsigned size, n; FILE *fp;
```

fread reads *n* items, each being *size* bytes long, from file stream *fp* into *buffer*.

Example

For an example of how to use this function, see the entry for **fopen**.

See Also

fwrite, **STDIO**

Diagnostics

fread returns zero upon reading EOF or on error; otherwise, it returns the number of items read.

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 the number of bytes read if the data were read successfully. If the value returned does not equal *n*, you have reached the end of the file or an error has occurred — usually the former.

Example

For examples of how to use this function, see the entries for **Fseek** and **Fcreate**.

See Also

gemdos, **TOS**

free — General function (libc)

Return dynamic memory to free memory pool

void free(ptr) char *ptr;

free helps you manage the arena. It returns to the free memory pool memory that had previously been allocated by **malloc**, **calloc**, or **realloc**. **free** marks the block indicated by *ptr* as unused, so the **malloc** search can coalesce 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. This most often occurs by storing data beyond 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]);
    Pterm(0);
}

```

See Also

gemdos, TOS

freopen — STDIO function (libc)

Open file stream for standard I/O

```
#include <stdio.h>
```

```
FILE *freopen (name, type, fp)
```

```
char *name, *type; FILE *fp;
```

freopen reinitializes the file stream *fp*. It closes the file currently associated with it, 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**” (for, respectively, read, write, append, and binary) to indicate the mode of the stream. For further discussion of the *type* variable, see the entry for **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

This example, called **match.c**, looks in **argv[2]** for the pattern given by **argv[1]**. If the pattern is found, the line that contains the pattern is written into the file **argv[3]** or to **stdout**.

```

#include <stdio.h>
#define MAXLINE 128
char buffer[MAXLINE];

```

```
main(argc,argv)
int argc; char *argv[];
{
    FILE *fpin, *fpout;

    if (argc != 3 && argc != 4)
        fatal("Usage: match pattern infile [outfile]");
    if (argc >= 3 && (fpin = fopen(argv[2], "r")==NULL))
        fatal("Cannot open input file");
    if ((fpout = freopen(argv[3], "w", stdout))==NULL)
        fatal("Cannot open output file");

    while (fgets(buffer, MAXLINE, fpin) != NULL)
    {
        if (prmatch(buffer, argv[1], 1))
            fputs(buffer, stdout);
    }

    if (!feof(fpin))
        fatal("read error");
    exit(0);
}

fatal(s)
char *s;
{
    fprintf(stderr, "match: %s\n", s);
    exit(1);
}
```

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)

Separate fraction and exponent

double frexp(*real*, *ep*) **double** *real*; **int** **ep*;

frexp breaks double-precision floating point numbers into fraction and exponent. It returns the fraction *m* of its *real* argument, such that $0.5 \leq m < 1$ or $m=0$, and stores the binary exponent *e* in location *ep*. These numbers satisfy the equation $real = m * 2^e$.

Example

This example prompts for a number, then uses **frexp** to break it into its fraction and exponent.

```

#include <stdio.h>

main()
{
    extern char *gets();
    extern double frexp(), atof();
    double real, fraction;
    int *ep;

    char string[64];
    for (;;)
    {
        printf("Enter number: ");
        if (gets(string) == 0)
            break;

        real = atof(string);
        fraction = frexp(real, ep);
        printf("%lf is the fraction of %lf\n",
               fraction, real);
        printf("%d is the binary exponent of %lf\n",
               *ep, real);
    }
}

```

See Also

atof, ceil, fabs, floor, ldexp, modf

fscanf — STDIO function (libc)

Format input from a file stream

```
#include <stdio.h>
```

```
int fscanf(fp, format, arg1, ... argN)
```

```
FILE *fp; char *format;
```

```
[data type] *arg1, ... *argN;
```

fscanf reads the file stream *fp*, and uses the string *format* to format the arguments *arg1* through *argN*, each of which must point to a variable of the appropriate data type.

fscanf returns either the number of arguments matched, or EOF if no arguments matched.

For more information on **fscanf**'s conversion codes, see **scanf**.

Example

The following example uses **fprintf** to write some data into a file, and then reads it back using **fscanf**.

```
#include <stdio.h>

main ()
{
    FILE *fp;
    char let[4];

    /* open file into write/read mode */
    if ((fp = fopen("tmpfile", "wr")) == NULL)
    {
        printf("Cannot open 'tmpfile'\n");
        exit(1);
    }

    /* write a string of chars into file */
    fprintf(fp, "1234");

    /* move file pointer back to beginning of file */
    rewind(fp);

    /* read and print data from file */
    fscanf(fp, "%c %c %c %c",
           &let[0], &let[1], &let[2], &let[3]);
    printf("%c %c %c %c\n",
           let[3], let[2], let[1], let[0]);
}
```

See Also

scanf, sscanf, 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)

Seek on file stream

#include <stdio.h>

int fseek(*fp*, *where*, *how*)

FILE **fp*; long *where*; int *how*;

fseek changes where the next read or write operation will occur within the file 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.

fseek differs from its cousin **lseek** in that **lseek** is a UNIX system call and takes a file number, whereas **fseek** is a STDIO function and takes a **FILE** pointer.

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, lseek, 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=Ffirst(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)

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 direc-

tory 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, `fselInput` alters the name in the buffer to which *directory* points in order 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.

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

Example

The following example demonstrates `fselInput`. It checks to see if the file you select is present or not.

```
#include <aesbind.h>
#include <gemdefs.h>
#include <osbind.h>

#define assert(x) if (!x) alertf(1, "[3][assert| %s |failed]", #x);
alertf(n, p) int n; char *p;
{
    static char buffer[512];
    sprintf(buffer, "%r", &p);
    return form_alert(n, buffer);
}

main()
{
    register char *cp;
    int result, button;

    static char prefix[128];
    static char separator[] = "\\.";
    static char suffix[16] = "*.*";
    static char filename[128];
    static char name[16];
    extern char *strchr();

    /* open application */
    appl_init();
```

```

/* build path for current directory */
cp = prefix;
*cp++ = Dgetdrv()+'A';
*cp++ = ':';
Dgetpath(cp, 0);

/* ensure mouse pointer is an arrow */
graf_mouse(ARROW, (int *)0);

for (;;) {
    /* build string of form "A:foo\\*.*" */
    strcpy(filename, prefix);
    strcat(filename, separator);
    strcat(filename, suffix);

    /* call fsel_input to select file */
    result = fsel_input(filename, name, &button);
    assert(filename[0] != 0);

    if (result == 0 || name[0] == 0 || filename[0] == 0 ||
        button == 0) {
        if (alertf(2, "[2] [Cancel|file|selection ] [Yes|No]")
            == 1)
            break;
        continue;
    }

    cp = strchr(filename, '\\');
    assert(cp != 0);
    *cp = 0;

    strcpy(prefix, filename);
    *cp++ = '\\';
    strcpy(suffix, cp);

    /* build query string */
    if (strchr(name, '*') || strchr(name, '?')) {
        strcpy(suffix, name);
        name[0] = 0;
        continue;
    }

    strcpy(cp, name);
    name[0] = 0;

    /* check if file is present */
    if (Fsfirst(filename, 0xFF) >= 0) {
        alertf(1, "[0] [File] %s |found | [Ok]",
            filename);
    } else {
        alertf(1, "[1] [File] %s |not found | [Ok]",
            filename);
    }
}

```

```
        /* see if user wishes to continue */
        if (alertf(1, "[2] [Try|another |file] [Yes|No]") == 2)
            break;
    }

    /* tidy up, exit */
    appl_exit();
    return 0;
}
```

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

If you specify volume label, no other type of file can be sought. 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 **AE_OK** (success) if the search succeeded, and **AEFILNF** (file not found) if it did not.

Example

For an example of this function, see the entry for **Fgetdta**.

See Also

Fsetdta, **Fsnnext**, **gemdos**, **stat.h**, TOS

Fsnnext — gemdos function 79 (**osbind.h**)

Search for next occurrence of file name

```
#include <osbind.h>
```

```
#include <stat.h>
```

```
int Fsnnext()
```

Fsnnext 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 **Fsnnext**. If **Fsnnext** 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**.

Fsnnext 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**)

Find file attributes

```
#include <stat.h>
```

```
fstat(descriptor, statptr) int 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)

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.

See Also

data types, executable file, library, portability

fwrite — **STDIO** function (libc)

Write onto file stream

#include <stdio.h>

int fwrite(*buffer*, *size*, *n*, *fp*)

char **buffer*; **unsigned** *size*, *n*; **FILE** **fp*;

fwrite writes *n* items, each of *size* bytes, from *buffer* onto the file stream *fp*.

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**

G

galaxy.a — Archive

galaxy.a is an archive that holds the source files for **galaxy**, a program that allows you to simulate on your Atari ST the birth and evolution of spiral galaxies. The source code is self-documenting, and is offered as an extended example of how to manipulate the Atari ST's graphics.

If you wish to compile **galaxy**, 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 galaxy.a
```

See Also

ar

gcvt — General function (libc)

Convert floating point number to ASCII string

```
char *gcvt(d, prec, buffer)
double d; int prec; char *buffer;
```

gcvt converts a floating point number into an ASCII string. Its operation resembles that of the **%g** operator to **printf**. **gcvt** converts its argument *d* into a NUL-terminated string of decimal numerals with a precision (i.e., the number of numerals to the right of the decimal point) of *prec*. Unlike its cousins **ecvt** and **fcvt**, **gcvt** uses a buffer that is defined by the caller. *buffer* must point to a buffer large enough to hold the result; 64 characters will always be sufficient.

When generating its output, **gcvt** will mimic **fcvt** if possible; otherwise, it mimics **ecvt**. **gcvt** returns *buffer*.

Example

For an example of this function, see the entry for **ecvt**.

See Also

ecvt, **fcvt**, **frexp**, **ldexp**, **modf**, **printf**

gem — Command

Run a GEM program

```
gem command args
```

gem allows you to run a GEM *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.

gem reads the environment, and will properly use the environmental variables **PATH** and **SUFF**. A GEM program will execute correctly if both the executable

and its associated resource file are located in a directory named in **PATH**.

Another way to use **gem** is 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 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**:

0x03	Cauxin	Read character from serial port
0x12	Cauxis	Return serial port input status
0x13	Cauxos	Return serial port output status
0x04	Cauxout	Write character to serial port
0x01	Cconin	Read character from console

0x0B	Cconis	Return console input status
0x02	Cconout	Write character to console
0x10	Cconos	Return console output status
0x0A	Cconrs	Read and edit string from console
0x09	Cconws	Write a string to the console
0x08	Cnecin	Read character from console, no echo
0x11	Cprnos	Check parallel port output status
0x05	Cprnout	Write character to parallel port
0x07	Crawcin	Read raw character from console
0x06	Crawio	Perform raw I/O with console
0x39	Dcreate	Create a subdirectory
0x3A	Ddelete	Remove a subdirectory
0x36	Dfree	Find free space on disk
0x19	Dgetdrv	Return current disk drive
0x47	Dgetpath	Return current directory
0x0E	Dsetdrv	Set the default drive
0x3B	Dsetpath	Set the current directory
0x43	Fattrib	Get/set file attributes
0x3E	Fclose	Close a file
0x3C	Fcreate	Create a file
0x57	Fdatetime	Get/set file's date stamp
0x41	Fdelete	Delete a file
0x45	Fdup	Duplicate a file's handle
0x46	Fforce	Force a file handle
0x2F	Fgetdta	Get a disk transfer address
0x3D	Fopen	Open a file
0x3F	Fread	Read a file
0x56	Frename	Rename a file
0x42	Fseek	Move a file pointer
0x1A	Fsetdta	Set disk transfer address
0x4E	Fsfirst	Search for first occurrence of file
0x4F	Fsnext	Search for next occurrence of file
0x40	Fwrite	Write into a file
0x48	Malloc	Allocate dynamic memory
0x49	Mfree	Free dynamic memory
0x4A	Mshrink	Shrink amount of allocated memory
0x4B	Pexec	Load or execute a process
0x4C	Pterm	Terminate a process
0x00	Pterm0	Terminate a TOS process
0x31	Ptermres	Terminate a process but keep in memory
0x20	Super	Enter supervisor mode
0x30	Sversion	Get current version of TOS
0x2A	Tgetdate	Get date
0x2C	Tgettime	Get time
0x2B	Tsetdate	Set date
0x2D	Tsettime	Set time

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>

(struct bpb *)Getbpb(device);

int device;

Getbpb returns a pointer to the BIOS parameter block for a given disk drive. This structure is described in the header file **bios.h**. *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 **NULL**.

Note that in the DRI bindings, **Getbpb** is declared as returning a **long**. The cast shown in the declaration is necessary to avoid an integer-pointer pun.

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_rdlenn);
    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 file stream

#include <stdio.h>

int **getc**(*fp*) **FILE** **fp*;

getc is a macro that reads a character from the file stream *fp*, and returns an **int**.

Example

The following example creates a simple copy utility. It opens the first file named on the command line and copies its contents into the second file named on the command line.

```
#include <stdio.h>

main(argc, argv)
int argc; char *argv[];
{
    int foo;
    FILE *source, *dest;

    if (--argc != 2)
        error("Usage: copy [source][destination]");

    if ((source = fopen(argv[1], "rb")) == NULL)
        error("Cannot open source file");
    if ((dest = fopen(argv[2], "wb")) == NULL)
        error("Cannot open destination file");

    while ((foo = getc(source)) != EOF)
        putc(foo, dest);
}
```

```
error(string)
char *string;
{
    printf("%s\n", string);
    exit (1);
}
```

See Also

fgetc, getchar, putc, 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, because **getc** is a complex macro, its use in expressions of too great a complexity may cause unforeseen difficulties. Use of the function **fgetc** may avoid this.

getchar — STDIO macro (stdio.h)

Read character from standard input

```
#include <stdio.h>
```

```
int getchar()
```

getchar is a macro that reads a character from the standard input. It is equivalent to **getc(stdin)**.

Example

The following example gets one or more characters from the keyboard, and echoes them on the screen.

```
#include <stdio.h>
main()
{
    int foo;
    while ((foo = getchar()) != EOF)
        putchar(foo);
}
```

See Also

getc, putchar, 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)

Read environmental variable

char *getenv(VARIABLE) char *VARIABLE;

A program may read variables from its *environment*. This allows the program to accept information that is specific to it. The environment consists of an array of strings, each having the form **VARIABLE=VALUE**. When called with the string **VARIABLE**, **getenv** reads the environment, and returns a pointer to the string **VALUE**.

Example

This example prints the environmental variable **PATH**.

```
#include <stdio.h>

main()
{
    char *env;
    extern char *getenv();

    if ((env = getenv("PATH")) == NULL)
    {
        printf("Sorry, cannot find PATH\n");
        exit(1);
    }
    printf("PATH = %s\n", env);
}
```

See Also

cc, **environment**, **envp**, **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); struct mpb *pointer;

Getmpb tells **TOS** to copy its memory parameter block into the **mpb** structure pointed to by *pointer*. This structure is described in the header file **bios.h**.

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 **Setcolor** to read and return the current settings of the color palette.

See Also

commands, Setcolor, setpal, 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 low resolution; one, medium resolution; and two, high 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 reztbl { int r_rez; char *r_name; } reztbl[] = {  
    GR_LOW, "low",  
    GR_MED, "medium",  
    GR_HIGH, "high",  
    -1, "unknown"
```

```
};
```

```

main() {
    register struct rextab *rp;
    register int rez;
    rez = Getrez();

    for (rp = rextab; 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)

Read string from standard input

#include <stdio.h>

char *gets(buffer) char *buffer;

gets reads characters from the standard input into a buffer pointed at by *buffer*. It stops reading as soon as it detects a newline character or EOF. **gets** discards the newline or EOF, appends a NUL character onto the string it has built, and returns another copy of *buffer*.

Example

The following example uses **gets** to get a string from the console; the string is echoed twice to demonstrate what **gets** returns.

```

#include <stdio.h>

main()
{
    char buffer[80];
    printf("Type something: ");
    fflush(stdout);
    printf("%s\n%s\n", gets(buffer), buffer);
}

```

See Also

buffer, fgets,getc, STDIO

Diagnostics

gets returns NULL if an error occurs or if EOF is seen before any characters are read.

Notes

Note that **gets** stops reading the input string as soon as it detects a newline character. If a previous input routine left a newline character in the standard input buffer, **gets** will read it and immediately stop accepting characters; to the user, it will appear as if **gets** is not working at all.

For example, if **getchar** is followed by **gets**, the first character **gets** will receive is

the newline character left behind by **getchar**. A simple statement will remedy this:

```
while (getchar() != '\n')
    ;
```

This throws away the newline character left behind by **getchar**; **gets** will now work correctly.

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 value whose bits indicate the following:

0-4	seconds, in two-second increments (0-29)
5-10	minutes (0-59)
11-15	hours (0-23)
16-20	day of the month (1-31)
21-24	month (1-12)
25-31	year (0-119, 0 indicates 1980)

Example

This example gets the keyboard time. 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();
```

```
    seconds = (time & 0x001F) << 1;
```

```
    minutes = (time >> 5) & 0x3F;
```

```
    hours = (time >> 11) & 0x1F;
```

```
    /* Get system time */
```

```
    /* Bits 0:4 */
```

```
    /* Bits 5:10 */
```

```
    /* 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

Kgettext, **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**)

Read word from file stream

#include <stdio.h>

int **getw**(*fp*) **FILE** **fp*;

getw reads a word (an **int**) from the file stream *fp*.

getw differs from **getc** in that **getw** gets and returns an **int**, whereas **getc** returns either a **char** promoted to an **int**, or EOF. To detect EOF while using **getw**, you must use **feof**.

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 end-of-file signal.

fgetw assumes that the bytes of the word it receives are in the natural byte ordering of the machine; see the entry on **byte ordering** for more information. This means that such files might not be portable between machines.

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 its 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 sus-

tained 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 */
    Cwcin();           /* 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 zero meridian is fixed.

See Also

gmtime, localtime, time, time.h, TIMEZONE

gmtime — Time function (libc)

Convert system time to calendar structure

```
#include <time.h>
```

```
tm *gmtime(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**, 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

GMT, localtime, time (overview), TIMEZONE

Notes

gmtime is useful only on a system whose time is set to GMT rather than to local time. The Mark Williams C time routines read the environmental variable **TIMEZONE** to translate GMT automatically into your local time, should you wish. See the entry on **TIMEZONE** for more information on how this works.

gmtime returns a pointer to a statically allocated data area that is overwritten by successive calls.

goto — C keyword

Unconditionally jump within a function

A **goto** command jumps to the area of the program introduced by a label. Note that a **goto** cannot cross a function boundary.

In the context of C programming, the most common use for **goto** is to exit from a control block or go to the top of a control block. It is used most often to write “rip-cord” routines, i.e., routines that are executed when an major error occurs too deeply within a program for the program to disentangle itself correctly.

Example

The following example demonstrates how to use **goto**.

```
#include <stdio.h>
```

```
main() {
    char line[80];
```

```
getline:
    printf("Enter line: ");
    fflush(stdout);
    gets(line);

/* a series of tests often is best done with goto's */
    if (*line == 'x')
    {
        printf("Bad line\n");
        goto getline;
    }

    else if (*line == 'y')
    {
        printf("Try again\n");
        goto getline;
    }

    else if (*line == 'q')
        goto goodbye;

    else
        goto getline;

goodbye:
    printf("Goodbye.\n");
    exit(0);
}
```

See Also

C keywords, C language

The C Programming Language, page 62

Notes

The C Programming Language describes **goto** as “infinitely-abusable”: *caveat utilitor*.

graf_dragbox — AES function (libaes)

Draw a draggable box

#include <aesbind.h>

int graf_dragbox(width, height, stx, sty, bx, by, bw, bh, finx, finy)

int width, height, stx, sty, bx, by, bw, bh, *finx, *finy;

graf_dragbox is an AES routine that allows the user to drag a box around the screen. It also sets a boundary rectangle that limits how far the box can be dragged. The boundary can be set to the entire screen, to a window, or to some other delimiter.

width and *height* give, respectively the width and height of the box being dragged, 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:

<i>Resolution</i>	<i>Width</i>	<i>Height</i>
High	640	400
Medium	640	200
Low	320	200

stx and *sty* give, respectively, the starting X and Y coordinates for the box. *finx* and *finy* point to the coordinates to which the box has been dragged; these values are set by the function.

bx, *by*, *bw*, and *bh* set, respectively, the X coordinate of the boundary rectangle, its Y coordinate, its width, and its height.

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)

Draw a growing box

```
#include <aesbind.h>
```

```
int graf_growbox(stx, sty, stw, sth, finx, finy, finw, finh)
```

```
int stx, sty, stw, sth, finx, finy, finw, finh;
```

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.

stx, *sty*, *stw*, and *sth* set, respectively, the X coordinate of the origin box (the box from which the growing box starts to grow), its Y coordinate, its width, and its height. *finx*, *finy*, *finw*, and *finh* set in the same way the dimensions of the finish box (the box toward which the growing box grows). The unit of measure for all eight arguments is the number of rasters for the screen. The number rasters on the screen varies with the degree of resolution, as follows:

<i>Resolution</i>	<i>Width</i>	<i>Height</i>
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, **graf_shrinkbox**, **window AES**, **gem**, **graf_shrinkbox**, **TOS**, **window**

graf_handle — AES function (libaes)

Get a VDI handle

```
#include <aesbind.h>
```

```
int graf_handle(chwidth, chheight, bwidth, bheight)
```

```
int *chwidth, *chheight, *bwidth, *bheight;
```

The AES routine **graf_handle** returns the handle for the physical workstation open for the desktop. It also returns the size of the default system font.

chwidth and *chheight* point, respectively, to the width and height of the default character cell. *bwidth* and *bheight* point, respectively, to the width and height of a square box (corrected for aspect ratio) that contains a character. This is the size of the window boxes. These values are set by GEM.

See Also

AES, **TOS**

Notes

A desk accessory that does not call **graf_handle** will not have its desk menu item displayed, and the desktop's desk menu item will fail to function, even though it is displayed.

The VDI handle that **graf_handle** returns is the handle that AES uses to implement all of its graphics library routines. You should not use this handle unless you wish to alter the AES graphics. For example, if you reset the fill pattern using the handle returned by **graf_handle**, you may or may not find the window manager using your fill pattern to fill the title bars of windows as it redraws them.

graf_mbox — AES function (libaes)

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)

Get the current mouse state

```
#include <aesbind.h>
```

```
int graf_mkstate(xptr, yptr, bptr, kptr) int *xptr, *yptr, *bptr, *kptr;
```

graf_mkstate is an AES routine that returns the current mouse state. *xptr* points to an integer that holds the X coordinate of the mouse pointer. *yptr* points to an integer that holds the Y coordinate of the mouse pointer. *bptr* points to an integer that indicates the button state when the event occurred: zero indicates up and one indicates down. Finally, *kptr* points to an integer that represents the states of the control, alt, and shift keys OR'd together, as follows:

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 AES, TOS

graf_mouse — AES function (libaes)

Change the shape of the mouse pointer

```
#include <aesbind.h>
```

```
int graf_mouse(form, shape) int form; int shape[37];
```

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	arrow (default)
1	TEXT_CRSR	vertical line (text cursor)
2	BUSY_BEE	bee
3	POINT_HAND	hand with pointing finger
4	FLAT_HAND	hand with extended fingers
5	THIN_CROSS	thin cross hairs
6	THICK_CROSS	thick cross hairs
7	OUTLN_CROSS	outlined cross hairs
255	USR_DEF	user-described shape
256	M_OFF	hide mouse pointer

257 M.ON

show mouse pointer

shape is a 37-word array that specifies a new shape for the pointer. This argument is ignored if *form* has 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>
#include <gemdefs.h>

/*
 * array used to build unique mouse-pointer shape.
 * ANSI C standard states that it's OK to end array
 * initialization with a ','.
 */
int mouse[37] = (
    7, 7, 1, 0, 1,
    0x7FFF, 0x7007, 0x780F, 0x5C1D, 0x4E39, 0x4771,
    0x4361, 0x4001, 0x4361, 0x4771, 0x4E39, 0x5C1D,
    0x780F, 0x7007, 0x7FFF, 0x0000, 0x7FFF, 0x7007,
    0x780F, 0x5C1D, 0x4E39, 0x4771, 0x4361, 0x4001,
    0x4361, 0x4771, 0x4E39, 0x5C1D, 0x780F, 0x7007,
    0x7FFF, 0x0000,
);

main()
(
    int counter;
    appl_init();

    /* draw "canned" mouse pointer shapes */
    for (counter = ARROW; counter <= OUTLN_CROSS; counter++) (
        graf_mouse(counter, (int *)0);
        evnt_keybd();
    )

    /* draw user-defined pointer shape */
    graf_mouse(USER_DEF, mouse);
    evnt_keybd();

    appl_exit();
    return(0);
)
```

For further examples, see the entries **evnt_multi**, **object**, **window**.

See Also

AES, **object**, **vsc_form**, **window AES**, **object**, **TOS**, **vsc_form**, **window**

Notes

Mixing AES mouse calls with VDI mouse calls can produce unpredictable results.

graf_mouse and **vsc_form** use the same 37-word mouse form descriptor. The call

```
graf_mouse(USER_DEF, form);
```

is exactly equivalent to:

```
int handle;
handle = graf_handle(&handle, &handle, &handle, &handle);
vsc_form(handle, form);
```

This is an instance of how the AES uses the VDI to implement the higher-level functions that it provides.

graf_rubbox — AES function (libaes)

Draw a rubber box

```
#include <aesbind.h>
```

```
int graf_rubbox(x, y, w, h, newwidth, newheight)
```

```
int x, y, w, h, *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. *x*, *y*, *w*, and *h* define the initial dimensions of the rubber box: respectively, they define its X coordinate, its Y coordinate, its width, and its height. All dimensions are in rasters.

newwidth and *newheight* point to the values for width and height to be set by the user’s manipulation of the box.

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 select a group of files at once uses **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)

Draw a shrinking box

```
#include <aesbind.h>
```

```
int graf_shrinkbox(beginx, beginy, beginw, beginh, endx, endy, endw, endh)
```

```
int beginx, beginy, beginw, beginh, endx, endy, endw, endh;
```

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 *beginx*, *beginy*, *beginw*, and *beginh* define the initial dimensions of the shrinking box: respectively, they set its X coordinate, Y coordinate, width, and height. *endx*, *endy*, *endw*, and *endh* set the same dimensions for the rectangle toward which the shrinking box shrinks. The unit of measure is the number of rasters for the screen, as follows:

<i>Resolution</i>	<i>Width</i>	<i>Height</i>
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, **graf_growbox** **AES**, **gem**, **graf_growbox**, **TOS**

graf_slidebox — AES function (libaes)

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". A slider is a box that the user can click to scroll through the contents of the file or directory being displayed.

This function is *not* usable for the window sliders, because the window manager is the only entity that knows where the object that defines those sliders is kept.

All that is needed to define a slider is a box with another box within it. A more complex slider can be made by making the primary boxes invisible and drawing icons within the slider and parent boxes.

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_slidebox 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.

Example

The following example draws a slider on the screen. By clicking it, you can move the slide bar back and forth; the program informs you of the new position of the slide bar.

```
#include <gemdefs.h>
#include <obdefs.h>

/* The slider is simply one box inside another */
#define PARENT 0
#define SLIDER 1
OBJECT object[] = {
    (-1, 1, 1, G_BOX, NONE, NORMAL,
     ((-1L<&0xFF)<<16)|(BLACK<<12)|(BLACK<<8)|(1<<4)|BLACK, 0, 0, 20, 1 ),
    ( 0, -1, -1, G_BOX, LASTOB, NORMAL,
     (1L<<16)|(BLACK<<12)|(BLACK<<8)|BLACK, 10, 0, 1, 1 ),
};

#define NOBJECT (sizeof object / sizeof *object)

typedef struct { int x, y, w, h; } Rectangle;
#define elements(r) r.x, r.y, r.w, r.h
#define pointers(r) &r.x, &r.y, &r.w, &r.h

typedef struct { int x, y, b, k; } Mouse_state;
#define melements(r) m.x, m.y, m.b, m.k
#define mpointers(r) &m.x, &m.y, &m.b, &m.k

alertf(n, p) int n; char *p;
{
    static char buffer[512];
    sprintf(buffer, "%r", &p);
    return form_alert(n, buffer);
}

/* Recompute slider position using graf_slidebox return */
slid_repos(op, np, ns, d, s)
register OBJECT *op;
int np, ns, d, s;
{
    if (d == 0)
        op[ns].ob_x = (((long)(op[np].ob_width -
        op[ns].ob_width)*s)/1000;

    else
        op[ns].ob_y = (((long)(op[np].ob_height -
        op[ns].ob_height)*s)/1000;
}
```

```
main()
{
    int s;          /* Slide position */
    Rectangle d;    /* Desktop rectangle */
    Rectangle r;
    Mouse_state m;

    appl_init();
    for (s = 0; s < NOBJECT; s += 1)
        rsrc_obfix(object, s);

    /* Get desktop rectangle and center slider */
    wind_get(0, WF_FULLXYWH, pointers(d));
    object[PARENT].ob_x = d.x + d.w / 2 - object[PARENT].ob_width / 2;
    object[PARENT].ob_y = d.y + d.h / 4;

    /* Loop until the alert'ed user quits */
    do {
        /* Redraw the slider */
        objc_draw(object, ROOT, 8, elements(d));

        /* Find the slider rectangle */
        objc_offset(object, SLIDER, &r.x, &r.y);
        r.w = object[SLIDER].ob_width;
        r.h = object[SLIDER].ob_height;

        /* Wait for the slider to be selected */
        do
            evnt_mouse(0, elements(r), mpointers(m));
        while ((m.b & 1) == 0);

        /* Let the AES track the slider */
        s = graf_slidebox(object, PARENT, SLIDER, 0);

        /* Compute the new slider position */
        slid_repos(object, PARENT, SLIDER, 0, s);
    } while (alertf(1, "[0]slider at %d ][Ok|Quit]", s) == 1);

    appl_exit();
    return 0;
}
```

See Also

AES, TOS

graf_watchbox — AES function (libaes)

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

H

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* 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**. See the entry for **FILE** for more information.

A *workstation handle* is used by the GEM VDI to identify a virtual device. It is returned by the routines **graf_handle**, **v_opnwk**, **v_opnvwk**. It 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.

A *window handle* identifies each handle as it is created, to distinguish it from all other created windows. It is returned by the routine **wind_create**.

See Also

AES, VDI, UNIX routines

header file — Overview

A *header file* is a file of C code 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 of 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 easier to maintain and to port to other environments.

See Also

#include, portability, stdio.h

help — Command

Print concise description of command

help command

help prints a concise description of the options available for each specified *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**. This file must be kept in a directory that is named in the environmental variable **LIBPATH**, or **help** will not be able to find it. 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**. Be sure to use the '#', as described above. Once you have edited **helpfile**, you must rebuild its index; otherwise, **help** will no longer work. To rebuild the **helpfile**, use the following command:

```
help -R foo
```

where **foo** is the name of any entry within **helpfile**.

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**, **mousehidden**, **showmouse**, **TOS**

HOME — Environmental variable

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 — Character constant

Mark Williams C recognizes the literal character '\t' as representing the ASCII horizontal tab character HT (octal 011). This character may be used as a character constant or in a string constant.

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)

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

I

if — Command

Execute a command conditionally

if *word1 word2 [word3]*

if is a command built into the microshell **msh**. It governs the conditional execution of commands: If **word1** executes successfully, then **word2** is executed; otherwise, the **word3**, if present, is executed. Each of the words may be a list of commands that is enclosed within parentheses.

Example

The command

```
if (cc -V foo.c >&bar) (cp foo.c b:\src) (me foo.c bar)
```

compiles the program **foo.c**. If the compilation proceeded correctly, then **foo.c** is copied into the directory **src**; however, if something went wrong, then the editor would be invoked to display both **foo.c** and the file into which all error messages had been redirected. This is useful if you keep your source files on a RAM disk.

See Also

commands, **equal**, **is_set**, **msh**, **not**, **while**

if — C keyword

Introduce a conditional statement

if is a C keyword that introduces a conditional statement. For example,

```
if (i==10)
    dosomething();
```

will **dosomething** only if **i** equals ten.

if statements can be used with the statements **else if** and **else** to create a chain of conditional statements. Such a chain can include any number of **else if** statements, but only one **else** statement.

See Also

C keywords, **C language**, **else**

The C Programming Language, page 51

#if — Preprocessor instruction

Include code conditionally

#if (*expression*)

#if is the initiator for a conditional statement that is processed by the C preprocessor **cpp**. This command tells **cpp** that if the following condition is met, then include the following lines of code in the program until it meets the next **#elif**, **#else**, or **#endif** statement.

An **#if** command can also be coupled with the **#else** and **#elif** commands to create several levels of conditions. For example:

```
#if (condition1)
    int foo = 1;
#elif (condition2)
    int foo = 2;
#else
    int foo = 3;
#endif
```

See Also

cc, cpp, #elif, #else, #endif

Notes

Note that all preprocessor commands must be left justified (set flush with the left margin), even within **#if** statements; otherwise, **cpp** will ignore them.

#ifdef — Preprocessor instruction

Include code conditionally

#ifdef *identifier*

An **#ifdef** instruction tells the C preprocessor to check if *identifier* has been defined. If the statement is true (that is, if *identifier* is defined), then **cpp** includes the following code up to the next **#elif**, **#else**, or **#endif** statement.

An **#ifdef** instruction can also be coupled with **#else** and **#elif** instructions to create several levels of conditions. For example:

```
#ifdef identifier
    int foo = 1;
#elif (condition1)
    int foo = 2;
#elif (condition2)
    int foo = 3;
#else
    int foo = 4;
#endif
```

Note that the **#ifdef** instruction can be used with the **-D** option to the **cc** command to generate different versions of a program at compile time. For example, with the following code:

```
#ifdef SMALL
    foo = 3;
#else
    foo = 4;
#endif
```

You can set **foo** to equal three with the **cc** command

```
cc -DSMALL example.c
```

Otherwise, `foo` will be set to four by default.

See Also

cc, cpp, #elif, #else, #endif, #if, #ifndef
The C Programming Language, page 208

Notes

Note that all preprocessor commands must be left justified (set flush with the left margin), even within `#ifdef` statements.

#ifndef — Preprocessor instruction

Include code conditionally

#ifndef identifier

An `#ifndef` instruction tells the C preprocessor to check if *identifier* has been defined. If the statement is true (that is, *identifier* is not defined), then `cpp` includes the following lines of code up to the next `#elif`, `#else`, or `#endif` statement.

An `#ifndef` instruction can also be coupled with `#else` and `#elif` instructions to create several levels of conditions. For example:

```
#ifndef SMALL
    int foo = 1;
#elif (condition1)
    int foo = 2;
#elif (condition2)
    int foo = 3;
#else
    int foo = 4;
#endif
```

You can set `foo` to one with the `cc` command

```
cc -USMALL example.c
```

See Also

cc, cpp, #elif, #else, #endif, #if, #ifdef
The C Programming Language, page 208

Notes

Note that all preprocessor commands must be left justified (set flush with the left margin), or they will not be recognized by `cpp`.

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 variable

INCDIR names the default directory within which the C preprocessor **cpp** seeks its header files. For example, the command

```
setenv INCDIR=a:\include
```

tells **cpp** to look for header files in directory **include** on drive A. This directory is searched, as is the directory that holds the C source files and the directories named with **-I** options to the **cc** command, if any.

It is recommended that you set **INCDIR** in your **profile** to ensure that it is always set correctly.

See Also

cc, **environment**, **environmental variable**

#include — Preprocessor instruction

Copy a header file into a program

```
#include <file.h>
```

```
#include "file.h"
```

#include is a statement processed by the C preprocessor **cpp**. Its operation is simple: **cpp** 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** options to the **cc** command line, and then in the directory named by the environmental variable **INCDIR**. Quotation marks tell **cpp** to look for *file.h* in the source file's directory, then in directories named with the **-I** options, and then in the directory named by the environmental variable **INCDIR**.

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

index — String function (libc)

Find a character in a string

```
char *index(string, c) char *string; char c;
```

index scans the given *string* for the first occurrence of the character *c*. If *c* is found, **index** returns a pointer to it. If it is not found, **index** returns NULL.

Note that having **index** search for NUL will always produce a pointer to the end of a string. For example,

```
char *string;
assert(index(string, 0)==string+strlen(string));
```

will never fail.

Example

For an example of this function, see the entry for **strncpy**.

See Also

memchr, **pnmatch**, **rindex**, **string**, **strchr**, **strpbrk**

The C Programming Language, page 67

Notes

This function is identical to the function **strchr**, which is described in the ANSI standard. Mark Williams C includes **strchr** in its libraries. It is recommended that it be used instead of **index** so that programs more closely approach strict conformity with the ANSI standard.

inherit — Command

Pass variable to child shell

inherit *variable* ...

The command **inherit** allows a sub-shell to inherit a *variable* set in its parent shell. *variable* must be a variable that had been set with the **set** command.

See Also

commands, **msh**, **set**, **setenv**

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 threshold-scale-delta
- 3 set parameter for Y axis threshold-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 — C keyword

Data type

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

C keywords, C language, data formats, 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**

is_set — Command

Check if an environmental variable is set

is_set [*in dir*] *name*

is_set is a test command that is built into the microshell, **msh**. It tests to see if the environmental variable *name* is set; **is_set** returns zero if *name* is set, and a value other than zero if it is not.

Example

The following command checks to see if the environmental variable **CMD** is set. If it is, the RAM-disk utility **rdy** is invoked in command-line mode; otherwise, **rdy** is invoked under the command **gem**, in graphics mode.

```
if (is_set CMD) rdy (gem rdy)
```

See Also

commands, **equal**, **if**, **msh**, **not**, **while**

isalnum — ctype macro (ctype.h)

Check if a character is a number or letter

```
#include <ctype.h>
```

```
int isalnum(c) int c;
```

isalnum tests whether the argument *c* is alphanumeric (0-9, A-Z, or a-z). It returns a number other than zero if *c* is of the desired type, and 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

ASCII, **ctype**

isalpha — ctype macro (ctype.h)

Check if a character is a letter

```
#include <ctype.h>
```

```
int isalpha(c) int c;
```

isalpha tests whether the argument *c* is a letter (A-Z or a-z). It returns a number other than zero if *c* is an alphabetic character, and 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

ASCII, **ctype**

isascii — ctype macro (ctype.h)

Check if a character is an ASCII character

```
#include <ctype.h>
```

```
int isascii(c) int c;
```

isascii tests whether the argument *c* is an ASCII character ($0 \leq c \leq 0177$). It returns a number other than zero if *c* is an ASCII character, and zero if it is not. Many other **ctype** macros will fail if passed a non-ASCII value 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**

isatty — General function (libc.a/isatty)

Check if a device is a terminal

```
isatty(fd); int fd;
```

isatty checks to see if a device is a terminal. Given the file descriptor *fd*, **isatty** returns non-zero if *fd* is attached to a terminal, and 0 if it is not.

See Also

FILE, **fileno**

isctrl — ctype macro (ctype.h)

Check if a character is a control character

```
#include <ctype.h>
```

```
int isctrl(c) int c;
```

isctrl tests whether the argument *c* is a control character (including a newline character) or a delete character. It returns a number other than zero if *c* is a control character, and zero if it is not. **isctrl** 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>
```

```
int isdigit(c) int c;
```

isdigit tests whether the argument *c* is a numeral (0-9). It returns a number other than zero if *c* is a numeral, and 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

ASCII, **ctype**

isleapyear — Time function (libc)

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>
```

```
int islower(c) int c;
```

islower tests whether the argument *c* is a lower-case letter (a-z). It returns a number other than zero if *c* is a lower-case letter, and 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

ASCII, **ctype**

isprint — **ctype** macro (**ctype.h**)

Check if a character is printable

#include <ctype.h>

int isprint(c) int c;

isprint is a macro that tests if *c* is printable, i.e, if it is neither a delete nor a control character. It returns a number other than zero if *c* is a printable character, and 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

ASCII, **ctype**

ispunct — **ctype** macro (**ctype.h**)

Check if a character is a punctuation mark

#include <ctype.h>

int 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 a number other than zero if the character tested is a punctuation mark, and 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

ASCII, **ctype**

isspace — **ctype** macro (**ctype.h**)

Check if a character prints white space

#include <ctype.h>

int isspace(c) int c;

isspace tests whether the argument *c* is a space, tab, newline, carriage return, or form-feed character. It returns a number other than zero if *c* is a white-space

character, and 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

ASCII, **ctype**

isupper — **ctype** macro (**ctype.h**)

Check if a character is an upper-case letter

```
#include <ctype.h>
```

```
int isupper(c) int c;
```

isupper tests whether the argument *c* is an upper-case letter (A-Z). It returns a number other than zero if *c* is an upper-case letter, and 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

ASCII, **ctype**

J

j0 — Mathematics function (libm)

Compute Bessel function

```
#include <math.h>
```

```
double j0(z) double z;
```

j0 computes the Bessel function of the first kind for order 0, for its argument *z*.

Example

This example, called **bessel.c**, demonstrates the Bessel functions **j0**, **j1**, and **jn**. 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)

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)

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 (overview), time.h, time_to_jday, tm_to_jday

Note

This function is of use mainly to astronomers, geographers, and historians.

jday_to_tm – Time function (libc)

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` (overview), `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 multi-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 enable. For a table of interrupts, see the entry for `Mfpint`.

See Also

`Jdisint`, `Mfpint`, `TOS`, `xbios`

jn — Mathematics function (`libm`)

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

K

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 */
};
```

kb_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_joyvec** 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_kbdsys** 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 the 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];
    long k_s;
} kst;
long ktm;

kbdvec(p) char *p;
{
    kst.k_c[0] = *p++;
    kst.k_c[1] = *p++;
    kst.k_c[2] = *p++;
    kst.k_c[3] = *p++;
    ktm = *((long *)0x4BA);

    /* translate four-character packet ... */
    /* ... into a long */
    /* one for joystick and mouse */
    /* packet time stamp */

    /* store four byte packet */
    /* NB: 'p' could be an odd address */

    /* system 200hz clock tick */
}

main()
{
    register struct kbdvbase *kbp;
    register void (*xx_joyvec)(), (*xx_mousevec)();
    register long ks, kt;

    kbp = Kbdvbase();
    xx_joyvec = kbp->kb_joyvec;
    kbp->kb_joyvec = kbdvec;
    xx_mousevec = kbp->kb_mousevec;
    kbp->kb_mousevec = kbdvec;
    ks = kst.k_s;

    /* keyboard vector table */
    /* save old joystick vector */
    /* install new joystick vector */
    /* ditto for mouse */

    /* initialize state record */

    while (Bconstat(BC_CON) == 0) {
        if (ks != kst.k_s) {
            ks = kst.k_s;
            kt = ktm;
            printf("%08lx %lu\n", ks, kt);
        }
    }
}
```

```

    Bconin(BC_CON);
    kbp->kb_joyvec = xx_joyvec;
    kbp->kb_mousevec = xx_mousevec;
    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 = CrawlIn()) != '\03') {
        CrawlOut(c);
    }
    Kbrate(old_delay,old_rate);
    printf( "\nRates restored.\n" );
}
```

See Also

TOS, xbios

keyboard — Technical information

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: Note that some keys produce different scan codes when used with the <shift> or <alt> key.

function keys

3B	3C	3D	3E	3F	40	41	42	43	44	
54	55	56	57	58	59	5A	5B	5C	5D	<shifted>

keyboard

01	78	79	7A	7B	7C	7D	7E	7F	80	81	82	83	<alt>	
02	03	04	05	06	07	08	09	0A	0B	0C	0D	29	OE	
0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	53
1D	1E	1F	20	21	22	23	24	25	26	27	28			2B
2A	60	2C	2D	2E	2F	30	31	32	33	34	35	36		
38	39										3A			

keypad

62	61	63	64	65	66
52	48	47	67	68	69
4B	50	4D	6A	6B	6C
			6D	6E	6F
			70	71	

The keyboard sold in North America 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, `evnt_keyboard`, `Keytbl`, TOS

Keytbl — xbios function 16 (`osbind.h`)

Set the keyboard's translation table

```
#include <osbind.h>
```

```
#include <xbios.h>
```

```
keytbl *Keytbl(unshifted, shifted, caplock) char *unshifted, *shifted, *caplock;
```

Keytbl sets the keyboard's translation tables.

On the Atari ST, each key generates a unique *scan code*. (See the entry for **keyboard** to find the code for each key.) The scan code is looked up in one of three translation tables. The table used depends upon the states of the shift keys: one table is used when no shift key is pressed, a second is used when the shift key is depressed, and a third is used when the caps-lock key is depressed. The scan code is then translated into the character found in the appropriate table.

The variables *shifted*, *unshifted*, and *caplock* each point to a translation table that you wish to load in place of a default table. Each table must be 128 bytes long. Setting one or more of these arguments to -1L tells **Keytbl** not to load a new key table.

Keytbl returns a pointer to the following structure:


```
struct keytbl {
    char *unshifted;
    char *shifted;
    char *capslock;
}
```

These point to the areas where **Keytbl** has written the current key tables.

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] = {X06lx\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("X3d,", *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**

Kgettext — Time function (libc)

Read time from intelligent keyboard's clock

```
#include <time.h>
tm *Kgettext();
```

Kgettext is a function that reads the time from the intelligent keyboard's clock. This clock is maintained apart from the other clocks on the Atari ST. **Kgettext** returns a pointer to the structure **tm**, which it initializes. **tm** is defined in the header file **time.h**. For more information about it, see the entry for **time**.

See Also

Ksettime, **Sgettext**, **time (overview)**, **time.h**

Notes

Unlike the function **Gettime**, which deals in two-second increments, **Kgettext** allows the programmer to work with clock ticks.

This function does not work properly on the Mega ST. To read the clock on the Mega ST, use the function **Sgettext**.

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)

Set time in intelligent keyboard's clock

#include <time.h>

int Ksettime(time) tm *time;

Ksettime is a function that sets the time on the intelligent keyboard's clock. This clock is maintained apart from the other clocks on the Atari ST. *time* points to a copy of the structure **tm**, 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

Kgettext, **time (overview)**, **time.h**

Notes

Unlike the function **Settime**, which deals with two-second increments, **Ksettime** works directly with seconds.

L

lc — Command

List directory's contents in columnar format
lc

The command **lc** prints the contents of the current directory. The contents are printed in multiple columns to make them easy to read. Names of directories are followed by a slash '/'.
See Also

commands, ls

lcalloc — General function (libc)

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.
- kfilename**
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 *libname.a*.
- o file**
Write output to *file* (default, **l.prg**.)
- R value**
Relocation base option. By default, **ld** links executable 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 executable 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.

Because version 3.0 changes the object format, the edition of `ld` shipped with version 3.0 does not work with objects compiled with Mark Williams C version 2.1.7 or earlier. To convert such objects to a format that `ld` recognizes, use the command `mwtomw`.

ldexp — General function (libc)

Combine fraction and exponent

`double ldexp(f, e) double f; int e;`

ldexp combines the fraction *f* with the binary exponent *e* to return a floating-point value *real* that satisfies the equation $real = m \cdot 2^e$.

See Also

`atof`, `ceil`, `fabs`, `floor`, `fraction`, `frexp`, `modf`

Lexicon — Introduction

The Mark Williams Lexicon is a new approach to documentation of computer software. The Lexicon is 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 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 in which it is kept.

The next lines briefly describe the item, then give the item's usage, where applicable. These are followed by 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* and the first edition of *The C Programming Language*. Diagnostics and notes, where applicable, conclude each entry.

Internally, the Lexicon has a tree structure. The "root" entry is the present entry, for **Lexicon**. Below this entry comes the set of *Overview* entries. Each Overview entry introduces a group of entries; for example, the Overview entry for **string** introduces all of the string functions and macros, lists them, and gives a lengthy example of how to use them.

Each entry cross-references other entries. These cross-references point up the documentation tree, toward an overview article and, ultimately, to the entry for

Lexicon itself. They also point down the tree to subordinate entries, and across to entries on related subjects. For example, the entry for **getchar** cross-references **STDIO**, which is its Overview article, plus **putchar** and **getc**, which are related entries of interest to the user. The Lexicon is designed so that you can trace from any one entry to any other, simply by following the chain of cross-references up and down the documentation tree.

Types of entries

There are several types of entries, as follows:

Command

These describe commands or utilities that run directly under TOS.

Definitions

These entries define technical terms and provide background information that is useful in C programming.

Library functions

These present functions or macros included with Mark Williams C. They include **ctype macros** (a macro that checks the type of data being handled); **debugging macros**; **general functions** (non-specialized C functions and macros); **mathematics library functions**; **STDIO functions**; **STDIO macros**; **string functions** (or routines used to manipulate character strings); and **time functions** (routines used to manipulate the time setting rendered by TOS).

Overview

Each of these entries gives an overview of a group of routines.

Symbols and constants

Data elements that are used while compiling or running programs; these include **environmental variables** 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).

Technical information

These give detailed information on technical issues. The articles describe **calling conventions**, **data formats**, and others.

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.

At the back of this manual is a list of all entries in the Lexicon, sorted by category. Check there for a complete list of the Overview entries, as well as for lists of all functions sorted by type.

Use the Lexicon

If, while reading an entry, you encounter a technical term that you do not understand, 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 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 — Library

GEM AES bindings

libaes is the library that holds the GEM AES binding routines. AES stands for *application environment services*. The routines contained in **libaes** 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** or print 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** and the runtime startup module **crtsg.o**. Second, it can be included by itself with the *library* option **-laes**. This option must come at the *end* of the **cc** command line, or the library will not be linked in.

See Also

AES, **aesbind.h**, **ar**, **crtsg.o**, **gemdefs.h**, **library**, **nm**, **TOS**

libc — Library

libc 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**. For a complete listing of the modules within **libc**, 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 — Library

libm is the archive file that holds the mathematics library.

See Also

ar, **library**, **mathematics library**, **math.h**, **nm**

LIBPATH — Environmental variable

Directories that hold libraries

LIBPATH names the directories that **cc** searches to find the compiler's executable programs and libraries. **make** also searches these directories for the files **mmacros** and **mactions**.

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

cc, **make**, **msh**, **setenv**

library — Overview

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 the standard C functions; and **libm.a**, which holds mathematical functions. You can use the archiver **ar** to create your own libraries of functions or edit existing libraries, or you can purchase such libraries from elsewhere. The sizes of the files in an existing library can be listed with the command **size**, and their symbol tables may be listed with the command **nm**.

See the entries for **mathematics library**, **string**, **STDIO**, and **UNIX routines** for information about many of the routines within these libraries.

See Also

ar, **function**, **libaes**, **libc**, **libm**, **libvdi**

libvdi — Library

GEM VDI bindings

libvdi is the library that holds the GEM VDI routines. VDI stands for *virtual device interface*. These routines perform low-level GEM graphics tasks. For a brief summary of these routines, see the entry for **VDI**.

libvdi's table of contents can be printed and its contents altered with the archiver **ar**.

This library can be called on the **cc** command line in either of two ways. First, the **-VGEM** will automatically link it in, plus the library **libaes** and the runtime startup module **crtsg.o**. Second, it can be included by itself with the library option **-lvdi**. 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 — Preprocessor instruction

Reset line numbering

#line number

#line number filename

#line manifest constants

#line is a C preprocessor instruction that resets the line numbering within a file. It takes three forms. The first, **#line number**, resets the current line number to *number*. The second, **#line number filename**, resets the line number to *number* and resets the name of the file that the compiler thinks is the source file to *filename*. Finally, the form **#line manifest constants** contains manifest constants that have been set by earlier preprocessor instructions, such as **#define**; when the constants are interpreted by the **cpp**, the **#line** instruction will then resemble one of the first two forms. Most often, it is used to ensure that error messages point to the correct line in the program's source code.

Note that this instruction normally is not used by a programmer. It is used by a program generator to associate errors in generated C code with the original sources; for example, the program generator **yacc** will use **#line** instructions to link the C code it generates with the **yacc** code written by the programmer.

See Also

cpp

The C Programming Language, page 208

Notes

The **#** of this instruction must appear in the *first*, or leftmost, column on a line, or it will be ignored by **cpp**.

Line A — Technical information

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 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 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 foreground 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;
    SKEWMASK = 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 documentation is not readily available. 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

Declare Atari line A routines

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

line feed — Character constant

Mark Williams C recognizes the literal character '\n' for the ASCII line feed character LF (octal 012). This character may be used as a character constant or in a string constant.

See Also

ASCII, character constant

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**.

Note that to read a file that includes line-feed characters, it must be opened in binary mode. See the entry for **fopen** for more information.

lmalloc — General function (libc)

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.

localtime — Time function (libc)

Convert system time to calendar structure

#include <time.h>

tm *localtime(timep) time_t *timep;

localtime converts the TOS internal time into the form described in the structure **tm**.

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 to be the number of seconds since January 1, 1970 0h00m00s GMT.

localtime returns a pointer to the structure, **tm**, which is also defined in **time.h**, as follows:

```

struct tm (
    int    tm_sec;      /* current time, second */
    int    tm_min;      /* current time, minute */
    int    tm_hour;     /* current time, hour */
    int    tm_mday;     /* day of the month */
    int    tm_mon;      /* month (0-11) */
    int    tm_year;     /* year */
    int    tm_wday;     /* day of the week */
    int    tm_yday;     /* day of the year */
    int    tm_isdst;    /* daylight savings flag */
);

```

The function **asctime** turns **tm** into an ASCII string.

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. See the Lexicon entry for **TIMEZONE** for more information on how Mark Williams C handles time zone settings.

Example

The following example recreates the function **asctime**. It builds a string somewhat different from that returned by **asctime** to demonstrate how to manipulate the **tm** structure.

```

#include <time.h>

char *month[] = {
    "January", "February", "March", "April",
    "May", "June", "July", "August", "September",
    "October", "November", "December"
};

char *weekday[] = {
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"
};

main()
{
    char buf[20];
    time_t tnum;
    tm *ts;
    int hour = 0;

    time(&tnum); /* get time from system */

    /* convert time to tm struct */
    ts=localtime(&tnum);

```



```
if(ts->tm_hour==0)
    sprintf(buf,"12:%02d:%02d A.M.",
        ts->tm_min, ts->tm_sec);

else
    if(ts->tm_hour>=12) {
        hour=ts->tm_hour-12;
        if (hour==0)
            hour=12;
        sprintf(buf,"%02d:%02d:%02d P.M.",
            hour, ts->tm_min,ts->tm_sec);
    } else
        sprintf(buf,"%02d:%02d:%02d A.M.", ts->tm_hour,
            ts->tm_min,ts->tm_sec);

printf("\n%s %d %s 19%d %s\n",
    weekday[ts->tm_wday], ts->tm_mday,
    month[ts->tm_mon], ts->tm_year, buf);

printf("Today is the %d day of 19%d\n",
    ts->tm_yday, ts->tm_year);

if(ts->tm_isdst)
    printf("Daylight Saving Time is in effect\n");
else
    printf("Daylight Saving Time is not in effect\n");
}
```

See Also

gmtime, time (overview), TIMEZONE

Notes

localtime returns a pointer to a statically allocated data area that is overwritten by successive calls.

log — Mathematics function (libm)

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)

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>
```

```
char *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++ = 0xAAAAAAAAAL;
    } else {
        printf("The logical screen is at %lx\n", lbase);
        printf("The physical screen is at %lx\n", pbase);
    }
    exit();
}
```

See Also

Physbase, Setscreen, TOS, xbios

long — C keyword

Data type

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

C keywords, C language, data formats, declarations, int

longjmp — General function (libc)

Return from a non-local goto

#include <setjmp.h>

int 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** provides a non-local *goto*.

longjmp restores an environment that had been saved by a previous **setjmp** call. It returns the value *rval* to the caller of **setjmp**, just as if the **setjmp** call had just returned. Note that **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)

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's contents

ls [-adflrtw] [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.
- f Flag all directories with a trailing backslash '\ '.
- 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.
- w Print output in columns; write a backslash '\ ' after the name of every directory.

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

lc, commands, msh

lseek — UNIX system call (libc)

Set read/write position

long lseek(*fd, where, how*)

int *fd, how*; long *where*;

lseek changes the *seek position*, or the point within a file where the next read or write operation is performed. *fd* is the file's file descriptor, which is returned by **open**.

where and *how* describe the new seek position. *where* gives the number of bytes that you wish to move the seek position; 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. For example,

```
position = lseek(filename, 100, 0);
```

moves the seek position 100 bytes past the beginning of the file; whereas

```
position = lseek(filename, 0, 1);
```

merely returns the current seek position, and does not change the seek position at all.

lseek differs from its cousin **fseek** in that **lseek** is an TOS call and uses a file

descriptor, whereas `fseek` is a C function and uses a `FILE` pointer.

See Also

STDIO, UNIX routines

Diagnostics

`lseek` returns `-1L` on an error, such as seeking to a negative position. If no error occurs, it returns zero.

Notes

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.

Note that some operating systems, such as MS-DOS, set the displacement from the file descriptor in bytes; others, such as the VAX VMS, set the displacement in sectors. If you want your programs to be fully portable, you should avoid handing an absolute value to `lseek`.

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; for example, the left operand of an assignment statement must be an lvalue. Some operators give lvalue results; for example, if `e` is a pointer expression, `*e` is an lvalue that designates the object to which `e` points.

Note that a *variable* can be used as an lvalue, whereas a constant cannot. For example, you cannot say

```
6 = (foo+bar);
```

A pointer is a variable, and can be manipulated within limits. An array name, however, is a constant and cannot be altered legally. Thus, the code

```
int foo[10];
int *bar;
foo = bar;
```

will generate an error message when you attempt to compile it, whereas the code

```
int foo[10];  
int *bar;  
bar = foo;
```

will not.

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

M

macro — Definition

A **macro** is a body of text that is given a name. When the name is used in a program, it is replaced with the text to which it refers; this is called *macro expansion*. 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 — Technical information

Introduce program's main function

A C program consists of a set of functions, one of which must be called **main**. This function is called from the runtime startup routine 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 returns to the runtime startup; it closes all open file streams and otherwise cleans up, and then returns control to the operating system, passing it the value returned by **main** as exit status.

In some situations (errors, for example), it may be necessary to stop a program, and you may not want to return to **main**. Here, you can use **exit**; it cleans up the debris left by the broken program and returns control directly 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 helps you build programs that consist of more than one file of source code.

Complex programs often consist of several *object modules*, each of which is the product of compiling a *source file*. A source file may refer to one or more **include** files, which can also be changed. Recompiling and relinking complicated programs can be difficult and tedious.

make regenerates programs automatically. It follows a specification of the structure of the program that you write into a file called **makefile**. **make** also checks the date and time that TOS has recorded for each source file and its corresponding object module; to avoid unnecessary recompilation, **make** will recompile a source file only if it has been altered since its object module was last compiled.

The makefile

A **makefile** consists of three types of instructions: *macro definitions*, *dependency definitions*, and *commands*.

A macro definition simply defines a macro for use throughout the **makefile**; for example, the macro definition

```
FILES=file1.o file2.o file3.o
```

Note the use of the equal sign '='.

A dependency definition names the object modules used to build the target program, and source files used to build each object module. It consists of the *target name*, or name of the program to be created, followed by a colon ':' and the names of the object modules that build it. For example, the statement

```
example: $(FILES)
```

uses the macro **FILES** to name the object modules used to build the program **example**. Likewise, the dependency definition

```
file1.o: file1.c macros.h
```

defines the object module **file1.o** as consisting of the source file **file1.c** and the header file **macros.h**.

Finally, a command line details an action that **make** must perform to build the target program. Each command line must begin with a space or tab character. For example, the command line

```
cc -o example $(FILES)
```

gives the **cc** command needed to build the program **example**. Note that the **cc** command lists the *object modules* to be used, *not* the source files.

Finally, you can embed comments within a **makefile**. **make** recognizes any line that begins with a pound sign '#' as being a comment, and ignores it.

make searches for **makefile** first in directories named in the environmental vari-

able **PATH**, and then in the current directory.

Dependencies

The **makefile** specifies which files depend upon other files, and how to recreate the dependent files. For example, if the target file **test** depends upon the object module **test.o**, the dependency is as follows:

```
test: test.o
    cc -o test 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 that **make** recognizes.

make also has a set of rules to regenerate dependent files. For example, for a source file with suffix **.c** and a dependent file with the suffix **.o**, the target **.c.o** gives the regeneration rule:

```
.c.o:
    cc -c $<
```

The **-c** option to the **cc** commands tells **cc** not to link or erase the compiled object module. **\$<** is a macro that **make** defines; it stands for the name of the file that causes the current action. The default suffixes and rules are kept in the files **mmacros** and **mactions**. The dependencies can be changed by editing these files. Both of these should be kept in one of the directories named in the **LIBPATH** environmental variable.

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=-V
```

The other built-in macros are:

\$* target name, minus suffix
\$@ full 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**.

Note that you can override any built-in macro by resetting its value in the environment.

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**, which is sought first in the directories named in the **PATH** environmental variable, and then in the current directory.
- i** Ignore all errors from commands, and continue processing. Normally, **make** exits if a command returns an error.
- n** Test only; suppresses actual execution of commands. Note that if **make** will not run due to memory limitations, you can use this option to generate a script whose commands can then be executed under **msh**; for example:

```
make -n > mscript; set verbose; . mscript; unset verbose
```

msh, however, will not pay attention to error status in the same way as **make**.

- p** Print all macro definitions and target descriptions.
- q** Return a zero exit status if the targets are up to date. Do not execute any commands.
- r** Do not use the built-in rules that describe 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 by 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 environmental parameter **PATH**.

See Also

as, **cc**, **commands**, **msh**

Diagnostics

make reports its exit status if it is 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

The order of items in **mmacros**\.**SUFFIXES** is significant. The consequent of a default rule (e.g., **.o**) must *precede* the antecedent (e.g., **.c**) in the entry **.SUFFIXES**. Otherwise, **make** will not work properly.

malloc — General function (libc)

Allocate dynamic memory

char *malloc(size) unsigned size;

malloc helps to manage a program's free-space arenas. 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** returns allocated memory to the free memory pool.

Each area allocated by **malloc** is rounded up to the nearest even number and preceded by an **unsigned int** that contains the true length. Thus, if you ask for one byte, you will get four, and the **unsigned** that precedes the newly allocated area will be set to four.

When an area is freed, its low order bit is turned on; consolidation occurs when **malloc** passes over an area as it searches for space. The end of each arena contains a block with a length of zero, followed by a pointer to the next arena. Arenas point in a circle.

The most common problem with **malloc** occurs when a program modifies more space than it allocates with **malloc**. This can cause later **mallocs** to go into a loop.

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**, **lmalloc**, **lrealloc**, **notmem**, **realloc**, **setbuf**

Diagnostics

malloc returns **NULL** if insufficient memory is available.

The related function **lmalloc** takes an unsigned long as its *size* argument, and therefore can allocate memory blocks that are larger than 64 kilobytes.

Notes

The commonest error associated with **malloc** is failing to declare it properly. You should always declare **malloc** as returning a pointer to **char**.

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 size of the largest contiguous block of memory. In either case, **Malloc** returns 0 upon failure.

Examples

This example displays the output of **Malloc** when given -1 as its argument.

```
#include <osbind.h>
#define MGRAIN 32768L

main() {
    register long f1, f2, f3, f4;
    register char *p1, *p2;

    f1 = (long)Malloc(-1L);
    p1 = Malloc(MGRAIN);
    f2 = (long)Malloc(-1L);
    p2 = Malloc(MGRAIN);
    f3 = (long)Malloc(-1L);
    Mfree(p1);
    f4 = (long)Malloc(-1L);
    Mfree(p2);

    printf("%lx %lx %lx %lx %lx\n", f1, f2, f3, f4, Malloc(-1L));
    exit(0);
}
```

See Also

gemdos, **Mfree**, **Mshrink**, **TOS**

Notes

As of this writing, **Malloc** appears to have some peculiarities. You should always use **Malloc** to allocate even-sized 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 given a name so it can 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. Note, too, that numerals are manifest constants by definition.

The use of manifest constants in programs helps to ensure that code is portable by isolating the definition of these elements in a single header file, where they need to be changed only once.

See Also

#define, **EOF**, **header file**, **NULL**, **portability**

mantissa — Definition

In mathematics, a **mantissa** is the fractional part of a logarithm. In the context of C, “mantissa” often is used to describe the fractional portion of a floating point number; according to Knuth, however, the proper term is *fraction*.

See Also

data formats, **double**, **float**, **frexp**

math.h — Header file

Declare 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., **HUGE_VAL**; the definition of structure **cpx**, which describes complex variables; definitions of internal compiler functions; and, finally, declarations of all mathematical functions.

See Also

library, **libm**, **mathematics library**

mathematics library — Overview

The following mathematics routines are available with Mark Williams C:

acos	calculate inverse cosine
asin	calculate inverse sine
atan	calculate inverse tangent
atan2	calculate inverse tangent of quotient
cabs	calculate complex absolute value
cos	calculate cosine
cosh	calculate hyperbolic cosine

exp	calculate exponent
fabs	calculate absolute value function
floor	calculate floor function
hypot	calculate hypotenuse
j0	calculate Bessel function, order 0
j1	calculate Bessel function, order 1
jn	calculate Bessel function, order <i>n</i>
log	calculate natural logarithm
log10	calculate common logarithm
pow	calculate power
sin	calculate sine
sinh	calculate hyperbolic sine
sqrt	calculate square root
tan	calculate tangent
tanh	calculate 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.

maxmem — External data

```
extern unsigned int maxmem;
```

maxmem is an external variable that sets the maximum size of the program's data area. You can set **maxmem** in your program to protect a portion of memory from the memory allocation routine **sbrk**; otherwise, **maxmem** is set to the end of physical memory by the C runtime startup routine.

See Also

—end, **malloc**, **sbrk**

me — Command

MicroEMACS screen editor
me [-e] [*file ...*]

me is the command for MicroEMACS, the screen editor for Mark Williams C. With MicroEMACS, you 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, while displaying the contents of each file in its own screen window.

Screen layout

If the command **me** is used without arguments, MicroEMACS opens an empty buffer. If used with one or more file name arguments, MicroEMACS will 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*. It is important to remember that the current position is always to the *left* of the cursor, and lies *between* two letters, rather than at one letter or another. For example, if the cursor is positioned at the letter 'k' of the phrase "Mark Williams", then the current position lies *between* the letters 'r' and 'k'.

Commands and text

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 control character can be inserted into the text by prefixing it with **<ctrl-Q>** (that is, hold down the **<control>** key and type the letter 'Q').

There are two types of commands to remove text. *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. Moving the cursor before you kill a line will empty the kill buffer, and write the line just killed into it.

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. MicroEMACS defines a word as consisting of all alphabetic characters, plus '_' 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.

Text can also be handled in blocks. MicroEMACS defines a block of text as all the text that lies between the *mark* and the current position of the cursor. For example, typing **<ctrl-W>** kills all text from the mark to the current position of the

cursor; this is useful when moving text from one file to another. When you invoke MicroEMACS, the mark is set at the beginning of the file; you can reset the mark to the cursor's current position by typing `<ctrl-@>`.

Using MicroEMACS with the compiler

MicroEMACS can be invoked automatically by the compiler command `cc` to help you repair all errors that occur during compilation. The `-A` option to `cc` causes MicroEMACS to be invoked automatically when an error occurs. The compiler error messages are displayed in one window, the source code in the other, and the cursor is at the line on which the first error occurred. When the text is altered, exiting from MicroEMACS automatically recompiles the file.

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.

The MicroEMACS help facility

MicroEMACS has a built-in help facility. With it, you can ask for information either for a word that you type in, or for a word over which the cursor is positioned. The MicroEMACS help file contains the bindings for all library functions and macros included with Mark Williams C.

For example, consider that you are preparing a C program and want more information about the function `fopen`. Type `<ctrl-X>?`. At the bottom of the screen will appear the prompt

Topic:

Type `fopen`. MicroEMACS will search its help file, find its entry for `fopen`, then open a window and print the following:

```
Open a stream for standard I/O
#include <stdio.h>
FILE *fopen (name, type) char *name, *type;
```

If you wish, you can kill the information in the help window and copy it into your program, to ensure that you prepare the function call correctly.

Consider, however, that you are checking a program written earlier, and you wish to check the call for a call to `fopen`. Simply move the cursor until it is positioned over one of the letters in `fopen`, then type `<esc>?`. MicroEMACS will open its help window, and show the same information it did above.

To erase the help window, type `<esc>2`.

Options

The following list gives the MicroEMACS commands. They are grouped by function, e.g., *Moving the cursor*. Some commands can take an *argument*, which specifies how often the command is to be executed. The default argument is 1. The command `<ctrl-U>` introduces an argument. By default, it sets the argument

to four. Typing **<ctrl-U>** followed by a number sets the argument to that number. Typing **<ctrl-U>** followed by one or more **<ctrl-U>**s multiplies 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. Same as **<ctrl-X>G**.
- <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, it will prompt for a line number. Same as **<esc>G**.
- <esc>!** Move the current line 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 zero.
- <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 set to one, 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.
- <ctrl-X> <ctrl-O>**
Kill blank lines at current position.
- <ctrl-Y>** (Yank back) 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> **
Kill the previous word.
- ** 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 into two windows. This command is 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 text in current window down by *argument* lines.
- <ctrl-X> <ctrl-P>**
Move text in 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> < Move to previous error.

Search and replace

<ctrl-R> (Reverse) Incremental search backward; a pattern is sought as each character is typed.

<esc>R (Reverse) Search toward the beginning of the file. Waits for entire pattern before search begins.

<ctrl-S> (Search) Incremental search forward; a pattern is sought as each character is typed.

<esc>S (Search) Search toward the end of the file. Waits for entire pattern before search begins.

<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 forward or backward.

<esc>@ Toggle case sensitivity for searches. By default, searches are case insensitive.

Keyboard macros

<ctrl-X> (Begin a macro definition. MicroEMACS collects everything typed until the next **<ctrl-X>**) for subsequent repeated execution. **<ctrl-G>** breaks the definition.

<ctrl-X>) End a macro definition.

<ctrl-X>E (Execute) Execute the keyboard macro.

Change case of text

<esc>C (Capitalize) Capitalize the next word.

<ctrl-X> <ctrl-L>
(Lower) Convert all text from current position to mark into lower case.

<esc>L (Lower) Convert the next word to lower case.

<ctrl-X> <ctrl-U>
(Upper) Convert all text 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. This is often used in C programs to preserve the current level of indentation.

<ctrl-M> (Return) If the following line is not empty, insert a new line; if empty, move to next line.

<ctrl-O> Open a blank line; that is, insert newline after the current position.

<tab> With argument, set tab fields at every *argument* characters. An argument of zero restores the default of eight characters. Note that setting the tab to any character other than eight causes space characters to be set in your file instead of tab 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.

<ctrl> <space>
Set mark at current position.

Help window

<ctrl-X>? Prompt for word for which information is needed.

<esc>? Search for word over which cursor is positioned.

<esc>2 Erase help window.

Miscellaneous

<ctrl-G> Abort a command.

<ctrl-L> Redraw 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 the characters before and after the current position.

<ctrl-U> Specify a numeric argument, as described above.

<ctrl-U> <ctrl-X> <ctrl-C>
Abort editing and re-compilation. Use this command to abort editing and return to TOS when you are using the -A option to the cc command.

<ctrl-X>F Set word wrap to *argument* column. If argument is one, set word wrap to cursor's current position.

<ctrl-X> <ctrl-X>
Mark the current position, then jump to the previous setting of the mark. This is useful when moving text from one place in a file to another.

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. The file on disk is not changed until you save the edited text. MicroEMACS prints a warning and prompts you whenever a command would cause it to lose changed text.

See Also
commands

Notes

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 to edit. If this happens when you first invoke a file, you should exit from the editor immediately. Otherwise, your file on disk will be truncated. If this happens in the middle of an editing session, however, delete text until the message disappears, then save your file and exit. Due to the way MicroEMACS works, saving a file after this error message has appeared will take more time than usual.

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 or pattern search commands.

Note that the current version of MicroEMACS, including source code, is proprietary to Mark Williams Company. The code may be altered or otherwise changed for your personal use, but it may *not* be used for commercial purposes, and it may not be distributed without prior written consent by Mark Williams Company.

MicroEMACS is based upon 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 **msk** 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 new disk has been inserted into a floppy-disk drive. *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

memchr — String function (libc)

Search a region of memory for a character

char *memchr(*region*, *character*, *n*)

char *region;

unsigned int character, n;

memchr searches the first *n* characters in *region* for *character*. It returns a pointer to *character* if it is found, or NULL if it is not.

Unlike the string-search function **strchr**, **memchr** searches a region of memory. Therefore, it does not stop when it encounters a null character.

See Also

strchr, **string**

memcmp — String function (libc)

Compare two regions

int memcmp(*region1*, *region2*, *count*)

char *region1, *region2;

unsigned int count;

memcmp compares *region1* with *region2* character by character for *count* characters.

If every character in *region1* is identical to its corresponding character in *region2*, then **memcmp** returns zero. If it finds that a character in *region1* has a numeric value greater than that of the corresponding character in *region2*, then it returns a number greater than zero. If it finds that a character in *region1* has a numeric value less than that of the corresponding character in *region2*, then it returns a

number less than zero.

For example, consider the following code:

```
char region1[13], region2[13];
strcpy(region1, "Hello, world");
strcpy(region2, "Hello, World");
memcmp(region1, region2, 12);
```

memcmp scans through the two regions of memory, comparing **region1[0]** with **region2[0]**, and so on, until it finds two corresponding “slots” in the arrays whose contents differ. In the above example, this will occur when it compares **region1[7]** (which contains ‘w’) with **region2[7]** (which contains ‘W’). It then compares the two letters to see which stands first in the character table used in this implementation, and returns the appropriate value.

See Also

strcmp, **string**, **strncmp**, **strstr**

Notes

memcmp compares regions of memory rather than strings; therefore, it does not stop when it encounters a null character.

memcpy — String function (libc)

Copy one region of memory into another

```
char *memcpy(region1, region2, n)
char *region1, *region2;
unsigned int n;
```

memcpy copies *n* characters from *region2* into *region1*. Unlike the routines **strcpy** and **strncpy**, **memcpy** copies from one region to another; therefore, it will not halt automatically when it encounters a null character.

memcpy returns *region1*.

See Also

memmove, **strcpy**, **string**, **strncpy**

Notes

If *region1* and *region2* overlap, the behavior of **memcpy** is undefined. *region1* should point to enough reserved memory to hold *n* bytes of data; otherwise, code or data will be overwritten.

memory allocation — Technical information

The following diagram shows how Mark Williams C allocates memory.

VIDEO RAM	highest address
ARENA AND FREE MEMORY	
STACK	
UNINITIALIZED DATA	uninitialized instructions & data
INITIALIZED DATA	private data, shared data, strings
TEXT CODE	instructions
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 runtime startup sets the stack size to two kilobytes (2,048 bytes) 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 an *even* constant that specifies the number of bytes to allocate.

Example

The example in the entry for **Physbase** displays system memory graphically.

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

memset — String function (libc)

Fill an area with a character

```
char *memset(buffer, character, n);  
char *buffer; int character; unsigned int n;
```

memset fills the first *n* bytes of the area pointed to by *buffer* with copies of *character*. It casts *character* to an **unsigned char** before filling *buffer* with copies of it.

memset returns the pointer *buffer*.

See Also

memchr, memcmp, memcpy, memmove, string

menu — Technical information

A **menu** is a graphics form that is used extensively by GEM. It is a specialized form of an AES object, which is defined by 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. 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 bar object

The **root** object's first child is the **bar** object. It describes the menu bar, 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 resolution, a character is 16 rasters high; in medium and low resolutions, 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**. This type of object is used only with menus. By design, the first (leftmost) title controls the drop-down menu that names the available GEM desk accessories.

The screen object

The **screen** object is the **root** 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** must equal the number of titles. Each **box** is of type **G_BOX**.

Each **box** 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. GEM stores in a buffer 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 bits. No **box** should exceed this limit, or debris will be left on the screen when the menu is erased.

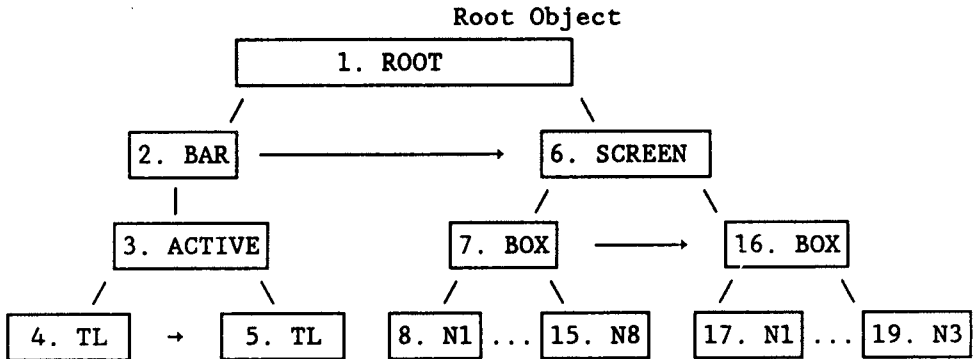
The name objects

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. 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 have been loaded.

Genealogical table

The following "genealogical 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:



You must invoke the menu with the function `menu_bar`; 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 in this example are sized by hand to fit with a screen that is 640 rasters across by 400 rasters high. If your screen does not match these dimensions, this example may not work. It will, however, show you how the elements of the menu object fit together.

```

#include <aesbind.h>
#include <gemdefs.h>
#include <obdefs.h>

```

```

OBJECT mask[] = { -1,-1,-1,G_BOX,LASTOB,NORMAL, 0x11c1L, 0, 0,
                  640, 400 };

#define MDESK 7
#define MCOM 16
#define MHORSE 17
#define MMOUSE 18
#define MPIG 19
#define MPARROT 20
#define MQUIT 22

OBJECT menu[] = {
    { -1, 1, 5, G_IBOX, NONE, NORMAL, 0, 0, 0, 80, 25 },
    { 5, 2, 2, G_BOX, NONE, NORMAL, (BLACK<<12)|(BLACK<<8), 0, 0, 80, 1+(2<<8) },
    { 1, 3, 4, G_IBOX, NONE, NORMAL, 0, 2, 0, 12, 1+(3<<8) },
    { 4, -1, -1, G_TITLE, NONE, NORMAL, (long)" Desk ", 0, 0, 6, 1+(3<<8) },
    { 2, -1, -1, G_TITLE, NONE, NORMAL, (long)" Menu ", 6, 0, 6, 1+(3<<8) },

    { 0, 6, 15, G_IBOX, NONE, NORMAL, 0, 0, 1+(3<<8), 80, 19 },
    { 15, 7, 14, G_BOX, NONE, NORMAL, ((-1L&0xFF)<<16)|(BLACK<<12)|(BLACK<<8),
      2, 0, 22, 8 },
    { 8, -1, -1, G_STRING, NONE, NORMAL, (long)" About menu", 0, 0, 22, 1 },
    { 9, -1, -1, G_STRING, NONE, DISABLED, (long)"-----", 0, 1, 22, 1 },

    { 10, -1, -1, G_STRING, NONE, NORMAL, (long)" Desk 1", 0, 2, 22, 1 },
    { 11, -1, -1, G_STRING, NONE, NORMAL, (long)" Desk 2", 0, 3, 22, 1 },
    { 12, -1, -1, G_STRING, NONE, NORMAL, (long)" Desk 3", 0, 4, 22, 1 },
    { 13, -1, -1, G_STRING, NONE, NORMAL, (long)" Desk 4", 0, 5, 22, 1 },
    { 14, -1, -1, G_STRING, NONE, NORMAL, (long)" Desk 5", 0, 6, 22, 1 },

    { 6, -1, -1, G_STRING, NONE, NORMAL, (long)" Desk 6", 0, 7, 22, 1 },
    { 5, 16, 22, G_BOX, NONE, NORMAL, ((-1L&0xFF)<<16)|(BLACK<<12)|(BLACK<<8),
      8, 0, 11, 7 },
    { 17, -1, -1, G_STRING, NONE, NORMAL, (long)" Cow", 0, 0, 11, 1 },
    { 18, -1, -1, G_STRING, NONE, NORMAL, (long)" Horse", 0, 1, 11, 1 },

    { 19, -1, -1, G_STRING, NONE, NORMAL, (long)" Mouse", 0, 2, 11, 1 },
    { 20, -1, -1, G_STRING, NONE, NORMAL, (long)" Pig", 0, 3, 11, 1 },
    { 21, -1, -1, G_STRING, NONE, NORMAL, (long)" Parrot", 0, 4, 11, 1 },
    { 22, -1, -1, G_STRING, NONE, DISABLED, (long)"-----", 0, 5, 11, 1 },
    { 15, -1, -1, G_STRING, LASTOB, NORMAL, (long)" Quit", 0, 6, 11, 1 },
};

#define NMENU(sizeof menu / sizeof menu[0])

alertf(n, p) int n; char *p;
{
    static char buffer[512];
    sprintf(buffer, "%r", &p);
    return form_alert(n, buffer);
}

main()
{
    int b[8], n, x, y, w, h;

```



```
/* open application; set pointer to arrow */
appl_init();
graf_mouse(ARROW, &n);

for (n = 0; n < NMENU; n += 1)
    rsrc_obfix(menu, n);

/* build window, draw object, open menu */
wind_get(0, WF_FULLXYWH, &x, &y, &w, &h);
objc_draw(mask, ROOT, MAX_DEPTH, x, y, w, h);
menu_bar(menu, 1);

/* wait for a message from the user */
for (;;) {
    evnt_mesag(b);
    /* b[0] holds the type of message */
    switch (b[0]) {
        /* if menu is clicked ... */
        case MN_SELECTED:
            /* ... b[4] holds entry clicked */
            switch(b[4]) {
                case MDESK:
                    alertf(1, "[0] [Menu |menu ] [Ok]");
                    break;

                case MCOW:
                    alertf(1, "[0] [MOO! ] [Ok]");
                    break;

                case MHORSE:
                    alertf(1, "[0] [NEIGH! ] [Ok]");
                    break;

                case MMOUSE:
                    alertf(1, "[0] [SQUEAK! ] [Ok]");
                    break;

                case MPIG:
                    alertf(1, "[0] [OINK! ] [Ok]");
                    break;

                case MPARROT:
                    alertf(1, "[0] [SQUAWK! ] [Ok]");
                    break;

                case MQUIT:
                    menu_bar(menu, 0);
                    appl_exit();
                    exit(0);

                default:
                    alertf(1, "[0] [item %d? ] [Ok]", b[4]);
                    break;
            }
        }

    menu_tnormal(menu, b[3], 1);
    break;
}
```

```

        default:
            alertf(1, "[0] [message %d? ] [Ok]", b[0]);
            break;
    }
}

```

See Also

AES, object, TOS, window

menu_bar — AES function (libaes)

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_ichck — AES function (libaes)

Write or erase a check mark next to a menu item

```
#include <aesbind.h>
```

```
#include <obdefs.h>
```

```
int menu_ichck(tree, item, eraseshow) OBJECT *tree; int item, eraseshow;
```

menu_ichck 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_ichck** 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)

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)

Add a name to the desk accessory menu list

```
#include <aesbind.h>
#include <obdefs.h>
int menu_register(accessory, textstring) int accessory; char *textstring;
```

menu_register is an AES routine that adds a name to the desk accessory menu list. *accessory* is the ID of the desk accessory. *textstring* points to the string of text inserted into the desk accessory menu. For more information about the desk accessory menu, see the entry for **menu**.

menu_register returns the desk accessory's identifier, from zero through five.

Example

For an example of this function, see the entry for **desk accessory**.

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)

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)

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 — Technical information

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 how the program works. More importantly, because the elements of an image are described logically rather than absolutely, it allows each element to be manipulated easily, and the image as a whole to be maneuvered. This lets you create images independent of 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 followed 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 gives a brief description.

3	0	<code>v_clrwk</code>	clear a virtual device
4	0	<code>v_updwk</code>	update workstation (flush buffers)
5	2	<code>v_exit_cur</code>	exit from alphabetic mode
5	3	<code>v_enter_cur</code>	enter alphabetic mode
5	20	<code>v_form_adv</code>	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_rfbbox	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	set polymarker type
19	0	vsm_height	set polymarker height
20	0	vsm_color	set polymarker color
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_rectl	draw a rectangular fill area
129	0	vs_clip	clip an area of the virtual device

See Also

TOS, v_meta_extents, 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)

Generate a temporary file name

char *mktemp(pattern) char *pattern;

mktemp generates a unique file name. It can be used, for example, to name intermediate data files.

Note that the functions **tmpnam** and **tempnam** each assemble a temporary file name and then call **mktemp**. These routines ease the difficulty in creating a proper name for a temporary file.

See Also

msh, tempnam, tmpnam

modf — General function (libc)

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 argument *real*, which is a value *f* in the range $0 \leq f < 1$. It also stores the integral part in the double location referenced by *ip*. These numbers satisfy the equation $real = f + *ip$.

Example

This example prompts for a number from the keyboard, then uses **modf** to calculate the number's fractional portion.

```
#include <stdio.h>

main()
{
    extern char *gets();
    extern double modf(), atof();
    double real, fp, *ip;
    char string[64];

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

        real = atof(string);
        fp = modf(real, ip);
        printf("%lf is the integral part of %lf\n",
               *ip, real);
        printf("%lf is the fractional part of %lf\n",
               fp, real);
    }
}
```

See Also

atof, ceil, fabs, floor, frexp, ldexp

modulus — Definition

Modulus is the operation that returns the remainder derived from a division operation. For example, 12 modulus four equals zero, because when 12 is divided by four it leaves no remainder. The term "modulo" also refers to the product of a modulus operation; in the above example, the modulo is zero. In C, the modulus operation is indicated with a percent sign '%'; therefore, 12 modulus 4 is written **12%4**.

The modulus operation often is used to trim numbers to a preset range. For example, if you wanted to create a list of single-digit random numbers, you would use the command:

```
rand()%10
```

This is demonstrated by the following example.

Example

This example prints a list of 20 single-digit random numbers. The random-number table is seeded with a portion of the current system time.

```
main()
{
    long nowhere; /* place to put unused pointer */
    int counter;

    srand((int)time(&nowhere));
    for (counter = 0; counter < 20; counter++)
        printf("%d\n", rand()%10);
}
```

See Also
operator

mousehidden — Command

Return how often mouse pointer has been hidden
mousehidden

mousehidden is a command that returns the number of times the mouse pointer has been hidden. Under GEM, if the mouse pointer has been hidden more than once, it must be “restored” as many times as it has been hidden before it reappears on the screen. **mousehidden** will tell you how many times you must restore the mouse pointer before it reappears.

See Also

commands, hidemouse, Line A, showmouse, TOS

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 command

msh also comes equipped with a history command. This command allows you to re-execute a command without having to retype it.

msh automatically records your previously typed commands into a buffer. The number of commands it "remembers" is set by the **history** variable: for example, typing

```
set history=8
```

tells **msh** to remember the last eight commands you have issued.

The history command is invoked with the exclamation point '!'. The command **!!** re-executes your last command. Therefore, the commands

```
ls -w
!!
```

will give you two columnar listings of the contents of your current directory. The command **!*name*** re-executes the last command with *name* that is in your history directory. For example, when you type the following list of commands:

```
ls -w
echo foo >stuff
rm stuff
!!s
```

the history command **!!s** reaches back and executes the command **ls -w**. To execute a previous command by its number, subtract its value from the current command. For example, **!-1** re-executes the previous command; it is a synonym for **!!**. To execute a command issued three commands ago, type **!-3**; this will execute **echo foo >stuff**.

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 `lname` 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.

Directories .bin and .cmd

msh has two built-in directories: `.bin` and `.cmd`. `.bin` holds **msh**'s built-in commands. It is searched automatically by **msh**, and so it does not need to be listed in the `PATH` environment parameter. `.cmd` holds user-defined commands. You can create a new command and load it into `.cmd` with the `set` command. For example, the following command to **msh** creates a new list command:

```
set in .cmd lc="ls -w"
```

This tells **msh** to equate the command `lc` with the string `ls -w`, which prints the contents of a directory in columnar format. `.cmd` must be included in the `PATH` environment, or it will not be searched by **msh** when you issue a command.

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.

Command substitutions

msh supports command substitutions. These allow you to embed a command within another command; the output of the inner command is automatically passed as input to the outer command. Command substitutions are indicated by quoting the inner command with grave accents. For example, the command:

```
pr 'ls -l'
```

first invokes the list command **ls** to read the contents of the current directory, and then passes its output to the pagination command **pr**, which paginates what **ls** produces and displays it on the standard output device.

One form of embedded command is included in **profile**: the command

```
date 'date -i'
```

resets the GEM time after a warm boot.

Note that the grave mark may be passed to **msh** as a literal character if it is preceded with a slash '/

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** Append the output of a command onto *file*. If *file* does not exist, create it and fill it with the output of the command.
- 2> file** Redirect material normally sent to the standard error device into *file*.
- 2>> file** Append material normally sent to the standard error device onto the end of *file*.
- 3> file** Redirect material normally sent to the printer into *file*.
- 3>> file** Append material normally sent to the printer onto the end of *file*.
- >& file** Redirect the output of a command and any diagnostic messages it produces into *file*.
- >>& 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**, **crt0.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.

cwdisk The current working device.

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.

msh has three aliases built into it, which extends the range of your command files. The alias **\$*** represents all of the arguments to the current command. For example, the command

```
cc -V $*
```

when placed into a file, compiles all of the files listed as arguments to that file.

\$# gives the number of arguments assigned to the current command. For example, the command

```
echo $#
```

prints 1 on the screen, which is the number of arguments to that command.

Finally, the alias **\$<** represents any line received from the standard input device, up to the newline character. For example, the following command gives a very slow version of the concatenation command, **cat**:

```
set in .cmd slowcat=(
while (set foo="$<" && not (equal "$foo" "")) (echo $foo)
)
```

Loops and conditional statements

msh supports conditional statements and loops. The basic conditional command is **if**. Its syntax is as follows:

```
if word1 word2 [ word3 ]
```

If **word1** executes successfully, then **word2** is executed; otherwise, the **word3**, if present, is executed. Commands can be grouped into statements by enclosing them within parentheses. The text within the parentheses can extend across as many lines of text without needing to precede newlines with backslashes. For example, the command

```
if (echo foo
    echo bar
    echo baz) (ls -l)
```

echoes the strings **foo**, **bar**, and **baz**, and then prints the contents of the current directory.

msh also contains a **while** command, which can control a conditional loop. Its syntax is as follows:

```
while word1 word2
```

As long as **word1** executes successfully, **word2** will also be executed. Each word may be a list of commands enclosed within parentheses.

msh contains two test commands, **equal** and **not**. **equal** compares two strings; it succeeds if the strings are identical, and fails if they are not. Its syntax is as follows:

```
equal argument1 argument2
```

Note that either argument can be a literal string, an integer, or an embedded command. **not** inverts the logical result of its argument.

The command **is_set** is also useful in building loops and conditional statements. This command takes the name of an environmental variable as its argument. It returns zero if the variable is set, and a number greater than zero if it is not. Its syntax is as follows:

```
is_set [ in dir ] name
```

which is much like that of the **set** command.

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.

msh also uses another file, called **postfile**, that restores the desktop environment when you exit from **msh**.

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*) **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 kept, and *length* indicates the amount of memory to be kept. **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.

mshversion — Command

Print current version of **msh**

mshversion

The command **mshversion** prints the version of the microshell **msh** that you are using. Typing **mshversion** returns a string of the form:

Mark Williams Micro Shell, version 3.0

See Also

commands, **msh**

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

List processor code numbers

The header file **mtype.h** assigns a code number to each of the processors supported by Mark Williams C compilers. These include the Intel i8086, i8088, i80186, and i80286; the Zilog Z8001 and Z8002; the DEC PDP-11 and VAX; the IBM 370, and the Motorola 68000.

See Also

header file, **portability**

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

mwtomw — Command

Convert old Mark Williams format to Mark Williams 3.0 format
mwtomw *filename*

The command **mwtomw** takes an executable file that had been compiled with Mark Williams C version 2.1.7 or earlier, and converts it to the format used with version 3.0. *filename* is the name of the executable file to convert.

See Also

commands, drtomw, ld

N

nested comments — Definition

Both *The C Programming Language* and the draft ANSI standard declare that comments cannot be nested. Earlier versions of Mark Williams C included a switch, called `-VCNEST`, that allowed a programmer to nest comments. This switch has been removed. Current and future versions of Mark Williams C abort compilation when they detect nested comments.

See Also

C language

newline — Character constant

Mark Williams C recognizes the literal character `'\n'` for the ASCII newline character LF (octal 012). This normally feeds the line and returns the carriage. This character may be used as a character constant or in a string constant.

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*. Each *file* argument must be a 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 that are in 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 Append the file name to the beginning of each output line.
- p Print symbols in the order in which they appear within the symbol table.
- r Sort in reverse-alphabetical order.
- u Print only undefined symbols.

By default, **nm** sorts symbol names alphabetically. Each symbol is followed by its value and its segment.

See Also

cc, commands, ld, size, strip

Notes

Because version 3.0 changes the format of executable files, the edition of **nm** shipped with version 3.0 does not work with executables linked with Mark Williams C version 2.1.7 or earlier. To convert such files to a format that **nm** recognizes, use the command **mwto_{nm}**.

nm now works with symbol tables larger than 64 kilobytes. It uses a balanced-tree to sort symbols, and will absorb as much memory as the system has available. When memory runs out, **nm** prints the symbols already sorted and continues with the remainder of the symbol table.

not — Command

Invert logical value of an argument

not *argument*

not is a test command that is built into the microshell **msh**. It inverts the logical value of *argument*; that is, it changes zero to one and any value other than zero to zero. *argument* may be either an absolute value or a value returned by another command.

Example

The following command prints the string **Not high res** if the monitor is not in high resolution, and it prints **High res** if the monitor is in high resolution.

```
(if (not (equal 'getrez 2))  
    (echo "Not high res") (echo "High res"))
```

Note that the command **getrez** returns two if the monitor is in high resolution.

See Also

commands, equal, if, is_set, msh, while

notmem — General function (libc)

Check if memory is allocated

int notmem(ptr) char *ptr;

notmem checks if a memory block has been allocated by **calloc**, **malloc**, **lmalloc**, **lmalloc**, or **realloc**. *ptr* points to the block to be checked. **notmem** searches the arena for *ptr*; it returns one if *ptr* is not a memory block obtained from **malloc**, **calloc**, or **realloc**, and zero if it is.

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. **Lmagic** is the “magic number” that identifies a load module; it always contains 0407. **Lflag** contains flags that indicate the type of the object module. **Lmachine** is the processor identifier. **Ltbase** is the start of the text segment. **Lentry** contains the machine address where execution of the module commences. **Lssize** 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**

nout.h — Header file

Describe output format **n.out**

#include <nout.h>

The header file **nout.h** contains the description of the output format **n.out**, which is created by Mark Williams C. It is used by the compiler, the assembler, and by the linker **ld** to create correctly formatted output files.

See Also

header file, n.out

NUL — Character constant

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 can, in fact, hold 49 printable characters plus **NUL**.

See Also

ASCII, character constant, NULL, string

NULL — Manifest constant

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

O

obdefs.h — Header file

Declare TOS objects and structures
#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)

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)

Change object's state

#include <aesbind.h>

#include <obdefs.h>

int objc_change(tree, object, junk, x, y, w, h, newstate, redraw)
OBJECT *tree; int object, junk, x, y, w, h, newstate, redraw;

objc_change is an AES routine that changes the state of an object within a named clipping rectangle. This is done by altering the member **ob_state** within the **OBJECT** structure. For more information on object states, see the entry for **object**.

objc_change is a simple extension to **objc_draw**, which allows you to reset **ob_state** and optionally skip redrawing the object.

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.

x, *y*, *w*, and *h* set, respectively, the X coordinate of the clipping rectangle, its Y coordinate, its width, and its height. **objc_change** will alter only the portion of the object that falls within this clipping rectangle.

newstate indicates the new state for the object, as follows:

0x00	NORMAL	Object is normal
0x01	SELECTED	Shown in reverse video
0x02	CROSSED	Object has 'X' drawn next to it
0x04	CHECKED	Check mark drawn next to object
0x08	DISABLED	Object redrawn in gray; unselectable
0x10	OUTLINED	Object is outlined
0x20	SHADOWED	Object has shadow drawn beneath it

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)

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)

Draw an object

#include <aesbind.h>

#include <obdefs.h>

int objc_draw(*tree*, *object*, *depth*, *x*, *y*, *w*, *h*)

OBJECT *tree; int object, depth, x, y, w, h;

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 the number of levels to which the object should be drawn, as follows: 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 tree.

x, *y*, *w*, and *h* set, respectively, the X coordinate of the clipping rectangle, its Y coordinate, its width, and its height.

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)

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 the character that follows the one being edited. This value is set by the 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)

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 descendants.

Finally, *mousex* and *mousey* give the coordinates of the mouse pointer.

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_offset — AES function (libaes)

Calculate an object's absolute screen position

```
#include <aesbind.h>
int objc_offset(tree, object, x, y)
OBJECT *tree; int object, *x, *y;
```

objc_offset 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. *x* and *y* give, respectively, the X and Y coordinates of the object. These are set by AES.

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

See Also

AES, object, TOS

objc_order — AES function (libaes)

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, TOS

object — Technical information

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 in a tree structure. 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 **example**. Like all object trees, **example** has a root object, **example[0]**. This object, in turn, has three children: **example[1]**, **example[2]**, and **example[3]**. Each of these three children has two siblings. For example, **example[2]**'s siblings are **example[1]** and **example[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, **example** names an array of objects. Each object's subscript depends on the order in which it is loaded 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 **example[1]** is 2, which points to **example[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, **example[0]** has a head pointer of 1, which indicates that **example[1]** is the first of its children, and a tail pointer of 3, which indicates that **example[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 that if object 1's **head** is set to two and its **tail** is set to seven, 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 object 2 and the last is object 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;           /* First child */
    int ob_tail;           /* Last child */
    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 the 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	Exit when button is pressed
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**. The former must be a selectable object that the user must touch with the mouse pointer and then click and release the button. The latter must *not* be a selectable object and exits when the user merely clicks the mouse

button, rather than when button is released.

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	'X' drawn in object
0x04	CHECKED	Check mark drawn next to object
0x08	DISABLED	Draw in shading rather than solid
0x10	OUTLINED	Draw border around object
0x20	SHADOWED	Draw shadow beneath object

The **SELECTED** specification is often used to show that an object has been selected, such as happens when you click an icon on the GEM desktop. The specifications **CROSSED**, **OUTLINED**, and **SHADOWED** are used only with boxes.

The specification can be changed as the program runs; for example, the specification in a menu object can change to indicate that the item is disabled or has been selected. You can either change an object's specification by hand, or you can use an AES library routine to do so. For example, **menu_tnormal** will change a menu entry's specification from **DISABLED** to **NORMAL** and vice versa, without your having to address that object directly within its tree. See **objc_change** for more information.

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 changing 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 uses 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:
--------------	--

20

<i>bits 0-3</i>	interior color
<i>bits 4-6</i>	interior pattern (0=empty, 7=solid)
<i>bit 7</i>	1=transparent, 0=opaque
<i>bits 8-11</i>	text color
<i>bits 12-15</i>	border color
<i>bits 16-23</i>	border thickness (-127 through 127)
<i>bits 24-31</i>	character index

Negative numbers draw the border outwards from the edge of the rectangle, whereas positive numbers draw the border inwards.

The codes for text and interior color are as follows:

0	WHITE
1	BLACK
2	RED
3	GREEN
4	BLUE
5	CYAN
6	YELLOW
7	MAGENTA
8	LWHITE
9	LBLACK
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 mnemonics in your program without having to remember the numeric code of each color. *Example:* To set a figure with a border width of one raster, a border color of black, a text color of black, the transparent bit off, the fill pattern of solid, and an interior color of white, use the following C code:

```
(1<<16)|(BLACK<<12)|(BLACK<<8)|(1<<7)|(7<<4)|WHITE)
```

This translates into the hexadecimal number `0x111F0L`.

G_BOXCHAR

27

This draws a rectangle with a single character inside it. It is used for elements like the “fuller” button on GEM windows. `ob_spec` is the same as for `G_BOX`, except that bits 24-31 encode the character to be displayed within the box.

G_BOXTEXT

22

This draws a box and writes text inside it. **ob_spec** points to the structure **TEDINFO**, which is described below.

G_BUTTON

26

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

29

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

30

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

25

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** is the same as for **G_BOX**.

G_ICON

31

This draws an icon on the screen. **ob_spec** points to the structure **ICONBLK**, which is described below.

G_IMAGE

23

This draws a user-defined shape on the screen. **ob_spec** points to the structure **BITBLK**, which is described below.

G_STRING

28

This writes a string. **ob_spec** points to the string being written.

G_TEXT

21

This writes formatted text. **ob_spec** points to the structure **TEDINFO**, which is described below.

G_TITLE

32

This creates a title on the menu bar. **ob_spec** points to the string to be written. This object is used only in a menu.

G_USERDEF

24

This is an object defined by the programmer. **ob_spec** points to the structure **USERBLK**, which is described below.

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 bit */
} 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** holds the index of the character for the icon in its low byte. The foreground (data) color is stored in bits 12-15, and the background (mask) color is stored in bits 8-11. The color codes are the same as those listed for **G_BOX**, above. **ib_xchar** and **ib_ychar** give, respectively, the X and Y coordinates of the character. **ib_xicon**, **ib_yicon**, **ib_wicon**, and **ib_hicon** give, respectively, the X coordinate, the Y coordinate, the width, and the height of the icon. **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 at the bottom of 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

In some versions of the AES, entry of an underscore to 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 is 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 you to define your 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 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. Whenever possible, you should use **resource**, the Mark Williams resource editor to design screen elements, or prepare a resource-description file that can be compiled with **rescomp**, the Mark Williams resource compiler. The following describes how to build an object by hand in C, to help you grasp the structure of objects and object trees.

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 be 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 is tedious, but it will save you time over trying to draw it "on the fly" on the screen.

Second, draw a "genealogical 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**.

Example

This example draws a set of seven nested rectangles on the screen. Typing any key returns you to **msh**.

```

#include <aesbind.h>
#include <gemdefs.h>
#include <obdefs.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]
 *        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]
 *        WHITE [Fill color]
 */

/* define object; widths and heights will be set elsewhere */
OBJECT fill[] = {
/* next/head/tail/type/ flags / state /spec./ X / Y/ W / H */
-1, 1, 1, G_BOX, DEFAULT, NORMAL, SPEC1, 0, 0, 0, 0,
0, 2, 2, G_BOX, DEFAULT, NORMAL, SPEC2, 0, 0, 0, 0,
1, 3, 3, G_BOX, DEFAULT, NORMAL, SPEC1, 0, 0, 0, 0,
2, 4, 4, G_BOX, DEFAULT, NORMAL, SPEC2, 0, 0, 0, 0,
3, 5, 5, G_BOX, DEFAULT, NORMAL, SPEC1, 0, 0, 0, 0,
4, -1, -1, G_BOX, DEFAULT, NORMAL, SPEC2, 0, 0, 0, 0
};

main()
{
    int nowhere = 0; /* For unused pointers */
    int i;
    int x, y, w, h; /* Dimensions of screen */

    appl_init(); /* Begin application */
    /*
     * get size of screen; set object dimensions.
     * "0" is desktop window, which always fills screen
     */
    wind_get(0, WF_FULLXYWH, &x, &y, &w, &h);
    fill[0].ob_width = w;
    fill[0].ob_height = h;

    for (i = 1; i < 6; i++) {
        fill[i].ob_x = w/12;
        fill[i].ob_y = h/12;
        fill[i].ob_width = w - ((w/6)*i);
        fill[i].ob_height = h - ((h/6)*i);
    }
}

```



```
/* Turn off mouse pointer */
graf_mouse(M_OFF, &nowhere);

/* Draw object */
objc_draw(fill, ROOT, MAX_DEPTH, 0, 0, w, h);

/* Wait for keybd event */
evnt_keybd();

/* Turn on mouse ptr */
graf_mouse(M_ON, &nowhere);

appl_exit();
exit(0);
}
```

See Also

AES, menu, obdefs.h, rescomp, resdecom, resource, 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

Print a hexadecimal dump of a file

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)

Open a file

int open(*file*, *type*) **char** **file*; **int** *type*;

open prepares a *file* to receive data, or to have its data read. When it can open *file*, **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**. *type* determines how the file is opened, as follows:

- | | |
|---|----------------|
| 0 | read only |
| 1 | write |
| 2 | read and write |

After *file* is opened, reading or writing begins at byte 0.

Example

This example copies the file named in **argv[1]** to the one named in **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***fdopen, file descriptor, fopen, 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 not be mixed 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 the operands 1 and 2 through the operation of addition; on the other hand, the statement

A>B

relates the operands A and B logically, by asserting that the former is greater than the latter; whereas

A=B

relates the operands A and B by assigning the value of the latter to the former. The following is a table of the C 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 complement
	bitwise inclusive OR
<<	shift left
>>	shift right
*	indirection
&	render an address
()	function indicator
[]	array indicator
->	structure pointer
.	structure member
? :	conditional expression

sizeof size of an object

For a table of the precedence of operators, see the entry for **precedence**.

See Also

precedence, sizeof

The C Programming Language, page 48.

osbind.h — Header file

Declare TOS functions

#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

P

path — Definition

A **path** is a sequence of names that are separated by a fixed character, '/' under the COHERENT or UNIX operating systems, or '\' under TOS or MS-DOS. Each name is a directory that contains the next-named directory; the only exception is the last name in the path, which may be the name of a file instead of a directory. For example, the COHERENT path

```
doc/letters/fred
```

names first the directory **doc**; then the directory **letters**, which is a sub-directory of **doc**; and finally **fred**, which can name either a directory or a file.

If a path begins with the separator character, then it is called *absolute*, and the first directory is a sub-directory of the root directory of the current physical device. Otherwise, the path is assumed to begin with the current directory. For example, the above example is a relative path that begins with the current directory; however, the following path names an absolute path for MS-DOS or TOS:

```
A:\doc\letters\fred
```

See Also

msh, **PATH**, **path.h**, **setenv**

path — General function (libc)

Build a path name for a file

```
#include <path.h>
```

```
#include <stdio.h>
```

```
char *path(path, filename, mode) char *path, *filename; int mode;
```

path is a general function that builds a path name for a file. *path* points to the list of directories to be searched for the file. You can use the function **getenv** to obtain the current definition of the environmental variable **PATH**; or use the default setting of **PATH** found in the header file **path.h**; or, you can define *path* by hand. *filename* is the name of the file for which **path** is to search. *mode* is the mode in which you wish to access the file, as follows:

- 1 execute the file
- 2 write to the file
- 4 read the file

path uses the function **access** to check the access status of *filename*. If **path** finds the file you requested and the file is available in the mode that you requested, it returns a pointer to a static area in which it has built the appropriate path name. It returns NULL if either *path* or *filename* are NULL, if the search failed, or if the requested file is not available in the correct mode.

Example

This example accepts a file name and a search mode. It then tries to find the file in one of the directories named in the **PATH** environmental variable.

```
#include <stdio.h>
#include <path.h>

main(argc, argv)
int argc; char *argv[];
{
    char *env, *pathname;
    extern char *getenv(), *path();
    int mode;

    if (argc != 3)
    {
        printf("Usage: findpath filename mode\n");
        exit(0);
    }

    if (((mode=atoi(argv[2]))>4) || (mode==3) || (mode<1))
    {
        printf("modes: 1=execute, 2=write, 3=read\n");
        exit(0);
    }

    env = getenv("PATH");
    if ((pathname = path(env, argv[1], mode)) != NULL)
    {
        printf("PATH = %s\n", env);
        printf("pathname = %s\n", pathname);
    }
    else
        printf("search failed\n");
}
```

See Also

access, **access.h**, **PATH**, **path.h**, **stdio**

path.h — Header file

Declare **path()**

#include <path.h>

path.h is a header file that declares the function **path**. It also contains a number of default definitions for variables, including **PATH** and **LIBPATH**.

See Also

path, **PATH**

PATH — Environmental variable

Directories that hold executable files

PATH names a default set of directories that are searched by **msh** when it seeks an executable file. You can set **PATH** with the command **setenv**. 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**.

See Also

msh, **path**, **path.h**, **setenv**

pattern — Definition

A **pattern** is any combination of ASCII characters and wildcard characters that can be interpreted by a command.

The function **pnmatch** compares two patterns and signals if they match.

See Also

egrep, **pnmatch**, **wildcard**

peekb — General function (libc)

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**

Notes

peekb is supplied for use in user-mode programs. Programs that run in supervisor mode, i.e., interrupt handlers, trap handlers, and boot sector programs, should access the memory locations directly with the following macro:

```
#define peekb(cp)  (*((char *)cp))
```

peekb does not work correctly in supervisor mode, which allows you to access memory locations directly.

peekl — General function (libc)

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**.

peekl is supplied for use in user-mode programs. Programs that run in supervisor mode, i.e., interrupt handlers, trap handlers, and boot sector programs, should access the memory locations directly with the following macro:

```
#define peekl(lp)  (*((long *)lp))
```

peekl does not work correctly in supervisor mode, which allows you to access memory locations directly.

peekw — General function (libc)

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**.

peekw is supplied for use in user-mode programs. Programs that run in supervisor mode, i.e., interrupt handlers, trap handlers, and boot sector programs, should access the memory locations directly with the following macro:

```
#define peekw(wp)  (*((int *)wp))
```

peekw does not work correctly in supervisor mode, which allows you to access memory locations directly.

perror — General function (libc)

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>
```

```
char *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.

Example

The following example uses **Physbase** and **Setscreen** to allow you to display graphically the entire system memory. Typing the up-arrow key scrolls the screen up the equivalent of one character row; typing the down-arrow key scrolls the screen down; typing **<Help>** lets you move the display to an absolute memory address; and typing **<Undo>** tells you the current memory base. Typing **<return>** exits. Moving the memory base to zero allows you to observe the operation of TOS, including stack, clocks, and peripheral devices; you are invited to manipulate the peripheral devices to observe them in action.

```

#include <osbind.h>
#include <gemdefs.h>
#include <osbind.h>
#include <stdio.h>

/* manifest constants */
#define UP_ARROW 0x4800
#define DN_ARROW 0x5000
#define HELP 0x6200
#define RETURN 0x1C0D
#define UNDO 0x6100

/* externs */
extern long atol();

```

```
main()
{
    char *oldphys;      /* base of default video display */
    char *setting;      /* base of maneuverable video display */
    char holder[50];
    unsigned long tmp;

    /* initialize base of video display, and begin */
    setting = oldphys = Physbase();
    appl_init();

    for (;;)
    {
        switch(evtnt_keybd())
        {
            /* move up one row */
            case UP_ARROW:
                /* round to row boundary */
                setting -= 1280;
                if (setting < 0)
                    setting = 0;
                /* reset base of physical display */
                Setscreen(-1L, setting, -1);
                break;

            /* move down one row */
            case DN_ARROW:
                setting += 1280;
                Setscreen(-1L, setting, -1);
                break;

            /* move to absolute address */
            case HELP:
                Setscreen(-1L, oldphys, -1);
                printf("Enter memory setting, decimal: ");
                fflush(stdout);
                tmp = atol(gets(holder));
                setting = (char *)tmp;

                if (setting < 0)
                    setting = 0;
                Setscreen(-1L, setting, -1);
                break;

            /* reset original video base, and exit */
            case RETURN:
                Setscreen(-1L, oldphys, -1);
                appl_exit();
                exit(0);
        }
    }
}
```

```

/* show current video memory base */
case UNDO:
    Setscreen(-1L, oldphys, -1);
    printf("Screen base is %lu\n", setting);
    evnt_keybd();
    Setscreen(-1L, setting, -1);
    break;

default:
    break;
}
}
)

```

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 is designed to be used with programs 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 are 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 suppress leading zeroes. For example, printing 1034 with mask **ZZZ,ZZZ** gives *<sp>* *<sp>* 1,034. Note that the comma is not a special character, but is printed literally.
- 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 all zeroes. For example, printing 070884 with mask **K9/K9/K9** gives 7/8/84.
- \$** Print 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 suppress 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.
- If you place a hyphen to the left of the mask, it is printed at the beginning of the number, but only if it is negative. For example, printing 105 with mask **-Z,ZZZ** yields <sp><sp>105, whereas printing -105 yields <sp><sp>-105.
- (This character acts like the minus sign '-', but prints a '('. For example, printing 105 with mask **(ZZZ)** gives <sp>105<sp>, 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 <sp><sp>5+, 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.

Example

For an example of **picture**, compile the source program **picture.c** with the option **-DTEST**.

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**, which is included with Mark Williams C. Note that **picture** is not included in a library.

pnmatch — String function (libc)

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 one if *pattern* matches *string*, and zero 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 zero or one: zero means that *pattern* must match *string* exactly, whereas one means that *pattern* can match any part of *string*. In the latter case, the wildcards '^' and '\$' can also be used in *pattern*.

Example

For an example of this function, see the entry for **fgets**.

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, Z8001, and on the PDP-11; they are 32 bits long on the i8086, LARGE model, Z8002, the 68000, and the VAX.

Note that failure to declare a function that returns a pointer 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 have the same size; however, 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, portability, pun

pokeb — General function (libc)

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

Notes

pokeb is supplied for use in user-mode programs. Programs that run in supervisor mode, i.e., interrupt handlers, trap handlers, and boot sector programs, should access the memory locations directly with the following macro:

```
#define pokeb(cp,c) (*((char *)cp) = c)
```

pokeb does not work correctly in supervisor mode, which allows you to access memory locations directly.

pokel — General function (libc)

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**.

pokel is supplied for use in user-mode programs. Programs that run in supervisor mode, i.e., interrupt handlers, trap handlers, and boot sector programs, should access the memory locations directly with the following macro:

```
#define pokel(lp,l) (*((long *)lp) = l)
```

pokel does not work correctly in supervisor mode, which allows you to access memory locations directly.

pokew — General function (libc)

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**.

pokew is supplied for use in user-mode programs. Programs that run in supervisor mode, i.e., interrupt handlers, trap handlers, and boot sector programs, should access the memory locations directly with the following macro:

```
#define pokew(wp,w) (*((int *)wp) = w)
```

pokew does not work correctly in supervisor mode, which allows you to access memory locations directly.

port — Definition

A **port** passes data to and receives data from a remote device.

See Also

aux, fclose, FILE, fopen, prn:, stream

portability — Technical information

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 a 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 characters"; 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 character.

4. Always use manifest constants, such as **EOF**, and make full use of **#define** statements.
5. Use header files to hold all machine-dependent declarations and definitions.
6. Declare everything explicitly. In particular, be sure to declare functions as **void** if they do not return a value; this avoids unforeseen problems with undefined return values.
7. Do not assume that integers and pointers have the same size or even the same kind of structure. Do not assume that pointers are all the same or can point anywhere. On the i8086, in **SMALL** model a pointer to a function addresses relative to the code segment, whereas a pointer to data addresses relative to the data segment. On some machines, character pointers are of a different size or structure than word pointers.
8. The constant **NULL** is defined as being different from any valid pointer. Use it and nothing else for that purpose.

See Also

#define, **header file**, **manifest constant**, **pointer**, **pun**, **void**

pow — Mathematics function (libm)

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.
- eck** Reset spacing represented by tab character. On input, expand tab character *c* to positions *k* plus one, two times *k* plus one, three times *k* plus one, etc. The default *c* is '\t'. The default value of *k* is eight.
- ick** Replacing spacing with the tab character. On input, replace spaces with the tab character *c* at positions *k* plus one, two times *k* plus one, three times *k* plus one, etc. The default *c* is '\t'. The default value of *k* is eight.
- ln** 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.
- sc** 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, as well as the header and footer space.
- wn** 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 following table summarizes the precedence of C operators. The are listed in descending order of precedence: those listed higher in the table are executed before those lower in the table. Operators listed on the same line have the same level of precedence.

<i>Operator</i>	<i>Associativity</i>
() [] -> .	Left to right
! ~ ++ -- - (type) * & sizeof	Right to left
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
	Left to right
&&	Left to right
	Left to right
? :	Right to left
= += -= *= /= %=	Right to left
,	Left to right

See Also

operators

The C Programming Language, page 48

printf — **STDIO** function (libc)

Format output

int **printf**(*format*, [*arg1*, ..., *argN*])

char **format*; [**data type**] *arg1*, ..., *argN*;

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 one or more digits gives the *precision*. For floating point (e, f, and g) conversions, the precision sets the number of digits printed after the decimal point. For string (s) conversions, the precision sets the maximum number of characters that can be used from the string. If the precision specification is given as an asterisk '*', the routine uses the next *arg* as an integer that gives 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, or with the draft ANSI standard.

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: <i>d.dddddesdd</i> , where there is always one digit before the decimal point and as many as the <i>precision</i> after it (the default is six). The exponent sign s may be either '+' or '-'. |
| f | Convert the float or double argument to a string 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 <i>precision</i> (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** Print 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 demonstrates many **printf** statements.

```
main()
{
    extern void demo_r();
    int precision = 1;
    int integer = 10;
    float decimal = 2.75;
    double bigdec = 27590.21;
    char letter = 'K';
    char buffer[20];

    strcpy (buffer, "This is a string.\n");

    printf("This is an int:   %d\n", integer);
    printf("This is a float:  %f\n", decimal);
    printf("Another float:   %3.*f\n", precision, decimal);
    printf("This is a double: %lf\n", bigdec);
    printf("This is a char:   %c\n", letter);
    printf("%s", buffer);
    printf("%s\n", "This is also a string.");

    demo_r("Print everything: %d %f %lf %c",
           integer, decimal, bigdec, letter);
    exit(0);
}

void demo_r(string)
char *string;
{
    printf("%r\n", (char **)&string);
}
```

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 evt_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 = evt_multi(which, CLICKS, BUTTON, DOWN,
                               0, norect, 0, 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 '?' modifier.

At present, **printf** does not return a meaningful value.

prn: — Operating system device

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.

See Also

daemon, **file**

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 — **xbios** function 36 (**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**, as follows:

```
struct prtblk {
    char *pb_blkptr;          /* Address of bit block or text string */
    int pb_offset;            /* Bit offset into block */
    int pb_width;             /* Width of area to dump, in pixels */
    int pb_height;           /* Height of block to dump, or zero for text print */
    int pb_left;              /* Pixels to left of block */
    int pb_right;             /* Pixels to right of block:
                             * enclosing bitmap is
                             * pb_right+pb_width+pb_left wide */
    int pb_srcres;            /* Source resolution, ala Getrez() */
    int pb_dstres;           /* Output resolution */
    int *pb_colpal;          /* Color palette, ala Setpalette() [sic] */
    int pb_type;             /* Printer type */
    int pb_port;             /* Printer port */
    int *pb_masks;          /* Halftone dithers */
};
```

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>
struct prtblk pb;

main() {
    int palette[16];
    register int i;
```

```
/* Determine printer characteristics */
i = Setprt(-1);
if (i & PR_DAISS)
    pb.pb_type = PB_DAISS;
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_DAISS) {
    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 = 640;
        pb.pb_height = 200;
        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 should never return.

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. Puns are supported by C'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**, and is handled as such. No trouble will arise if the program is run on a machine that defines an **int** and a pointer to have the same length (e.g., i8086 SMALL model); however, such code cannot be transported to an environment in which this is not the case (e.g., i8086 LARGE model).

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. This function may not do anything when the AES is in ROM.

See Also

TOS, xbios

putc — STDIO macro (stdio.h)

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**. It opens an ASCII file and prints its contents on the screen. For another example of **putc**, see the entry for **getc**.

```
#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, getc, 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

For an example of this routine, see the entry for **getchar**.

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)

Write string to standard output

#include <stdio.h>

void puts(string) char *string

puts appends a newline character to the argument *string* and writes the result on the standard output.

Example

The following uses **puts** to write a string on the screen.

```
#include <stdio.h>

main()
{
    puts("This is a string.\n");
}
```

See Also

fputs, **STDIO**

putw — **STDIO** macro (stdio.h)

Write word to stream

#include <stdio.h>

int putw(word, fp) int word; FILE *fp;

The macro **putw** writes *word* onto the file stream *fp*. It returns the value written.

putw differs from **putc** in that **putw** writes an **int**, whereas **putc** writes a **char** that is promoted to an **int**.

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 genuine end-of-file flag.

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

Q

qsort — General function (libc)

Sort arrays in memory

```
void 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 this function, see the entry for **malloc**.

See Also

shellsort, **strcmp**, **strncmp**

The Art of Computer Programming, vol. 3

Notes

qsort differs from the other sorting function, **shellsort**, in that it uses a recursive algorithm that makes heavy use of the stack.

R

rand — General function (libc)

Generate pseudo-random numbers

```
int rand()
```

rand generates a set of pseudo-random numbers. It returns integers in the range 0 to 32,767, and purportedly has a period of 2^{32} . **rand** will always return the same series of random numbers unless you change its *seed*, or beginning-point, with **srand**.

Example

This example demonstrates 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 data are read is dictated by the order in which they stream past the tape head. Random-access memory (RAM) is an example of random access. Hard disks and floppy disks 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; a more accurate name would be "read/write memory", to contrast RAM 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 numbers**

rc_copy — AES function (libaes)

Copy a rectangle

```
#include <aesbind.h>
```

```
int rc_copy(oldrect, newrect) int oldrect[4], newrect[4];
```

rc_copy is an AES routine that copies a rectangle from one part of the screen to another. *oldrect* and *newrect* hold, respectively, the rectangle being copied and the area to which it is being copied. Each array holds the following information:

<i>rectangle</i> [0]	X point (upper left corner)
<i>rectangle</i> [1]	Y point (upper left corner)
<i>rectangle</i> [2]	width
<i>rectangle</i> [3]	height

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. If you do not, you may inadvertently copy a rectangle over an element in low memory, such as a RAM disk.

rc_equal – AES function (libaes)

Compare two rectangles

#include <aesbind.h>

int rc_equal(rect1, rect2) int rect1[4], rect2[4];

rc_equal is an AES routine that compares two rectangles. *rect1* and *rect2* hold the two rectangles being compared. Each array holds the following information:

<i>rectangle[0]</i>	X point (upper left corner)
<i>rectangle[1]</i>	Y point (upper left corner)
<i>rectangle[2]</i>	width
<i>rectangle[3]</i>	height

rc_equal returns zero if the rectangles are not identical, and one if they are.

See Also
AES, TOS

rc_intersect – AES function (libaes)

Check if two rectangles intersect

#include <aesbind.h>

int rc_intersect(rect1, rect2) int rect1[4], rect2[4];

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 array holds the following information:

<i>rectangle[0]</i>	X point (upper left corner)
<i>rectangle[1]</i>	Y point (upper left corner)
<i>rectangle[2]</i>	width
<i>rectangle[3]</i>	height

The values within the array *rect2* will be changed to the coordinates of the area common to both rectangles, or to meaningless values 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)

Calculate overlap between two rectangles

```
#include <aesbind.h>
```

```
void rc_union(rect1, rect2) int rect1[4], rect2[4];
```

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 array holds the following information:

<i>rectangle</i> [0]	X point (upper left corner)
<i>rectangle</i> [1]	Y point (upper left corner)
<i>rectangle</i> [2]	width
<i>rectangle</i> [3]	height

The values within the array *rect2* will be changed to the coordinates of the rectangle that encloses the overlapping rectangles. These variables are set to meaningless values 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.

rdy — Command

Create, save, and load rebootable RAM disk

```
gem rdy
```

```
rdy [CMD=command FILE=filename ...]
```

rdy is the Mark Williams C utility that creates, saves, and loads a RAM disk. The RAM disk that **rdy** makes has, among others, the following properties:

- The RAM disk will survive system resets. Pressing the reset button on the back of the computer will not cause the RAM disk to be erased.
- The RAM disk can be made the system's boot disk. This allows you to reboot your system and load all desk accessories from the RAM disk.
- The RAM disk can be set to substitute for any physical drive from C through P, and to any size that fits into the memory available on your machine.
- The RAM disk can be copied, contents and all, into an executable file; this file can then be loaded directly into memory. This simplifies the task of recreating the RAM disk after your computer has been turned off.

rdy is designed to work either under **msh** by using a command-line interface, or from the GEM desktop by using a graphics interface.

All source code for **rdy** including its resource files and header file, is included in the archive **rdy.a**. See the entry for the archiver **ar** for information on how to extract the contents of **rdy.a** for alteration and compilation.

How rdy works

rdy goes through the following steps when it builds a RAM disk.

1. It writes a prototype RAM disk and copies it into a file. The size of the RAM disk, its device name (e.g., "C"), whether it should be the boot device, and the name of the file into which the prototype should be copied, are all set either according to variables supplied by the user or, if no such variables are set, according to built-in prototypes.
2. It loads the RAM disk into memory. After it does so, the system automatically warm boots. **rdy** automatically updates various system tables, so that TOS knows that the RAM disk is present, but it is up to the user to install the icon for the RAM disk on the GEM desktop.
3. **rdy** can then be told to back up the installed RAM disk, plus whatever files you have copied into it, into an executable file.
4. If you wish, **rdy** will remove one of its RAM disks. Note that the only certain way to remove a RAM disk is either through **rdy** itself or by cycling power on your computer.
5. Finally, **rdy** can read an installed RAM disk, a back-up file, or a prototype file, and print its parameters on the screen.

To operate **rdy**, you must tell it the *command* that you want it to execute (e.g., create a prototype file or load a RAM disk into memory), and then supply the necessary variables the command needs (e.g., if you are getting information about a prototype file, the name of the file you want **rdy** to read).

As noted above, you can pass this information to **rdy** either through a command-line interface under the shell **msh**, or through a graphics interface directly from the GEM desktop. **rdy** reads the environment and looks for the environmental variable **CMD**; if it is not set, or if it set to **NULL** (which is always the case when running from the GEM desktop), **rdy** reads its associated resource file and runs through the graphics interface; otherwise, it ignores its graphics routines and operates through an ordinary command-line interface. Each interface is described in detail below.

Using the command-line interface

As noted above, **rdy** checks its environment for the environmental variable **CMD**. If **CMD** is found, **rdy** invokes the command-line interface; otherwise, it invokes the graphics interface.

The set of environmental variables that **rdy** uses is as follows:

BOOT boot flag for the RAM disk
CMD choose one of:
 DROP - remove a RAM disk from memory
 HELP - give information on rdy
 LIST - list RAM disks active in memory or saved in file
 LOAD - load a RAM disk into memory from a file
 MAKE - create a new RAM disk file
 SAVE - save a RAM disk from memory into a file
DISK drive identifier of the RAM disk
FILE file name for RAM-disk prototype or backup
ROOT size of the RAM disk root directory, in 512-byte sectors
SIZE size of the RAM disk data area, in kilobytes

CMD and the environmental variables that **rdy** uses can be set either by using the **setenv** command to implant them into the **msh** environment, or by setting them on the **rdy** command line itself. As far as **rdy** is concerned, typing

```
setenv CMD=LOAD
setenv FILE="a:\bin\ramdisk.rdy"
rdy
```

is equivalent to typing

```
rdy CMD=LOAD FILE="a:\bin\ramdisk.rdy"
```

The only difference is the command **setenv** fixes the variables **CMD** and **FILE** within the environment, where they can be read by **rdy** and other programs, whereas passing them on the command line means that they disappear when **rdy** has finished its work.

To build a new RAM disk on your machine, use the following script:

1. Decide how large a RAM disk you want. To perform compiles, a RAM-disk must be at least 100 kilobytes. A 512-kilobyte machine can support a 100- to 200-kilobyte RAM disk, where a 1,024-kilobyte machine can support a RAM disk of up to 512 kilobytes. The rest of this example will demonstrate building a 100-kilobyte RAM disk.
2. Enter the microshell **msh**. Now, type the command:

```
rdy CMD=MAKE DISK=C SIZE=100 FILE="a:\ramdisk.rdy"
```

Change the **SIZE** and **DISK** parameters to suit your preferences. **rdy** will create a description of the RAM disk, and write it into the file **ramdisk.rdy**, and then return you to **msh**.

3. Reinvoke **rdy**, as follows:

```
rdy CMD=LOAD FILE=rdydisk.ram
```

rdy will now load the prototype RAM disk into memory. Note that during the loading process, your system will warm boot and return you to the **GEM**

desktop.

4. Use the desktop's **install** utility to install the RAM disk on the desktop. You may wish to rename the icon, and otherwise modify the desktop. When you have done so, save the desktop settings.
5. Re-enter **msh**. Now, use **cd** to move to the RAM disk. Configure the disk as you wish: create directories and copy files into it that you use often. On a small RAM disk, the MicroEMACS editor performs very well out of the RAM disk; on a large RAM disk, you may wish to move the entire compiler and linker onto it.
6. Now, place a newly formatted disk into drive A, and invoke **rdy** as follows:

```
rdy CMD=SAVE FILE="a:\ramdisk.dta"
```

There is nothing magical about the file name in the above example; you may call the file whatever you wish. **rdy** will copy an image of the entire RAM disk into the file **ramdisk.dta** on drive A. The next time you need to cold boot the system, you can use **rdy** to copy the contents of this file back into the RAM disk; this should save you considerable amounts of time.

Your RAM disk is now ready.

Using the graphics interface

As noted above, if **rdy** does not find the parameter **CMD** defined either in its environment or on its command line, it will automatically invoke its graphics interface. The graphics interface is easier to use than the command-line interface, especially by users unfamiliar with **rdy**, but offers a narrower range of options.

To see how the graphics interface works, type **exit** to leave **msh**, and then click the icon labelled **rdy.prg**. The screen will clear, and in a moment, a new menu bar will appear at the top of the screen. The title at the left of the menu bar, called **Desk**, gives you access to all desk accessories. The title at the right, **Read Me**, describes how **rdy** works. You can use this feature to refresh your memory. The title in the center, **Options**, lets you command **rdy** to perform one or more tasks for you.

If you pass the mouse pointer over the **Options** title, a menu will drop down. This menu has six entries, as follows:

Create a RAM disk

Write a prototype RAM disk into a file named by the user (default, **a:\ramdisk.rdy**).

Load a RAM disk

This loads a prototype or backup RAM disk into memory. Note that the system will warm boot automatically as soon as the disk is installed.

Back up a RAM disk

Copy a RAM disk and its contents, into a file. This file can later be loaded back into memory.

Remove a RAM disk

Erase a RAM disk from memory. Note that the only sure way to remove a RAM disk is either with this command, or by cycling power on your computer. Note that the system will warm boot automatically as soon as the disk is removed.

Get data on a RAM disk

Read a RAM disk, a prototype file, or a backup file, then print information about it on the standard output device.

Quit

Exit from rdy.

To begin, click the first entry, **Create a RAM disk**. A series of dialogues will ask you to describe the RAM disk that you want to build.

The first dialogue box asks you how much RAM your system has, either 512 kilobytes or 1024 kilobytes. Click the appropriate button.

The next dialogue asks the size of the RAM disk you wish to create. Again, click the appropriate button. Note that your RAM disk should be large enough to hold a significant number of files, but not so big that it stops you from loading any program that you use frequently. A good rule of thumb is to use a RAM disk that takes up approximately between one quarter and one half of the RAM on your machine.

The next dialogue asks the name of the drive you wish to call your RAM disk. You should not use a drive that is already taken up by another device, such as a logical partition on your hard disk, or by another RAM disk. If you do so, rdy will not be able to load your RAM disk.

rdy asks the name of the file in which to store the prototype RAM disk. Then, it asks you if you want this RAM disk to be your system's boot disk. When you have answered these questions, rdy displays the configuration of the new RAM disk, and ask you if it is correct. If you answer "No", you will return to the rdy desktop; otherwise, the new prototype RAM disk will be written.

Finally, rdy asks if you wish to load the new RAM disk. If you answer "No", rdy returns you to its desktop. Answer "Yes", which tells rdy to load the new file. Note that as it installs a new RAM disk, rdy warm boots your system. Do not be alarmed when the screen clears and you are returned to the GEM desktop: this indicates that the RAM disk has been loaded successfully.

Now, you should install your new disk on the GEM desktop. To do so, first single-click the icon for one of your existing storage devices; then move the mouse pointer to the **Options** title on the menu bar, and double-click the entry **Install Disk Drive**. Change the name of the drive from its old setting to the name of your RAM disk (e.g., from **A** to **D**), and then type in the name that you want to appear under the icon (e.g., "RAM DISK"). Then click the button labelled **Install**. The

desktop will return with the new icon displayed.

Finally, you should create a new directory (or “folder”, in Atari jargon) named **tmp**. Do so by clicking the “New Folder” entry on the Desktop’s File menu, and following its directions.

Working with a RAM disk

A RAM disk improves the speed with which you work by reducing the amount of time the compiler needs to read a file. Even a small RAM disk will speed your work greatly if it is used properly. For example, the compiler writes a temporary files to pass information between its four phases; writing the temporary files onto the RAM disk eliminates the time taken by writing these files onto a disk and reading them back again. Test compiles have shown that this change alone will reduce the time needed to compile and link a large program by more than half.

To take full advantage of your RAM disk, you will need to tell the Mark Williams microshell **msh** that it exists and how you want it to be used. To do so, you must edit the file **profile**, which **msh** reads when you invoke it; make the following two changes. First, the line that begins **TMPDIR=** indicates where you wish to store temporary files; this line should be changed to the name of your RAM disk. For example, if your RAM disk is named **D**, this line should read as follows:

```
TMPDIR=D:\
```

Then, the line that begins **PATH=** lists for **msh** all the directories where it should look for executable files. Your RAM disk should go near the beginning of that list. For example, this line may read as follows:

```
PATH=.cmd,,a:\bin, b:\bin
```

If your RAM disk is named as drive **D**, change the **PATH** description to the following:

```
PATH=.cmd,d:\,,a:\bin,b:\bin
```

This tells **msh** that the RAM disk should be searched for executable files *before* either of the floppy disk drives; naturally, a RAM disk can be searched much more quickly than a floppy disk drive, which will save you time.

We suggest that you not attempt to alter the **profile** until *after* you have installed Mark Williams C and have read the chapter in the manual that introduces **msh**. If you alter the **profile** too radically without knowing how it works, you may confuse **msh** and create difficulties for yourself.

The bootable RAM disk

As noted above, **rdy** can create a RAM disk that is defined to the system as its boot disk. TOS will look for its boot file in directory **\auto** on that device, and look for its desk accessories in its root directory. The following describes how to use **rdy** in order to take advantage of this and other more advanced features.

1. Create a RAM disk using the steps listed above. Load it into memory and create its icon. Then, double-click the RAM disk's icon to open it. Create two new folders for it, called **tmp** and **auto**. If you are running a hard disk, put the hard disk driver into the **auto** folder; then press the reset button to warm boot. This allows you to access the hard disk throughout the rest of this routine. Then double-click the RAM disk's icon to reopen it.
2. Now, drag into the RAM disk the programs and utilities you want to store there. Some programmers prefer to keep **msh** and MicroEMACS (or another preferred editor) in a folder called **bin** on the RAM disk, because they are used constantly. Be sure to leave enough room on the RAM disk to hold the compiler's temporary files.
3. Save the desktop and copy the file **desktop.inf** onto the RAM disk if it was not written there. Note that if you have a hard disk with drive C, TOS will insist on writing **desktop.inf** there.
4. The next step is to test the RAM disk by warm booting. Press the reset button. The floppy disk drive should run for a second as the system looks for a boot block; then, the programs in the RAM disk's **auto** folder will run. Finally, the desktop you saved should appear as the desktop initializes.

If anything is missing or wrong, go back, fix it, and test again until everything is right.

5. The next step is to back up your configured RAM disk. Put a blank floppy disk into drive A. Format it with the volume name **coldstrt.dsk**, copy the files **rdy.prg** and **rdy.rsc** onto it, and make an **auto** folder on it.

Then click **rdy.prg**, select **Back up a RAM disk**, and follow its directions to save the contents of the loaded RAM disk into file **a:\auto\coldstrt.prg**. Note that a backed-up RAM disk is an executable file in its own right; you do not need to invoke **rdy** to load it into memory.

6. Now that the back up is finished, you have a disk from which you can cold start your system easily. To test whether all will work correctly, turn off your computer for a few moments, then turn it on again. Drive A should be selected for several seconds while TOS loads the program **coldstrt.prg** into memory. The screen will flash as the RAM disk warm boots to install itself. Then, the programs installed in the **auto** folder of the RAM disk will run and take effect. If you installed a hard disk driver, you should see the hard disk initialize. Finally, the desktop that you saved on the RAM disk will be displayed as the desktop initializes.

The screen should not flash after the initial reset, but it often does. This could be due to any number of reasons. The most easily fixed is a loose cable: make sure that the cables that plug into the back of your ST are pressed all the way into their sockets.

You now have a RAM disk that you can use either to warm boot or cold boot your system. When ever you need to cold boot your system, simply place the boot disk you just created into floppy disk drive A, turn on the computer, let it boot, then put the boot disk away.

When an error occurs, you have reason to believe that TOS or the AES has corrupted itself, or a program enters an infinite loop, just push the reset button. The machine will reboot off the RAM disk in a few seconds, and all the contents of the RAM disk will be exactly as you left them.

Problems can occur from a number of sources. If a program had a file open on the RAM disk when you reset the system, the file may not have been completely written to the RAM disk. Removing the file name may not recover all the clusters allocated to the file. These lost clusters may become a problem if you continue to reset out of the program, because the RAM disk will eventually run out of data clusters for files. This can be fixed by saving your work to a floppy or hard disk, removing the RAM disk, and reloading it from your cold boot disk.

The RAM disk occupies high memory, beyond the value of the system variable **phystop**, which is normally 32 kilobytes past the start of the video display. If a program writes into this memory, it will destroy your RAM disk and all its contents will be lost. This is most easily done by not clipping graphics to the screen correctly: then, if you scribble past the border of the screen, the virtual image you create will overwrite the RAM disk.

You probably will want to update your cold boot RAM disk from time to time. If you are replacing an old version of a file with a new version, you can simply copy the new version in and replace the old back-up RAM disk file. However, if you are changing the structure or contents of the RAM disk, then do it in the following order.

First, remove the old files from your loaded RAM disk. Remove the old folders from your loaded ramdisk. Create the new folders on your loaded RAM disk; then copy the new files onto your loaded RAM disk.

Then, place **coldstrt.dsk** into the floppy drive, run **rdy.prg**, and save the RAM disk image to **a:\auto\coldstrt.prg**. If you change the contents of your RAM disk in another order, you may get a much larger RAM disk image than necessary.

A saved RAM disk contains all the data clusters up to the last one allocated. You can eliminate fragmentation of your RAM disk data clusters by simply copying all the files and folders from your RAM disk onto a floppy disk, deleting all the files and folders from your RAM disk, and copying the files and folders back onto the RAM disk from the floppy disk.

You can also use **rdy.prg** from a shell command file; for example:

```
setenv CMD=SAVE FILE=a:\auto\coldstrt.prg DISK=c; rdy
```

You may wish to store this command in a file on your cold boot disk. Be sure to set the variable **DISK** to the correct drive identifier for your RAM disk!

Finally, enjoy the speed and convenience of your new RAM disk. You may wish to spend some of this time studying the sources for **rdy**; they are stored in the archive file **rdy.a**. See the entry for the archiver **ar** for information on how to extract files from the archive.

See Also
commands, TOS

Notes

The Supra hard disk autoboot can sometimes interfere with RAM disks built with **rdy**. To use RAM disks with the Supra hard disk autoboot, do the following:

1. Create a new RAM disk of the desired size and drive; this must not conflict with a hard-drive identifier. Make it non-bootable. Give this file a name like **ramXXXD.prg**, where **XXX** is the size of the RAM disk, in kilobytes, and **D** is the letter of the RAM disk's identifier.
2. If you already have an **\auto** fold on partition C, move all of its files into another folder, and delete the **\auto** folder.
3. Create a new **\auto** folder.
4. Put the RAM disk into the new **\auto** folder *first*.
5. Move all of the files that were moved out of the **\auto** folder into the new **\auto** folder.

When the system boots for the first time after a power-up, the RAM disk will load and then cause the system to warm-boot. On warm-boot, the RAM disk loader portion of the RAM disk file sees that a drive is already loaded with the given drive specifier and exits without loading the RAM disk again and without rebooting the system. The rest of the programs in the **\auto** folder are then executed.

rdy.a — Archive

rdy.a is an archive that holds the source files for **rdy**, the Mark Williams utility that creates rebootable RAM disks on the Atari ST.

If you wish to recompile **rdy**, 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 rdy.a
```

See Also
ar, rdy

read — UNIX system call (libc)

Read from a file

int read(*fd*, *buffer*, *n*) int *fd*; char **buffer*; int *n*;

read reads up to *n* bytes of data from the file descriptor *fd* and writes them into *buffer*. The amount of data actually read may be less than that requested if **read** detects EOF. The data are read beginning 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 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, such as bad file descriptor, bad *buffer* address, or physical read error.

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**.

readonly — C keyword

Storage class

readonly is a C keyword that modifies data declarations. As its name implies, the **readonly** modifier declares that data are to be read only; this helps protect key data against casual modification by the user or another programmer.

See Also

C keywords, C language, keyword

Notes

The draft ANSI standard for the C language eliminates this keyword.

read-only memory — Definition

As its name suggests, **read-only memory**, or ROM, is memory that can be read but not overwritten. 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

realloc — General function (libc)

Reallocate dynamic memory

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

realloc helps you 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*.

Example

For an example of this function, see the entry for **calloc**.

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.

real number — Definition

A **real number** is any number of the set of rational numbers or irrational numbers.

See Also

float, rational number, integer, irrational number

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 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

register — C keyword

Storage class

register is a C keyword that declares a class of data storage. A variable so declared will be stored in a register, which may increase the speed with which it is read by a program.

See also

auto, C keywords, C language, extern, register variable, static

register — Definition

A **register** is special high-speed memory within a microprocessor that can be addressed concisely and 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.

The routines in the Mark Williams C libraries generally assume that they have been called from C programs; thus, they may freely overwrite any registers that the compiler overwrites in its generated code.

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

rescomp — Command

Resource compiler

rescomp [-v] [-o outfile] infile[.ext]

The command **rescomp** compiles *infile*, which must be a file of resource-description text into a GEM resource.

The option **-v** tells **rescomp** to report statistics as it compiles.

The option **-o** changes the name of the three files it produces to *outfile*. By default, **rescomp** names its output files after *infile*. When the compiler creates these files, it gives them the extensions **.rsc** for the resource file, **.rsd** for the compiled resource description, and **.h** for the C header file.

See the section in the introduction on the **Resource compiler and decompiler** for a summary of the resource description language.

See Also

resdecom, **resource**

resdecom — Command

Resource decompiler

resdecom [-o *outfile*].**ext**]] [-d *defile*].**ext**]] [-mv] [-] *infile*].**ext**]

The command **resdecom** decompiles a GEM resource into a file of resource-description language. This can be useful when you want to change a line of text within a resource, globally change a string which appears in more than one place within a resource, or if you want to create a simple resource without using the Resource Editor. It is easy to track changes between versions of your resource by comparing decompiled resource files.

Decompiled resources often take much more room than their compiled counterparts. The text descriptions of some objects are more compact, but images, icons, buttons and compound objects take up more space.

To decompile an existing resource set into a resource description file, use the program **resdecom.prg**. Its options are as follows:

-d *defile*].**ext**]

Specify the name of the definition file.

-o *outfile*].**ext**]

Rename the output file that the decompiler creates. The default is the name of the resource you are decompiling, with the extension **.rdl**.

-m Force menus to be treated as forms.

-v Verbose option: decompile with messages.

- Send output to the standard output.

resdecom looks for two files that are labeled with the extensions **.rsc** and **.rsd**. It reads them and writes a resource description file that takes the names of the resource files, adding the extension **.rdl**.

The file produced by **resdecom** can be edited with MicroEMACS or most other text editors. You can then recompile it into a resource by using the resource com-

piler **rescomp**.

See Also

rescomp, **resource**

resource — Command

Invoke the resource editor
resource

The command **resource** invokes the Mark Williams Resource Editor. A resource editor simplifies the creation of icons, menus, dialogue boxes, forms, and alerts. In general, it helps you to design and implement graphics interfaces for your programs.

resource encodes objects that you display and manipulate on the editor's desktop. With **resource**, you can move objects around the screen, and edit each until it is as you want it to appear with your GEM application program. It then fills in the X, Y, height, and width coordinates, and records the relative position of each object within its object tree. **resource** also allows you to name each object. It then produces a header file that contains the names and their "handles," so you can refer to each object easily from within your program.

In addition to the C header file, **resource** produces two other files that contain information that your application program will use to reproduce the interface you have created. One file, with the suffix **.rsc**, is the resource file called by your application program. The other is a "name and type" definition file with the suffix **.rsd**. This definition file is used only by **resource** and by the resource decompiler **resdecom**. It is not used by the application program.

To use the editor, you must have an Atari ST system with TOS in ROM, at least one disk drive, a monochrome or color monitor in medium or high resolution, and **Mark Williams C for the Atari ST**.

The **Mark Williams Resource Editor** is designed to work in medium or high resolution. Many of the dialogues in the Resource Editor contain large amounts of information and will not work correctly in low resolution.

The editor also has the following limitations:

- The structure of a resource file limits it to 64 kilobytes.
- A text string cannot be longer than 65 bytes.
- The colors in an object are limited to white, black, red, and green, and the thickness of its border to four rasters (inside and out).

To run **resource**, you must install the files **resource.prg** and **resource.rsc** into the same directory; the directory should be one of those named in the environmental variable **PATH**.

To run the Resource Editor from **msh**, the Mark Williams micro-shell, type:

resource

at the prompt. If you want to invoke the Resource Editor from the GEM desktop, double-click the mouse on **resource.prg**.

See Also

rescom, **resdecom**, **object**, **menu**

return — C keyword

Return a value and control to calling function

return is a C statement that returns a value from a function to the function that called it. **return** can be used without a value, to return control of the program to the calling function; also, the calling function is free to ignore the value **return** hands it. Note that it is good programming practice to declare functions that return nothing to be of type **void**.

Note that a function can return only one value to the function that called it. Most often, this value is used to signal whether the function performed successfully or not.

See Also

C keywords, **C language**

The C Programming Language, page 68

rewind — STDIO function (libc)

Reset file pointer

#include <stdio.h>

int rewind(*fp*) FILE **fp*;

rewind resets the file pointer to the beginning of stream *fp*. It is a synonym for **fseek(*fp*, 0L, 0)**.

Example

For an example of this routine, see the entry for **fscanf**.

See Also

fseek, **STDIO**

Diagnostics

rewind returns EOF if an error occurs; otherwise, it returns zero.

rindex — String function (libc)

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 uses **rindex** to help strip a sample file name of the path information.

```
#include <stdio.h>
#define PATHSEP '\\'          /* path name separator */
extern char *rindex();
extern char *basename();

main()
{
    char *testpath = "A:\\foo\\bar\\baz";

    printf("Before massaging: %s\n", testpath);
    printf("After massaging: %s\n", basename(testpath));
}

char *basename(path)
char *path;
{
    char *cp;
    return (((cp = rindex(path, PATHSEP)) == NULL)
        ? path : ++cp);
}
```

See Also

index, **memchr**, **string**, **strrchr**

Notes

This function is identical to the function **strrchr**, which is described in the ANSI standard. Mark Williams C includes **strrchr** in its libraries. It is recommended that it be used instead of **rindex** so that programs more closely approach strict conformity with the ANSI standard.

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 directories

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 — xbios function 15 (osbind.h)

Configure the serial port

```
#include <osbind.h>
```

```
#include <xbios.h>
```

```
long Rsconf(speed, flow, UCR, RSR, TSR, SCR)
```

```
int speed, flow, UCR, RSR, TSR, SCR;
```

Rsconf configures the serial port. *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 a 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.

Rsconf returns a **long** that holds the old *UCR*, *RSR*, *TSR*, and *SCR*, in that order.

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\n");
}
```

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.

rsconf — Command

Configure 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, 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 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, **RSR**, **TSR**, and **SCR** set, respectively, the control register, the receive status, the transmission status, and the synchronous character register. See **Rsconf** for more information on the values to which these arguments can be set. Setting each to -1 will cause them to be ignored by TOS.

See Also

commands, **Rsconf**, **TOS**

rsrc_free — AES function (libaes)

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, you should use this routine before loading a second resource file. **rsrc_free** 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)

Get the address of a resource object

```
#include <aesbind.h>
```

```
int rsrc_gaddr(type, index, address) int type, index; OBJECT **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 resource file. *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)

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)

Change the form of an object's coordinates

#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, because only then is the resolution of the screen 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.

Example

For an example of this function, see **menu**.

See Also

AES, TOS

rsrc_saddr — AES function (libaes)

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 — Overview

The C runtime startup is a routine that is linked with a C program as the first part of an executable program. It performs the functions needed to start and terminate the C environment. To begin the program, it initializes the stack and calls **main**; to conclude the program, it calls **exit** with the return value from **main**.

Three C runtime startup routines are available on Mark Williams C for the Atari ST: **crt0.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 **crt0.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 **Ncrtsg.o**. The **crtsd.o** start-up routine can be called with the option **-VGEMACC** or with the **name** option **Ncrtsd.o**.

See Also

calling conventions, cc, crts0.o, crttd.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.

Unlike an lvalue, an rvalue can be either a variable or a constant.

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(r_or_w, buffer, n, rec, drive) int r_or_w, n, rec, drive; char *buffer;
```

Rwabs reads from or writes data to a disk drive. *r_or_w* indicates the task to perform, as follows:

0	read
1	write
2	read, no medium change
3	write, no medium change

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. If *buffer* is set to zero, then the status set in the argument *r_or_w* is used to set the drive's medium change status.

Rwabs returns zero if all went well, and a number less than zero if an error occurred.

See Also

bios, TOS

S

sbrk — General function (libc)

Increase a program's data space

char *sbrk(increment)

unsigned int increment;

sbrk increases a program's data space by *increment* bytes. It increments the variable **p_hitpa** of the base page, which points to the end of the program's data space. See **basepage.h** for more information on **p_hitpa**. Note that the memory allocation routine **malloc** calls **sbrk** should you attempt to allocate more space than is available in the program's data space.

sbrk returns a pointer to the previous setting of **p_hitpa** if the requested memory is available, or **((char *)-1)** if it is not.

See Also

basepage.h, **malloc**, **maxmem**

Notes

sbrk will not increase the size of the program data area if the physical memory requested exceeds the physical memory allocated by TOS, or if the requested memory exceeds the limit set in the user-defined variable **maxmem**. **sbrk** does not keep track of how space is used; therefore, memory seized with **sbrk** cannot be freed. *Caveat utilitor.*

scanf — STDIO function (libc)

Accept and format input

#include <stdio.h>

int scanf(format, arg1, ... argN)

char *format; [data type] *arg1, ... *argN;

scanf reads the standard input, and uses the string *format* to specify a format for each *arg1* through *argN*, 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.

scanf reads the standard input until the return key is pressed. Inappropriate characters are thrown away; e.g., it will not try to write an alphabetic character into an **int**.

The following modifiers can be used within the conversion string:

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 ***.

It is important to remember that **scanf** reads up, but not through, the newline character; the newline remains in the standard input device's buffer until you dispose of it somehow. Programmers have been known to forget to empty out the buffer before calling **scanf** a second time, which leads to unexpected results.

Example

The following example uses **scanf** in a brief dialogue with the user.

```
#include <stdio.h>

main()
{
    int left, right;

    printf("No. of fingers on your left hand: ");
    fflush(stdout);
    scanf("%d", &left);
    while(getchar() != '\n'); /* eat newline char */

    printf("No. of fingers on your right hand: ");
    fflush(stdout);
    scanf("%d", &right);
    while(getchar() != '\n');

    printf("You've %d left fingers, %d right, & %d total\n",
           left, right, left+right);
}
```

See Also

fscanf, **sscanf**, **STDIO**

The C Programming Language, page 147

Diagnostics

scanf returns the number of arguments filled. It returns EOF if no arguments can be filled or if an error occurs.

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 are 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.

It is not recommended that **scanf** be used to obtain a string from the keyboard: use **gets** to obtain the string, and **sscanf** to format it.

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 — Technical information

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	<i>row col</i> Position cursor at <i>row</i> , <i>col</i> , which are row/column numbers plus 040 (space character)
<esc>bc	Set foreground color to <i>c</i>
<esc>cc	Set background color to <i>c</i>
<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)

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)

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 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.

A second internal directory, **.cmd**, set aside for the user to install functions with the **set** command. For example, the command

```
set in .cmd off="cursconf 3"
```

installs the command **off** into **.cmd**, 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)

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 file stream *fp* and the *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 contains at least **BUFSIZ** bytes.

See Also

STDIO

setcol — Command

Reset a color

setcol entry, color

setcol is a command that uses the **xbios** function **Setcolor** to reset a color. *entry* is the entry in the color palette that you wish to reset, from zero through 15. *color* is the three-digit number that indicates the color to which you wish to set *entry*.

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.

```
#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)

Perform non-local goto

#include <setjmp.h>

int setjmp(env) jmp_buf env;

The function call is the only mechanism that C provides to transfer control between functions. This mechanism, however, 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

Define **setjmp()** and **longjmp()**

#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 *entry1* ... *entry16*

setpal is a command that uses the **xbios** function **Setpalette** (*sic*) to reset the system's color palette. The arguments *entry1* through *entry16* each is a three-digit number that specifies the color code for the corresponding entry in the Atari color palette. If fewer than 16 arguments are given, only that many entries in the palette will be changed.

To alter a specific entry in the color palette, use the command **setcol**.

See Also

commands, getpal, setcol, Setpalette, TOS

Setpalette — xbios function 6 (osbind.h)

Set the screen's color palette

```
#include <osbind.h>
```

```
#include <xbios.h>
```

```
void Setpalette(palette) int palette[16];
```

Setpalette (*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() {  
    Setpalette( 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. It can be used to display any part of the ST's memory as a bit map. *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 changing from a color resolution to a monochrome resolution will warm start the machine and put you back to a correct resolution for your monitor. Changing from low to medium resolution, or vice versa, will create a distorted image that can be corrected with, respectively, the commands **ltom** and **mtol**.

See Also

commands, **getrez**, **Getrez**, **TOS**

Notes

If you enter **msh** or run a GEM program without restoring the resolution, unpredictable results will be evident.

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 | low resolution |
| 1 | medium resolution |
| 2 | high resolution |

Setting any variable to a negative number ensures that that variable will be ignored.

Example

This example demonstrates **Setscreen**. For another example, see the entry for **Physbase**.

```
#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

Notes

If you change the resolution of the screen with this routine, the screen will be cleared. Under some circumstances, the previous screen base will also be cleared. Therefore, it is best to switch screen bases and then change resolutions, rather than doing both with one call.

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.

Sgettextime — Time function (libc)

Read time from intelligent keyboard's clock

```
#include <time.h>
```

```
tm *Sgettextime();
```

Sgettextime is a function that reads the time from the intelligent keyboard's clock. This clock is maintained apart from the other clocks on the Atari ST. **Sgettextime** returns a pointer to the structure **tm**, which it initializes. **tm** is defined in the header file **time.h**. For more information about it, see the entry for **time**.

See Also

Kgettextime, Ssettime, time (overview), time.h, tm

Notes

Unlike the function **Gettime**, which deals in two-second increments, **Sgettextime** allows the programmer to work with clock ticks.

Unlike the related function **Kgettextime**, **Sgettextime** works on the Mega ST.

shelEnvrn — AES function (libaes)

Search for an environmental variable

```
#include <aesbind.h>
```

```
int shelEnvrn(parameter, name) char *parameter, *name;
```

shelEnvrn is an AES routine that searches for a particular environmental variable in the desktop's environment. *name* points to the name of the variable 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. **shelEnvrn** al-

ways returns one.

Example

The following example uses the `shel` library to exchange information with the environment.

```
#include <aesbind.h>

alertf(n, p) int n; char *p;
{
    static char buffer[512];
    sprintf(buffer, "%r", &p);
    return form_alert(n, buffer);
}

main()
{
    char *cp;
    char cmd[128], tail[128];
    int retval;

    appl_init();

    retval = shel_envrn(&cp, "PATH=");
    alertf(1, "[0][shel_envrn |returns |%d |Ok]", retval);
    alertf(1, "[0][PATH= is |%s |Ok]", cp);

    retval = shel_read(cmd, tail);
    alertf(1, "[0][shel_read |returns |%d |Ok]", retval);
    alertf(1, "[0][command is |%s |Ok]", cmd);
    alertf(1, "[0][tail is |%s |Ok]", tail);

    retval = appl_find("SHEL");
    alertf(1, "[0][appl_find SHEL is |%d |Ok]", retval);

    retval = appl_find("shel");
    alertf(1, "[0][appl_find shel is |%d |Ok]", retval);

    retval = appl_find("MSH");
    alertf(1, "[0][appl_find MSH is |%d |Ok]", retval);

    if (alertf(1, "[2][ invoke shel ][No|Yes]") == 2) {
        retval = shel_write(1, 1, 1, "shel.prg", "sock it to me");
        alertf(1, "[0][ shel_write | returns | %d |Ok]", retval);
    }

    appl_exit();
    return 0;
}
```

See Also

AES, TOS

Notes

`shel_envrn` can find a variable only in the desktop environment, *not* the environment of the current process. Due to the design of the AES, it can return only that part of the environment which fits into a small buffer. The sixth character of the

desktop environment is always set to ‘;’ by the desktop because the **PATH** environment passed by the ROM always has a null character in that position.

shel_find — AES function (libaes)

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)

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.

Example

For an example of this function, see **shel_envrn**.

See Also

AES, **TOS**

Notes

Even after a command and a tail is passed successfully through **shel_read**, **shel_write** returns two copies of the command, instead of the command and its tail.

shel_write — AES function (libaes)

Tell desktop which application to run next

```
#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. In GEM terms, it combines **Pterm** and optionally **Pexec**. In UNIX terms, it combines **exit** and optionally **exec**. In effect, it tells the desktop to continue.

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.

Example

For an example of this function, see **shel_envrn**.

See Also

AES, TOS

shellsort — General function (libc)

Sort arrays in memory

```
void shellsort(data, n, size, comp)
```

```
char *data; int n, size; int (*comp)();
```

shellsort is a generalized algorithm for sorting arrays of data in primary memory. It 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. 84ff, 114ff

Notes

shellsort differs from the sort function **qsort** in that it uses an iterative algorithm that does not require much stack.

short — C keyword

Data type

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** equals 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

C keywords, C language, data format, data type, declarations

show — Command

Display a stored screen image

show *screenfile*

show displays a screen image that has been stored either with the command **snap**, or with one of several graphics editors. *screenfile* is the name of the file in which the screen image is stored. *screenfile* can be in any of the following formats, as indicated by its suffix:

.PI1	DEGAS uncompressed screen images, low resolution
.PI2	DEGAS uncompressed screen images, medium resolution
.PI3	DEGAS uncompressed screen images, high resolution
.NEO	Neochrome uncompressed screen images
.MUR	COLR editor screen murals
.PIC	Atari Logo SAVEPIC files

show checks the size of each file to confirm that it is of the correct type; if it is of the wrong size for its type, **show** exits silently.

Note that you may need to alter the image's resolution to resolve it on your current device. This can be done with the battery of commands **htom**, **ltom**, **mtoh**, and **mtol**. For example, to display the low-resolution DEGAS file **foo.pi1** on a high-resolution monitor, use the following command line:

```
show foo.pi1 ; ltom ; mtoh
```

show can also be used with the command **snap** to convert an image from one format to another. For example, to convert the high-resolution DEGAS file **foo.pi3** to Neochrome format, use the following command line:

```
show foo.pi1 ; htom ; mtol ; snap foo.neo
```

See Also

commands, htom, ltom, mtoh, mtol, 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, mousehidden, TOS

signal.h — Header file

Define Atari ST signals

#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)

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)

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 *file*, which must be a relocatable object module. The total size is given in decimal, and the size of each segment is given in both decimal and hexadecimal. All sizes are in bytes.

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.

size prints out the size of each segment, as follows:

c	code
d	data
e	extra
s	stack
a1	auxiliary 1
a2	auxiliary 2
a3	auxiliary 3
a4	auxiliary 4

The suffix **x** (for extension) on a segment identifier indicates the difference between the initialized size in the file and the minimum size in memory. For programs compiled under Mark Williams C, this only appears as **dx** and indicates the size of the arena.

See Also

cc, commands, cpp, nm, strip

Notes

Because version 3.0 changes the object format, the edition of **size** shipped with version 3.0 does not work with objects compiled with Mark Williams C version 2.1.7 or earlier. To convert such objects to a format that **size** recognizes, use the command **mwtomw**.

sizeof — C keyword

Return size of a data element

sizeof is a C operator that returns a constant **int** that is the size of any given data element. The element examined can be a data object, a portion of a data object, or a type cast. **sizeof** returns the size of the element in **chars**; for example

```
long foo;
sizeof(foo);
```

returns four, because a **long** is as long as four **chars**.

Note that **sizeof** is especially useful in **malloc** routines, and when you need 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 of your code.

See Also

C keywords, C language, 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 *scrfile*

snap takes a “snapshot” of the screen’s image, and writes it into *scrfile*. **snap** stores a screen image in any of the following formats, as indicated by the suffix to *scrfile*:

.PI1	Degas uncompressed screen images, low resolution
.PI2	Degas uncompressed screen images, medium resolution
.PI3	Degas uncompressed screen images, high resolution
.NEO	Neochrome images
.MUR	COLR editor screen murals
.PIC	Atari Logo SAVEPIC files

For example, typing

```
snap foo.mur
```

saves the current screen image into file **foo.mur**. It can then be redisplayed with the **show** command.

See Also

commands, show, TOS

sort — Command

Sort lines of text

sort [-bcdfinru] [-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 (octal 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.

-tc 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

sprintf — **STDIO** function (libc)

Format output

```
#include <stdio.h>
```

```
int 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**.

Example

For an example of this function, see the entry for **sscanf**.

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.

At present, **sprintf** does not return a meaningful value.

sqrt — **Mathematics** function (libm)

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

When a domain error occurs (i.e., when *z* is negative), **sqrt** sets **errno** to **EDOM** and returns zero.

rand — General function (libc)

Seed random number generator

void rand(seed) int seed;

rand uses *seed* to initialize the sequence of pseudo-random numbers returned by **rand**. Different values of *seed* initialize different sequences.

Example

For an example of this function, see the entry for **rand**.

See Also

rand

The Art of Computer Programming, vol. 2

sscanf — STDIO function (libc)

Format input

#include <stdio.h>

int 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**.

Example

This example uses **sprintf** to create a string, and then reads it with **sscanf**. It also illustrates a common problem with this routine.

```
#include <stdio.h>
```

```
main()
```

```
{
```

```
    char string[80];
```

```
    char s1[10], s2[10];
```

```
    sprintf(string, "123456789012345678901234567890");
```

```
    sscanf(string, "%9c", s1);
```

```
    sscanf(string, "%10c", s2);
```

```
    printf("27s is the string\n", string);
```

```
    printf("%s: first 9 characters in string\n", s1);
```

```
    printf("%s\n", s2);
```

```
    printf("comes from not leaving space for terminator\n");
```

```
}
```

*See Also***fscanf, scanf, STDIO***The C Programming Language*, page 150*Diagnostics*

sscanf returns the number of arguments filled. It returns zero if no arguments can be filled or if an error occurs.

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 are in the correct data format, such as data that were written 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 = 16000L;
```

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

`\auto, _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 error — Definition

The **standard error** is the peripheral device or file where programs write error messages by default. It is defined in the header file **stdio.h** under the abbreviation **stderr**, and by default is the computer's monitor.

See Also

freopen, header file, msh, standard input, standard output, stdio.h

standard input — Definition

The **standard input** is the device or file 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 the operating system, a shell, or **freopen**.

See Also

freopen, header file, msh, standard error, standard output, stdio.h

standard output — Definition

The **standard output** is the device or file where programs write output by default. It is defined in the header file **stdio.h** under the abbreviation **stdout**, and in most instances is defined to be the computer's monitor.

See Also

freopen, header file, msh, standard error, standard input, stdio.h

stat — General function (libc)

Find file attributes

```
#include <stat.h>
```

```
int stat(file, statptr)
```

```
char *file; struct stat *statptr;
```

stat returns a structure that contains the GEM-DOS attributes of a file. This function is included to maintain compatibility with the UNIX and COHERENT operating systems.

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**:

```
struct stat {
    short st_dev;           /* set only under COHERENT */
    short st_ino;           /* set only under COHERENT */
    short st_mode;          /* attributes */
    short st_nlink;         /* set only under COHERENT */
    short st_uid;           /* set only under COHERENT */
    short st_gid;           /* set only under COHERENT */
    short st_rdev;          /* set only under COHERENT */
    long st_size;           /* file size (in bytes) */
    time_t st_atime;        /* time last accessed */
    time_t st_mtime;        /* time last modified */
    time_t st_ctime;        /* time created */
};
```

The following summarizes the legal settings for **st_mode**, which sets the file's attributes:

S_IJRON	0x01	read-only file
S_IJHID	0x02	hidden from search
S_IJSYS	0x04	system, hidden from search
S_IJVOL	0x08	volume label in first 11 bytes
S_IJDIR	0x10	directory
S_IJWAC	0x20	written to and closed

The entry **st_size** gives the size of the file, in bytes.

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.

See Also

fstat, ls, msh, open, stat.h

Diagnostics

stat returns -1 if an error occurs, e.g., the file cannot be found. Otherwise, it returns zero.

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

header file, stat

static — C keyword

Declare storage class

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 used within a function, it can be accessed only by that function; when used outside a function, it can be accessed only by functions that are defined within the same source file as the 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 overwriting 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, C keywords, C language, extern, register variable, storage class

The C Programming Language, page 80

stderr — Definition

stderr is an abbreviation for *standard error*. It is defined in the header file **stdio.h**.

See Also

stdin, stdio.h, stdout, standard error

stdin — Definition

stdin is an abbreviation for *standard input*. It is defined in the header file **stdio.h**.

See Also

standard input, stderr, stdio.h, stdout

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 file stream
fdopen	open a file stream for I/O
feof	discover a file stream's status
ferror	discover a file stream's status
fflush	flush an output buffer
fgetc	get a character
fgets	get a string
fgetw	get a word
fileno	get a file descriptor
fopen	open a file stream
fprintf	format and print to a file stream
fputc	output a character
fputs	output a string
fputw	output a word
fread	read a file stream
freopen	open a file stream
fscanf	format and read from a file stream
fseek	seek in a file stream
ftell	return file pointer position
fwrite	write to a file stream
getc	get a character
getchar	get a character
gets	get a string
getw	get a word
printf	print a formatted string
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 file-stream buffers

sprintf	format and print to a string
sscanf	format and read from a string
ungetc	return character to file stream

STDIO routines are buffered by default.

See Also

buffer, FILE, Lexicon, stdio.h, stream, UNIX routines

The C Programming Language, page 166

stdio.h — Header file

Declarations and definitions for I/O

stdio.h is a header file that defines several manifest constants used in standard 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**.

Example

For an example of how to redirect **stdout** from within a program, see the entry for **system**.

See Also

standard output, stderr, stdin, stdio.h

stime — Time function (libc)

Set the operating system time

#include <time.h>

int stime(timep) time_t *timep;

stime sets the operating 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 **long**.

Example

For an example of using this function from the **\auto** directory, see the entry for **\auto**.

Example

The following example prints the time, then uses **stime** to reset the time by one hour.

```
#include <time.h>
main()
{
    long nowhere; /* buffer to put unwanted things */

    /* print current time */
    printf("%s\n", ctime);

    /* subtract one hour (3600 seconds) from current time */
    if (stime((time(&nowhere) - 3600)) == -1)
    {
        printf("Cannot reset time.\n");
        exit(1);
    }

    /* print altered time */
    printf("%s\n", ctime);

    /* add one hour to current time, to correct above */
    if (stime((time(&nowhere) + 3600)) == -1)
    {
        printf("Cannot re-reset time.\n");
        exit(1);
    }

    /* print fixed time, to confirm correction */
    printf("%s\n", ctime);
}
```

See Also

date, time (overview)

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 — Technical information

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)

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

For an example of this function, see the entry for **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 point to enough space to hold itself and *string2*; otherwise, another portion of the program or operating system may be overwritten.

strchr — String function (libc)

Find a character in a string

char *strchr(string, character);

char *string; int character;

strchr searches for *character* within *string*. The null character at the end of *string* is included within the search.

strchr returns a pointer to the first occurrence of *character* within *string*. If *character* is not found, it returns NULL.

See Also

memchr, strcspn, string, strpbrk, strrchr, strspn, strstr, strtok

Notes

This is equivalent to the function **index**, which is also included with Mark Williams C.

strcmp — String function (libc)

Compare two strings

int strcmp(string1, string2) char *string1, *string2;

strcmp compares *string1* with *string2* lexicographically. It returns zero if the strings are identical, returns a number less than zero if *string1* occurs earlier alphabetically than *string2*, and returns a number greater than zero if it occurs later. This routine is compatible with the ordering routine needed by **qsort**.

Example

For examples of this function, see the entries for **string** and **malloc**.

See Also

memcmp, qsort, shellsort, string, strncmp, strcspn, strspn, strstr

The C Programming Language, page 101

strcpy — String function (libc)

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*.

Example

See **string**. For an example of using this function in a TOS application, see the entry for **Fgetdta**.

See Also

memcpy, string, strncpy

The C Programming Language, page 100

Notes

string1 must point to enough space to hold *string2*, or another portion of the program or operating system may be overwritten.

strcspn — String function (libc)

Length one string excludes characters in another

unsigned int strcspn(string1, string2)

char *string1, *string2;

strcspn compares *string1* with the characters in *string2*. It then returns the length, in characters, for which *string1* consists of characters *not* found in *string2*.

See Also

memchr, strchr, string, strpbrk, strrchr, strspn, strstr, strtok

stream — Definition

The term **stream** applies to any entity that can be named and from which bits can flow, such as a device or a file. The name “stream” reflects the fact that the C programming environment does not depend upon record descriptors and other devices that predetermine what form data can assume; rather data, from whatever source, are seen to be a flow of bytes whose significance is set entirely by the program that reads them.

For example, whether 16 bits forms an **int**, two **chars**, and should be used as an absolute value or a bit map, is entirely up to the program that receives it. It is also irrelevant to the program that processes these 16 bits whether they come from the keyboard, from a file on disk, or from a peripheral device.

See Also

bit, byte, data formats, file

strerror — String function

Translate an error number into a string

char *strerror(*error*); int *error*;

char *strerror(int *error*);

strerror helps to generate an error message. It takes the argument *error*, which presumably is an error code generated by an error condition in a program, and may return a pointer to the corresponding error message.

See Also

perror, string

Notes

strerror returns a pointer to a static array that may be overwritten by a subsequent call to **strerror**.

strerror differs from the related function **perror** in the following ways: **strerror** receives the error number through its argument *error*, whereas **perror** reads the global constant **errno**. Also, **strerror** returns a pointer to the error message, whereas **perror** writes the message directly into the standard error stream.

strerror and **perror** must return the same error message when handed the same error number.

string — Overview

The character string is a common formation 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 normally is used as the starting point of the string; note that a pointer to a string holds only this address. Note, too, that an array of 20 characters can hold 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 string for a character
memchr	search buffer for a character
memcmp	compare two buffers
memcpy	copy one buffer into another
memset	initialize a buffer
pnmatch	match a string pattern
rindex	search string for a character
strcat	concatenate two strings
strchr	find a character in a string
strcmp	compare two strings
strcpy	copy one string into another
strcspn	return length for which strings do not match
strerror	translate error number into string
strlen	measure a string
strncat	concatenate two strings
strncmp	compare two strings
strncpy	copy one string into another
strpbrk	find first occurrence of any character in string
strrchr	find rightmost occurrence of character
strspn	return length for which strings match
strstr	find one string within another
strtok	break a string into tokens

Example

This example reads from `stdin` up to `NNAMES` names, each of which is no more than `MAXLEN` characters long. It then removes duplicate names, sorts the names, and writes the sorted list to the standard output. It demonstrates the functions `shellsort`, `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 strcmp();
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 *), strcmp);
    for (index=0; index < count; index++)
        printf("%s\n", array[index]);
    exit(0);
}

strcmp(s1, s2)
register char **s1, **s2;
{
    extern int strcmp();
    return(strcmp(*s1, *s2));
}
```

See Also

ASCII, Lexicon

Notes

The draft ANSI standard for the C language allows adjacent string literals, e.g.:

```
"hello" "world"
```

Mark Williams C now supports this standard. Note, however, that this feature may not be portable to all other compilers. Also, because it departs from the Ker-

nighan and Ritchie description of C, it will generate a warning message if you use the compiler's **-VSBOOK** option.

strip — Command

Strip debug, relocation, and symbol tables from executable file

strip -drs file ...

strip removes the symbol table, relocation information, and debug tables from a file. It makes the executable file or object module noticeably smaller.

strip recognizes the following options:

- d** Keep debug information. If this option is *not* used, all debug information used by the debuggers **csd** and **db** is removed.
- r** Keep relocation information. Note that this is not the GEM-DOS relocation information.
- s** Keep the symbol table.

See Also

Notes

Because version 3.0 changes the object format, the edition of **strip** shipped with version 3.0 does not work with objects compiled with Mark Williams C version 2.1.7 or earlier. To convert such objects to a format that **strip** recognizes, use the command **mwto mw**.

strlen — String function (libc)

Measure the length of a string

int strlen(string) char *string;

strlen measures *string*, and returns its length in bytes, *not* including the NUL terminator. This is 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)

Append one string onto 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*.

Example

For an example of this function, see the entry for **strncpy**.

See Also

strcat, **string**

Notes

string1 should point to enough space to hold itself and *n* characters of *string2*. If it does not, a portion of the program or operating system may be overwritten.

strncmp — String function (libc)

Compare two strings

int 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, returns a number less than zero if *string1* occurs earlier alphabetically than *string2*, and returns a number greater than zero if it occurs later. This routine is compatible with the ordering routine needed by **qsort**.

Example

For an example of this function, see the entry for **strncpy**.

See Also

memcmp, **strcmp**, **strcspn**, **string**, **strspn**, **strstr**

strncpy — String function (libc)

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.

Example

This example, called **swap.c**, reads a file of names, and changes them from the format

```
first_name [middle_initial] last_name
```

to the format

```
last_name, first_name [middle_initial]
```

It demonstrates **strncpy**, **strncat**, **strncmp**, and **index**.

```
#include <stdio.h>
#define NNAMES 512
#define MAXLEN 60

char *array[NNAMES];
char gname[MAXLEN], lname[MAXLEN];
extern int strncmp(), strcmp();
extern char *strcpy(), *strncpy(), *strncat(), *index();

main(argc, argv)
int argc; char *argv[];
{
    FILE *fp;
    register int count, num;
    register char *name, string[60], *cptr, *eptr;
    unsigned glength, length;

    if (--argc != 1)
    {
        fprintf(stderr, "Usage: swap filename\n");
        exit(1);
    }

    if ((fp = fopen(argv[1], "r")) == NULL)
        printf("Cannot open %s\n", argv[1]);
    count = 0;
    while (fgets(string, 60, fp) != NULL)
    {
        if ((cptr = index(string, '.')) != NULL)
        {
            cptr++;
            cptr++;
        } else if ((cptr = index(string, ' ')) != NULL)
            cptr++;

        strcpy(lname, cptr);
        eptr = index(lname, '0');
        *eptr = ',';

        strncat(lname, " ");
        glength = (unsigned)(strlen(string) - strlen(cptr));
        strncpy(gname, string, glength);

        name = strncat(lname, gname, glength);
        length = (unsigned)strlen(name);
        array[count] = malloc(length + 1);

        strcpy(array[count], name);
        count++;
    }
}
```

```
    for (num = 0; num < count; num++)  
        printf("%s\n", array[num]);  
    exit(0);  
}
```

See Also

memcpy, strepy, string

Notes

string1 must point to enough space to hold itself and *string2*; otherwise, a portion of the program or operating system may be overwritten.

strpbrk — String function

Find first occurrence in a string of any character from another string

char *strpbrk(string1, string2)

char *string1, *string2;

strpbrk returns a pointer to the first character in *string1* that matches any character in *string2*. It returns NULL if no character in *string1* matches a character in *string2*. The set of characters that *string2* points to is sometimes called the “break string”. For example,

```
char *string = "To be, or not to be: that is the question.";  
char *brkset = ",;";  
strpbrk(string, brkset);
```

returns the value of the pointer **string** plus six; this points to the comma, which is the first character in the area pointed to by **string** to match any character in the string pointed to by **brkset**.

See Also

strchr, strcspn, string, strpbrk, strchr, strspn, strstr, strtok

Notes

strpbrk resembles the function **strtok** in functionality, but unlike **strtok**, it preserves the contents of the strings being compared. It also resembles the function **strchr**, but lets you search for any one of a group of characters, rather than for one character alone.

strrchr — String function

Search for rightmost occurrence of a character in a string

char *strrchr(string, character)

char *string; int character;

strrchr looks for the last, or rightmost, occurrence of *character* within *string*. *character* is declared to be an **int**, but is handled within the function as a **char**. Another way to describe this function is to say that it performs a reverse search for a character in a string.

strchr returns a pointer to the rightmost occurrence of *character*, or NULL if *character* could not be found within *string*.

See Also

memchr, **strchr**, **strcspn**, **string**, **strpbrk**, **strspn**, **strstr**, **strtok**

Notes

strchr is identical to the function **rindex**, which is included with Mark Williams C.

strspn — String function

Return length for which one string includes characters in another

unsigned int strspn(string1, string2)

char *string1, *string2;

strspn returns the length for which *string1* initially consists only of characters that are found in *string2*. For example,

```
char *s1 = "hello, world";
char *s2 = "kernighan & ritchie";
strcspn(s1, s2);
```

returns two, which is the length for which the first string initially consists of characters found in the second.

See Also

memchr, **strchr**, **strcspn**, **string**, **strpbrk**, **strchr**, **strstr**, **strtok**

strstr — String function

Find one string within another

char *strstr(string1, string2)

char *string1, *string2;

strstr looks for *string2* within *string1*. The terminating null character is not considered part of *string2*.

strstr returns a pointer to where *string2* begins within *string1*, or NULL if *string2* does not occur within *string1*.

For example,

```
char *string1 = "Hello, world";
char *string2 = "world";
strstr(string1, string2);
```

returns *string1* plus seven, which points to the beginning of **world** within **Hello, world**. On the other hand,

```
char *string1 = "Hello, world";
char *string2 = "worlds";
strstr(string1, string2);
```


returns NULL because **worlds** does not occur within **Hello, world**.

See Also

memchr, **strchr**, **strcspn**, **string**, **strpbrk**, **strrchr**, **strspn**, **strtok**

struct — C keyword

Data type

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 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 one structure to be assigned to another, provided the two structures are of the same type. It also allows structures to be passed by and returned by 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, **C keywords**, **C language**, **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 manipulated 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 — Technical information

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 C compilers transform structure arguments and structure returns into structure pointers. Note that the use of structure assignment, structure arguments, or structure returns may create problems when porting the code to another C compiler.

See Also

portability, struct, structure

Notes

Note that because this feature deviates from the description of the C language found in *The C Programming Language*, compiling with the **-VSBOOK** option will flag all points where it occurs in your program.

SUFF – Environmental variable

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 privilege mode

```
long Super(stack) char *stack;
```

Super manipulates the Atari ST's privilege mode. *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 **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 lddiv_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)

Swap a pair of bytes

void 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.

Example

This example prompts for an integer; it then prints the integer both as you entered it, and as it appears with its bytes swapped.

```
#include <stdio.h>

main() {
    int word;

    printf("Enter an integer: \n");
    scanf("%d", &word);
    printf("The word is 0x%x\n", word);
    swab(&word, &word, 2);
    printf("The word with bytes swapped is 0x%x\n", word);
}
```

See Also
byte ordering

switch — C keyword

Test a variable against a table

switch is a C keyword that lets you perform a number of tests on a variable in a convenient manner. For example,

```
while(foo < 10)
    switch(foo) {
        case 1:
            dosomething();
            break;
        case 2:
            somethingelse();
            break;
        case 3:
            anotherthing();
            break;
        default:
            break;
    }
}
```

is equivalent to

```
while(foo < 10) {
    if(foo == 1) {
        dosomething();
        continue;
    } else if(foo == 2) {
        somethingelse();
        anotherthing();
        continue;
    } else if(foo == 3) {
        /* Note: compiler eliminate duplicate code */
        anotherthing();
        continue;
    } else
        break;
}
```

Note that **switch** is always used with the **case** statement, and nearly always with the **default** statement.

See Also

break, **C keywords**, **C language**, **case**, **default**, **keyword**, **while**
The C Programming Language, page 54

system — General function (libc)

Pass a command to TOS for execution

int system(commandline) char *commandline;

system passes *commandline* to the Mark Williams microshell, which loads it into memory and executes it. **system** executes commands exactly as if they had been typed directly into the shell.

system uses the environmental variables **SHELL** and **PATH** to find the command line processor. **SHELL** defaults to **msh.prg**, but you can substitute any other command processor that can evaluate the command lines passed through **system** by **make**, **me**, or **db**.

Example

This example uses **system** to call the **msk** command **ls** to list all C programs in the present directory. It redirects its output into the file **list.fil**.

```
#include <stdio.h>

FILE *newfp;
int oldstdout;

main()
{
    extern int system();

    reopen("list.fil");
    system("dir *.c");
    rclose();
}

/*
 * Redirect stdout prior to system() call.
 * You can't redirect child process's I/O
 * but you can redirect main()'s and let the child inherit it.
 */

reopen(tofile)
char *tofile;
{
    if ((newfp = fopen(tofile, "w")) == NULL)
        fatal("cannot open output file \"%s\"", tofile);

    /* Duplicate stdout so it can be restored later */
    if ((oldstdout = dup(fileno(stdout))) == -1)
        fatal("dup failed");

    /* Force duplication of new file handle as stdout */
    if (dup2(fileno(newfp), fileno(stdout)) == -1)
        fatal("dup2 failed");
}

/*
 * Terminate redirection
 */

rclose()
{
    /* Restore old stdout */
    if (dup2(oldstdout, fileno(stdout)) == -1)
        fatal("dup2 failed");
    /* Close the extra handle */
    if (close(oldstdout) != 0)
        fatal("cannot close old stdout");
    fclose(newfp);
}
```



```
/*
 * Fatal error
 */
fatal(p)
char *p;
{
    fprintf(stderr, "redirect: %r\n", &p);
    exit(1);
}
```

See Also

execve, exit, msh, Pexec

Notes

No shell variable that has been set with the **set** command is duplicated.

system variables — Technical information

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.

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_critc/long

Points to the critical error handler.

0x408/etv_term/long

Points to routine that ends a program.

0x04C/etv_xtra/long

0x420/memvalid/int

Check if the memory controller's configuration is valid.

0x424/memcntl/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/timr_ms/int

Clock rate, in microseconds.

0x444/fverify/int

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/defshiftmd/int

If Atari shifted from monochrome to color, new resolution is set here: zero indicates low resolution; one, medium resolution.

0x44C/sshiftmd/int

Screen resolution, as follows: zero, low resolution; one, medium resolution; two, high resolution.

0x44E/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

Points to beginning of video RAM.

0x462/vbclock/long

Number of vertical blank interrupt routines.

0x466/frclock/long

Number of vertical blank routines executed.

0x46A/hdv_init/long

Points to hard-disk initialization.

0x46E/swv_vec/long

Points to routine to change screen resolution.

0x472/hdv_bpb/long

Points to fetch BIOS parameter block for hard disk.

0x476/hdv_rw/long

Points to read/write routine for hard disk.

0x47A/hdv_boot/long

Points to routine to reboot hard disk.

0x47E/hdv_mediach/long

Points to routine to handle medium change for hard disk.

0x482/cmdload/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**.

0x48E/themd/four 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.

0x4AE/sav_context/long

Points to temporary areas used by exception-handling routines.

0x4B2/bufl0/long

Points to head of data sector list.

0x4B2/bufl1/long

Points to head of file allocation table (FAT).

0x4BA/hz_200/long

Counter for 200-Hz system clock.

0x4BE/the_env/four 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.

0x4EE/prt_cnt/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.

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

T

tail — Command

Print the end of a file

tail [+*n*[*bcl*]] [*file*]

tail [-*n*[*bcl*]] [*file*]

tail copies the last part of *file*, or of the standard input if none is named, 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; the default is lines. If no *number* is specified, a default of -10 is assumed.

See Also

commands, **egrep**

Notes

Because **tail** buffers data measured from the end of the file, large counts may not work.

tan — Mathematics function (libm)

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)

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)

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 the string of letters that will prefix the temporary name; this string should not be more than three or four 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**, **tmpnam**

tetd_to_tm — Time function (libc)

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 (overview), 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 — Command

Time the execution of a command

time "*command arguments*"

time lets you time the execution of a command under the microshell, **msh**. *command* is the command to execute, and *arguments* its arguments. For example, typing

```
time "cc hello.c"
```

times how long it takes the compiler to compile **hello.c**. Note that *command* and its arguments must be enclosed within quotation marks, for **msh** to parse the command line correctly.

When *command* is finished, **time** prints on the standard output device the amount of time needed to execute the command, in hours, minutes, seconds, and hundredths of a second.

See Also

commands, msh

time — Time function (libc)

Get current time

```
#include <time.h>
```

```
time_t time(tp) time_t *tp;
```

time reads the current system time. *tp* points to a data element 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 the number of seconds since January 1, 1970, 0h00m00s GMT.

Example

For an example of this function, see the entries for **asctime** and **time**.

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 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. Mark Williams C uses three variables to describe time:

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. Mark Williams C defines **time_t** as a 32-bit number that holds the number of seconds since January 1, 1970, 0h00m00s GMT.

struct tm

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 [1-31] */
    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 [0-366] */
    int tm_isdst; /* daylight savings time flag */
} tm_t;
```

The ANSI standard also describes a number of time functions, as follows:

asctime	convert tm to ASCII string
clock	return time since system was turned on
ctime	return an ASCII string that gives local time
difftime	compute difference between calendar times
gmtime	return tm for Greenwich Mean Time
localtime	return tm for local time
stime	set system time (UNIX/COHERENT-compatible)
time	return time_t

To print out the local time, a program must perform the following tasks: First, read the system time with **time**. Then, it must pass **time**'s output to **localtime**, which breaks it down into the **tm** structure. Next, it must pass **localtime**'s output to **asctime**, which transforms the **tm** structure into an ASCII string. Finally, it must pass the output of **asctime** to **printf**, which displays it on the standard output device. See the entry for **asctime** for an example of such a program.

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 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 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** structure to Julian date
jday_to_tm convert Julian date to **tm** 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** to TOS time
tetd_to_tm convert TOS time to **tm**

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** and the IKBD clock:

Kgettextime turn IKBD time to **tm**
Ksettime turn **tm** to IKBD time

Finally, the Atari **gemdos** and **xbios** routines include a number of functions that directly manipulate system time, as follows:

Fdftime	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 example 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**, **tm_to_tetd**, **Tsetdate**, and **Tsettime**.

```
#include <time.h>
#include <osbind.h>
tm getdate(p)
char *p;
{
    static tm 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 *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 *) (x), "x");

main(argc, argv)
int argc; char *argv[];
{
    tm *tp;
    tetd_t td;
    time_t t;
    time_t temp;

    if (argc > 1) {
        tp = getdate(argv[1]);
        td = tm_to_tetd(tp);
        stime(&t);
```

```

        Ksettime(tp);
        Settime(td);
        Tsetdate(td.g_date);
        Tsettime(td.g_date);
    }

    temp = time(0L);
    display(localtime(&temp));
    display(gmtime(&temp));
    display(Kgettime());
    display(tetd_to_tm(Gettime()));
    display(tetd_to_tm(((long)Tgetdate())<<16)|(unsigned)Tgettime()));
}

```

See Also

Lexicon

time — Command

Print current time/time execution of a command

time

time *command*

time " *command arguments* "

The command **time** performs two different tasks, depending upon whether it is used with or without arguments.

When **time** is typed without any arguments, it prints the date and time. The date and time are presented in a string of the form:

```
Thu Apr 7 10:35:53 1988 CDT
```

The extension "CDT" stands for "Central Daylight Time". Daylight savings time will be returned only if the macro **TIMEZONE** is set properly in your **profile**. See **TIMEZONE** for more information.

If **time** is used with one or more arguments, it times the execution of a command. For example, typing **time ls** prints the contents of the current directory, then prints a string of the form:

```
00:00:02.340
```

which states how long the command took to execute.

If you wish to time a command that takes arguments, you must enclose the command and its arguments within quotation marks. For example, to time how long it takes to compile the program **window.c** with the **-VGEM** option to the compiler, use the command:

```
time "cc -VGEM window.c"
```

See Also

commands, date, msh, time (overview)

time.h — Header file

Give time-description structure

#include <time.h>

time.h is a header file that contains descriptions and declarations for elements used to manipulate system time under TOS.

See Also

time

time_to_jday — Time function (libc)

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 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.

See Also

jday_to_time, jday_to_tm, time (overview), time.h, tm_to_jday

TIMEZONE — Environmental variable

Time zone information

TIMEZONE=standard:offset[:daylight:date:date:hour:minutes]

TIMEZONE is an environmental parameter that holds information about the user's time zone. This information is used by Mark Williams C's time routines to construct their description of the current time and day.

To set the **TIMEZONE** parameter, use the **set** command, as follows:

```
set TIMEZONE=[description]
```

```
setenv TIMEZONE=[description]
```

where [description] is the string that describes your time zone. What this string consists of will be described below. Most users write this command into the file **profile**, so that **TIMEZONE** is set automatically whenever they invoke **msh**.

The description string

A **TIMEZONE** description string consists of seven fields that are separated by colons. Fields 1 and 2 must be filled; fields 3 through 7 are optional.

Field 1 gives the name of your standard time zone. Field 2 gives the time zone's offset from Greenwich Mean Time in minutes. Offsets are positive for time zones west of Greenwich and negative for time zones east of Greenwich. For example, users in Chicago set these fields as follows:

```
TIMEZONE=CST:360
```

CST is an abbreviation for Central Standard Time, that area's time zone; and 360 refers to the fact that Chicago's time zone is 360 minutes (six hours) behind that of Greenwich.

Field 3 gives the name of the local daylight saving time zone. In Chicago, for example, this field would be set as follows:

```
TIMEZONE=CST:360:CDT
```

CDT is an abbreviation for Central Daylight Time. The absence of this field indicates that your area does not use daylight saving time.

Fields 4 and 5 specify the dates on which daylight saving time begins and ends. If field 3 is set but fields 4 and 5 are not, changes between standard time and daylight saving time will be assumed to occur at the times legislated in the United States in 1986: at 2 A.M. standard time on the first Sunday in April, and at 2 A.M. daylight saving time on the last Sunday in October.

Fields 4 and 5 each consist of three numbers separated by periods. The first number specifies which occurrence of the day 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. For example, in Chicago fields 4 and 5 are set to the following:

```
TIMEZONE=CST:360:CDT:1.1.4:-1.1.10
```

If the first number in either field is set to zero, then the last two numbers are assumed to indicate an absolute date. This is done because some countries switch to daylight saving time on the same day each year, instead of a given day of the week.

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. In Chicago, these are set as follows:

```
TIMEZONE=CST:360:CDT:1.1.4:-1.1.10:2:60
```

The '2' of field 6 indicates that the switch to daylight savings time occurs at 2 A.M. The '60' of field 7 indicates that daylight savings time changes the local time by 60 minutes. Although 60 minutes is the standard change, some regions of the world

shift by 30, 45, 90, or 120 minutes; the last shift is also called “double daylight saving time”.

Under the microshell **msh**, it usually is 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

This environmental variable should be set only if you have set your computer system’s time to conform with Greenwich Mean Time. Otherwise, it will cause such functions as **localtime** to incorrectly offset the time they return.

The time zone **time** returns depends on how the time zone was originally set. If **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). If you live outside that time zone, you may wish to edit **TIMEZONE** to reflect your local time zone.

For those requiring more information on this subject, much research has been performed by astrologers. See *Time Changes in the World*, compiled by Doris Chase Doane (three volumes, Hollywood, CA, Professional Astrologers, Inc., 1970).

tm_to_jday — Time function (libc)

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** return the current time in this format. For more information on **tm_t**, see the entry for **time**.

tm_to_jday returns the structure **jday_t**, which is defined in the header file **time.h**. **jday_t** to consist 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.

See Also

jday_to_time, jday_to_tm, time (overview), time.h, time_to_jday

tm_to_tetd — Time function (libc)

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** (overview), **time.h**

TMPDIR — Environmental variable

TMPDIR names the directory into which Mark Williams C writes its temporary files. If this variable is not set, the default is the directory in which the source files are kept. Note that this variable need be set only if space is a problem on the storage device that holds your current directory. For example, the command

```
set TMPDIR=a:\tmp
```

typed at the system prompt tells **cc** to write temporary files in the directory **tmp** on drive A:.

It is a good idea to set **TMPDIR** so that your temporary files are written onto a RAM disk. This will speed compilation noticeably.

See Also

cc, **environment**, **environmental variable**

tmpnam — General function (libc)

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.

Unlike its cousin **tempnam**, **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.h

Convert characters to ASCII

#include <ctype.h>

int toascii(c) int c;

toascii takes any integer value *c*, keeps the low seven bits unchanged, and changes the others to zero; this, in effect, transforms the integer value to an ASCII character. **toascii** then returns the transformed integer. If *c* is already a valid ASCII character, it is returned unchanged.

Example

This example prompts for a file name. It then opens the file and prints its contents, while converting all non-alphanumeric characters to alphanumeric.

```
#include <ctype.h>
#include <stdio.h>

main()
{
    FILE *fp;
    int ch;
    int filename[20];

    printf("Enter file name: ");
    fflush(stdout);
    gets(filename);
    if ((fp = fopen(filename, "r")) != NULL) {
        while ((ch = fgetc(fp)) != EOF)
            putchar(isascii(ch) ? ch : toascii(ch));
    }
    else printf("Cannot open %s\n", filename);
}
```

See Also

ctype

tolower — General function (libc)

Convert characters to lower case

int tolower(c) int c;

tolower converts the letter *c* to lower case. **tolower** returns *c* converted to lower case. If *c* is not a letter or is already lower case, then **tolower** returns it unchanged.

Example

The following example demonstrates **tolower** and **toupper**. It reverses the case of every character in a text file.

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: ");
    fflush(stdout);
    gets(filename);

    if ((fp = fopen(filename,"r")) != NULL)
    {
        while ((ch = fgetc(fp)) != EOF)
            putchar(isupper(ch) ? tolower(ch) : toupper(ch));
    } else
        printf("Cannot open %s.\n", filename);
}
```

See Also

ctype, **_tolower**, **toupper**

_tolower — **ctype** macro (**ctype.h**)

Convert letter to lower case

```
#include <ctype.h>
int _tolower(c) int c;
```

_tolower is a macro that converts *c* to lower case and returns it. If *c* is not a letter, the result is undefined.

_tolower differs from its cousin **tolower** in that **_tolower** is a macro that does not check whether its argument is in fact an alphanumeric character, whereas **tolower** is a function that does check its argument.

Example

This example opens a file of text and reverses the cases of all characters. It demonstrates **_tolower** and **_toupper**.

```
#include <ctype.h>
#include <stdio.h>
```

```
main(argc, argv)
int argc; char *argv[];
{
    FILE *fp;
    int ch;

    if (--argc != 1)
        fatal("Usage: example filename");

    if ((fp = fopen(argv[1], "r")) == NULL)
        fatal("Cannot open file for reading");

    while ((ch = fgetc(fp)) != EOF)
    {
        if ((isascii(ch) != 0) && ch != '\r')
            fatal("Not a text file");

        if (isalpha(ch) != 0)
            fputc((isupper(ch) ? _tolower(ch) : _toupper(ch)),
                stdout);
        else
            fputc(ch, stdout);
    }

    fatal(message)
    char *message;
    {
        fprintf(stderr, "%s\n", message);
        exit(1);
    }
}
```

See Also

ctype, **tolower**, **_toupper**

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 generated 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, **crtso.o**. Note, however, that it differs from the default runtime startup **crtso.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 — General function (libc)

Convert characters to upper case

#include <ctype.h>

int toupper(c) int c;

toupper is a macro that converts the letter *c* to upper case and returns the converted character. If *c* is not a letter or is already upper case, then **toupper** returns it unchanged.

Example

For examples of this routine, see the entries for **ctype** and **tolower**.

See Also

ctype, tolower, _toupper

_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.

_toupper differs from its cousin **toupper** in that **_toupper** is a macro that does not check whether its argument is in fact an alphanumeric character, whereas **toupper** is a function that does check its argument.

Example

For an example of this routine, see the entry for **_tolower**.

See Also

ctype, **_tolower**, **toupper**

Tsetdate — gemdos function 43 (osbind.h)

Set a new date

```
#include <osbind.h>
```

```
long Tsetdate(i) inti;
```

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 checking — Technical information

Every expression has a *type*, such as **int**, **char**, or **double**. C is not strongly typed, which means that it allows different types to be mixed relatively freely, and be changed (or **cast**) from one type to another.

Mark Williams C checks types more strictly than the C standard implies. 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

cast, cc, type promotion

typedef — C keyword

Define a new data type

typedef is a C facility that lets you define new data types. Such definitions are always made in terms of existing data types; for example,

```
typedef long time_t;
```

establishes the data type **time_t**, 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

C keyword, C language, declarations, manifest constants, portability, storage class

The C Programming Language, page 140

type promotion — Technical information

In arithmetic expressions, Mark Williams C promotes one signed type to another signed type by sign extension, and promotes one unsigned type to another unsigned type 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

U

#undef — Preprocessor instruction

Undefine a manifest constant

#undef *variable*

#undef tells the C preprocessor **cpp** to suspend the current value of *variable*, which had been defined with the **#define** command. For example, in a long program *variable* may be undefined for one function, where its current value conflicts with other elements of the function. The variable can be redefined or restored to its original value with another **#define** statement.

See Also

cc (-D option), **cpp**, **#define**

ungetc — STDIO function (libc)

Return character to input stream

#include <stdio.h>

int 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**. No more than one character can be pushed back into any stream at once. A call to **fseek** will nullify the effects of a previous **ungetc**.

Example

The following example opens a file and returns how many lines and sentences it contains. A sentence is defined as being any passage of text that ends in a period.

```
#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 — C keyword

Multiply declare a variable

A **union** defines an area of storage that can accept any one of several types of data. In effect, it is a multiple declaration of a variable. 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. For example, the declaration

```

union {
    int number;
    double bignumber;
    char *stringptr;
} example;
```

allows **example** to hold either an **int**, a **double**, or a pointer to a **char**, whichever is needed at the time. The elements of a **union** are accessed like those of a **struct**: for example, to access **number** from the above example, type **example.number**.

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

For an example of how to use a **union** in a program, see the entry for **byte ordering**.

See Also

C keywords, **C language**, **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 version 7 of 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:

chdir	change working directory
chmod	change a file's 'mode'
chown	change a file's owner

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
execve	execute command from within program
_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, **STDIO**

unlink — UNIX system call (libc)

Remove a file

int unlink(name) char *name;

unlink removes the directory entry for the given file *name*, which in effect erases *name* from the disk. Note that *name* cannot be open when **unlinked**. The name is a historical artifact.

Example

This example removes the files named on the command line.

```
main(argc, argv)
int argc; char *argv[];
{
    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 it cannot remove a file, and zero if it can.

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.

unset uses the same sort of “in directory” syntax as the **set** command. For example, the command

```
unset in .bin
```

erases **.bin** and everything in it; whereas

```
unset in .bin ls lc cd echo
```

will remove **ls**, **lc**, **cd**, and **echo** from **.bin**.

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 — C keyword

Data type

The **unsigned** modifier tells the compiler to treat the variable as an unsigned value. In effect, this doubles the largest positive value storage in that type, and changes the lowest storage value to zero. Note that the 68000 uses “two’s complement” storage, not sign magnitude.

See Also

C keywords, **C language**, **data type**

The C Programming Language, page 34

V

v_arc — VDI function (libvdi)

Draw a circular arc

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
void v_arc(handle, x, y, radius, beginangle, endangle)
```

```
int handle, x, y, radius, beginangle, endangle;
```

v_arc is a VDI routine that draws a circular arc. *handle* is the virtual device's VDI handle. *x* and *y* 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)

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 as the preset fill pattern.

handle is the virtual device's VDI handle. *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 draw 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 <aesbind.h>
#include <gemdefs.h>
#include <vdibind.h>

#define CLICKS 1 /* no. of clicks expected */
#define BUTTON 1 /* which button; 1=leftmost */
#define BUTTONSTATE 1 /* button state; 1=down */

/* global line A variables used by vdi; MUST be included */
int ctrl[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, 1, 1 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* Place for unwanted data */
int nowhere = 0;

main()
{
    /* declarations used by evt_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 */
    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 */
    int depth; /* depth of rubberbox */

    /* OK, here we go ... */
    appl_init();
    graf_mouse(ARROW, &nowhere);
    v_opvwk(work_in, &vdihandle, work_out);

    /* set clipping array to size of screen */
    cliparray[2] = work_out[0];
    cliparray[3] = work_out[1];
}
```

```

/* set clipping rectangle; draw rectangle */
vs_clip(vdihandle, 1, cliparray);
vsf_perimeter(vdihandle, 1);

for(;;) {
    selection = evt_multi(which, CLICKS, BUTTON,
        BUTTONSTATE, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        buffer, 0, 0, &mousex, &mousey,
        &nowhere, &nowhere, &key, &nowhere);

    switch(selection) {
    case MU_KEYBD:
        switch((char)key) {
        case 'r':
            v_clswwk(vdihandle);
            appl_exit();
            exit(0);

        case 't':
        case 'T':
            type = (++type%5);
            vsf_interior(vdihandle, type);
            v_hide_c(vdihandle);
            v_bar(vdihandle, xyarray);
            v_show_c(vdihandle, 0);
            break;

        case 's':
        case 'S':
            style = ((style%24)+1);
            vsf_style(vdihandle, style);
            v_hide_c(vdihandle);
            v_bar(vdihandle, xyarray);
            v_show_c(vdihandle, 0);
            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);
        v_hide_c(vdihandle);
        v_bar(vdihandle, xyarray);
        v_show_c(vdihandle, 0);
        break;

    default:
        break;
    }
}

```

See Also

TOS, vr_recfl, VDI

v_bit_image — VDI function (libvdi)

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:

- | | |
|---|-------------------------|
| 0 | ignore aspect ratio |
| 1 | honor pixel ratio |
| 2 | 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 it. *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. The even-numbered entries set the X coordinates, and the odd-numbered entries the Y coordinates.

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)

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 to 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, and *xyarray[2]* and *xyarray[3]* give the coordinates for the upper right-hand 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 equal *cells* times *rows*.

See Also

TOS, VDI, **vq_cellarray**

Notes

This routine is not present in the resident VDI. It may not be present in the GDOS.

v_circle — VDI function (libvdi)

Draw a circle

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
void v_circle(handle, x, y, radius) int handle, x, y, radius;
```

v_circle is a VDI routine that draws a circle. *handle* is the virtual device's VDI handle. *x* and *y* 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 <aesbind.h>
#include <gemdefs.h>
#include <vdi.h>

#define ASTERISK 3 /* draw an asterisk marker */
#define BUTTON 1 /* which button; 1=leftmost */
#define CLICKS 1 /* no. of clicks expected */
#define DOWN 1 /* mouse button is down */
#define UP 0 /* mouse button is up */
#define XOR 3 /* 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 to calculate radius */
int xyarray[4];

/* array used by v_pmarker() */
int xymarker[2];

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 1, 1 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* throw-away declaration */
int nowhere = 0;

main()
{
    /* declarations used by evnt_multi() */
    int selection; /* code for event */
    unsigned int which = (MU_KEYBD | MU_BUTTON); /* place for AES messages */
    int buffer[8]; /* mouse X coordinate */
    int mousex; /* mouse Y coordinate */
    int mousey; /* key typed by user */
    unsigned key;

    /* misc declarations */
    int vdihandle; /* virtual device's handle */
    int writectr = 0; /* write modes */
    int fillctr = 1; /* circle fill styles */
    int n = 0; /* keep track of xyarray */
}
```

```

appl_init();
graf_mouse(ARROW, &nowhere);
v_opnvwk(work_in, &vdihandle, work_out);

/* set clipping rectangle to size of screen */
cliparray[2] = work_out[0];
cliparray[3] = work_out[1];
vs_clip(vdihandle, 1, cliparray);

vsf_interior(vdihandle, 2);
vsf_perimeter(vdihandle, 1);
vsm_height(vdihandle, 3);
vsm_type(vdihandle, ASTERISK);

for(;;) {
    selection = evtnt_multi(which, CLICKS, BUTTON, DOWN,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, buffer, 0, 0,
        &mousex, &mousey, &nowhere, &nowhere, &key,
        &nowhere);

    switch(selection) {
        /* if keyboard is pressed ... */
        case MU_KEYBD:
            if ((char)key == '\r') {
                v_clswnk(vdihandle);
                appl_exit();
                exit(0);
            }
            if (((char)key == 'w') || ((char)key == 'W'))
                writectr++;
            break;

        /* if user presses a mouse button ... */
        case MU_BUTTON:
            evtnt_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 */
    int leg2;                /* second leg */
    int radius;              /* radius of circle=hypotenuse */
    extern double hypot();    /* declare hypot() */

    leg1 = abs(xyarray[2] - xyarray[0]);
    leg2 = abs(xyarray[3] - xyarray[1]);
    /* note casts of variables */
    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)

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 is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

v_clrwk – VDI function (libvdi)

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)

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 all tasks needed to close the device 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)

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 is not available in the ROM-resident VDI. 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)

Fill an outlined area

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
void v_contourfill(handle, x, y, color)
```

```
int handle, x, y, 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. *x* and *y* 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 <aesbind.h>
#include <gemdefs.h>
#include <vdibind.h>

#define BLACK 1
#define BUTTON 1
#define CLICKS 1
#define DOWN 1
#define REPLACE 1
#define UP 0
#define XOR 3

/* code for color black */
/* which button; 1 = leftmost */
/* no. of clicks expected */
/* mouse button is down */
/* make writing mode REPLACE */
/* mouse button is up */
/* 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, 1, 1 };

/* array used by evnt_multi(), for writing AES messages */
int buffer[8];

/* 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 */
int nowhere = 0;

main()
{
    /* declarations used by evnt_multi() */
    int selection;          /* code for event */
    unsigned key;           /* scan code of key */
    int mousex;             /* mouse X coordinate */
    int mousey;             /* mouse Y coordinate */
    int vdihandle;          /* virtual device's handle */
    int flag = 0;           /* has line been drawn yet? */
    int i = 0;              /* counter */

    /* OK, here we go ... */
    appl_init();
    graf_mouse(ARROW, &nowhere);
    v_opnvwk(work_in, &vdihandle, work_out);

    cliparray[2] = work_out[0];
    cliparray[3] = work_out[1];
    vs_clip(vdihandle, 1, cliparray);
    vsf_interior(vdihandle, 2);
    vsf_style(vdihandle, 23);
    vswr_mode(vdihandle, XOR);

    for(;;) {
        selection = evnt_multi(MU_KEYBD | MU_BUTTON,
                               CLICKS, BUTTON, DOWN, 0, 0, 0, 0, 0,
                               0, 0, 0, 0, 0, buffer, 0, 0, &mousex,
                               &mousey, &nowhere, &nowhere, &key,
                               &nowhere);

        switch(selection) {
            case MU_KEYBD:
                if ((char)key == '\r') {
                    v_clsvwk(vdihandle);
                    appl_exit();
                    exit(0);
                }

                if (((char)key == 'w') || ((char)key == 'W')) {
                    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)) {

```

See Also
TOS, VDI

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)

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.

Example

For an example of this function, see the entry for **v_enter_cur**.

See Also

TOS, **v_curhome**, **v_curleft**, **v_curright**, **v_curup**, VDI

v_curhome — VDI function (libvdi)

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.

Example

For an example of this function, see the entry for **v_enter_cur**.

See Also

TOS, **v_curdown**, **v_curleft**, **v_curright**, **v_curhome**, VDI

v_curleft — VDI function (libvdi)

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.

Example

For an example of this function, see the entry for **v_enter_cur**.

See Also

TOS, **v_curdown**, **v_curhome**, **v_curright**, **v_curup**, VDI

v_curright — VDI function (libvdi)

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.

Example

For an example of this function, see the entry for **v_enter_cur**.

See Also

TOS, **v_curdown**, **v_curhome**, **v_curleft**, **v_curup**, VDI

v_curtext — VDI function (libvdi)

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.

Example

For an example of this function, see the entry for **v_enter_cur**.

See Also

TOS, VDI

v_curup — VDI function (libvdi)

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.

Example

For an example of this function, see the entry for **v_enter_cur**.

See Also

TOS, **v_curdown**, **v_curhome**, **v_curleft**, **v_currigh**t, VDI

v_dspcur — VDI function (libvdi)

Move mouse pointer to point on screen

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
void v_dspcur(handle, x, y) int handle, x, y;
```

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. *x* and *y* are, respectively, the X and Y coordinates to which the mouse cursor will be moved.

See Also

TOS, VDI

v_eeol — VDI function (libvdi)

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.

Example

For an example of this function, see the entry for **v_enter_cur**.

See Also

TOS, **v_eeos**, VDI

v_eeos — VDI function (libvdi)

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.

Example

For an example of this function, see the entry for **v_enter_cur**.

See Also

TOS, **v_eeol**, VDI

v_ellarc — VDI function (libvdi)

Draw an elliptical arc

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
void v_ellarc(handle, x, y, xradius, yradius, beginangle, endangle)
```

```
int handle, x, y, xradius, yradius, beginangle, endangle;
```

v_ellarc is a VDI routine that draws an elliptical arc. *handle* is the virtual device's VDI handle. *x* and *y* give, respectively, the X and Y coordinates of the center of the imaginary ellipse, of which the curve being drawn is 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 <aesbind.h>
#include <gemdefs.h>
#include <stdio.h>
#include <vdibind.h>
```

```
#define ESCAPE 0x1B
#define REPLACE 1
#define RCUNDED 2
#define XOR 3
```

```
/* ASCII escape code */
/* REPLACE writing mode */
/* put rounded ends on lines */
/* 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, 1, 1 };

/* arrays used by drawline() */
xyvert[] = { 1, 1, 1, 1 };
xyhoriz[] = { 1, 1, 1, 1 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* keep system from scribbling over itself */
int nowhere = 0;

main()
{
    unsigned key;           /* key pressed 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);
    v_opvwk(work_in, &vdihandle, work_out);

    /* set clipping array to match screen dimensions */
    cliparray[2] = work_out[0];
    cliparray[3] = work_out[1];

    /* set line arrays to suit screen dimensions */
    xyvert[0] = work_out[0]/2;
    xyvert[1] = 1;
    xyvert[2] = work_out[0]/2;
    xyvert[3] = work_out[1];

    xyhoriz[0] = 1;
    xyhoriz[1] = work_out[1]/2;
    xyhoriz[2] = work_out[0];
    xyhoriz[3] = work_out[1]/2;

    /* set clipping rectangle & line style */
    vs_clip(vdihandle, 1, cliparray);
    vsl_ends(vdihandle, ROUNDED, ROUNDED);

    /* and execute the program */
    for(;;) {
        printf("Type <return> to continue, <esc> to exit.\n");
        key = evt_keybd();
        switch((char)key) {

```

```
/* user wants to exit */
case ESCAPE:
    v_clsvwk(vdihandle);
    appl_exit();
    exit(0);

/* user wants to continue */
case 'r':
    drawlines(vdihandle);
    /* Enter X radius */
    xradius=getdata("Enter X radius (screen, 0-320)");
    xyhoriz[0] = (work_out[0]/2) - xradius;
    xyhoriz[2] = (work_out[0]/2) + xradius;
    drawlines(vdihandle);

    /* Enter Y radius */
    yradius=getdata("Enter Y radius (screen, 0-200)");
    xyvert[1] = (work_out[1]/2) - yradius;
    xyvert[3] = (work_out[1]/2) + yradius;
    drawlines(vdihandle);

    /* Enter beginning angle */
    beginangle=getdata("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, work_out[0]/2, work_out[1]/2,
             xradius, yradius, beginangle, endangle);
    break;

default:
    break;
}

}

/* draw cross-hairs on screen */
drawlines(handle)
int handle;
{
    printf("\033E\n");
    vsl_width(handle, 1);
    v_pline(handle, 2, xyvert);
    v_pline(handle, 2, xyhoriz);
    return;
}
```

```

/* get dimensions of arc from user */
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, x, y, xradius, yradius)
```

```
int handle, x, y, xradius, yradius;
```

v_ellipse is a VDI routine that draws an ellipse. *handle* is the virtual device's VDI handle. *x* and *y* 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. Pressing <return> exits.

```

#include <aesbind.h>
#include <gemdefs.h>
#include <vdibind.h>

#define DOWN 1                /* mouse button is down */
#define CLICKS 1              /* no. of clicks expected */
#define BUTTON 1              /* which button; 1=leftmost */

/* 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, 1, 1 };

```

```
/* 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 */
int nowhere = 0;

main()
{
    /* declarations used by evt_multi() */
    int selection; /* code for event that occurred */
    unsigned int which = (MU_KEYBD | MU_BUTTON); /* place to write messages */
    int buffer[8]; /* mouse X coordinate */
    int mousex; /* mouse Y coordinate */
    int mousey; /* key typed by user */
    unsigned key;

    /* misc declarations */
    int vdihandle; /* virtual device's handle */
    int writectr = 0; /* cycle through write modes */
    int fillctr = 1; /* cycle through fill styles */
    int width; /* box width */
    int height; /* box height */
    int xcoord; /* X coordinate of center */
    int ycoord; /* Y coordinate of center */
    int xradius; /* X radius of ellipse */
    int yradius; /* Y radius of ellipse */

    /* OK, here we go ... */
    appl_init();
    graf_mouse(ARROW, &nowhere);
    v_opvwk(work_in, &vdihandle, work_out);

    /* set clipping array to match screen dimensions */
    cliparray[2] = work_out[0];
    cliparray[3] = work_out[1];
    vs_clip(vdihandle, 1, cliparray);

    vsf_interior(vdihandle, 2);
    vsf_perimeter(vdihandle, 1);
    vsm_height(vdihandle, 3);

    for(;;) {
        selection = evt_multi(which, CLICKS, BUTTON,
                               DOWN, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                               buffer, 0, 0, &mousex, &mousey,
                               &nowhere, &nowhere, &key, &nowhere);

        switch(selection) {
            case MU_KEYBD:
                if ((char)key == '\r') {
                    v_clsvwk(vdihandle);
                    appl_exit();
                    exit(0);
                }
        }
    }
}
```

```

        if (((char)key=='w')||((char)key=='W')) {
            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_ellipse

Notes

v_ellipse can only create ellipses that are oriented horizontally or vertically. It cannot create ellipses that are oriented diagonally.

v_ellipse — VDI function (libvdi)

Draw an elliptical pie slice

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
void v_ellipse(handle, x, y, xradius, yradius, beginangle, endangle)
```

```
int handle, x, y, xradius, yradius, beginangle, endangle;
```

v_ellipse is a VDI routine that draws an elliptical pie slice. *handle* is the virtual device's VDI handle. *x* and *y* give, respectively, the X and Y coordinates of the imaginary ellipse of which v_ellipse 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_ellarc

v_enter_cur — VDI function (libvdi)

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.

Example

The following example creates a simple screen editor using VDI text calls. Note that it does not save anything you type, or have the capacity to write what you type into a file, although these features can be added. The keystrokes resemble those used by the MicroEMACS editor.

```
#include <aesbind.h>
#include <vdibind.h>

/* control characters used in example */
#define CTRLA 0x01 /* move to beginning of line */
#define CTRLB 0x02 /* move back one character */
#define CTRLF 0x06 /* move forward one character */
#define CTRLH 0x08 /* home cursor */
#define CTRLK 0x0B /* kill line of text */

#define CTRLN 0x0E /* move to next line */
#define CTRLP 0x10 /* move up to previous line */
#define CTRLR 0x12 /* toggle reverse video */
#define CTRLW 0x17 /* kill text to end of screen */
#define CTRLX 0x18 /* secondary set of keystrokes */
#define CTRLZ 0x1A /* exit from program */

/* standard VDI arrays */
int contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

/* arrays used by v_opnvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 1, 1 };

/* place to write junk */
long nowhere;

main()
{
    int row, column, videoflag;
    int vdihandle;
    char key[2];
    char string[50];
```

```
key[1] = '\0';
videoflag = 0;

appl_init();
v_opnvwk(work_in, &vdihandle, work_out);

/* set clipping area */
cliparray[2] = work_out[0];      /* width of screen */
cliparray[3] = work_out[1];      /* height of screen */
vs_clip(vdihandle, 1, cliparray);

/* enter text mode */
v_enter_cur(vdihandle);

/* accept characters from the keyboard and process them */
for(;;) {
    key[0] = (char) evnt_keybd();

    switch (key[0]) {
        case '\r':
            v_curdown(vdihandle);
            /* Note: no break */

        case CTRLA:
            vq_curaddress(vdihandle, &row, &column);
            vs_curaddress(vdihandle, row, 1);
            break;

        case CTRLB:
            v_curleft(vdihandle);
            break;

        case CTRLF:
            v_currright(vdihandle);
            break;

        case CTRLH:
            v_curhome(vdihandle);
            break;

        case CTRLK:
            v_eeol(vdihandle);
            break;

        case CTRLN:
            v_curdown(vdihandle);
            break;

        case CTRLP:
            v_curup(vdihandle);
            break;
    }
}
```



```
case CTRLR:
    if ((videoflag%2) == 0)
        v_rvon(vdihandle);
    else
        v_rvoff(vdihandle);
    videoflag++;
    break;

case CTRLW:
    v_eeos(vdihandle);
    break;

case CTRLX:
    switch((char)evnt_keybd()) {
        /* print out current position */
        case '=':
            vq_curaddress(vdihandle,
                           &row, &column);
            vs_curaddress(vdihandle,
                           24, 1);
            sprintf(string, "Row: %2d Column %2d",
                    row, column);
            v_curtext(vdihandle, string);
            vs_curaddress(vdihandle,
                           row, column);
            break;
    }
    break;

case CTRLZ:
    v_exit_cur(vdihandle);
    v_clswnk(vdihandle);
    exit(0);

default:
    v_curtext(vdihandle, key);
    break;
}
}
```

See Also

TOS, v_exit_cur, VDI

v_exit_cur — VDI function (libvdi)

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.

Example

For an example of this function, see the entry for `v_enter_cur`.

See Also

TOS, `v_enter_cur`, VDI

v_fillarea — VDI function (libvdi)

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 <aesbind.h>
```

```
#include <gemdefs.h>
```

```
#include <vdibind.h>
```

```
#define ASTERISK 3
```

```
#define CLICKS 1
```

```
#define BUTTON 1
```

```
#define BUTTONSTATE 1
```

```
#define ESC 0x1B
```

```
/* no. of clicks expected */
```

```
/* which button; 1=leftmost */
```

```
/* button state expected; 1=down */
```

```
/* 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 v_pmarker() */
```

```
int xymarker[2];
```

```
/* array used by v_fillarea() */
```

```
int xypoly[80];
```

```
/* array used by vs_clip() */
```

```
int cliparray[] = { 1, 1, 1, 1 };
```

```
/* arrays used by v_opvwk() */
```

```
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
```

```
int work_out[57];
```

```
/* throw-away declaration */
int nowhere = 0;

main()
{
    /* declarations used by evnt_multi() */
    int selection;                /* code for event */
    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 */
    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);
    v_opnvwk(work_in, &vdihandle, work_out);

    /* set clipping array to match screen dimensions */
    cliparray[2] = work_out[0];
    cliparray[3] = work_out[1];
    vs_clip(vdihandle, 1, cliparray);

    vsm_height(vdihandle, 3);
    vsm_type(vdihandle, ASTERISK);

    for(;;) {
        selection = evnt_multi(which, CLICKS, BUTTON,
                                BUTTONSTATE, 0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, buffer, 0, 0, &mousex, &mousey,
                                &nowhere, &nowhere, &key, &nowhere);

        switch(selection) {
            case MU_KEYBD:
                switch((char)key) {
                    case '\r':
                        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':
case 'T':
    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':
case 'S':
    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)

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)

See if a given pixel is set

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
void v_get_pixel(handle, x, y, flag, color)
```

```
int handle, x, y, *flag, *color;
```

v_get_pixel is a VDI routine that indicates whether a pixel is set. *handle* is the virtual device's VDI handle. *x* and *y* 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* indicates the color of the pixel, if it is set. For a table of color codes, see the entry for **v_opnwk**.

See Also

TOS, **VDI**

v_gtext — VDI function (libvdi)

Draw graphics text

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
void v_gtext(handle, x, y, text) int text, x, y; char *text;
```

v_gtext is a VDI routine that draws graphics text on the screen. *handle* is the virtual device's VDI handle. *x* and *y* 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, as follows:

vst_effects
vst_alignment
vst_rotation
vst_height

set special effects
 set text alignment
 set angle of rotation
 set text height

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 <aesbind.h>
#include <gemdefs.h>
#include <vdibind.h>

#define ESCAPE 0x1B /* ASCII code for <esc> */
#define RESERVED 0

/* 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, 1, 1 };

/* arrays used by drawline() */
xyvert[] = { 1, 1, 1, 1 };
xyhoriz[] = { 1, 1, 1, 1 };

/* 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, 1, 2 };
int work_out[57];

/* throw-away declaration */
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);
    v_opnvwk(work_in, &vdihandle, work_out);
```

```
/* set clipping area to match screen dimensions */
cliparray[2] = work_out[0];
cliparray[3] = work_out[1];
vs_clip(vdihandle, 1, cliparray);
drawtext(vdihandle);

/* set drawing arrays to match screen dimensions */
xyvert[0] = work_out[0]/2;
xyvert[1] = 1;
xyvert[2] = work_out[0]/2;
xyvert[3] = work_out[1];

xyhoriz[0] = 1;
xyhoriz[1] = work_out[1]/2;
xyhoriz[2] = work_out[0];
xyhoriz[3] = work_out[1]/2;

for(;;) {
    key = evt_keybd();
    switch((char)key) {
        case '\r':
            graf_mouse(M_ON, &nowhere);
            v_clsvwk(vdihandle);
            appl_exit();
            exit(0);

        case 'e':
        case 'E':
            vst_effects(vdihandle, effect);
            if (++effect > 32)
                effect = 1;
            drawtext(vdihandle);
            break;

        case 'h':
        case 'H':
            halign++;
            vst_alignment(vdihandle, (halign%3), (valign%6),
                &nowhere, &nowhere);
            /* legal H value 0-2, V value 0-5 */
            drawtext(vdihandle);
            break;

        case 'r':
        case 'R':
            /* Note: ST draws text only at right angles */
            angle += 900;
            if (angle > 3500)
                angle = 0;
            vst_rotation(vdihandle, angle);
            drawtext(vdihandle);
            break;
    }
}
```

```

        case 's':
        case 'S':
            vst_height(vdihandle, size, &nowhere,
                       &nowhere, &nowhere, &nowhere);
            /* character size in rasters */
            if (++size > 26)
                size = 1;
            drawtext(vdihandle);
            break;

        case 'v':
        case 'V':
            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, work_out[0]/2, work_out[1]/2, 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)

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)

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. To display the mouse pointer again, use **v_show_c**.

Example

For an example of this function, see the entry for **v_bar**.

See Also

TOS, **v_show_c**, VDI

Notes

Mixing VDI mouse calls with AES mouse calls can produce unpredictable results.

v_justified — VDI function (libvdi)

Justify graphics text

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
void v_justified(handle, x, y, string, length, charsp, wordsp)
```

```
int handle, x, y, length, charsp, wordsp; char *string;
```

v_justified is a VDI routine that justifies a string 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. *x* and *y* 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

v_meta_extents — VDI function (libvdi)

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)

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_opnvwk**.

handle is the device handle for the screen. Because the GEM desktop has already opened the screen device, you must use the AES routine **graf_handle** to obtain the VDI handle for the screen, as follows:

```
handle = graf_handle(cw, ch, bw, bh);
```

In this example, *handle* is the VDI handle, which is returned by **graf.handle**. *cw* and *ch* point to integers that hold, respectively, the width and height of a character to be displayed, and *bw* and *bh* point to integers that set, respectively, the width and height of a character cell in the screen device.

work_out is an array of 57 integers that are set by **v_opnwk**. 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

As of this writing, 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 one. To set device attributes, use the appropriate attribute functions, as listed in the entry for **VDI**.

v_opnwk — VDI function (libvdi)

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 physical workstation. This routine should be used to open all physical workstations *except* the screen, because the screen is opened by the GEM desktop 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 LMAGENTA
- 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 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 more information.

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. 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. GDOS is required to use NDC.

handle is the device's VDI handle, and is set by TOS. You can obtain the VDI handle with the AES function **graf_handle**. See the entry for **v_opnvwk** or **graf_handle** for more information.

work_out is an array of 57 integers that is filled in by **v_opnwk**, as follows:

- 0 width of device, in rasters (number of X coordinates)
- 1 height of device, in rasters (number of Y coordinates)
- 2 uses precision scaling? (0=yes, 1=no)
- 3 width of one pixel, microns
- 4 height of one pixel, microns
- 5 number of possible character heights (0=continuous scaling)
- 6 number of line types
- 7 number of possible line widths (0=continuous scaling)
- 8 number of marker types
- 9 number of possible marker sizes (0=continuous scaling)
- 10 number of text fonts available
- 11 number of fill styles available
- 12 number of cross-hatching styles available
- 13 number of colors that can be shown simultaneously
- 14 number of generalized drawing primitives (GDP's)
- 15-24 first 10 GDP'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 GDP 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 number of colors supported:
 0=more than 32,767; 1=monochrome; >2=number of colors
- 40 Cursor control devices: 1=keyboard only; 2=keyboard and
 mouse
- 41 number of mappable devices: 1=keyboard, 2=another device
- 42 number of choice devices: 1=function keys, 2=another key field
- 43 number 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 is not available in the ROM-resident VDI. 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)

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)

Draw a circular pie slice

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
void v_pieslice(handle, x, y, radius, beginangle, endangle)
```

```
int handle, x, y, radius, beginangle, endangle;
```

v_pieslice is a VDI routine that draws a circular pie slice. *handle* is the virtual device's VDI handle. *x* and *y* 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)

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. For the 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. *howmany* is the number of end points being created. *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 *howmany*. 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.

```
#include <aesbind.h>
#include <gemdefs.h>
#include <vdi-bind.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 & vs_clip */
int xyarray[] = { 1, 1, 1, 1 };
int cliparray[] = { 1, 1, 1, 1 };

/* 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 */
int nowhere = 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 */
    int button = 1; /* which button; 1=leftmost */
    int buttonstate = 1; /* button state; 1=down */
    int buffer[11]; /* place to write AES messages */

    int mousex; /* mouse X coordinate */
    int mousey; /* mouse Y coordinate */
    int vdihandle;

    /* OK, here we go ... */
    appl_init();
    graf_mouse(ARROW, &nowhere);
    v_opvwk(work_in, &vdihandle, work_out);
}
```



```
cliparray[0] = 0;
cliparray[1] = 1;
cliparray[2] = work_out[0];
cliparray[3] = work_out[1];
vs_clip(vdihandle, 1, cliparray);

vsl_width(vdihandle, 5);
vsl_type(vdihandle, 1);
vsl_ends(vdihandle, 1, 1);

for(;;) {
    selection = evnt_multi(which, clicks, button,
        buttonstate, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        buffer, 0, 0, &mousex, &mousey,
        &nowhere, &nowhere, &nowhere, &nowhere);

    switch(selection) {
        case MU_KEYBD:
            v_clswnk(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;
    }
}
```

See Also

TOS, VDI, vql_attributes, vsl_color, vsl_ends, vsl_type, vsludsty, vsl_width

v_pmarker — VDI function (libvdi)

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)

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 lets you use the mouse to draw rectangles on the screen.

```
#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 vs_clip() */
int cliparray[] = { 1, 1, 1, 1 };

/* array used by v_rbox() */
int xyarray[] = { 1, 1, 1, 1 };

/* 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 */
int nowhere = 0;
```

```
main()
{
    /* declarations used by evnt_multi() */
    int selection; /* code for event */
    unsigned int which = (MU_KEYBD | MU_BUTTON);
    int clicks = 1; /* no. of clicks expected */
    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 */

    /* open application */
    appl_init();

    /* turn mouse pointer to arrow */
    graf_mouse(ARROW, &nowhere);

    /* open screen device */
    v_opnvwk(work_in, &vdihandle, work_out);

    /* set clipping rectangle */
    cliparray[2] = work_out[0];
    cliparray[3] = work_out[1];
    vs_clip(vdihandle, 1, cliparray);

    /* set perimeter for drawn rectangle */
    vsf_perimeter(vdihandle, 1);

    for(;;) {
        /* wait for something to happen */
        selection = evnt_multi(which, clicks, button,
                               buttonstate, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                               buffer, 0, 0, &mousex, &mousey, &nowhere,
                               &nowhere, &key, &nowhere);
        switch(selection) {
            /* if keyboard is pressed, exit */
            case MU_KEYBD:
                v_clsvwk(vdihandle);
                appl_exit();
                exit(0);
        }
    }
}
```

```

/* if button is pressed, draw a rectangle */
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_rfbbox(vdihandle, xyarray);
    graf_mouse(M_ON, &nowhere);
    break;

default:
    break;
}
)
)

```

See Also

TOS, VDI, v_rfbbox

v_rfbbox — VDI function (libvdi)

Draw a filled, rounded rectangle

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
void v_rfbbox(handle, xyarray) int handle, xyarray[4];
```

v_rfbbox 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.

See Also

TOS, VDI, v_rbox

v_rmcur — VDI function (libvdi)

Remove last mouse pointer from the screen

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
void v_rmcur(handle) int handle;
```

v_rmcure is a VDI routine that removes the last mouse pointer from the screen. *handle* is the virtual device's VDI handle.

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)

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.

Example

For an example of this function, see the entry for **v_enter_cur**.

See Also

TOS, **v_rvon**, VDI

v_rvon — VDI function (libvdi)

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.

Example

For an example of this function, see the entry for **v_enter_cur**.

See Also

TOS, **v_rvoff**, VDI

v_show_c — VDI function (libvdi)

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` 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.

Example

For an example of this function, see the entry for `v_bar`.

See Also

TOS, `v_hide_c`, VDI

Notes

Mixing VDI mouse calls with AES mouse calls can produce unpredictable results.

v_updwk — VDI function (libvdi)

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 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 is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

v_write_meta — VDI function (libvdi)

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 in-

formation 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 is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

VDI — Technical information

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 (“bitblit”) 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, such as printers, plotters, video cameras, and the screen.

The screen device can, in addition, have one or more “virtual devices” output to it. A virtual device is a logical description of the screen that is handed to the screen driver for display.

Every time this virtual device is changed, the device driver updates the screen to reflect this change. More than one virtual device may be created for the screen. This allows you to create a series of “transparencies” that can be manipulated independent of each other and overlaid.

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 objects 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.

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. The NDC system can only be used with the VDI GDOS.

VDI components

The VDI 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.

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 physical 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 functions, 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 uses five global arrays of integers: `intin[]`, `intout[]`, `ptsin[]`, `ptsout[]`, and `ctrl[]`. The last of these should be declared as having 12 members; the others should each 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 `applinit` and close with `applexit`.

VDI library routines

The VDI library routines are declared in the header file `vdibind.h`, and are stored in the library `libvdi`. 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.

<code>v_arc</code>	draw a circular arc
<code>v_bar</code>	draw an outlined, filled rectangle
<code>v_bit_image</code>	print a bit-image file
<code>v_cellarray</code>	create an array of colored cells
<code>v_circle</code>	draw a circle
<code>v_clear_disp_list</code>	clear a printer's display list
<code>v_clrwk</code>	clear a virtual device
<code>v_clsvwk</code>	close the screen device
<code>v_clswk</code>	close a virtual device
<code>v_contourfill</code>	draw a filled polygon
<code>v_curdown</code>	move the text cursor down one row
<code>v_curhome</code>	move the text cursor to upper left corner
<code>v_curleft</code>	move the text cursor left one column
<code>v_curright</code>	move the text cursor right one column
<code>v_curtext</code>	write a string of text characters
<code>v_curup</code>	move the text cursor up one row
<code>v_dspcur</code>	move the mouse pointer to specified location
<code>v_eeol</code>	erase from text cursor to end of line
<code>v_eeos</code>	erase from text cursor to end of screen
<code>v_ellarc</code>	draw an elliptical arc
<code>v_ellipse</code>	draw an ellipse
<code>v_ellpie</code>	draw an elliptical pie segment
<code>v_enter_cur</code>	enter text mode
<code>v_exit_cur</code>	exit from text mode
<code>v_fillarea</code>	flood an enclosed area with fill pattern
<code>v_form_adv</code>	advance the page on a hard-copy device
<code>v_get_pixel</code>	find if a particular pixel has been set
<code>v_gtext</code>	output graphics text
<code>v_hardcopy</code>	dump virtual device to hard-copy device
<code>v_hide_c</code>	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_rfbbox	draw rectangular fill area with rounded corners
v_rmcur	remove last graphics cursor from screen
v_rvoff	turn off reverse video for character text
v_rvon	turn on reverse video for character 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	query 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
vsLudsty	set user-defined polyline style
vsL_width	set polyline width
vsm_choice	query choice devices, sample mode
vsm_color	set polymarker color
vsm_height	set polymarker height
vsm_locator	query locator devices, sample mode
vsm_string	query string devices, sample mode
vsm_type	set polymarker type
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 alignment
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 <vdi-bind.h>

#define BUTTON 1          /* which button; 1 = leftmost */
#define CENTER 1          /* indicates centering of text */
#define CENTERX 320       /* center of screen, X coord */
#define CENTERY 200       /* center of screen, Y coord */
#define CLICKS 1          /* no. of clicks expected */
#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 of timer; 5 ms. */
#define XOR 6             /* 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 */
/* solid black rectangle */
int shape[] = {
    0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
    0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
    0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
    0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF
};

/*
 * source form block */
FDB cat = { shape, 16, 16, 1, 0, 1, 0, 0, 0 };

/* target form block */
FDB screen = { 0L, 0, 0, 0, 0, 0, 0, 0, 0 };

/* initial position on screen */
int oldarray[] = {
    0, 0, 16, 16, CENTERX, CENTERY, CENTERX+16, CENTERY+16 };

/* new position on screen */
int newarray[] = {
    0, 0, 16, 16, CENTERX, CENTERY, CENTERX+16, CENTERY+16 };

/* throw-away declaration */
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 */
        Vsync();

        /* blit cat */
        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!" in big letters */
    write(stdout, "\07", 1);
    v_gtext(vdihandle, 320, 200, text);
    evt_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 wants another game ... */
/* ... clear screen again ... */
v_clrwk(vdihandle);
/* ... move cat to center of screen ... */
newarray[4] = oldarray[4] = CENTERX;
newarray[5] = oldarray[5] = CENTERY;
newarray[6] = oldarray[6] = (CENTERX + 16);
newarray[7] = oldarray[7] = (CENTERX + 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, exit */
v_clsvwk(vdihandle);
appl_exit();
exit(1);
}
}
```

See Also

AES, libvdi, Line A, metafile, TOS, vdibind.h

Notes

A RAM version of GDOS is now available, and it is shipped with several different products. At present, the standard VDI supports drivers only for the screen device, and acknowledges only raster coordinates.

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

vdibind.h — Header file

Declarations for VDI 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.

See Also

aesbind.h, 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 a version number 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 C that, when compiled 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.c
```

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 **.c**, so that the compiler will know that it is a C-source file. Also, be sure to include this file on the **cc** command line when you compile the program.

See Also

commands, msh

vertical tab — Character constant

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. The vertical tab character is white space; in particular, **isspace** returns "true" for **'\v'**.

See Also

ASCII, character constant

vex_butv — VDI function (libvdi)

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. 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**

Notes

If the button interrupt routine is executed in assembly language, note the following:

Invoke the application-dependent code with a **JSR** instruction. The register d0.w contains the mouse button keys. When complete, use the instruction **RTS**, with the mouse button state stored in d0.w.

vex_curv — VDI function (libvdi)

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**

Notes

If the cursor routine is executed in assembly language, note the following:

Invoke the application-dependent code with a **JSR** instruction. Upon entry to the routine, the registers d0.w and d1.w registers contain, respectively, the X and Y positions of the cursor. If the application-dependent code does not draw its own cursor, a **JSR** instruction should be performed to the address returned in **contrl[9]** and **contrl[10]** (which are initialized by the functions **v_opnwk** and **v_opnvwk**, with d0.w and d1.w holding, respectively, the X and Y positions at which to draw the cursor. This causes VDI to draw a cursor. When complete, perform an **RTS** instruction.

vex_motv — VDI function (libvdi)

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**

Notes

If the mouse-movement routine is executed in assembly language, note the following:

Invoke the application-dependent code with a **JSR** instruction. On entry, the registers d0.w and d1.w contain, respectively, the X and Y positions of the mouse. When complete, execute a **JSR** instruction, with d0.w and d1.w holding, respectively, the X and Y positions of the mouse.

vex_timv — VDI function (libvdi)

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.

If the timer interrupt routine is written in assembly language, invoke the application-dependent code via a **JSR** routine. When finished, execute an **RTS** instruction.

vm_filename — VDI function (libvdi)

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 al-

phabetic string that is terminated with NUL.

See Also

TOS, v_meta_extent, v_write_meta, VDI

Notes

This routine is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

void — C keyword

Data type

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.

See Also

C keywords, C language, declarations

volatile — C keyword

Qualify an identifier as frequently changing

The type qualifier **volatile** marks an identifier as being frequently changed, either by other portions of the program, by the hardware, by other programs in the execution environment, or by any combination of these. This alerts the translator to re-fetch the given identifier whenever it encounters an expression that includes the identifier. In addition, an object marked as **volatile** must be stored at the point where an assignment to this object takes place.

See Also

C keyword, const

Notes

Although Mark Williams C recognizes this keyword, the semantics are not implemented in this release. Thus, storage declared to be **volatile** might have references removed by optimizations that the compiler performs. The compiler will generate a warning if the peephole optimizer is enabled and the keyword **volatile** is detected.

vq_cellarray — VDI function (libvdi)

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*, **rowused*, **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. 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-hand 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 points to an integer that holds the number of horizontal cells used in each row. *rowused* points to the number of rows in the array that were actually used. *status* points to 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 is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

vq_chcells — VDI function (libvdi)

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* point, respectively, to 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];
```

```
/* print an alert box on the screen */
```

```
alertf(n, p) int n; char *p;
```

```
{
```

```
    static char buffer[512];  
    sprintf(buffer, "%r", &p);  
    return form_alert(n, buffer);
```

```
}
```

```
main()
```

```
{
```

```
    int nowhere = 0;  
    int vdihandle;  
    int rows;  
    int columns;
```

```
    appl_init();  
    v_opvwk(work_in, &vdihandle, work_out);
```

```
    vq_chcells(vdihandle, &rows, &columns);  
    alertf(1, "[1] [Rows: %d] [Columns: %d] [OK]", rows, columns);
```

```
    v_clsvwk(vdihandle);  
    appl_exit();  
    exit(0);
```

```
}
```

See Also
TOS, VDI

vq_color — VDI function (libvdi)

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* is 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)

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* point to integers into which the function will write, respectively, the row and the column in which the text cursor is positioned.

Example

For an example of this function, see the entry for **v_enter_cur**.

See Also
TOS, VDI, **vs_curaddress**

vq_extnd — VDI function (libvdi)

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 inquiry 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 *work_out*, 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*)

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)

Check mouse position and button state

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
void vq_mouse(handle, status, x, y) int handle, *status, *x, *y;
```

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. *x* and *y* 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)

Find if graphics tablet is available

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
int 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 the graphics tablet: zero indicates that the tablet is not available, and one indicates that it is.

See Also

TOS, VDI

Notes

This routine is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

vqf_attributes — VDI function (libvdi)

Read the area fill's current attributes


```
#include <aesbind.h>
#include <vdibind.h>
void vqL_attributes(handle, attrib) int handle, attrib[5];
```

vqL_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:

<i>attrib</i> [0]	Fill type. For a table of fill types, see the entry for vsf_interior .
<i>attrib</i> [1]	Fill color. For a table of color codes, see the entry for v_opnwk .
<i>attrib</i> [2]	Fill pattern. For a table of fill patterns, see the entry for vsf_style .
<i>attrib</i> [3]	Writing mode: one indicates replace mode; two, transparent mode; three, XOR (exclusive or) mode; and four, reverse transparent mode.
<i>attrib</i> [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)

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* points to an integer that will hold the current 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)

Read the polyline's current attributes

```
#include <aesbind.h>
#include <vdibind.h>
void vqL_attributes(handle, attrib) int handle, attrib[6];
```

vqlAttributes 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, **vql_attributes**, **vqm_attributes**, **vqt_attributes**

vqm_attributes — VDI function (libvdi)

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)

Inquire if an error occurred with the Polaroid Palette

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
int vqp_error(handle) int handle;
```

The VDI supports a driver for the Polaroid Palette, a camera that shoots color transparencies. When the driver is loaded, the VDI routine **vqp_error** 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_inmessage*, *vsp_save*, *vsp_style*

Notes

This routine is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

vqp_films — VDI function (libvdi)

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 supports a driver for the Polaroid Palette, a camera that shoots color transparencies. When the driver is loaded, the VDI routine **vqp_films** returns the names of the five 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 is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

vqp_state — VDI function (libvdi)

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, *planes, *indices[8][2];
```

The VDI supports a driver for the Polaroid Palette, a camera that shoots color transparencies. When the driver is loaded, the VDI routine *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 is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

vqt_attributes — VDI function (libvdi)

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)

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)

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. 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:

<i>sizes[0]</i>	bottom line to baseline
<i>sizes[1]</i>	descent line to baseline
<i>sizes[2]</i>	half line to baseline
<i>sizes[3]</i>	ascent line to baseline
<i>sizes[4]</i>	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:

<i>adjust[0]</i>	increase in character width
<i>adjust[1]</i>	left offset
<i>adjust[2]</i>	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)

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_opnvwk** 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];

/* show an alert box on the screen */
alertf(n, p) int n; char *p;
{
    static char buffer[512];
    sprintf(buffer, "%r", &p);
    return form_alert(n, buffer);
}

main()
{
    int nowhere = 0;
    int vdihandle;
    int info[32]; /* array used by vqt_name */
    int i;

    /* open application */
    appl_init();
    v_opnvwk(work_in, &vdihandle, work_out);
```

```

/* return code and description of all screen fonts */
for (i=1; i <= work_out[10]; i++)
    alertf(1, "[1] [Font %d|%s] [OK]",
           vqt_name(vdihandle, i, info), info);

/* clscse device, exit */
v_clsvwk(vdihandle);
appl_exit();
exit(0);
)

```

See Also

TOS, VDI, *vst_font*

vqt_width — VDI function (libvdi)

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.

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)

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 <time.h>
#include <vdibind.h>

#define USER 4                /* 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, 1, 1 };

/* arrays used by v_opnvwk() */
int work_in[] = { 1, 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 */
int nowhere = 0;

main()
{
    int vdihandle;                /* virtual device's handle */
    int key;

    /* open application */
    appl_init();

    /* open screen device */
    v_opnvwk(work_in, &vdihandle, work_out);

    /* set clipping array */
    xyarray[2] = work_out[0];
    xyarray[3] = work_out[1];
    vs_clip(vdihandle, 1, xyarray);
```

```

/* hide the mouse pointer; set interior to user-defined */
v_hide_c(vdihandle);
vsf_interior(vdihandle, USER);

dofill(vdihandle);

for(;;) {
    key = evnt_keybd();
    if ((char)key == '\r') {
        v_show_c(vdihandle);
        v_clsvwk(vdihandle);
        appl_exit();
        exit(1);
    } else
        dofill(vdihandle);
}

dofill(vdihandle)
int vdihandle;
{
    int counter;

    /* seed random-number routine with system time */
    srand((int)time(&nowhere));

    /* fill buffer with random numbers */
    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)

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 it is dealing with the output of the physical device, 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 be set to the same value in both *sourcemfdb* and *destmfdb*. *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)

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 (erasc 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, *sourcemfd* and *destmfd* point to the “memory form definition blocks” 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 be set to the same value in both *sourcemfdb* and *destmfdb*. *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 example lets you copy one portion of the screen to another. When you click the mouse the first time, you draw a rectangle on the screen; clicking the mouse again lets you 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 BUTTON 1
#define CLICKS 1
#define DOWN 1
#define FUJI 4
#define HOLLOW 0
#define REPLACE 1
#define RETURN 0x0D
#define W_KEY 0x77
#define XOR 3

/* which button; 1=leftmost */
/* no. of clicks expected */
/* mouse button is down */
/* "fuji" fill pattern */
/* make fill type hollow */
/* make writing mode REPLACE */
/* code for <return> */
/* code for W key */
/* 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() */
int cliparray[] = { -1, 1, 1, 1 };

/* 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[] = { 1, 1, 1, 1 };

/* arrays used by vro_cpyfm() */
int copyarray[8];

/* throw-away declaration */
int nowhere = 0;

main()
{
    /* declarations used by evt_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 */
    int mousex;                   /* mouse X coordinate */
    int mousey;                   /* mouse Y coordinate */

    /* misc declarations */
    int vdihandle;                /* virtual device's handle */
    int logic = 3;                /* logic type */
    FDB holder = { 0L, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }; /* used by vro_cpyfm; all zeros */

    /* open application */
    appl_init();

    /* set mouse pointer to arrow */
    graf_mouse(ARROW, &nowhere);

    /* open screen device */
    v_opnvwk(work_in, &vdihandle, work_out);

    /* set clipping array */
    cliparray[2] = work_out[0];
    cliparray[3] = work_out[1];
    vs_clip(vdihandle, 1, cliparray);

    /* set interior of rectangle to Atari "fuji" */
    vsf_interior(vdihandle, FUJI);

    /* turn on rectangle perimeter */
    vsf_perimeter(vdihandle, 1);

    /* turn off mouse pointer */
    graf_mouse(M_OFF, &nowhere);

    /* draw rectangle; turn on pointer again */
    xyarray[0] = work_out[0]/3;
    xyarray[1] = work_out[1]/3;
    xyarray[2] = work_out[0]/3;
    xyarray[3] = work_out[1]/3;
    v_bar(vdihandle, xyarray);
    graf_mouse(M_ON, &nowhere);
}

```

```

for(;;) {
    /* wait for use to do something */
    selection = evnt_multi(which, CLICKS, BUTTON,
        DOWN, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        buffer, 0, 0, &mousex, &mousey,
        &nowhere, &nowhere, &key, &nowhere);

    switch(selection) {
        /* if keyboard is touched ... */
        case MU_KEYBD:
            /* if <return>, exit */
            if ((char)key == RETURN) {
                v_clsvwk(vdihandle);
                appl_exit();
                exit(0);
            }
            /* if "W" is pressed, change pattern */
            if ((char)key == W_KEY)
                logic++;
            break;

        /* if mouse button is pressed, draw rubberbox */
        case MU_BUTTON:
            getarray(vdihandle, mousex, mousey);
            v_bar(vdihandle, xyarray);
            vswr_mode(vdihandle, REPLACE);

            graf_mouse(M_OFF, &nowhere);
            /* do the blitting */
            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 */
    int height;               /* box height */
    int newx;                  /* X coordinate */
    int newy;                  /* Y coordinate */

    /* 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)

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 jargon, 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)

Find location of mouse cursor when a key is pressed

```
#include <aesbind.h>
```



```
#include <vdibind.h>
void vrq_locator(handle, x, y, xout, yout, key)
int handle, x, y, *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. *x* and *y* 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, you must write a loop if you want to terminate on a particular key.

See Also

TOS, VDI, **vsm_locator**

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)

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 jargon, 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)

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	<i>key</i> plus ten
Cursor down	<i>key</i> minus ten
Shift/cursor up	<i>key</i> plus one
Shift/cursor down	<i>key</i> 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)

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[2]; 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.

color is an array of two integers that set the color indices: *color*[0] sets the index for the foreground color, and *color*[1] sets the index for the background color. The color indices are 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

sourcemfdb and *destmfdb* point to the "memory form definition blocks 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 be set to the same value in both *sourcemfdb* and *destmfdb*. *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)

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:

<i>xyarray[0]</i>	X coordinate of first corner
<i>xyarray[1]</i>	Y coordinate of first corner
<i>xyarray[2]</i>	X coordinate of opposite corner
<i>xyarray[3]</i>	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)

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)

Move text 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 text 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 text cursor.

Example

For an example of this function, see the entry for `v_enter_cur`.

See Also

TOS, VDI, `vq_curaddress`

vs_palette — VDI function (libvdi)

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)

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.

Once the new shape is loaded with `vsc_form`, it can be called with `graf_mouse(USER_DEF, form)`. The header file `gemdefs.h` must be included to use this call. `graf_mouse` is another way to set the mouse to a user-defined form; it is implemented using `vsc_form`.

See Also

TOS, VDI

vsf_color — VDI function (libvdi)

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)

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 (erased, set to color 0) |
| 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)

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)

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 diagonal "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)

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)

Set input mode for logical input device

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
int vsin_mode(handle, device, mode) int handle, device, mode;
```

vsln_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.

vsln_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)

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)

Attach ends to a line

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

void vslends(handle, beginning, end) int handle, beginning, end;

vslends 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)

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, **vsludsty**

vsludsty – VDI function (libvdi)

Set user-defined line type

#include <aesbind.h>

#include <vdibind.h>

void vsludsty(handle, pattern) int handle, pattern;

vsludsty 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 **vsludsty**, it must be loaded with the function **vsl_type**.

See Also

TOS, **v_pline**, VDI, **vsl_type**

vsl_width — VDI function (libvdi)

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)

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)

Set a polymarker'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)

Set a polymarker's height

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
int vsm_height(handle, height) int handle, height;
```

vsm_height is a VDI routine that set the height of a polymarker. *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)

Return mouse pointer's position

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
int vsm_locator(handle, x, y, xout, yout, key)
```

```
int handle, x, y, *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. *x* and *y* 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 `vsin_mode(handle, 1, 2)` must be entered, which will place the locator device into sample mode.

vsm_string — VDI function (libvdi)

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)

Set polymarker's 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 polymarker 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)

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 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	<i>key</i> plus ten
Cursor down	<i>key</i> minus ten
Shift/cursor up	<i>key</i> plus one
Shift/cursor down	<i>key</i> 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)

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)

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)

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)

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)

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 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)

Set special effects for graphics text

```
#include <aesbind.h>
```

```
#include <vdibind.h>
```

```
int vst_effects(handle, effects) int handle, effects;
```

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)

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 examining 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 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)

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)

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 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 is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

vst_point — VDI function (libvdi)

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. It resets the characters into printer's points; one point equals 1/72 of an inch. The related function **vst_height** resets character height, but uses absolute values rather than points.

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)

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 entry 36 of the array *work_out[]*, which is returned by **v_opnwk** or **v_opnvwk**. Zero indicates no, and one indicates yes.

As of this writing, the Atari ST can rotate text only in 90-degree increments.

vst_unload_fonts — VDI function (libvdi)

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)

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 ex-

clusive 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

W

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.

L W C

See Also

commands

while — C keyword

Introduce a loop

while(*condition*)

while is a C keyword that introduces a conditional loop. *condition* is tested on reiteration of the loop, and the loop ends when *condition* is no longer satisfied. For example,

```
while (foo < 10)
```

introduces a loop that will continue until the variable **foo** is reset to ten or greater. Note that the statement

```
while (1)
```

will loop forever, unless interrupted by a **break**, **goto**, or **return** statement.

See Also

break, C keywords, C language, **continue**, **do**, **for**

The C Programming Language, page 56

while — Command

Execute a conditional loop

while *word1 word2*

while is a command built into the microshell, **msh**. It controls the operation of conditional loops: as long as **word1** executes successfully, **word2** is executed.

while is often used with the test commands **equal** and **not**.

See Also

commands, equal, if, is_set, msh, not

wildcards — Definition

Wildcards are characters that, under special circumstances, can represent a range of ASCII characters. Another name for them is “metacharacters”. The wildcards available under **msh** are as follows:

- ? Match any one character.
- * Match any number of characters, or no characters at all.
- [] 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 all lower-case letters or [0-9] for all numerals.
- / Remove the special meaning of a wildcard.

See Also

egrep, Ffirst, msh, patterns, pnmach

wind_calc — AES function (libaes)

Calculate a window's rectangle

```
#include <aesbind.h>
```

```
int wind_calc(type, kind, x, y, w, h, xptr, yptr, wptr, hptr)
```

```
int type, x, y, w, h, *xptr, *yptr, *wptr, *hptr; unsigned int kind;
```

wind_calc is an AES routine that calculates the rectangle of a window. If *type* is zero, then *x*, *y*, *w*, and *h* specify the working rectangle of the window (the area within which text or icons are written), and **wind_calc** computes the total rectangle of the window (the entire area the window occupies on the screen). *x*, *y*, *w*, and *h* stand, respectively, for the X coordinate of the window's upper left-hand corner, the Y coordinate of the upper left-hand corner, the width, and the height. All are given in pixels.

If *type* is one, then *x*, *y*, *w*, and *h* specify the total rectangle of the window, and **wind_calc** computes the working rectangle of the window.

The computed rectangle is stored into the integers pointed to by *xptr*, *yptr*, *wptr*, and *hptr*.

kind lists the “gadgets” that appear in the window. The gadgets must be coded as follows:

0x001	NAME	title name
0x002	CLOSER	"close" bar
0x004	FULLER	"full" box
0x008	MOVER	"move" bar
0x010	INFO	information line
0x020	SIZER	"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"

If this list is not complete, then **wind_calc** will not calculate the rectangle correctly.

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)

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. A closed window is not visible on the screen, but 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)

Create a window

```
#include <aesbind.h>
```

```
int wind_create(kind, x, y, w, h) unsigned int kind; int x, y, w, h;
```

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	CLOSER	"close" bar
0x004	FULLER	"full" box
0x008	MOVER	"move" bar
0x010	INFO	information line
0x020	SIZER	"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).

x, *y*, *w*, and *h* give, respectively, the window's X coordinate, its Y coordinate, its width, and its height. The X and Y coordinates are always for the window's upper left-hand corner. These values are returned by the function **wind_get** when given the request **WF_FULLED**. These values are also often used to reset the size of the window when the "full" box is clicked, but nothing requires that this be done.

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)

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)

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 **graf_mkstate**. **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)

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. A window's handle is first set by the routine **wind_create**, and is passed to your program via messages generated by the window manager.

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 value of WF_CURRXYWH
7	WF_FULLXYWH	window's greatest possible size (set by wind_create)
8	WF_HSLIDE	<i>output1</i> set to relative position of horizontal slider (1-1,000; 1=leftmost)
9	WF_VSLIDE	<i>output1</i> set to relative position of vertical slider (1-1,000; 1=top)
10	WF_TOP	<i>output1</i> 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	<i>output1</i> set to size of horizontal slider relative to scroll bar; -1 is minimal (small box), 1-1,000 is relative size
16	WF_VSLSIZE	<i>output1</i> set to size of vertical slider relative to scroll bar; -1 is minimal (small box), 1-1,000 is relative size

When used with the macros named above, **wind_get** returns zero if an error occurred, and a number greater than zero if one did not. However, when used with the macros **WF_NAME**, **WF_INFO**, or **WF_NEWDESK**, all of which may be used with the function **wind_set**, **wind_get** always returns one, which indicates success, but it returns no useful information.

Example

For an example of this routine, see the entry for **window**.

See Also

AES, TOS, window

wind_open — AES function (libaes)

Open or reopen a window

```
#include <aesbind.h>
```

```
int wind_open(handle, x, y, w, h) int handle, x, y, w, h;
```

wind_open is an AES routine that opens or reopens a window. *handle* is the window's handle, as set by **wind_create**. *x*, *y*, *w*, and *h* give, respectively, the X coordinate of the window to be opened, its Y coordinate, its width, and its height.

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)

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:
<i>input1</i> and <i>input2</i> give address
passed as two-word pointer |
| 3 | WF_INFO | Point to new information line for window:
<i>input1</i> and <i>input2</i> give address
passed as two-word pointer |
| 5 | WF_CURRXYWH | Window's total area:
<i>input1</i> gives X coordinate
<i>input2</i> gives Y coordinate
<i>input3</i> gives width
<i>input4</i> gives height |
| 8 | WF_HSLIDE | <i>input1</i> set to relative position of
horizontal slider (1-1,000; 1=leftmost) |
| 9 | WF_VSLIDE | <i>input1</i> set to relative position of
vertical slider (1-1,000; 1=top) |
| 10 | WF_TOP | <i>handle</i> gives handle of window
to be topmost |
| 14 | WF_NEWDESK | Address of new default GEM desktop:
<i>input1</i> and <i>input2</i> give address
passed as two-word pointer
<i>input3</i> gives starting object in tree |
| 15 | WF_HSLSIZE | Set size of horizontal slider
<i>input1</i> gives size of slider, 1-1,000 |
| 16 | WF_VSLSIZE | Set size of vertical slider
<i>input1</i> gives size of slider, 1-1,000 |

Ordinarily, **wind_set** returns zero if an error occurred, and a number greater than zero if it was successful. Note the following exceptions: When used with the mac-

ros **WF_PREVXYWH** or **WF_FULLXYWH**, which can be used with the function **wind_get**, **wind_set** returns success but does nothing. When the values used with **WF_VSLIDE** or **WF_HSLIDE** are out of bounds (i.e., greater than 1,000 or less than one), **wind_set** returns success and sets the slider to the closest legal value.

Example

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

See Also

AES, **TOS**, **window**

Notes

The window manager does not make its own copies of the strings specified by **WF_NAME** or **WF_INFO**. Thus, changing either of these strings after they are passed with **wind_set** will change the appearance of the window as well.

wind_update — AES function (libaes)

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 and keep extraneous material from being drawn over the window as it is being redrawn. *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_MCTRL	End mouse control through user: lock window
3	BEG_MCTRL	Begin mouse control through user: unlock

From the time that

```
wind_update(BEG_MCTRL);
```

is called, the AES suspends mouse event handling until the matching

```
wind_update(END_MCTRL);
```

is called. **graf_mkstate** may be used to poll the mouse state while **BEG_MCTRL** is active. The menu bar, any displayed window gadgets, any mouse events requested via **envt_mouse**, **envt_button**, or **envt_multi** and any dialogues requested through **form_do** will be inactive until **END_MCTRL** is encountered. This is useful if a desk accessory wishes to perform a dialogue over another application's dialogue screen without activating the application dialogue.

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 — Technical information

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 an integer *handle* for the window, which can be used to identify it to all other functions. 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 be 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. If you fail to set the **WF_NAME** or **WF_INFO** after creating a window, random text may be written into those areas. All the rest of the attributes may be left to default values without hazard.

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. Word 0 is the type of message being sent. Word 1 is the handle of the application that

sends the message. Word 2 is the number of bytes in the message beyond the standard 16 bytes. Normally, a program that handles windows does not need to examine word 1 and 2. Words 3 through 7 give information specific to the message.

The following gives the types of messages that are relevant to handling windows:

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's 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 lowest position, and 1,000 the highest.

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.

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 full 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`. 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 AES organizes the interior of each window 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 is more than one window 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 redrawn 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, &x, &y, &w, &h);
```

and the dimensions of the next rectangle with this call:

```
wind_get(handle, WF_NEXTXYWH, &x, &y, &w, &h);
```

AES returns zero for the width and height when it reaches the end of its 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, sliders, 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 randomly defined mask. Clicking either slider redraws the mask.

```
#include <aesbind.h>
#include <gemdefs.h>
#include <obdefs.h>
#include <osbind.h>

#define MAXWIND      4
#define GRAIN 64
#define KIND (0xFFF&~INFO) /* Everything but INFO */

typedef struct { int x, y, w, h; } Box;

/* C note: a macro can fill an array as well as call a function */
#define elements(r) r.x, r.y, r.w, r.h
#define pointers(r) &r.x, &r.y, &r.w, &r.h

int aes_handle;
int vdi_handle;
```

```

int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];
int contrl[12], intin[128], intout[128], ptsin[128], ptsout[128];

OBJECT mask = { -1, -1, -1, G_BOX, LASTOB, NORMAL,
                0x11C1L, 0, 0, 640, 400 };

main()
{
    register int w;      /* Current window handle */
    register int nw;     /* Current window count */
    int mb[8];          /* Message buffer */
    Box Full;           /* Filled area of window */
    Box Prev;           /* Previous area of window */
    Box Work;           /* Working area of window */
    Box Temp;           /* Miscellaneous Box */
    int nowhere; /* Unused pointers point here */

    aes_handle = appl_init();
    graf_mouse(ARROW, &nowhere);

    /* alter size of mask for resolution of screen */
    if (Getrez() < 2) /* i.e., screen is medium/low res */
        mask.ob_height = 200;
    if (Getrez() < 1) /* i.e., screen is low res */
        mask.ob_width = 320;

    /* summon the VDI; set some ways it works */
    v_oprvwk(work_in, &vdi_handle, work_out);
    vswr_mode(vdi_handle, 1); /* "replace" mode */
    vsf_perimeter(vdi_handle, 0); /* no perimeter on shapes */
    vsf_interior(vdi_handle, 4); /* user-defined fill */

    /* get size of work area for window 0 (desktop) */
    wind_get(0, WF_FULLXYWH, pointers(Full));

    /* mask the screen with grey */
    objc_draw(&mask, ROOT, 0, elements(Full));

    /* create, set, and open windows */
    for (nw = 0; nw < MAXWIND; nw += 1) {
        /* some variables used only in this block */
        static char *ordinal[] = { "st", "nd", "rd", "th" };
        static char t[MAXWIND][32];

        Temp = Full;
        if ((w = wind_create(KIND, elements(Temp))) < 0)
            break;
        Temp.w /= 2; Temp.x += rand() % Temp.w;
        Temp.h /= 2; Temp.y += rand() % Temp.h;
        sprintf(t[nw], "%dxs window", nw+1, ordinal[nw%3 ? 3 : nw]);

        /* set the "gadgets" on each window */
        wind_set(w, WF_NAME, t[nw], 0, 0);

        /* "sliders" set to a random number */
        wind_set(w, WF_HSLIDE, rand() % 1000, 0, 0, 0);
        wind_set(w, WF_VSLIDE, rand() % 1000, 0, 0, 0);
    }
}

```

```
/* set slider sizes relative to scroll bar */
wind_set(w, WF_HSLSIZE, Temp.w, 0, 0, 0);
wind_set(w, WF_VSLSIZE, Temp.h, 0, 0, 0);

/* draw window with "star wars" effect */
graf_growbox(1,1,1,1, elements(Temp));
wind_open(w, elements(Temp));
}

/* clicking a "gadget" generates a message */
while (nw > 0) {
    /* mb[3] gives handle of window */
    evnt_mesag(mb); w = mb[3];

    /* mb[0] holds the message */
    switch(mb[0]) {

        /* redraw window */
        case WM_REDRAW:
            redraw:
                /* get settings of window from window manager */
                wind_get(w, WF_HSLIDE, pointers(Temp)); Prev.x=Temp.x;
                wind_get(w, WF_VSLIDE, pointers(Temp)); Prev.y=Temp.y;
                wind_get(w, WF_HSLSIZE, pointers(Temp)); Prev.w=Temp.w;
                wind_get(w, WF_VSLSIZE, pointers(Temp)); Prev.h=Temp.h;
                wind_get(w, WF_WORKXYWH, pointers(Work));

                graf_mouse(M_OFF, &nowhere); /* Hide mouse */
                wind_update(BEG_UPDATE); /* Lock window */
                wind_get(w, WF_FIRSTXYWH, pointers(Temp));

                /* "walk the rectangles" to redraw window interior */
                while (Temp.w || Temp.h) {
                    if (Temp.w && Temp.h)
                        display(&Prev, &Work, &Temp);
                    wind_get(w, WF_NEXTXYWH, pointers(Temp));
                }

                wind_update(END_UPDATE); /* Unlock window */
                graf_mouse(M_ON, &nowhere); /* Show mouse */
                continue;

        /* make window the "top", or active, window */
        case WM_TOPPED:
            wind_set(w, WF_TOP, 0, 0, 0, 0);
            continue;

        /* have window fill the whole screen */
        case WM_FULLED:
            wind_get(w, WF_PREVXYWH, pointers(Prev));
            wind_get(w, WF_CURRXYWH, pointers(Temp));
            wind_get(w, WF_FULLXYWH, pointers(Full));
```

```

        if (rc_equal(&Prev, &Full))
            continue;
        wind_set(w, WF_CURRXYWH, elements(Prev));
    } else {
        wind_set(w, WF_CURRXYWH, elements(Full));
    }
}

resize:
wind_get(w, WF_WORKXYWH, pointers(Temp));
wind_set(w, WF_HSLSIZE, Temp.w, 0, 0, 0);
wind_set(w, WF_VSLSIZE, Temp.h, 0, 0, 0);
continue;

/* resize window */
case WM_SIZED:
    Temp = *(Box *) (mb+4);
    if (w & 1) {
        wind_calc(1, KIND, elements(Temp),
            pointers(Temp));
        Temp.w -= Temp.w % GRAIN;
        if (Temp.w < 2*GRAIN) Temp.w = 2*GRAIN;
        Temp.h -= Temp.h % GRAIN;
        if (Temp.h < 2*GRAIN) Temp.h = 2*GRAIN;
        wind_calc(0, KIND, elements(Temp),
            pointers(Temp));
    }

    wind_get(w, WF_CURRXYWH, pointers(Prev));
    if (rc_equal(&Temp, &Prev))
        continue;
    wind_set(w, WF_CURRXYWH, elements(Temp));
    goto resize;

/* window moved on screen */
case WM_MOVED:
    wind_set(w, WF_CURRXYWH, mb[4], mb[5], mb[6], mb[7]);
    continue;

/* horizontal slider clicked */
case WM_HSLID:
    wind_get(w, WF_HSLIDE, pointers(Temp));
    if (Temp.x == mb[4])
        continue;
    wind_set(w, WF_HSLIDE, mb[4], 0, 0, 0);
    goto redraw;

/* vertical slider clicked */
case WM_VSLID:
    wind_get(w, WF_VSLIDE, pointers(Temp));
    if (Temp.x == mb[4])
        continue;
    wind_set(w, WF_VSLIDE, mb[4], 0, 0, 0);
    goto redraw;

```

```
/* arrow or scroll bar clicked */
case WM_ARROWED:
(
    /* Note parens: limits variables to this block */
    static int dc[] = { -5, 5, -1, 1 };
    register int t;

    /*
     * mb[4] holds action: 0=page up, 1=page down,
     * 2=row up, 3= row down, 4=page left, 5=page
     * right, 6=column left, 7=column right
     */
    t = mb[4];

    if (t <= 3) {
        wind_get(w, WF_VSLIDE, pointers(Temp));
        t = dc[t] + Temp.x;
        if (t > 1000) t = 1000;
        if (t < 0) t = 0;
        if (t == Temp.x)
            continue;
        wind_set(w, WF_VSLIDE, t, 0, 0, 0);
    } else if (t <= 7) {
        wind_get(w, WF_HSLIDE, pointers(Temp));
        t = dc[t-4] + Temp.x;
        if (t > 1000) t = 1000;
        if (t < 0) t = 0;
        if (t == Temp.x)
            continue;
        wind_set(w, WF_HSLIDE, t, 0, 0, 0);
    } else
        continue;
    goto redraw;
}

/* close window */
case WM_CLOSED:
    wind_get(w, WF_CURRXYWH, pointers(Temp));
    wind_close(w);
    graf_shrinkbox(1,1,1,1, elements(Temp));
    wind_delete(w);
    nw -= 1;
    continue;

/* unanticipated message */
default:
    alertf(1, "[0] [Message %u received] [okay]", mb[0]);
    continue;
}

}

/* close everything, tidy up, exit gracefully */
v_clswk(vdi_handle);
appl_exit();
exit(0);
}
```

```

/* print an alert box */
alertf(n, p) int n; char *p;
{
    static char buffer[512];
    sprintf(buffer, "%r", &p);
    return form_alert(n, buffer);
}

/* redraw the screen */
display(lp, wp, rp)
Box *lp, *wp, *rp;
/* lp == logical page offsets and sizes 0 .. 999 */
/* wp == working area of window */
/* rp == clipped Box to fill */
{
    Box T; /* VDI temporary box */
    Box U; /* Second temporary box */
    int pattern[16]; /* User-defined pattern */
    register int x, y; /* Working-area scanner */

    T = *rp; T.w += T.x-1; T.h += T.y-1;

    /* set clipping rectangle */
    vs_clip(vdi_handle, 1, &T);

    for (x = 0; x < wp->w; x += GRAIN)
        for (y = 0; y < wp->h; y += GRAIN) {
            T.x = wp->x + x; T.w = GRAIN;
            T.y = wp->y + y; T.h = GRAIN;
            U = *rp;

            if (! rc_intersect(&T, &U))
                continue;

            /* fill window with randomly generated pattern */
            make_pattern(lp->x+(x/GRAIN), lp->y+(y/GRAIN), pattern);
            vsf_udpat(vdi_handle, pattern, 1);
            T.w += T.x-1; T.h += T.y-1;
            v_bar(vdi_handle, &T);
        }
}

/* create a pattern for filling each window */
make_pattern(x, y, texture)
register unsigned x, y, *texture;
{
    register unsigned d, z, dz;

    /* fill char array with prime numbers */
    static unsigned char dzs[] = {
        1,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,
        1,67,71,73,79,83,89,97,101,103,107,109,113,127,
        31,137,139,149,151,157,163,167,173,179,181,191,
        93,197,199,211,223,227,229,233,239,241,251
    };
}

```



```

/* randomly select a prime number */
srand(y*x);
dz = dzs[rand() % sizeof(dzs)];

/* perform magic with primes
 * to tile interior of window
 */
d = x % 100;
x /= 100;
if (x & 1) d = 100 - d;
d = (d * 256L) / 100;

for (y = 0; y < 16; y += 1)
    texture[y] = 0;

for (; d > 0; d -= 1) {
    for (z = rand(); ; z += dz) {
        x = 1 << ((z >> 4) & 15);
        y = z & 15;
        if ((texture[y] & x) == 0)
            break;
    }
    texture[y] ^= x;
}
}

```

See Also

AES, gem, gemdefs.h, object, TOS

write — UNIX system call (libc)

Write to a file

int write(*fd*, *buffer*, *n*)

int *fd*; char **buffer*; int *n*;

write writes *n* bytes of data, beginning at address *buffer*, into the file *fd*. 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 -1 if an error occurred before the **write** operation commenced, such as a bad file descriptor *fd* 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.

X

xbios — TOS function

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:

24	Bioskeys	restore the default keyboard table
64	Blitmode	get/set blitter mode
21	Cursconf	set the cursor's configuration
32	Dosound	pass data to the sound daemon
10	Flopfmt	format a floppy disk
8	Floprd	read a floppy disk
19	Flopver	verify a floppy disk
9	Flopwr	write to a floppy disk
4	Getrez	read the current screen resolution
23	Gettime	read the current system time
28	Giaccess	write to the GI sound chip registers
25	Ikbdws	send commands to the intelligent keyboard
0	Initmous	initialize the mouse
14	Iorec	get a pointer to the serial device input record
26	Jdisint	disable an interrupt
27	Jenabint	enable an interrupt
16	Keytbl	create a new keyboard table
34	Kbdvbase	get a pointer to a set of keyboard routines
35	Kbrate	set the keyboard's repeat rate
3	Logbase	get the screen's logical base
13	Mfpint	initialize interrupt routine in multi-function port
12	Midiws	send string to musical instrument digital interface
29	Offgibit	turn off a bit in the sound chip's A port
30	Ongibit	turn on a bit in the sound chip's A port
2	Physbase	get the physical base of the screen
18	Protobt	create a prototype boot routine
36	Prtblk	print a dump of the screen
39	Puntaes	make AES go away
17	Random	generate a pseudo-random number
15	Rsconf	configure the RS-232 (serial) port

20	Scrdmp	print a dump of the screen
7	SetColor	set a color
5	Setscreen	set the screen parameters
6	Setpalette	set the color palette
33	Setprt	configure the printer port
22	Settime	set the system time
38	Supexec	run a function under supervisor mode
37	Vsync	synchronize with the screen refresh
31	Xbtimer	initialize a timer on the multi-function port

See Also

osbind.h, **TOS**

Notes

No **xbios** function checks device numbers. Passing an invalid device number to one will crash the system.

xbios and **bios** traps can be nested to a level of three deep. This occurs either when an interrupt-level routine calls an **xbios** or **bios** function while an **xbios** or **bios** function is executing, or when an **xbios** or **bios** function itself traps to the **xbios** or **bios**. A dangerous situation may occur if an **xbios** or **bios** function is called by a routine that is executed by an interrupt handler or can be invoked asynchronously. In these situations, the level of nesting can quickly exceed the limit of three.

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

Declare **xbios** constants and structures

```
#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.

Example

This example sets the timer to a value (an ambiguous time) and loops that many times while incrementing a memory location, then does the same with a register variable.

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 Xbtimer.prg Xbtimer.c setrte.s
```

The following gives the text of `Xbtimer.c`:

```
#include <osbind.h>
unsigned long a_lock;

void timetick()                /* the clock interrupt handler */
{
    setrte();                  /* make this an int return */
    a_lock++;                  /* increment the lock */
    Xbtimer(0, 0, 0, 0L);      /* disable the timer */
}

main()
{
    unsigned long tick=0;      /* slow counter */
    register unsigned long tock=0; /* fast counter */

    a_lock = 0;                /* clear the timer flag */
    Xbtimer(0, 7, 100, timetick); /* declare the timer */
    while ( a_lock == 0 )      /* do the memory loop */
        tick++;                /* print statistics */
    printf("Using memory, loop executed %ld times.\n", tick);

    a_lock = 0;                /* clear flag */
    Xbtimer(0, 7, 100, timetick); /* declare timer */
    while ( a_lock == 0 )      /* do register loop */
        tock++;                /* then print stats. */
    printf("Using register, loop executed %ld times.\n", tock);
    Pterm0();                  /* exit */
}
```

See Also

TOS, `xbios`

XOFF — Manifest constant

XOFF is a flow-control signal used with asynchronous communications. Usually, it consists of a `<ctrl-S>` character (octal 023). It 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
ASCII, XON

XON — Manifest constant

XON is a flow-control signal used with asynchronous communications. Usually, it consists of a **<ctrl-Q>** character (octal 021). It 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
ASCII, XOFF

Permuted Listing of Lexicon Entries

AES function (libaes):

appl_exit	Exit from an application	182
appl_find	Get another application's handle	182
appl_init	Initiate an application	182
appl_read	Read a message from another application	183
appl_tplay	Replay AES activity	183
appl_trecord	Record user actions	183
appl_write	Send a message to another application	184
evnt_button	Await a specific mouse button event	320
evnt_dclick	Get/set double-click interval	321
evnt_keybd	Await a keyboard event	321
evnt_mesag	Await a message	322
evnt_mouse	Wait for mouse to enter specified rectangle	324
evnt_multi	Await one or more specified events	325
evnt_timer	Wait for a specified length of time	328
form_alert	Display an alert box	361
form_center	Center an object on the screen	362
form_dial	Reserve/free screen space for dialogue	362
form_do	Handle user input in form dialogue	363
form_error	Display a TOS error	364
fsel_input	Select a file	375
graf_dragbox	Draw a draggable box	398
graf_growbox	Draw a growing box	399
graf_handle	Get a VDI handle	400
graf_mbox	Move a box	400
graf_mkstate	Get the current mouse state	401
graf_mouse	Change the shape of the mouse pointer	401
graf_rubbox	Draw a rubber box	403
graf_shrinkbox	Draw a shrinking box	403
graf_slidebox	Track the slider within a box	404
graf_watchbox	Draw a watched box	406
menu_bar	Show or erase the menu bar	481

menu_ichk.	Write or erase a check mark next to a menu item.	481
menu_ienable	Enable or disable a menu item	481
menu_register.	Add a name to the desk accessory menu list	482
menu_text	Replace text of a menu item	482
menu_tnormal.	Display menu title in normal or reverse video	483
objc_add.	Redefine a child object within an object tree	507
objc_change	Change object's state	507
objc_delete	Delete an object from an object tree	508
objc_draw	Draw an object	508
objc_edit.	Edit a text object	509
objc_find	Find if mouse pointer is over particular object	509
objc_offset	Calculate an object's absolute screen position	510
objc_order	Reorder a child object within the object tree	510
rc_copy.	Copy a rectangle	555
rc_equal	Compare two rectangles.	556
rc_intersect.	Check if two rectangles intersect	556
rc_union.	Calculate overlap between two rectangles	557
rsrc_free.	Free memory allocated to a set of resources	576
rsrc_gaddr	Get the address of a resource object	576
rsrc_load	Load a resource file into memory	577
rsrc_obfix	Change the form of an object's coordinates	577
rsrc_saddr.	Store address of a free string or a bit image	578
scrp_read	Read the scrap directory	584
scrp_write.	Write to the scrap directory	585
shel_envrn	Search for an environmental variable	596
shel_find	Search BPATHR for file name	598
shel_read	Let an application identify the program that called it.	598
shel_write.	Tell desktop which application to run next	598
wind_calc	Calculate a window's rectangle	769
wind_close	Close a window and preserve its handle	770
wind_create	Create a window	770
wind_delete	Delete a window and free its resources	771
wind_find	Determine if the mouse pointer is in a window	772
wind_get.	Get information about a window	772
wind_open	Open or reopen a window	773
wind_set.	Set specified fields within the window	774
wind_update	Lock or unlock a window.	775
Archive:		
galaxy.a		382
me.a		471
rdy.a		565
bios function:		
Bconin	Receive a character	219
Bconout	Send a character to a peripheral device	220
Bconstat.	Return the input status of a peripheral device	220

Bcstat.	Read the output status of a peripheral device	221
Drvmap	Get a map of the logical disk drives.	304
Getbpb.	Get pointer to BIOS parameter block for a disk drive.	385
Getmpb	Copy memory parameter block	388
Getshift	Get or set the status flag for shift/alt/control keys	392
Mediach	Check whether disk has been changed.	471
Rwabs	Read or write data on a disk drive	579
Setexc	Get or set an exception vector.	588
Tickcal.	Return system timer's calibration.	640
C keyword:		
auto.	Note an automatic variable.	214
break.	Exit from loop or switch statement.	226
case.	Introduce entry in switch statement	240
char.	Data type	255
const.	Qualify an identifier as not modifiable.	263
continue.	Force next iteration of a loop	263
default.	Default label in switch statement.	291
do.	Introduce a loop	300
double.	Data type	303
else.	Introduce a conditional statement.	313
entry.	Undefined keyword	315
enum.	Declare a type and identifiers	315
extern.	Declare storage class.	331
float.	Data type	350
for.	Control a loop.	360
goto.	Unconditionally jump within a function	397
if.	Introduce a conditional statement.	411
int.	Data type	416
long.	Data type	450
readonly.	Storage class	566
register.	Storage class	567
return.	Return a value and control to calling function	571
short.	Data type	600
sizeof.	Return size of a data element	602
static.	Declare storage class.	610
struct.	Data type	624
switch.	Test a variable against a table.	629
typedef.	Define a new data type	658
union.	Multiply declare a variable	661
unsigned.	Data type	664
void.	Data type	722
volatile.	Qualify an identifier as frequently changing	722
while.	Introduce a loop	768
Character constant:		
backspace.		218

carriage return	239
horizontal tab	409
line feed	445
newline	503
NUL	506
vertical tab	719
Command:	
ar	The librarian/archiver 185
as68toas	Convert Motorola assembler 205
as	Assembler for Atari ST 189
cat	Concatenate files 241
cc	Compiler controller 243
cd	Change directory 253
chmod	Change the modes of a file 257
cmp	Compare bytes of two files 259
cp	Copy a file 265
cpp	C preprocessor 265
cursorconf	Set the cursor's configuration 273
date	Print/set the date and time 277
db	Assembler-level symbolic debugger 278
df	Measure free space on disk 296
diff	Summarize differences between two files 299
drtomw	Convert from DRI to Mark Williams format 303
drvprs	Check if a drive is present on the machine 304
echo	Repeat/expand an argument 309
egrep	Extended pattern search 310
equal	Compare two arguments 318
exit	Exit from a BmshR shell 329
file	Name a file's type 347
gem	Run a GEM program 382
getcol	Get a color value 387
getpal	Get the color palette settings 389
getphys	Get the base of the physical screen's display 390
getrez	Get screen's current resolution 390
help	Print concise description of command 408
hidemouse	Hide the mouse pointer 409
htom	Redraw screen from high to medium resolution 410
if	Execute a command conditionally 411
inherit	Pass variable to child shell 415
is_set	Check if an environmental variable is set 418
kbrate	Reset the keyboard's repeat rate 429
kick	Force TOS to reread the disk cache 433
lc	List directory's contents in columnar format 434
ld	Link relocatable object files 434
ls	List directory's contents 451

ltom	Redraw the screen from low to medium resolution . . .	453
make	Program building discipline	455
me	MicroEMACS screen editor	463
mf	Measure space left in RAM.	486
mkdir	Create a directory	490
mousehidden	Return how often mouse pointer has been hidden . . .	492
msh	492
mshversion	Print current version of BmshR.	500
msleep	Stop executing for a specified time	500
mtoh	Redraw the screen from medium to high resolution . .	501
mtol	Redraw the screen from medium to low resolution . .	501
mv	Rename files or directories	501
mwto mw	Convert objects to Mark Williams 3.0 format.	502
nm	Print a program's symbol table	503
not	Invert logical value of an argument.	504
od	Print a hexadecimal dump of a file	520
pr	Paginate and print files	538
pwd	Print the name of the current directory	550
rdy	Create, save, and load rebootable RAM disk	557
rescomp	Resource compiler	568
resdecomp	Resource decompiler.	569
resource	Invoke the resource editor	570
rmdir	Remove directories.	572
rm	Remove files.	572
rsconf	Configure the serial port	575
setcol	Reset a color	586
setenv	Set an environmental variable.	587
setpal	Reset the color palette.	590
setphys	Reset physical screen's display space.	591
setprt	Reset the printer port.	591
setrez	Reset the screen resolution.	592
set	Set an BmshR variable	585
show	Display a stored screen image	600
showmouse	Redisplay the mouse pointer.	601
size	Print the size of an object module	602
sleep	Stop executing for a specified time	603
snap	Save a screen image	603
sort	Sort lines of text	604
strip	Strip tables from executable file.	619
tail	Print the end of a file	636
time	Print current time/time execution of a command . . .	645
time	Time the execution of a command	640
tos	Execute GEM-DOS program.	652
touch	Update modification time of a file.	655
uniq	Remove/count repeated lines in a sorted file	662

unset	Discard a shell variable	664
unsetenv	Discard an environmental variable	664
version	Print/create a version string	718
wc	Count words, lines, and characters in files	768
while	Execute a conditional loop	768
cctype macro (ctype.h):		
_tolower	Convert letter to lower case	651
_toupper	Convert letter to upper case	655
isalnum	Check if a character is a number or letter.	418
isalpha	Check if a character is a letter.	419
isascii	Check if a character is an ASCII character	419
isctrl	Check if a character is a control character	420
isdigit	Check if a character is a numeral.	420
islower	Check if a character is a lower-case letter.	420
isprint	Check if a character is printable.	421
ispunct	Check if a character is a punctuation mark.	421
isspace	Check if a character prints white space	421
isupper	Check if a character is an upper-case letter.	422
toascii	Convert characters to ASCII	650
Debugging macro:		
assert	Check assertion at run time	210
Definition:		
address		177
alignment		181
arena		186
argc	Argument passed to main	187
argv	Argument passed to main	187
array		188
ASCII		206
auto		214
BIOS		222
bit		223
bit map		224
boot		226
buffer		227
byte		228
cast		240
cc0		249
cc1		249
cc2		249
cc3		249
compound number		262
daemon		276
directory		300
environ		316

environment	316
envp Argument passed to main	317
executable file	328
field	347
file	347
file descriptor	349
FILE Descriptor for a file stream	348
flexible arrays	350
fraction	367
function	380
GMT	396
handle	408
interrupt	416
lvalue	453
macro	455
manifest constant	462
mantissa	462
modulus	491
n.out	505
nested comments	503
nybble	506
object format	520
operator	523
path	526
pattern	528
pointer	535
port	537
precedence	539
process	544
pun	548
random access	554
ranlib	554
rational number	555
read-only memory	566
real number	567
record	567
register	568
register variable	568
rvalue	579
stack	607
standard error	608
standard input	608
standard output	608
stderr	610
stdin	610

stdout	612
stream	616
structure	624
wildcards	769
Environmental variable:	
HOME	409
INCDIR	414
LIBPATH	Directories that hold libraries 440
PATH	Directories that hold executable files 527
SUFF	625
TIMEZONE	Time zone information 646
TMPDIR	649
Example:	
example	Give an example of Mark Williams Lexicon format . . . 172
picture	Format numbers under mask 533
External data:	
_end	314
_stksize	613
maxmem	463
gemdos function:	
Cauxin	Read a character from the serial port 241
Cauxis	Check if characters are waiting at serial port. 242
Cauxos	Check if serial port is ready to receive characters 243
Cauxout	Write a char to the serial port. 243
Cconin	Read a character from the standard input 250
Cconis	Find if a character is waiting at standard input 250
Cconos	Check if console is ready to receive characters 251
Cconout	Write a character onto standard output 252
Cconrs	Read and edit a string from the standard input 252
Cconws	Write a string onto standard output 253
Cncin	Perform modified raw input from standard input 259
Cprnos	Check if printer is ready to receive characters 267
Cprnout	Send a character to the printer port 267
Crawcin	Read a raw character from standard input 268
Crawio	Perform raw I/O with the standard input. 268
Dcreate	Create a directory 288
Ddelete	Delete a directory 289
Dfree	Get information on a drive's free space 297
Dgetdrv	Find current default disk drive 298
Dgetpath	Get the current directory name 298
Dsetdrv	Make a drive the current drive 305
Dsetpath	Set the current directory 306
Fattrib	Get and set file attributes 332
Fclose	Close a file 333
Fcreate	Create a file 334

Fdatetime	Get or set a file's date/time stamp	337
Fdelete	Delete a file	338
Fdup	Generate a substitute file handle	340
Fforce	Force a file handle	342
Fgetdta	Get a disk transfer address	343
Fopen	Open a file	360
Fread	Read a file	367
Frename	Rename a file	368
Fseek	Move a file pointer	373
Fsetdta	Set disk transfer address	378
Fsfirst	Search for first occurrence of a file	378
Fsnext	Search for next occurrence of file name	379
Fwrite	Write into a file	381
Malloc	Allocate dynamic memory	461
Mfree	Free allocated memory	487
Mshrink	Shrink amount of allocated memory	500
Pexec	Load or execute a process	530
Pterm0	Terminate a TOS process	547
Ptermres	Terminate a process but keep it in memory	547
Pterm	Terminate a process	547
Super	Enter privilege mode	625
Version	Get the version number of TOS	628
Tgetdate	Get the current date	638
Tgettime	Get the current time	639
Tsetdate	Set a new date	656
Tsettime	Set a new time	658
General function (libc):		
abort	End program immediately	173
abs	Return the absolute value of an integer	173
access	Check if a file can be accessed in a given mode	174
atof	Convert ASCII strings to floating point	212
atoi	Convert ASCII strings to integers	213
atol	Convert ASCII strings to long integers	213
calloc	Allocate dynamic memory	238
ecvt	Convert floating-point numbers to strings	309
exit	Terminate a program	329
fcvt	Convert floating point numbers to ASCII strings	336
free	Return dynamic memory to free memory pool	368
frexp	Separate fraction and exponent	370
fstat	Find file attributes	379
gcvt	Convert floating point number to ASCII string	382
getenv	Read environmental variable	388
isatty	Check if a device is a terminal	419
lcalloc	Allocate dynamic memory	434
ldexp	Combine fraction and exponent	437

lmalloc	Allocate dynamic memory	446
longjmp	Return from a non-local goto	450
lrealloc	Reallocate dynamic memory	451
malloc	Allocate dynamic memory	459
mktemp	Generate a temporary file name	490
modf	Separate integral part and fraction	490
notmem	Check if memory is allocated	504
path	Build a path name for a file	526
peekb	Extract a byte from memory	528
peekl	Extract a long from memory	528
peekw	Extract a word from memory	529
perror	System call error messages	529
pokeb	Insert a byte into memory	536
pokel	Insert a long into memory	536
pokew	Insert a long into memory	537
qsort	Sort arrays in memory	552
rand	Generate pseudo-random numbers	553
realloc	Reallocate dynamic memory	567
sbrk	Increase a program's data space	580
setjmp	Perform non-local goto	589
shellsort	Sort arrays in memory	599
srand	Seed random number generator	606
stat	Find file attributes	609
swab	Swap a pair of bytes	629
system	Pass a command to TOS for execution	630
tempnam	Generate a unique name for a temporary file	637
tmpnam	Generate a unique name for a temporary file	649
tolower	Convert characters to lower case	650
toupper	Convert characters to upper case	655
Header file:		
access.h	Define manifest constants used by access()	175
aesbind.h	Declare GEM AES routines	181
assert.h	Define assert()	211
basepage.h	Define TOS base page structure	218
bios.h	Declare bios constants and structures	223
canon.h	Canonical conversion for the 68000	239
ctype.h	Header file for data tests	273
errno.h	Error numbers used by errno()	319
gemdefs.h	GEM structures and definitions	383
gemout.h	GEM-DOS file formats and magic numbers	385
linea.h	Declare Atari line A routines	445
math.h	Declare mathematics functions	462
mtype.h	List processor code numbers	501
nout.h	Describe output format Bn.outR	505
obdefs.h	Declare TOS objects and structures	507

osbind.h	Declare TOS functions	525
path.h	Declare path().	527
setjmp.h	Define setjmp() and longjmp().	589
signal.h	Define Atari ST signals	601
stat.h	Definitions and declarations used to obtain file status	610
stdio.h	Declarations and definitions for I/O	612
time.h	Give time-description structure	646
vdibind.h	Declarations for VDI routines	718
xbios.h	Declare xbios constants and structures	787
Introduction:		
Lexicon		437
Library:		
libaes	GEM AES bindings	439
libc		439
libm		440
libvdi	GEM VDI bindings.	440
Linker-defined symbol:		
edata		310
end		314
etext		320
Manifest constant:		
CLK_TCK		258
EOF		317
NULL		506
XOFF		788
XON		789
Mathematics function (libm):		
acos	Calculate inverse cosine.	176
asin	Calculate inverse sine	210
atan2	Calculate inverse tangent.	212
atan	Calculate inverse tangent.	211
cabs	Complex absolute value function	234
ceil	Set numeric ceiling	254
cos	Calculate cosine.	264
cosh	Calculate hyperbolic cosine.	264
exp	Compute exponent.	330
fabs	Compute absolute value.	332
floor	Set a numeric floor	353
hypot	Compute hypotenuse of right triangle	410
j0	Compute Bessel function	423
j1	Compute Bessel function	424
jn	Compute Bessel function	425
log10	Compute common logarithm.	449
log	Compute natural logarithm	448
pow	Compute a power of a number	538

sin	Calculate sine	601
sinh	Calculate hyperbolic sine	601
sqrt	Compute square root	605
tan	Calculate tangent.	636
tanh	Calculate hyperbolic cosine.	636
Operating system device:		
aux	Logical device for serial port	216
con	Logical device for the console	263
prn:	TOS logical device for parallel port	544
Overview:		
C keywords		230
C language		230
character constant		255
commands		260
ctype		271
declarations		290
header file		408
library		440
mathematics library		462
runtime startup		578
STDIO		611
string		617
time		641
TOS		652
UNIX routines		662
Preprocessor instruction:		
#assert	Check assertion at compile time.	211
#define	Define a variable as manifest constant.	291
#elif	Include code conditionally	312
#else	Include code conditionally	313
#endif	End conditional inclusion of code	314
#ifdef	Include code conditionally	412
#if	Include code conditionally	411
#ifndef	Include code conditionally	413
#include	Copy a header file into a program	414
#line	Reset line numbering	441
#undef	Undefine a manifest constant	660
Runtime startup:		
crt0.o	Default C runtime startup	269
crtsd.o	C runtime startup, GEM environment.	270
crtsg.o	C runtime startup, GEM environment.	270
STDIO function (libc):		
fclose	Close stream	333
fopen	Open a stream for standard I/O.	338
fflush	Flush output stream's buffer.	341

fgetc	Read character from stream	342
fgets	Read line from stream	345
fgetw	Read integer from stream	346
fileno	Get file descriptor	349
fopen	Open a stream for standard I/O	358
fprintf	Print formatted output onto file stream	365
fputc	Write character onto file stream	365
fputs	Write string to file stream	366
fputw	Write an integer to a stream	366
fread	Read data from file stream	367
freopen	Open file stream for standard I/O	369
fscanf	Format input from a file stream	371
fseek	Seek on file stream	372
ftell	Return current position of file pointer	380
fwrite	Write onto file stream	381
gets	Read string from standard input	391
getw	Read word from file stream	394
printf	Format output	540
puts	Write string to standard output	550
rewind	Reset file pointer	571
scanf	Accept and format input	580
setbuf	Set alternative stream buffers	586
sprintf	Format output	605
sscanf	Format input	606
ungetc	Return character to input stream	660
STDIO macro (stdio.h):		
clearerr	Present stream status	257
feof	Discover stream status	340
ferror	Discover stream status	340
getchar	Read character from standard input	387
getc	Read character from file stream	386
putchar	Write a character to standard output	549
putc	Write character to stream	548
putw	Write word to stream	550
String function (libc):		
index	Find a character in a string	415
memchr	Search a region of memory for a character	472
memcmp	Compare two regions	472
memcpy	Copy one region of memory into another	473
memset	Fill an area with a character	476
pnmatch	Match string pattern	534
rindex	Find a character in a string	571
strcat	Append one string to another	614
strchr	Find a character in a string	614
strcmp	Compare two strings	615

strcpy	Copy one string into another.	615
strcspn	Length one string excludes characters in another	615
strerror	Translate an error number into a string.	616
strlen	Measure the length of a string.	619
strncat	Append one string onto another.	619
strncmp	Compare two strings.	620
strncpy	Copy one string into another.	620
strpbrk	Find first occurrence in a string.	622
strrchr	Search for rightmost occurrence of a character.	622
strspn		623
strstr	Find one string within another	623
Technical information:		
AES		177
bombs	68000 processor exceptions.	225
byte ordering		228
calling conventions		234
data formats		276
data types		276
desk accessory		292
error codes		319
keyboard		430
Line A		441
main	Introduce program's main function.	455
memory allocation		473
menu		476
metafile		483
object		511
portability		537
screen control		583
storage class		614
structure assignment		624
system variables		632
type checking		658
type promotion		659
VDI		710
window		776
Time function (libc):		
asctime	Convert time structure to ASCII string	209
clock	Get number of clock ticks since system boot	258
ctime	Convert system time to an ASCII string.	271
dayspermonth	Return number of days in a given month.	278
difftime	Return difference between two times	300
gmtime	Convert system time to calendar structure	397
isleapyear	Indicate if a year was a leap year	420
jday_to_time	Convert Julian date to system time	424

jday_to_tm	Convert Julian date to system calendar format	424
Kgettext	Read time from intelligent keyboard's clock	432
Ksettime	Set time in intelligent keyboard's clock	433
localtime	Convert system time to calendar structure	446
Sgettext	Read time from intelligent keyboard's clock	596
stime	Set the operating system time	612
tetd_to_tm	Convert IKBD time to system calendar format	637
time_to_jday	Convert system time to Julian date	646
time	Get current time	641
tm_to_jday	Convert calendar format to Julian time	648
tm_to_tetd	Convert system calendar format to IKBD time	649
TOS function:		
bios	Call an input/output routine in the TOS BIOS	222
gemdos	Call a routine from GEM-DOS	383
xbios	Call a routine from the extended TOS BIOS	786
UNIX data:		
errno	External integer for return of error status	318
UNIX system call (libc):		
_exit	Terminate a program	329
chdir	Change working directory	256
chmod	Change file protection modes	256
chown	Change ownership of a file	257
close	Close a file	258
creat	Create/truncate a file	269
dup2	Duplicate a file descriptor	308
dup	Duplicate a file descriptor	307
execve	Execute a command from within a program	328
lseek	Set read/write position	452
open	Open a file	522
read	Read from a file	566
unlink	Remove a file	663
write	Write to a file	784
VDI function (libvdi):		
v_arc	Draw a circular arc	665
v_bar	Draw a rectangle	665
v_bit_image	Print a bit image file	668
v_cellarray	Draw a table of colored cells	669
v_circle	Draw a circle	669
v_clear_disp_list	Clear a printer's display list	672
v_clrwk	Clear the virtual workstation	672
v_clswwk	Close the screen virtual device	673
v_clswk	Close a virtual workstation	673
v_contourfill	Fill an outlined area	674
v_curdown	Move text cursor down one row	677
v_curhome	Move text cursor to the home position	677

v_curleft	Move text cursor left one column	677
v_currigh	Move text cursor right one column	678
v_curtext	Write alphabetic text	678
v_curup	Move text cursor up one row	678
v_dspcur	Move mouse pointer to point on screen	679
v_eeol	Erase text from cursor to end of screen	679
v_eeos	Erase from text cursor to end of screen	679
v_ellarc	Draw an elliptical arc	680
v_ellipse	Draw an ellipse	683
v_ellpie	Draw an elliptical pie slice	685
v_enter_cur	Enter text mode	686
v_exit_cur	Exit from text mode	688
v_fillarea	Draw a complex polygon	689
v_form_adv	Advance the page on a printer	692
v_get_pixel	See if a given pixel is set	692
v_gtext	Draw graphics text	692
v_hardcopy	Write the screen to a hard-copy device	695
v_hide_c	Hide the mouse pointer	696
v_justified	Justify graphics text	696
v_meta_extents	Update extents header of metafile	697
v_opnvwk	Open the virtual screen device	697
v_opnwk	Open a virtual workstation	698
v_output_window	Dump a portion of a virtual device to a printer	702
v_pieslice	Draw a circular pie slice	702
v_pline	Draw a line	702
v_pmarker	Draw a marker	704
v_rbox	Draw a rounded rectangle	705
v_rfbbox	Draw a filled, rounded rectangle	707
v_rmcur	Remove last mouse pointer from the screen	707
v_rvoff	End reverse video for alphabetic text	708
v_rvon	Display alphabetic text in reverse video	708
v_show_c	Show the mouse cursor	708
v_updwk	Update a virtual workstation	709
v_write_meta	Write a metafile item	709
vex_butv	Set new button interrupt routine	719
vex_curv	Set new cursor interrupt routine	720
vex_motv	Set new mouse movement interrupt routine	720
vex_timv	Set new timer interrupt routine	721
vm_filename	Rename a metafile	721
vq_cellarray	Return information about cell arrays	723
vq_chcells	Find how many characters virtual device can print	724
vq_color	Check/set color intensity	725
vq_curaddress	Get the text cursor's current position	725
vq_extnd	Perform extend inquire of VDI virtual device	725
vq_key_s	Check control key status	726

vq_mouse	Check mouse position and button state	727
vq_tabstatus	Find if graphics tablet is available	727
vqf_attributes	Read the area fill's current attributes	727
vqin_mode	Determine mode of a logical input device	728
vql_attributes	Read the polyline's current attributes	728
vqm_attributes	Read the marker's current attributes	729
vqp_error	Inquire if an error occurred with the Polaroid Palette	730
vqp_films	Get films supported by driver for Polaroid Palette.	730
vqp_state	Read current settings of the Polaroid Palette driver.	731
vqt_attributes	Read the graphic text's current attributes.	731
vqt_extent	Calculate a string's length	732
vqt_fontinfo	Get information about special effects for graphics text	733
vqt_name	Get name and description of graphics text font	734
vqt_width	Get character cell width.	735
vr_recfl	Draw a rectangular fill area	735
vr_trnfm	Transform a raster image.	737
vro_cpyfm	Copy raster form, opaque.	738
vrq_choice	Return status of function keys when any key is pressed	743
vrq_locator	Find location of mouse cursor when a key is pressed	743
vrq_string	Read a string from the keyboard	744
vrq_valuator	Return status of shift and cursor keys	745
vrt_cpyfm	Copy raster form, transparent.	745
vs_clip	Set the virtual device's clipping rectangle	747
vs_color	Set color intensity	748
vs_curaddress	Move text cursor to specified row and column	748
vs_palette	Select color palette on medium-resolution screen	749
vsc_form	Draw a new shape for the mouse pointer	749
vsf_color	Set a polygon's fill color.	750
vsf_interior	Set a polygon's fill type	750
vsf_perimeter	Set whether to draw a perimeter around a polygon.	750
vsf_style	Set a polygon's fill style.	751
vsf_udpat	Define a fill pattern	752
vsin_mode	Set input mode for logical input device	752
vsl_color	Set a line's color	753
vsl_ends	Attach ends to a line.	753
vsl_type	Set a line's type.	754
vsl_udsty	Set user-defined line type.	754
vsl_width	Set a line's width.	755
vsm_choice	Return last function key pressed	755
vsm_color	Set a polymarker's color	756
vsm_height	Set a polymarker's height	756
vsm_locator	Return mouse pointer's position	757
vsm_string	Read a string from the keyboard	757
vsm_type	Set polymarker's type	758
vsm_valuator	Return shift/cursor key status.	759

vsp_message	Suppress messages from Polaroid Palette device.	760
vsp_save	Save to disk current setting of Polaroid Palette driver .	760
vsp_state	Set the Polaroid Palette driver.	760
vst_alignment	Realign graphics text	761
vst_color	Set color for graphics text	762
vst_effects	Set special effects for graphics text	762
vst_font	Select a new font.	763
vst_height	Reset graphics text height, in absolute values	763
vst_load_fonts	Load fonts other than the standard font.	764
vst_point	Reset graphics text height, in printer's points	765
vst_rotation	Set angle at which graphic text is drawn	765
vst_unload_fonts	Unload fonts.	766
vswr_mode	Set the writing mode	766
xbios function:		
Bioskeys	Reset the keyboard to its default	223
Blitmode	Get/set blitter configuration	224
Cursconf	Get or set the cursor's configuration	274
Dosound	Start up the sound daemon	301
Flopfmt	Format tracks on a floppy disk	353
Floprd	Read sectors on a floppy disk	356
Flopver	Verify a floppy disk	357
Floppwr	Write sectors on a floppy disk	358
Getrez	Read the current screen resolution.	390
Gettime	Read the current time.	393
Giaccess	Access a register on the GI sound chip	394
Ikbdws	Write a string to the intelligent keyboard device.	413
Initmous	Initialize the mouse	415
Iorec	Set the I/O record	417
Jdisint	Disable interrupt on multi-function peripheral device . .	425
Jenabint	Enable a multi-function peripheral port interrupt. . . .	425
Kbdvbase	Return a pointer to the keyboard vectors	427
Kbrate	Get or set the keyboard's repeat rate.	429
Keytbl	Set the keyboard's translation table	431
Logbase	Read the logical screen's display base	449
Mfpint	Initialize the MFP interrupt	486
Midiws	Write a string to the MIDI port	488
Offgibit	Clear a bit in the sound chip's A port	521
Ongibit	Turn on a bit in the sound chip's A port	521
Physbase	Read the physical screen's display base	531
Protobt	Generate a prototype boot sector	544
Prtblk	Print a dump of the screen.	545
Puntaes	Disable AES.	548
Random	Generate a 24-bit pseudo-random number	553
Rsconf	Configure the serial port	573
Scrdmp	Print a dump of the screen.	582

Setcolor	Set one color	586
Setpalette	Set the screen's color palette.	590
Setprt	Get or set the printer's configuration	591
Setscreen	Set the video parameters	592
Settime	Set the current time.	593
Supexec	Run a function under supervisor mode	626
Vsync	Synchronize with the screen.	767
Xbtimer	Initialize the MFP timer	787

Index

!	494	<ctrl-V>	52
#assert	211	<ctrl-W>	57
#define	291	<ctrl-X>	66
#elif	312	<ctrl-X>	83
#else	313	<ctrl-X>!	81
#endif	314	<ctrl-X>(. . .	80
#if	411	<ctrl-X>).	80
#ifdef	412	<ctrl-X>1.	73, 76
#ifndef	413	<ctrl-X>2.	75
#include	414	<ctrl-X><	83
#line	441	<ctrl-X><ctrl-B>	73
#undef	660	<ctrl-X><ctrl-C>	53, 56
\$	14, 494	<ctrl-X><ctrl-F>	71
\$*	95	<ctrl-X><ctrl-N>	77
\$<	95	<ctrl-X><ctrl-P>	77
\$?	95	<ctrl-X><ctrl-R>	71
\$@	95	<ctrl-X><ctrl-S>	53
&&	496	<ctrl-X><ctrl-V>	71
*	23, 37, 494	<ctrl-X><ctrl-W>	66, 70
-	92, 98	<ctrl-X><ctrl-Z>	77
/	495	<ctrl-X>>	83
;	496	<ctrl-X>B	78
<ctrl-@>	57	<ctrl-X>E	80
<ctrl-A>	51	<ctrl-X>F	60
<ctrl-B>	51	<ctrl-X>K	74
<ctrl-C>	81	<ctrl-X>N	76
<ctrl-D>	54	<ctrl-X>P	76
<ctrl-E>	51	<ctrl-X>Z	76
<ctrl-F>	51	<ctrl-Y>	56
<ctrl-G>	64	<ctrl-Z>	66
<ctrl-L>	59	<ctrl>	48
<ctrl-N>	52		55
<ctrl-P>	52	<esc>	48
<ctrl-T>	59	<esc>!	77
<ctrl-U>	68	<esc>%	64
		<esc>2	83
		<esc><	53
		<esc>	55
		<esc>>	53
		<esc>?	83
		<esc>B	51
		<esc>C	58
		<esc>D	55
		<esc>F	51
		<esc>L	59
		<esc>R	63

<code><esc>S</code>	62	<code>\$*</code>	29, 498
<code><esc>U</code>	59	<code>\$<</code>	30, 498
<code><esc>V</code>	53	aliases	498
<code><return></code>	51, 63	aliases	29
<code>?</code>	22, 494	aliasing	498
<code>[</code>	494	alignment	181
<code>[]</code>	22	altering stack size	39
<code>]</code>	494	<code>appl_exit</code>	182
<code>__end</code>	314	<code>appl_find</code>	182
<code>__exit</code>	329	<code>appl_init</code>	182
<code>__stksize</code>	613	<code>appl_read</code>	183
<code>__tolower</code>	651	<code>appl_tplay</code>	183
<code>__toupper</code>	655	<code>appl_trecord</code>	183
<code>{</code>	494	<code>appl_write</code>	184
<code>}</code>	23	<code>ar</code>	185
<code> </code>	24, 496	archive	95-96
<code>&</code>	24, 496	arena	186
<code> </code>	496	<code>argc</code>	187
<code>}</code>	494	arguments	68
A		arguments	
abort	173	default value	68
abs	173	deleting	69
access	174	increasing or decreasing	68
access.h	175	selecting values	68
accessory		with create window commands	76
<i>see</i> desk accessory		with enlarge window command	77
acos	176	with scrolling commands	77
address	177	with shrink window command	77
AES	177	<code>argv</code>	187
compiling with AES library	177	array	188
aesbind.h	181	arrow keys	51
aggregate		as	39, 189
<i>see</i> array		as68toas	205
alert	114	ASCII	206
alias		ASCII file	
<code>\$#</code>	30, 498	<i>see</i> FILE	
		asctime	209
		asin	210
		assembler	95
		assembly-language generator	33
		assembly-language programs	39
		assert	210
		assert.h	211
		atan	211
		atan2	212
		Atari ST references	8

atod.c	37
atof	212
atoi	213
atol	213
auto	214
auto size	107
auto snap	107
automatic mode	34, 82
aux	21, 216

B

backspace	218
backspace key	51
basepage.h	218
Bconin	219
Bconout	220
Bconstat	220
Bcostat	221
begin-macro command	80
beginning of text command	53
bibliography	7
binary files	
<i>see</i> FILE	
BIOS	222
bios	222
bios.h	223
Bioskeys	223
bit	223
bit map	224
Blitmode	224
block kill command	57
bombs	225
boot	226
border	131
break	226
buffer	227
definition	70
delete	74
for killed text	56
how differs from file	70
move text from one b. to another	72
name on command line	50
naming	70
need unique names	73

number allowed at one time	73
prompting for new name	73
replace with named file	71
status command	73
status window	73
switch b.	71
with windows	78
buffer status command	73
use with windows	79
buffer status window	73
byte	228
byte ordering	228

C

C keywords	230
C language	230
C preprocessor	33
cabs	234
calling conventions	234
calloc	238
cancel a command	64
canon.h	239
capitalization	58
carriage return	239
case	240
case sensitivity	34
cast	240
cat	22, 241
Cauxin	241
Cauxis	242
Cauxos	243
Cauxout	243
cc	33, 243
automatic mode	34
MicroEMACS mode	34, 82
-VGEM	654
-VGEMACC	654
cc0	33, 249
cc1	33, 249
cc2	33, 249
cc3	33, 249
Cconin	250
Cconis	250
Cconos	251
Cconout	252

Cconrs	252	rmdir	19
Cconws	253	set	16
cd	18, 253	setenv	17
ceil	254	unset	17
char	255	while	28, 499
character constant	255	command files	498
character quotations	495	command line	89, 92-93
character, copy		command line	
see memcpy		buffer name	50
character, fill an area with		changed file name	66
see memset		file name	50
character, reverse search for		file name changed	70
see strchr		interpretation	49
character, search for in region of memory		macro definition	93
see memchr		options	92
character, search for in string		target specification	93
see strchr, strchr		command processor	14
character, search string for		command separators	496
see strpbrk		;	24
chdir	256		24
chmod	256-257	&	24
chown	257	commands	260
clearerr	257	commands (MicroEMACS)	
CLK_TCK	258	arguments	68
clock	258	block kill text	56
close	258	buffer status	73
cmp	259	cancel	64
Cnecin	259	capitalization	58
code generator	33	cursor movement display	51
colon	88, 95	exiting from MicroEMACS	65
command		file and buffer	70
cd	18	giving c. to TOS	81
cp	19	increase power	68
equal	28, 499	keyboard macros	80
error	92, 98	lowercase	58
history	29, 493	MicroEMACS	49
if	27	move text	56
is_set	30, 499	program interrupt	81
mkdir	18	redraw screen	59
mv	19	saving text	65
nm	41	search and replace	64
not	28, 499	searching	62
od	41	switch buffers	71
printing	93	uppercase	58
rm	20	window manipulation	75
		word wrap	60

comment 89
 compare strings
 see **strspn**, **strcspn**
 compare two regions
 see **memcmp**
 compatibility 108
 compiling and debugging 82
 compiling from the GEM desktop 34
 compiling with Mark Williams C 33
 compiling without linking 38
 compound number 262
 con: 21, 263
 conditional statements 499
 const 263
 continue 263
 control characters 48
 control key 48, 121
 copy 108, 122
 copy a region of memory
 see **memcpy**
 copying text 79
 copying trees and objects 109
 cos 264
 cosh 264
 cp 19, 265
 cpp 33, 265
 Cprnos 267
 Cprnout 267
 Crawcin 268
 Crawio 268
 creat 269
 crts0.o 269
 crtso.o 270, 654
 crtsg.o 270, 654
 csd C Source Debugger 39
 ctime 271
 ctype 271
 ctype.h 273
 Cursconf 274
 cursconf 273
 cursor movement
 arrow keys 51
 back 51
 beginning of text 53
 end of text 53

forward 51
 left 51
 line position 52
 move within window 77
 next line 52
 previous line 52
 right 51
 screen down 52
 screen up 52
 scroll down 77
 scroll up 77
 cwd 27, 498
 cwdisk 27, 498

D

daemon 276
 data formats 276
 data types 276
 date 277
 dayspermonth 278
 db 278
 setting registers 281
 Dcreate 288
 Ddelete 289
 debug option (cc) 92
 debugging programs 39
 declarations 290
 default rules 94
 delete 108, 123
 delete buffer command 74
 delete key 55
 delete text, versus killing 54
 deleting with arguments 69
 desk accessory 292
 desktop 102
 device redirection, example
 see **system**
 df 296
 Dfree 297
 Dgetdrv 298
 Dgetpath 298
 diff 299
 difftime 300
 directories 18
 directory 300

.cmd.	26	envp.	317
display	108	EOF.	317
capitalization.	58	equal	318
commands	62	equal command.	28
file and buffer commands.	70	erase text	53
keyboard macro commands.	80	by line	55
kill and move commands	56	deletion of spaces.	54
killing and deleting.	54	erasing spaces	54
movement commands	51	to the left.	55
redraw	58	to the right.	54
return indent	58	errno	318
transpose.	58	errno.h	319
text and exiting.	65	error codes.	319
do	300	error message, return text of	
Dosound	301	see strerror	
double	303	error messages	141
double colon.	95	error status	92, 98
drtomw.	303	errors.	98
Drvmap	304	escape key	48
drvprs	304	etext	320
Dsetdrv.	305	evnt_button	320
Dsetpath	306	evnt_dclick.	321
dup	307	evnt_keybd	321
dup2	308	evnt_mesag	322
		evnt_mouse	324
E		evnt_multi	325
echo.	22-23, 309	evnt_timer.	328
ecvt	309	example	172
edata	310	executable files	36, 328
edit	112	executable program	34
egrep	310	execute macro command	80
else	313	execve	328
embedded commands	26	exit	329
end	314	exit button.	361
end macro command	80	exit status	98
end of text command	53	exiting from MicroEMACS.	65
enlarge window command	76	exp	330
enlarge window command		extended commands.	66
with arguments.	77	extern.	331
entry	315		
enum	315	F	
environ	316	fabs	332
environment.	316, 497	factor.c	37
environmental variables.	17	Fattrib	332
		Fclose.	333

-
- | | | | |
|---|----------|---|-----|
| <code>fclose</code> | 333 | <code>Flopver</code> | 357 |
| <code>Fcreate</code> | 334 | <code>Flopwr</code> | 358 |
| <code>fcvt</code> | 336 | folders | |
| <code>Fdatetime</code> | 337 | creating | 102 |
| <code>Fdelete</code> | 338 | <code>Fopen</code> | 360 |
| <code>fdopen</code> | 338 | <code>fopen</code> | 358 |
| <code>Fdup</code> | 340 | <code>for</code> | 360 |
| <code>feof</code> | 340 | <code>form</code> | 110 |
| <code>ferror</code> | 340 | editing | 110 |
| <code>fflush</code> | 341 | <code>form_alert</code> | 361 |
| <code>Fforce</code> | 342 | <code>form_center</code> | 362 |
| <code>fgetc</code> | 342 | <code>form_dial</code> | 362 |
| <code>Fgetdta</code> | 343 | <code>form_do</code> | 363 |
| <code>fgets</code> | 345 | <code>form_error</code> | 364 |
| <code>fgetw</code> | 346 | <code>fprintf</code> | 365 |
| <code>field</code> | 347 | <code>fputc</code> | 365 |
| <code>FILE</code> | 348 | <code>fputs</code> | 366 |
| <code>file</code> | 105, 347 | <code>fputw</code> | 366 |
| definition | 70 | <code>fraction</code> | 367 |
| how differs from buffer | 70 | <code>Fread</code> | 367 |
| name on command line | 50 | <code>fread</code> | 367 |
| naming | 70 | <code>free</code> | 368 |
| rename | 71 | <code>free image</code> | 114 |
| replace buffer with named f. | 71 | editing | 115 |
| with windows | 78 | <code>free string, editing</code> | 114 |
| write to new f. | 70 | <code>Frename</code> | 368 |
| <code>file descriptor</code> | 349 | <code>freopen</code> | 369 |
| <code>file modification time</code> | 93 | <code>frexp</code> | 370 |
| <code>file name substitutions</code> | 494 | <code>fscanf</code> | 371 |
| <code>file operations</code> | 108 | <code>Fseek</code> | 373 |
| <code>file option</code> | 92 | <code>fseek</code> | 372 |
| <code>file redirection</code> | 496 | <code>fsel_input</code> | 375 |
| <code>file-name substitutions</code> | 22 | <code>Fsetdta</code> | 378 |
| <code>fileno</code> | 349 | <code>Fsfirst</code> | 378 |
| <code>fill an area with a character</code> | | <code>Fsnext</code> | 379 |
| <i>see</i> <code>memset</code> | | <code>fstat</code> | 379 |
| <code>find one string within another</code> | | <code>ftell</code> | 380 |
| <i>see</i> <code>strstr</code> | | <code>function</code> | 380 |
| <code>flatten</code> | 123 | <code>Fwrite</code> | 381 |
| <code>flexible arrays</code> | 350 | <code>fwrite</code> | 381 |
| <code>float</code> | 350 | | |
| <code>floating-point numbers</code> | 36 | G | |
| <code>floor</code> | 353 | <code>galaxy.a</code> | 382 |
| <code>Flopfmt</code> | 353 | <code>gcvt</code> | 382 |
| <code>Floprd</code> | 356 | <code>gem</code> | 382 |

gemdefs.h	383	horizontal tab	409
gemdos	383	htom	410
gemout.h	385	hyphen	92
Getbpb	385	hypot	410
getc	386		
getchar	387	I	
getcol	387	I/O redirection	20
getenv	388, 497	icon editing	117
Getmpb	388	icon dialogue	116
getpal	389	if	411
getphys	390	if command	27
Getrez	390	ignore errors option	92, 98
getrez	390	lkbdws	413
gets	391	image editing	117
Getshift	392	INCDIR	414
Gettime	393	include file	108
getw	394	<i>see header file</i>	
Giaccess	394	incomplete resource set	108
GMT	396	index	415
gmtime	397	<i>see strchr</i>	
goto	397	inherit	415
graf_dragbox	398	initialization	
graf_growbox	399	<i>see array</i>	
graf_handle	400	Initmous	415
graf_mbox	400	int	416
graf_mkstate	401	intelligent keyboard	427
graf_mouse	401	interrupt	98, 416
graf_rubberbox		introduction	1
<i>see graf_rubbox</i>		lorec	417
graf_rubbox	403	is_set	30, 418
graf_shrinkbox	403	isalnum	418
graf_slidebox	404	isalpha	419
graf_watchbox	406	isascii	419
		isatty	419
H		iscntrl	420
handle	408	isdigit	420
header files	108, 408	isleapyear	420
help	408	islower	420
in MicroEMACS	83	isprint	421
help window	83	ispunct	421
hide	123	isspace	421
hidemouse	409	isupper	422
history command	29		
HOME	17, 409	J	

-
- j0 423
 - j1 424
 - jday_to_time 424
 - jday_to_tm 424
 - Jdisint 425
 - Jenabint 425
 - jn 425
 - joystick 427
 - K**
 - Kbdvbase 427
 - Kbrate 429
 - kbrate 429
 - keyboard 430
 - keyboard macros 80
 - Keytbl 431
 - Kgettime 432
 - kick 433
 - kill text
 - block 57
 - versus deleting 54
 - Ksettime 433
 - L**
 - lc 434
 - lcalloc 434
 - ld 434
 - ldexp 437
 - level 131
 - Lexicon 437
 - introduction 171
 - libaes 439
 - libaes.a, compiling with 177
 - libc 439
 - libm 37, 440
 - LIBPATH 440
 - library 440
 - libvdi 440
 - Line A 441
 - line feed 445
 - linea.h 445
 - linker 34
 - linking without compiling 38
 - lmalloc 446
 - loading 109
 - localtime 446
 - log 448
 - log10 449
 - Logbase 449
 - logical devices 21
 - long 450
 - longjmp 450
 - loops 499
 - lowercase text 59
 - lrealloc 451
 - ls 451
 - ls 22
 - lseek 452
 - ltom 453
 - lvalue 453
 - M**
 - macro 80, 93, 455
 - definition 89, 93
 - printing 93
 - mactions 94
 - main 455
 - make 455
 - make a directory 18
 - makefile 88, 92
 - Malloc 461
 - malloc 459
 - manifest constant 462
 - manipulating peripheral devices 427
 - mantissa 462
 - manual
 - how to use 5
 - user reaction report 6
 - Mark Williams C
 - description 1
 - environments 1
 - hardware requirements 2
 - processors supported 1
 - math.h 462
 - mathematics library 37, 462
 - maxmem 463
 - me 15, 463
 - me.a 471

Mediach	471	Visit file:	72, 74
memchr	472	Write file:	66, 70
memcmp	472	meta characters.	48
memcpy	473	metafile.	483
memory allocation	473	mf	486
memory, copy		Mfpint	486
<i>see</i> memcpy		Mfree	487
memset.	476	micro-shell.	14
menu	111, 113, 476	MicroEMACS	15
editing	113	advanced editing with	67
menu_bar	481	beginning to use	48
menu_ichack	481	exiting from	65
menu_ienable	481	invoking	49
menu_register.	482	quit without saving text.	56
menu_text	482	saving text	53
menu_tnormal	483	<i>see</i> me	
message (MicroEMACS)		Midiws	488
!	81	mkdir	18, 490
[Done]	62	mktemp	490
[End macro]	80	mmacos	94
[end]	81	modf	490
[Mark set]	57	modification time.	93
[Old buffer]	72	modulus	491
[Read XX lines]	72, 74	mousehidden	492
[Start macro]	80	move (MicroEMACS)	
[Wrap at column XX]	61	cursor	51
[Wrote XX lines]	53, 66, 70	text	56
Arg: X.	61, 68	text from one buffer to another	72
Buffer name:.	73	within window command	77
Discard changes [y/n]?	74	moving trees and objects	109
failing i-search forward	63	msh	14, 492
i-search forward.	62	!	494
Kill buffer:	74	\$	494
Name:	71	\$#	498
New string	65	\$*	498
Not found	63	\$<	498
Not now	80	&&	496
Old string	65	*	494
Query replace,		.cmd directory	494
[oldstring] -> [newstring].	65	/	495
Quit [y/n]?	56	2>	496
Read file:	71	2>>	496
Reverse search [xxxxx]:	63	3>	496
Search:	63	3>>	496
Use buffer:	78	;	496
		<	496

-
- > 496
 - >& 496
 - >> 496
 - >>& 496
 - ? 494
 - [] 494
 - aliases 498
 - aliases 29
 - aliasing 498
 - character quotations 495
 - command files 498
 - command separators 24, 496
 - command substitution 495
 - definition 14
 - embedded commands 26, 495
 - environment 497
 - equal command 499
 - file redirection 496
 - file-name substitutions 22, 494
 - history command 493
 - how to enter 14
 - I/O redirection 20
 - is_set command 499
 - not command 499
 - parentheses 499
 - pipes 496
 - profile 24, 499
 - prompt 14, 27
 - quoted strings 495
 - scripts 498
 - setting internals 16
 - variable substitution 493
 - while command 499
 - { } 494
 - | 496
 - |& 496
 - || 496
 - msh directory
 - .cmd 26
 - msh environment
 - see **PATH**
 - msh redirection operator
 - 2> 21
 - 2>> 21
 - 3> 21
 - 3>> 21
 - < 21
 - > 20
 - >& 21
 - >> 20
 - >>& 21
 - msh substitution
 - * 23
 - ? 22
 - [] 22
 - { } 23
 - Mshrink 500
 - mshversion 500
 - msleep 500
 - mtoh 501
 - mtol 501
 - mtype.h 501
 - multiple copying of killed text 57
 - multiple source files 36
 - mv 19 501
 - mwtomw 502
 - N
 - n.out 505
 - name dialogue 109
 - nested comments 503
 - newline 503
 - next error 83
 - next line command 52
 - nm 503
 - nm 41
 - no execution option 92
 - no rules option 93
 - not 504
 - not command 28
 - not modifiable, type qualifier
 - see **const**
 - notmem 504
 - nout.h 505
 - NUL 506
 - NULL 506
 - number of buffers allowed 73
 - nybble 506
 - O

- obdefs.h 507
- objc_add 507
- objc_change 507
- objc_delete 508
- objc_draw 508
- objc_edit 509
- objc_find 509
- objc_offset 510
- objc_order 510
- object 106, 511
 - copying 122
 - deleting 123
 - flatten 123
 - hide 123
 - manipulating 121
 - moving 122
 - new 116
 - resizing 122
 - retype 123
 - snap 123
 - sort 123
 - unhide 123
- object format 520
- object module 34, 38
- octal or hexadecimal dump 41
- od 41, 520
- Offgibit 521
- Ongibit 521
- open 522
- operator 523
- optimization 33
- optimizer/object generator 33
- options 92, 107
 - auto size 107
 - auto snap 107
- Options
 - compatibility 108
- osbind.h 525
- output conversion 34
- P**
- packet
 - joystick 427
- mouse 427
- parser 33
- PATH 527
- path 526
- path name 18
- path.h 527
- pattern 528
- peekb 528
- peekl 528
- peekw 529
- peripheral devices 427
- perror 529
- Pexec 530
- Physbase 531
- picture 533
- pipes 24, 496
- pnmatch 534
- pointer 535
- pokeb 536
- pokel 536
- pokew 537
- port 537
- portability 537
- postfile 500
- pow 538
- pr 538
- precedence 539
- previous error 83
- previous line command 52
- print option 93
- print symbol tables 41
- printf 540
- printing 93
- prn: 21, 544
- process 544
- processor exceptions
 - see bombs*
- profile 24, 499
- program
 - maintenance 96
 - specification 88, 92
- program interrupt command 81
- prompt 14, 498
 - device sensitive 27
- Protobt 544
- Prtblk 545

Pterm. 547
 Pterm0 547
 Ptermres 547
 pun 548
 Puntaes 548
 putc 548
 putchar 549
 puts 550
 putw 550
 pwd 550

Q

qsort 552
 quitting MicroEMACS. 53
 quoted strings. 23, 495

R

RAM

see random access

rand. 553
 Random 553
 random access 554
 ranlib 554
 rational number 555
 rc_copy 555
 rc_equal 556
 rc_intersect 556
 rc_union 557
 rdy 557
 rdy.a 565
 read 566
 read-only memory 566
 readonly 566
 real number 567
 realloc 567
 record 567
 redirecting input and output. 20
 redirecting to peripheral devices. 21
 redirection
 see system
 redraw screen. 59
 region of memory, copy
 see memcpy

region of memory, search for character

see memchr

regions, compare

see memcmp

register 567-568

 overwriting. 568

register declaration

see register variable

register variable 568

removing a directory. 19

removing a file 20

rename file 71, 108

replace buffer with named file. 71

rescomp 568

resdecom. 569

resetting registers 281

resize 122

resource 570

resource compiler 125

resource decompiler 125

resource description

 BNF. 133

 border. 131

 bordercolor. 129

 box_spec 128

 C NAME. 133

 colors 129

 data 130

 extended 129

 fill 130

 free image 127

 free string 127

 image_spec. 129

 interior 131

 justify 130

 level 131

 mask 131

 menu 127

 name_and_level. 128

 object_spec. 127

 offset 132

 options 132

 pattern 131

 size 131

 string_spec. 128

 template 130

text	131	screen editor	15
text_spec	128	screen forward movement	52
textcolor	131	screen redraw	59
transparent	130	screen up command	53
trees	127	scroll down command	77
validation	130	scroll down command	
resource description language	126	with arguments	77
resource set		scroll up command	77
creating	102	scroll up command	
restore (yank back) killed text	56	with arguments	77
return	571	scrip_read	584
return indent	60	scrip_write	585
return value	98	search	
retype	123	forward	62
reverse search	63	reverse	63
reverse search for character in string		search and replace command	64
see strchr		search for character in a string	
rewind	571	see strchr , strchr	
rindex	571	search for character in region of memory	
see strchr		see memchr	
rm	20, 572	search string for character	
rmdir	19, 572	see strpbrk	
ROM		set	16, 498, 585
see read-only memory		setbuf	586
Rsconf	573	setcol	586
rsconf	575	Setcolor	586
rsrc_free	576	setenv	17, 497, 587
rsrc_gaddr	576	Setexc	588
rsrc_load	577	setjmp	589
rsrc_obfix	577	setjmp.h	589
rsrc_saddr	578	setpal	590
rules option	93	Setpalette	590
runtime startup	578	setphys	591
rvalue	579	Setprt	591
Rwabs	579	setprt	591
		setrez	592
		Setscreen	592
		Settime	593
S		setting environmental variables	17
saving text	53, 65	setting the environment	497
sbrk	580	setting variables	16
scanf	580	Sgettime	596
Scrdmp	582	shel_envrn	596
screen backwards movement	52	shel_find	598
screen control	583	shel_read	598
screen down command	52	shel_write	598

-
- shell scripts 498
 - shell variable
 - cwd 498
 - cwd 27
 - cwdisk 27, 498
 - prompt 498
 - status 498
 - shell variables 498
 - shellsort 599
 - short 600
 - show 600
 - showmouse 601
 - shrink window command 77
 - with arguments 77
 - signal.h 601
 - silent option 93, 98
 - sin 601
 - sinh 601
 - size 602
 - sizeof 602
 - sleep 603
 - snap 123, 603
 - sort 123, 604
 - source file 38
 - special targets 98
 - specification 88, 92
 - sprintf 605
 - sqrt 605
 - srand 606
 - sscanf 606
 - stack 607
 - stack size 39
 - standard error 608
 - standard input 608
 - standard output 608
 - startup
 - see runtime startup
 - stat 609
 - stat.h 610
 - static 610
 - status 498
 - stderr 610
 - stdin 610
 - STDIO 611
 - stdio.h 612
 - stdout 612
 - stime 612
 - storage class 614
 - store command 66
 - strcat 614
 - strchr 614
 - strcmp 615
 - strcpy 615
 - strcspn 615
 - stream 616
 - strerror 616
 - string 617
 - string, comparison
 - see strspn, strcspn
 - string, find one within another
 - see strstr
 - string, reverse search for character
 - see strrchr
 - string, search for character
 - see strpbrk
 - string, search for character in
 - see strrchr, strchr
 - strip 619
 - strlen 619
 - strncat 619
 - strncmp 620
 - strncpy 620
 - strpbrk 622
 - strchr 622
 - strspn 623
 - strstr 623
 - struct 624
 - structure 624
 - structure assignment 624
 - SUFF 625
 - Super 625
 - Supexec 626
 - Sversion 628
 - swab 629
 - switch 629
 - switch buffer command 78
 - system 630
 - system variables 632
- T**

tail	636	tolower	650
tan	636	TOS	652
tanh	636	tos	652
target	93, 98	touch	655
target		touch option (make)	93
line	95	toupper	655
printing	93	transpose characters	59
program	93	tree	105, 110
specification	93	Tsetdate	656
tempnam	637	Tsettime	658
test	111	type checking	658
test suites	96	type promotion	659
tetd_to_tm	637	type qualifier, not modifiable	
text (MicroEMACS)		<i>see const</i>	
block kill	57	typedef	658
capitalize	58		
erase	53	U	
erase to left	55	ungetc	660
erase to right	54	unhide	123
kill by lines	55	union	661
lowercase	59	uniq	662
move	56	UNIX routines	662
move from one buffer to another	72	unlink	663
multiple copying of killed t.	57	unset	664
restore (yank back)	56	unset	17
saving	53, 65	unsetenv	497, 664
uppercase	59	unsettling environmental variables	17
write to new file	66	unsettling variables	16
yank back (restore)	56	unsigned	664
text dialogue	119	uppercase text	59
text of error message, return		user reaction report	6
<i>see strerror</i>			
Tgetdate	638	V	
Tgettime	639	v_arc	665
Tickcal	640	v_bar	665
time	640-641, 645	v_bit_image	668
time.h	646	v_cellarray	669
time_to_jday	646	v_circle	669
TIMEZONE	646	v_clear_disp_list	672
title strings	114	v_clrwk	672
tm_to_jday	648	v_clswwk	673
tm_to_tetd	649	v_clswk	673
TMPDIR	649	v_contourfill	674
tmpnam	649	v_curdown	677
toascii	650		

v_curhome	677	creating new file	74
v_curleft	677	moving text between buffers	72
v_curright	678	prompting for buffer name	73
v_curtext	678	vm_filename	721
v_curup	678	void	722
v_dspcur	679	volatile	722
v_eeol	679	vq_cellarray	723
v_eeos	679	vq_chcells	724
v_ellarc	680	vq_color	725
v_ellipse	683	vq_curaddress	725
v_ellpie	685	vq_extnd	
v_enter_cur	686	<i>see</i> vq_extnd	
v_exit_cur	688	vq_extnd	725
v_fillarea	689	vq_key_s	726
v_form_adv	692	vq_mouse	727
v_get_pixel	692	vq_tabstatus	727
v_gtext	692	vqf_attributes	727
v_hardcopy	695	vqin_mode	728
v_hide_c	696	vql_attributes	728
v_justified	696	vqm_attributes	729
v_meta_extents	697	vqp_error	730
v_opnvwk	697	vqp_films	730
v_opnwk	698	vqp_state	731
v_output_window	702	vqt_attributes	731
v_pieslice	702	vqt_extent	732
v_pline	702	vqt_fontinfo	733
v_pmarker	704	vqt_name	734
v_rbox	705	vqt_width	735
v_rfbox	707	vr_recfl	735
v_rmcu	707	vr_trnfm	737
v_rvoff	708	vro_cpyfm	738
v_rvon	708	vrq_choice	743
v_show_c	708	vrq_locator	743
v_updwk	709	vrq_string	744
v_write_meta	709	vrq_valuator	745
variable substitution	493	vrt_cpyfm	745
VDI	710	vs_clip	747
vdibind.h	718	vs_color	748
version	718	vs_curaddress	748
vertical tab	719	vs_palette	749
vex_butv	719	vsc_form	749
vex_curv	720	vsf_color	750
vex_motv	720	vsf_interior	750
vex_timv	721	vsf_perimeter	750
visit command (MicroEMACS)	71	vsf_style	751
		vsf_udpat	752

vsin_mode	752	wind_set	774
vsl_color	753	wind_update.	775
vsl_ends	753	window.	776
vsl_type.	754	window (MicroEMACS)	
vsl_udsty	754	buffer status.	73
vsl_width.	755	buffer status command use.	79
vsm_choice.	755	copying text among.	79
vsm_color	756	definition.	75
vsm_height	756	enlarge	76
vsm_locator	757	move within	77
vsm_string.	757	moving text among.	79
vsm_type.	758	multiple w.	75
vsm_valuator	759	number possible	76
vsp_message.	760	one w.	76
vsp_save	760	saving text	79
vsp_state	760	scroll down.	77
vst_alignment.	761	scroll up	77
vst_color	762	shifting between	76
vst_effects	762	shrink.	77
vst_font.	763	use with editing.	78
vst_height	763	using multiple buffers	78
vst_load_fonts.	764	word wrap	60
vst_point	765	write	784
vst_rotation	765	write text to new file	66, 70
vst_unload_fonts	766		
vswr_mode.	766	X	
Vsync.	767	xbios	786
		xbios.h	787
W		Xbtimer	787
warm boot		XOFF.	788
see boot		XON	789
wc	768	Y	
while	768	yank back text	56, 69
while command (msh).	28		
wildcard	769		
*	37		
?	37		
wind_calc.	769		
wind_close	770		
wind_create	770		
wind_delete	771		
wind_find	772		
wind_get	772		
wind_open	773		

User Reaction Report

To keep this manual free of errors and to help us improve Mark Williams C, we would appreciate it if you send us your reactions. Please fill in the form below, detach it, and mail it to us. Thank you.

**Mark Williams Company
1430 W. Wrightwood Avenue
Chicago, IL 60614**

Name: _____

Company: _____

Address: _____

Phone: _____

Date: _____

Version and hardware used:

Did you find any errors in the manual?

Can you suggest any improvements to the manual?

Did you find any bugs in the software?

Can you suggest improvements or enhancements to the software?

Additional comments:

User Reaction Report

To keep this manual free of errors and to help us improve Mark Williams C, we would appreciate it if you send us your reactions. Please fill in the form below, detach it, and mail it to us. Thank you.

**Mark Williams Company
1430 W. Wrightwood Avenue
Chicago, IL 60614**

Name: _____

Company: _____

Address: _____

Phone: _____

Date: _____

Version and hardware used:

Did you find any errors in the manual?

Can you suggest any improvements to the manual?

Did you find any bugs in the software?

Can you suggest improvements or enhancements to the software?

Additional comments:

