

# Mark Williams C

for the  
Atari ST

Featuring New  
Resource Editor!



Mark Williams Company

## Contents

<b>1. Introduction</b> . . . . .	
What is Mark Williams C? . . . . .	
Hardware requirements. . . . .	
Changes from release 2.0 . . . . .	
How to use this manual. . . . .	
User registration and reaction report. . . . .	
Technical support. . . . .	
Bibliography. . . . .	
Atari ST information. . . . .	
<b>2. Installing and Running Mark Williams C</b> . . . . .	
Back up your disks! . . . . .	
Installing Mark Williams C . . . . .	
Using Mark Williams C with single-sided floppy disks . . . . .	
Using two double-sided floppy drives . . . . .	
Introducing the Mark Williams micro-shell . . . . .	
What is msh? . . . . .	
How to enter msh . . . . .	
Editing a file . . . . .	
Setting the shell's internal variables . . . . .	
Setting the environment. . . . .	
Directories . . . . .	
Renaming, moving, copying, and removing files . . . . .	
Redirecting input and output. . . . .	
Redirecting to peripheral devices . . . . .	
Logical devices. . . . .	
File-name substitutions . . . . .	
Quoted strings. . . . .	
Joining and separating commands. . . . .	
The profile file. . . . .	
Embedded commands . . . . .	
The .cmd directory . . . . .	

Williams Company.

Information that is the property of Mark Williams Company. It shall not be copied, 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 warranty of fitness for any particular purpose.

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

Page 5      Printing 5 4 3 2 1

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

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
<b>3. Compiling with Mark Williams C . . . . .</b>	<b>33</b>
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
<b>4. Introduction to MicroEMACS . . . . .</b>	<b>47</b>
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

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
<b>5. make Programming Discipline . . . . .</b>	<b>85</b>
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
<b>6. Introduction to the Resource Editor . . . . .</b>	<b>99</b>
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
<b>7. Resource Compiler and Decompiler . . . . .</b>	<b>125</b>
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
<b>8. Error Messages . . . . .</b>	<b>141</b>
<b>9. The Lexicon . . . . .</b>	<b>171</b>
example . . . . .	Give an example of Mark Williams Lexicon format . . . . .
abort . . . . .	End program immediately . . . . .

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

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

<b>clock</b>	Get number of clock ticks since system boot	258
<b>close</b>	Close a file	258
<b>cmp</b>	Compare bytes of two files	259
<b>Cnecin</b>	Perform modified raw input from standard input	259
<b>commands</b>		260
<b>compound number</b>		262
<b>con</b>	Logical device for the console	263
<b>const</b>	Qualify an identifier as not modifiable	263
<b>continue</b>	Force next iteration of a loop	263
<b>cos</b>	Calculate cosine	264
<b>cosh</b>	Calculate hyperbolic cosine	264
<b>cp</b>	Copy a file	265
<b>cpp</b>	C preprocessor	265
<b>Cprnos</b>	Check if printer is ready to receive characters	267
<b>Cprnout</b>	Send a character to the printer port	267
<b>Crawcin</b>	Read a raw character from standard input	268
<b>rawio</b>	Perform raw I/O with the standard input	268
<b>c_eat</b>	Create/truncate a file	269
<b>cr_s0.o</b>	Default C runtime startup	269
<b>crtd.o</b>	C runtime startup, GEM environment	270
<b>crt_g.o</b>	C runtime startup, GEM environment	270
<b>ctime</b>	Convert system time to an ASCII string	271
<b>ctype</b>		271
<b>ctype.h</b>	Header file for data tests	273
<b>curconf</b>	Set the cursor's configuration	273
<b>Cursconf</b>	Get or set the cursor's configuration	274
<b>daemon</b>		276
<b>data formats</b>		276
<b>data types</b>		276
<b>date</b>	Print/set the date and time	277
<b>dayspermonth</b>	Return number of days in a given month	278
<b>db</b>	Assembler-level symbolic debugger	278
<b>Dcreate</b>	Create a directory	288
<b>Ddelete</b>	Delete a directory	289
<b>declarations</b>		290
<b>default</b>	Default label in switch statement	291
<b>#define</b>	Define a variable as manifest constant	291
<b>desk accessory</b>		292
<b>df</b>	Measure free space on disk	296
<b>Dfree</b>	Get information on a drive's free space	297
<b>Dgetdrv</b>	Find current default disk drive	298
<b>Dgetpath</b>	Get the current directory name	298
<b>diff</b>	Summarize differences between two files	299
<b>diffime</b>	Return difference between two times	300
<b>directory</b>		300

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

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

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

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

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

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

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

<b>Prtblk</b>	Print a dump of the screen.	545
<b>Pterm</b>	Terminate a process.	547
<b>Pterm0</b>	Terminate a TOS process.	547
<b>Ptermres</b>	Terminate a process but keep it in memory	547
<b>pun</b>		548
<b>Puntaes</b>	Disable AES.	548
<b>putc</b>	Write character to stream	548
<b>putchar</b>	Write a character to standard output	549
<b>puts</b>	Write string to standard output	550
<b>putw</b>	Write word to stream	550
<b>pwd</b>	Print the name of the current directory	550
<b>qsort</b>	Sort arrays in memory	552
<b>rand</b>	Generate pseudo-random numbers	553
<b>Random</b>	Generate a 24-bit pseudo-random number	553
<b>random access</b>		554
<b>ranlib</b>		554
<b>rational number</b>		555
<b>rc_copy</b>	Copy a rectangle	555
<b>rc_equal</b>	Compare two rectangles.	556
<b>rc_intersect</b>	Check if two rectangles intersect	556
<b>rc_union</b>	Calculate overlap between two rectangles	557
<b>rdy</b>	Create, save, and load rebootable RAM disk	557
<b>rdy.a</b>		565
<b>read</b>	Read from a file	566
<b>readonly</b>	Storage class	566
<b>read-only memory</b>		566
<b>realloc</b>	Reallocate dynamic memory	567
<b>real number</b>		567
<b>record</b>		567
<b>register</b>	Storage class	568
<b>register</b>		568
<b>register variable</b>		568
<b>rescomp</b>	Resource compiler	568
<b>resdecom</b>	Resource decompiler.	569
<b>resource</b>	Invoke the resource editor	570
<b>return</b>	Return a value and control to calling function	571
<b>rewind</b>	Reset file pointer	571
<b>index</b>	Find a character in a string	571
<b>m</b>	Remove files.	572
<b>m</b>	Remove directories.	572
<b>mdir</b>		573
<b>l3conf</b>	Configure the serial port	575
<b>r3conf</b>	Configure the serial port	575
<b>rsrc_free</b>	Free memory allocated to a set of resources	576
<b>rsrc_gaddr</b>	Get the address of a resource object	576
<b>rsrc_load</b>	Load a resource file into memory	577

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

sqrt	Compute square root	605
srand	Seed random number generator	606
scanf	Format input	606
sack		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
stime	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
strchr	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		641
time	Print current time/time execution of a command	645
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

## xx Mark Williams C for the Atari ST

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

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

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

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

## Introduction 3

`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.

## Introduction 5

`lck` 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.

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 **resdecomp** 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.

Kernighan, B.W.; Plaager, 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.

Gerita, 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.PRG**. 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** options 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.PRG` 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>.

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

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:

```
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.

- 7 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 $A
echo "$A"
echo "$B"
echo "$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

dash 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 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 dash, it automatically reads a file called profile and executes all of the commands it finds there. By altering your profile, you can customize dash 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, $drive\command
setenv SUFF=.prg,.cos,.ctp
setenv LIBPATH=$drive\lib,$drive\bin,
setenv TMPDIR=$drive\temp
setenv INCDIR=$drive\include
setenv TIMEZONE=CST:0:CDT
set prompt='X'
set history=8
The first line,
```

sets the variable drive to a:. This means that the variable \$drive will be interpreted as a: by dash.

```
The next line,
setenv PATH=.cmd, $drive\command
```

sets the PATH environmental variable, which tells dash where to find executable files. The first directory, .cmd, stands for dash's internal command directory. This tells dash to check and see if the command you have typed in is built into dash itself. The rest of the command

```
,$drive\command
```

tells dash 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, dash will not be able to execute the rest of the commands in profile.

The next lines,

```
setenv SUFF=.prg,.cos,.ctp
setenv LIBPATH=$drive\lib,$drive\bin,
setenv TMPDIR=$drive\temp
setenv INCDIR=$drive\include
setenv TIMEZONE=CST:0:CDT
```

set the environmental variables that dash 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 **!*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**.

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
crt0.o
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 TOS function `Setexc` 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 `db`, 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 `csd`, 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.

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

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 `cad`, 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
```

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.

<ctrl-F>	Move forward one space
<esc>F	Move forward one word
<ctrl-E>	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.

<ctrl-B>	Move back one space
<esc>B	Move back one word
<ctrl-A>	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.

`<esc><`        Move to beginning of text  
`<esc>>`        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 `msb`.

### 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*.

<code>&lt;ctrl-D&gt;</code>	Delete one character to the right
<code>&lt;esc&gt;D</code>	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>&lt;del&gt;</code>	Delete one character to the left
<code>&lt;ctrl-H&gt;</code>	Delete one character to the left
<code>&lt;esc&gt;&lt;del&gt;</code>	Kill one word to the left
<code>&lt;esc&gt;&lt;ctrl-H&gt;</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 `<del>`. The second `e` of the word `three` has vanished.

Type `<esc><del>`. 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><del>`. 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><del>` command can be used to erase words throughout your text.

If you wish to erase a word to the left yet preserve both spaces that are around it, position the cursor at the space immediately to the right of the word and type `<esc><del>`. If you wish to erase a word to the left plus the space that immediately follows it, position the cursor under the first letter of the *next* word and then type `<esc><del>`.

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

### Erasing lines of text

Finally, the following command erases a line of text:

<code>&lt;ctrl-K&gt;</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:

```
<ctrl-@>      Set mark
<ctrl-W>      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, reinvoked the editor to edit the file `example1.c`, as you did before. The following sections will perform some exercises with this file.

### Search forward

`<ctrl-S>` Search forward incrementally  
`<esc>S` 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: *msg;`

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:

<ctrl-X><ctrl-S>	Save text
<ctrl-X><ctrl-W>	Write text to a new file
<ctrl-Z>	Save text and exit
<ctrl-X><ctrl-C>	Exit without saving text

You have used two of these commands already: the *save* command <ctrl-X><ctrl-S> and the *quit* command <ctrl-X><ctrl-C>, which respectively allow you to save text or to exit from MicroEMACS without saving text. (Commands that begin with <ctrl-X> are called *extended* commands; they are used frequently in the advanced editing to be covered in the second half of this tutorial.)

### Write text to a new file

<ctrl-X> <ctrl-W>	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 *f* 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 reinvok 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:

<code>&lt;ctrl-X&gt;2</code>	Create a window
<code>&lt;ctrl-X&gt;1</code>	Delete extra windows
<code>&lt;ctrl-X&gt;N</code>	Move to next window
<code>&lt;ctrl-X&gt;P</code>	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:

```
<ctrl-X>Z      Enlarge window
<ctrl-X><ctrl-Z>  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:

```
<ctrl-X><ctrl-N>  Scroll down
<ctrl-X><ctrl-P>  Scroll up
<esc>I           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>I`. 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>I`. (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.

## 88 Mark Williams C for the Atari ST

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 mccoy.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 `mccoy.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
mccoy.o: rifle.h pistol.h
```

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

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

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

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 h'phen, '-', 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 **msk**. 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 = factora.o atoda.o
OBJ2 = factorb.o atodb.o

factor.prg :: $(OBJ1)
    cc -c $(OBJ1) -lm

factor.prg :: $(OBJ2)
    cc -c $(OBJ2) -lm

```

This **makefile** tells **make** to do the following: (1) Check if either **factora.o** or **atoda.o** is younger than **factor.prg**. (2) If either one is, regenerate **factor.prg** using this version of these files. (3) If neither **factora.o** nor **atoda.o** 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 decompiler.

---

## 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 **.rtd**. 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:

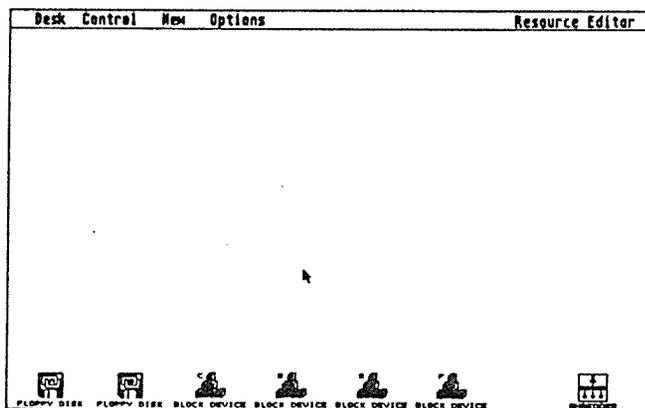
```
gem resource
```

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:

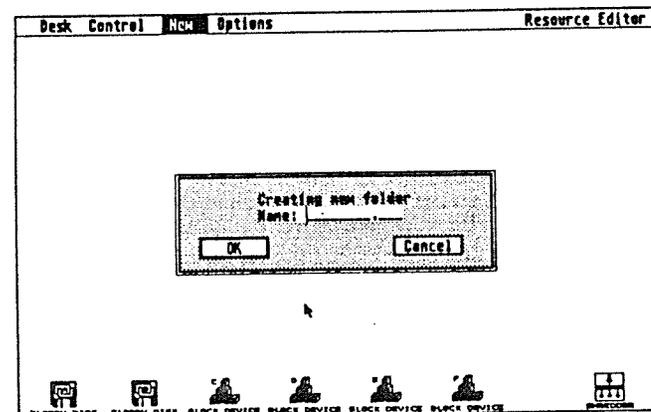


A menu bar extends across the top of the screen, and two types of icons are shown: In the above example, the first six icons are for file systems and the last represents the *Shredder*. Each file-system icon represents a RAM disk, a floppy disk, or a logical segment on a hard disk.

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**

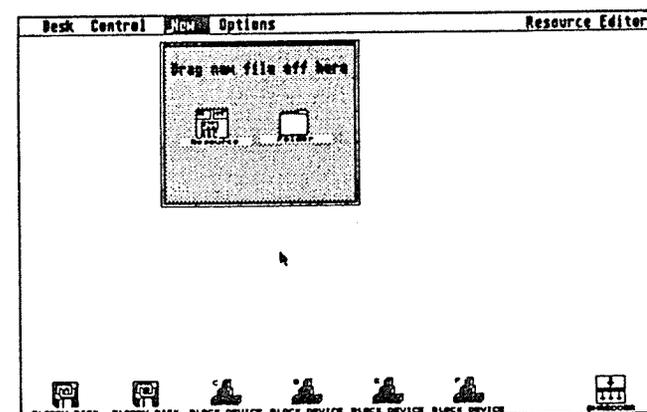
Click this item to close the top window entirely, even if clicking its close box, in the upper left corner of the window, would have taken it up to the next level.

**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:



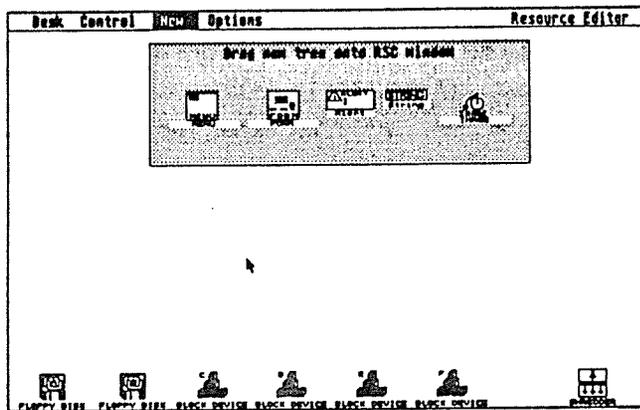
To open a new folder or resource, drag the appropriate icon onto an open area of the desktop.

**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:



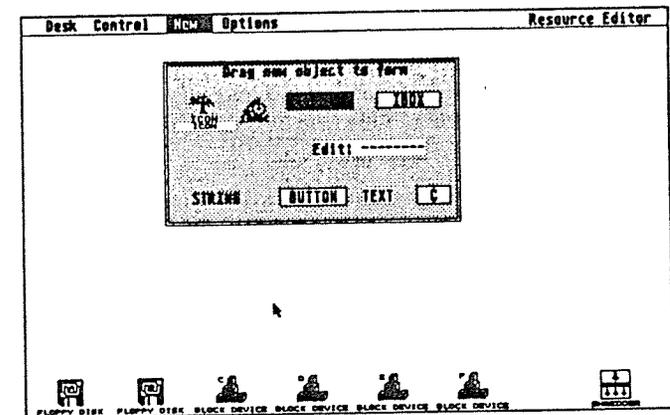
These trees are described in detail later in this section.

### 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

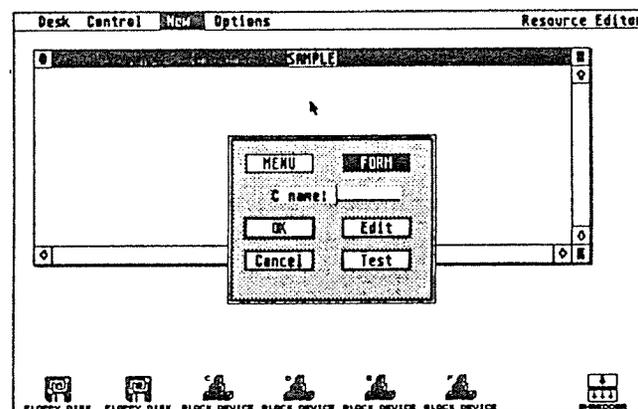
The different types of trees are described as follows:

### 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

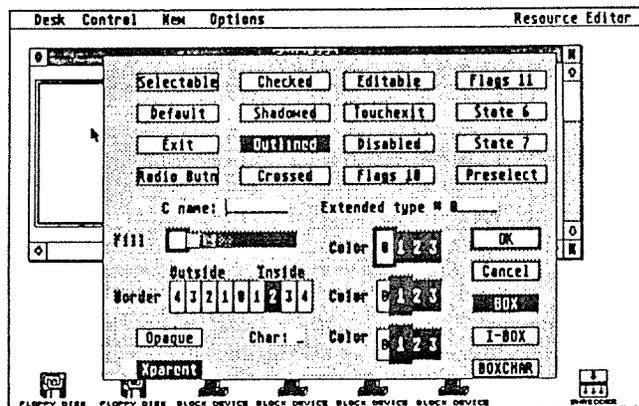
If the form can be treated as a menu, the button *MENU* will be enabled on the name dialogue. Usually this is the case only when you have previously changed type from *MENU* to *FORM* to edit the menu in non-standard ways (see *MENU*, below).

### 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 menu, 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.

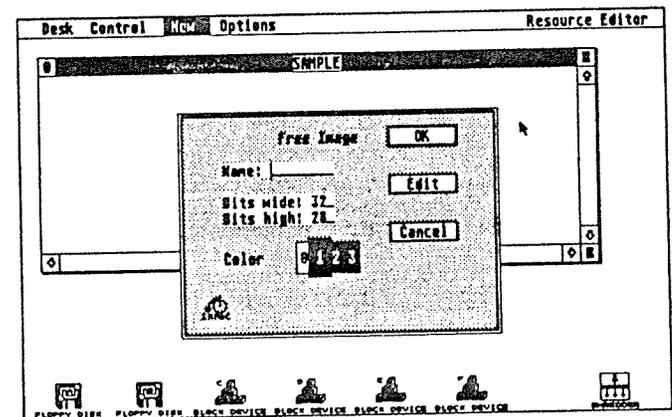
The icon displayed in an alert is adjusted from the name dialogue. You can edit the string and buttons within the alert by clicking the *Edit* button or by double-clicking the alert's icon.

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.

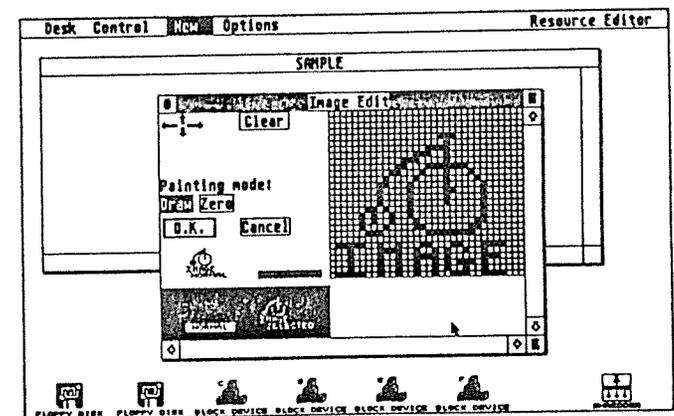
### Image

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:



## 116 Mark Williams C for the Atari ST

## 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.

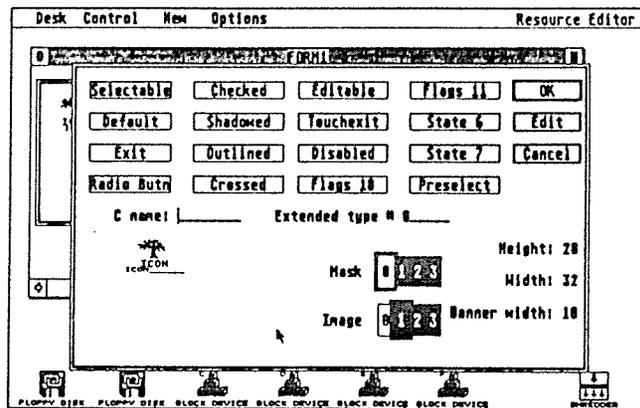
See the Lexicon article on *object* for a fuller discussion of objects and object trees.

## 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* partabox 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:

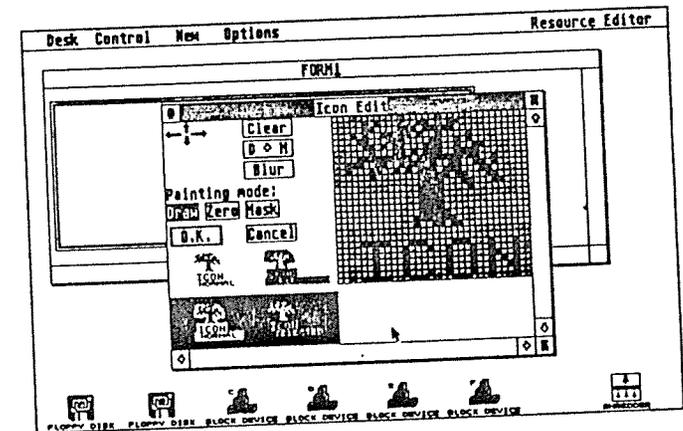


On the icon dialogue, the icon character and banner fields may be dragged to reposition them relative to the icon. The banner as displayed on the dialogue is left justified, whereas it is centered when the icon is displayed normally. The banner width field is in characters, and is limited to 20. Changes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

Mask	Data	Effect	Edit Display Color
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:

<b>Clear</b>	Zero both data and mask.
<b>D to M</b>	Copy the data image to the mask image.
<b>Blur</b>	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 <i>D to M</i> and the <i>Blur</i> a few times to give a halo of mask around the image.
<b>Arrows</b>	Clicking the arrows in the top left of the display shifts the image and mask one pixel in the indicated direction.
<b>Cancel</b>	Abandon changes.
<b>OK</b>	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.

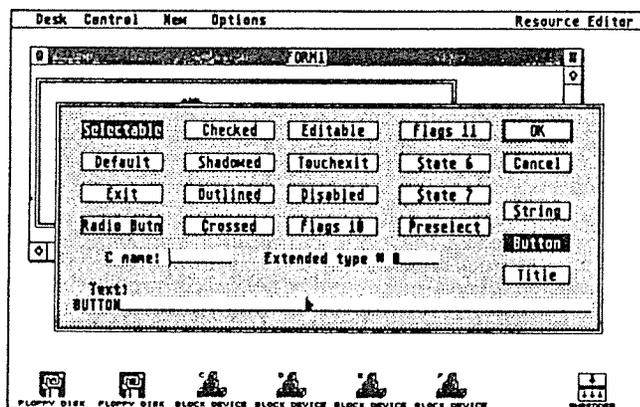
When you create a new editable field by changing a TEXT item to FTEXT or FBOXTEXT, remember to set the EDITABLE flag.

### 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. *Remember to unhide any objects you have hidden before you save the resource.*
- 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. *Snap* 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 **resdecomp**, the resource decompiler.

**resdecomp** 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 **resdecomp.prg**. Its command line is as follows:

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

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

**resdecomp** 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

```
resdecomp -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

```
resdecomp bar
```

tells **resdecomp** 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 bracked 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 MENUNAME .
```

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:

<b>box</b>	name_and_level	box_spec
<b>ibox</b>	name_and_level	box_spec
<b>boxcharacter</b>	name_and_level	box_spec
<b>button</b>	name_and_level	string_spec
<b>string</b>	name_and_level	string_spec
<b>title</b>	name_and_level	string_spec
<b>boxtext</b>	name_and_level	text_spec
<b>boxedit</b>	name_and_level	text_spec
<b>text</b>	name_and_level	text_spec
<b>edit</b>	name_and_level	text_spec
<b>icon</b>	name_and_level	icon_spec
<b>image</b>	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 **rescomp** 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
            : lbox 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
                  | blue1

```

```

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 **resdec**.

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.

## 142 Mark Williams C for the Atari ST

*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 to 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

## 148 Mark Williams C for the Atari ST

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.

## 152 Mark Williams C for the Atari ST

- 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

## 154 Mark Williams C for the Atari ST

- 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 (**cpp**, 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 (cpp, 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.

## 170 Mark Williams C for the Atari ST

---

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  $n \geq 0$ , 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,
                "Xs 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** stands for *application environment services*. It draws and manipulates predefined graphics elements, such as icons, pull-down menus, and windows. It is the highest level of GEM, and the one that 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.

<code>appLexit</code>	tell the AES that the program is exiting
<code>appLfind</code>	get another application's handle
<code>appLinit</code>	initialize a new application
<code>appLread</code>	read a message from another process
<code>appLtplay</code>	replay recorded AES events
<code>appLtrecord</code>	record AES events
<code>appLwrite</code>	send a message to another process
<code>evnt_button</code>	await a mouse-button event
<code>evnt_dclick</code>	set/get double-clicking speed
<code>evnt_keybd</code>	await a keyboard event
<code>evnt_mesag</code>	await a message
<code>evnt_mouse</code>	wait for mouse to enter a rectangle
<code>evnt_multi</code>	await more than one event
<code>evnt_timer</code>	wait a given amount of time

<code>form_alert</code>	perform an alert dialogue
<code>form_center</code>	center dialogue box on screen
<code>form_dial</code>	reserve/release dialogue box
<code>form_do</code>	use dialogue box
<code>form_error</code>	display preset error box
<code>fseLinput</code>	display/run file selector box
<code>graf_growbox</code>	draw expanding box outline
<code>graf_handle</code>	return VDI handle
<code>graf_mbox</code>	draw moving box
<code>graf_mkstate</code>	return current mouse states
<code>graf_mouse</code>	change mouse pointer's shape
<code>graf_rubbox</code>	draw box that expands with mouse pointer
<code>graf_shrinkbox</code>	draw a shrinking outline
<code>graf_slidebox</code>	find center of box's "slider"
<code>graf_watchbox</code>	check if mouse pointer is within box
<code>menu_bar</code>	display/erase menu bar
<code>menu_ichck</code>	display/remove checks by menu items
<code>menu_ienable</code>	enable/disable menu items
<code>menu_register</code>	name desk accessory on desk menu
<code>menu_text</code>	change text of menu item
<code>menu_tnormal</code>	show menu title in normal/reverse video
<code>objc_add</code>	add an object to object tree
<code>objc_change</code>	change an object's state
<code>objc_delete</code>	delete object from object tree
<code>objc_draw</code>	draw an object
<code>objc_edit</code>	edit text within an object
<code>objc_find</code>	find if mouse is over an object
<code>objc_offset</code>	return location of object on the screen
<code>objc_order</code>	change order of object within its tree
<code>objc_set</code>	compute object's location
<code>rc_copy</code>	copy a rectangle
<code>rc_equal</code>	compare two rectangles
<code>rc_intersect</code>	calculate overlap of rectangles
<code>rc_union</code>	combine rectangles
<code>rsrc_free</code>	free memory allocated to resource
<code>rsrc_gaddr</code>	get address of data structure
<code>rsrc_load</code>	load resource file into RAM
<code>rsrc_obfix</code>	convert character coordinates
<code>rsrc_saddr</code>	store index to data structure
<code>scrp_read</code>	find name of scrap directory
<code>scrp_write</code>	set name of scrap directory

<b>shelenvrn</b>	search for environmental variable
<b>shel_find</b>	find a file name
<b>shel_read</b>	return name of parent program
<b>shel_write</b>	tell desktop which application to run next
<b>wind_calc</b>	calculate window size
<b>wind_close</b>	close a window
<b>wind_create</b>	create a window
<b>wind_delete</b>	delete window
<b>wind_find</b>	find a window under mouse pointer
<b>wind_get</b>	get information about a window
<b>wind_open</b>	open a window
<b>wind_set</b>	set values for window
<b>wind_update</b>	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_clsvwk(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**

### 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

### **appLexit** – AES function (libaes)

Exit from an application  
**#include <aesbind.h>**  
**int appLexit()**

**appLexit** 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.

**appLexit** 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, **appLinit**, TOS

### **appLfind** – AES function (libaes)

Get another application's handle  
**#include <aesbind.h>**  
**int appLfind(name) char name[9];**

**appLfind** 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 "

**appLfind** returns the handle if it is found, and -1 if an error occurred. This requires that the application be started with **shLwrite**.

*See Also*

AES, TOS

### **appLinit** – AES function (libaes)

Initiate an application  
**#include <aesbind.h>**  
**int appLinit()**

**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

### **appLread** – AES function (libaes)

Read a message from another application  
**#include <aesbind.h>**  
**int appLread(handle, length, buffer) int handle, length; char \*buffer;**

**appLread** is an AES routine that helps to read a message sent by the function **appLwrite**. The first 16 bytes of such a message are read by either of the functions **evnt\_mesag** or **evnt\_multi**; **appLread** 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, **appLwrite**, **evnt\_mesag**, TOS

### **appLtplay** – AES function (libaes)

Replay AES activity  
**#include <aesbind.h>**  
**int appLtplay(buffer, number, speed) char \*buffer; int number, speed;**

**appLtplay** is an AES routine that replays a set of AES events. These events must be recorded with the function **appLtrecord**. *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. **appLtplay** always returns one.

*See Also*

AES, **appLtrecord**, TOS

### **appLtrecord** – 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\_multf**. 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.
- f** 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**, **crtso.o**, **crtso.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	(0nn)

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:	<b>shri</b>	shared instruction
	<b>shrd</b>	shared data
private:	<b>prvi</b>	private instruction
	<b>prvd</b>	private data
uninitialized:	<b>bssi</b>	uninitialized instruction
	<b>bssd</b>	uninitialized data
	<b>strn</b>	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
.bssl      .shrl
.prvd      .strn
.prvl
```

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 **is** 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.

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

<i>abcd</i>	<i>dn, dn</i>	
<i>abcd</i>	<i>-(an), -(an)</i>	
<i>abcd</i>		C100
<i>addx</i>		D140
<i>addx.b</i>		D100
<i>addx.l</i>		D180
<i>sbcd</i>		8100
<i>subx</i>		9140
<i>subx.b</i>		9100
<i>subx.l</i>		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.

<i>add</i>	<i>dn, ea</i>	
<i>add</i>	<i>ea, dn</i>	
<i>add</i>		D040
<i>add.b</i>		D000
<i>add.l</i>		D080
<i>sub</i>		9040
<i>sub.b</i>		9000
<i>sub.l</i>		9080

**ADDA Type:** These instructions accept any source effective address. The *cmp* instruction cannot combine byte operations with address-register sources.

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

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

sgt	ea	5EC0
shi	ea	52C0
sbs	ea	54C0
sle	ea	5FC0
slo	ea	55C0
sls	ea	53C0
slt	ea	5DC0
smi	ea	5BC0
sne	ea	56C0
spl	ea	5AC0
st	ea	50C0
subi	\$n,ea	0440
subi.b	\$n,ea	0400
subi.l	\$n,ea	0480
svc	ea	58C0
svs	ea	59C0
tas	ea	4AC0
tst	ea	4A40
tst.b	ea	4A00
tst.l	ea	4A80

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

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

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

and	dn,ea	
and	ea,dn	
and		C040
and.b		C000
and.l		C080
or		8040
or.b		8000
or.l		8080

**ANDI Type:** These instructions combine an immediate source operand with either a data-alterable effective address destination operand or the status register. The

whole status register or only the low byte is selected, depending on whether the operation size is word or byte.

andi	\$n,ea	
andi	\$n,sr	
andi		0240
andi.b		0200
andi.l		0280
eori		0A40
eori.b		0A00
eori.l		0A80
ori		0040
ori.b		0000
ori.l		0080

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

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

roxr.l E090

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

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

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

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

**JMP Type:** These instructions require control-effective addresses.

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

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

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

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

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

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

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

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

bcc	l	6400
bcc.s	l	6400
bcs	l	6500
bcs.s	l	6500
beq	l	6700
beq.s	l	6700
bge	l	6C00
bge.s	l	6C00
bgt	l	6E00
bgt.s	l	6E00
bhi	l	6200
bhi.s	l	6200
bhs	l	6400
bhs.s	l	6400
ble	l	6F00
ble.s	l	6F00
blo	l	6500
blo.s	l	6500
bls	l	6300
bls.s	l	6300
blt	l	6D00
blt.s	l	6D00
bmi	l	6B00
bmi.s	l	6B00
bne	l	6600

bne.s	l	6600
bpl	l	6A00
bpl.s	l	6A00
bra	l	6000
bra.s	l	6000
bsr	l	6100
bsr.s	l	6100
bvc	l	6800
bvc.s	l	6800
bvs	l	6900
bvs.s	l	6900
cmpm	(an)+,(an)+	B148
cmpm.b	(an)+,(an)+	B108
cmpm.l	(an)+,(an)+	B188
dbcc	dn,l	54C8
dbcs	dn,l	55C8
dbeq	dn,l	57C8
dbf	dn,l	51C8
dbge	dn,l	5CC8
dbgt	dn,l	5EC8
dbhi	dn,l	52C8
dbhs	dn,l	54C8
dble	dn,l	5FC8
dblo	dn,l	55C8
dbls	dn,l	53C8
dblt	dn,l	5DC8
dbmi	dn,l	5BC8
dbne	dn,l	56C8
dbpl	dn,l	5AC8
dbra	dn,l	50C8
dbt	dn,l	50C8
dbvc	dn,l	58C8
dbvs	dn,l	59C8
exg	rn,rn	C100
ext	dn	4880
ext.l	dn	48C0
link	an,\$n	4E50
moveq	\$n,dn	7000
nop		4E71
reset		4E70
rte		4E73
rtr		4E77
rts		4E75
stop	\$n	4E72
swap	dn	4840
trap	\$n	4E40

```

trapv          4E76
unkl          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	&		Amperсанд
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 &lt;math.h&gt;

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 &lt;assert.h&gt;

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 &lt; 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 &lt;assert.h&gt;

assert.h is the header file that defines the macro assert.

*See Also*

assert, header file

**atan** — Mathematics function (libm)

Calculate inverse tangent

#include &lt;math.h&gt;

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  $[-\text{PI}/2, \text{PI}/2]$ .

#### 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  $[-\text{PI}, \text{PI}]$ . 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**

**atoi, atol, float, long, printf, scanf**

**Notes**

No overflow checks are performed. **atoi** 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**, **intin**, and **intout** arrays. You can place **msh.prg** into **\auto** and enter it automatically when you boot your system; however, subsequent attempts to run any GEM application through **msh** generates effects that are unpredictable and usually unwelcome.

**Example**

The following example shows a few things that you can do in a program that is placed in **\auto**. It demonstrates the functions **Cursconf**, **Iorec**, **Kbrate**, **linea0**, **Ptermres**, **Raconf**, **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], ptain[128], ptsout[128];
        static int *ctrl[4];
        linea0();
        INTIN = intin;

        INTOUT = intout;
        PTSIN = ptain;
        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_MIDI); 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_HOMO|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_hltpe-BP->p_lowtpe, 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:**.