L

lc – Command

List directory's contents in columnar format
lc

The command **lc** prints the contents of the current directory. The contents are printed in multiple columns to make them easy to read. Names of directories are followed by a slash '/'.

See Also
commands, ls

lcalloc – General function (libc)

Allocate dynamic memory
char *lcalloc(*count, size***)**
unsigned long *count, size*;

lcalloc is one of a set of routines that helps you to manage the computer's free memory, or *arena*. **lcalloc** calls **lmalloc** to obtain a block large enough to contain *count* items of *size* bytes each; it then initializes the block to zeroes and returns a pointer to it. Dynamic memory that is no longer needed can be returned to the free memory pool with the function **free**.

Unlike the related function **calloc**, **lcalloc** takes arguments that are **unsigned longs**; therefore, it can allocate memory blocks that are larger than 64 kilobytes.

See Also
arena, calloc, free, lmalloc, lrealloc, malloc, notmem, realloc

Diagnostics
lcalloc returns NULL if insufficient memory is available.

ld – Command

Link relocatable object files
ld *[option ...] file ...*

A compiler translates a file of source code into a *relocatable object*. This relocatable object cannot be executed by itself, for calls to routines stored in libraries have not yet been resolved. **ld** combines, or *links*, relocatable object files with libraries produced by the archiver **ar** to construct an executable file. For this reason, **ld** is sometimes called a *linker*, a *link editor*, or a *loader*.

ld scans its arguments in order and interprets each option as described below. Each non-option argument is either a relocatable object file, produced by **cc**, **as**, or **ld**, or a library archive produced by **ar**. It rejects all other arguments and prints a diagnostic message.

Each relocatable file argument is bound into the output file if its machine type matches the machine type of the first file so bound; if it does not, a diagnostic message is generated. The symbol table of the file is merged into the output symbol table and the list of defined and undefined symbols updated appropriately. If the file redefines a symbol defined in an earlier bound module, the redefinition is reported and the link continues. The last such redefinition determines the value that the symbol will have in the output file, which may be acceptable but is probably an error.

Each library archive argument is searched only to resolve undefined references, i.e., if there are no undefined symbols, the linker goes to the next argument immediately. The library is searched from first module to last and any module that resolves one or more undefined symbols is bound into the output exactly as an explicitly named relocatable file is bound. The library is searched repeatedly until an entire scan adds nothing to the executable file.

The order of modules in a library is important in two respects: it will affect the time required to search the library, and, if more than one module resolves an undefined symbol, it can alter the set of library modules bound into the output.

A library will link faster if the undefined symbols in any given library module are resolved by a library module that comes later in the library. Thus, the low-level library modules, those with no undefined symbols, should come at the end of the library, whereas the higher-level modules, those with many undefined symbols, should come at the beginning. The library module **ranlib.sym**, which is maintained by the **ar s** modifier, provides **ld** with a compressed index to the symbols defined in the library. But even with the index, the library will link much faster if the modules occur in top-down rather than bottom-up order.

A library can be constructed to provide a type of "conditional" linking if alternate resolutions of undefined symbols are archived in a carefully thought-out order. For instance, **libc.a** contains the modules

finit.o
exit.o
_finish.o

in precisely the order given, though some other modules may intervene. **finit.o** contains most of the internals of the STDIO library, **exit.o** contains the **exit()** function, and **_finish.o** contains an empty version of **_finish()**, the function that **exit()** calls to close STDIO streams before process termination. If a program uses any STDIO routines, macros, or data, then **finit.o** will be bound into the output with its version of **finish()**. If a program uses no STDIO, then the "dummy" **_finish.o** will be bound into the output because it is the first module that defines **_finish()** that the linker encounters after **exit.o** adds the undefined reference. This saves approximately 3,000 bytes. To set the order of routines within a library, use the archiver **ar**; this, of course, has its own entry in the Lexicon.

The available options are as follows:

-d Define common regions even if relocation information is retained. By default, **ld** leaves common areas undefined if there are undefined symbols or if the **-r** option is specified.

-l*filename*
Link with the object file *filename*. This option is used to link programs to access code or data at fixed locations outside the program being linked, such as a library burned into a ROM or the fixed low memory locations documented by Atari.

-l *name*
An abbreviation for the libraries named in the environmental variable **LIBPATH**. **ld** searches each directory named in **LIBPATH** for a file named **lib***name*.**a**.

-o *file*
Write output to *file* (default, **l.prg**.)

-R *value*
Relocation base option. By default, **ld** links executeable files to run at the *user-base* for the computer. In almost all cases, the *user-base* is zero. If the **-R** option is used, **ld** will link the executeable to run at *value* instead of at zero. *value* can be set to any C-style constant, or to a symbol name that **ld** can find in the object files and archives being linked; remember that a C-accessible symbol *must* end with an underscore character '_'. This option is used primarily to produce output files that can be burned into ROM. These programs must make their own provisions for relocating initialized data and other tasks.

-r Retain relocation information in the output, and issue no diagnostic message for undefined symbols. By default **ld** discards relocation information from the output if there are no undefined symbols.

-s Strip the symbol table from the output. The same effect may be obtained by using **strip**. The **-s** and **-r** options are mutually exclusive.

-u *symbol*
Add *symbol* to the symbol table as a global reference, usually to force the linking of a particular library module.

-X Discard local compiler-generated symbols of the form 'L...'.

-x Discard all local symbols.

See Also
ar, as, cc, commands, n.out

Notes
If you are linking a program by hand (that is, running **ld** independently from the **cc** command), be sure to include the appropriate run-time start-up routine with the **ld** command line; otherwise, the program will not link correctly.

Because version 3.0 changes the object format, the edition of **ld** shipped with version 3.0 does not work with objects compiled with Mark Williams C version 2.1.7 or earlier. To convert such objects to a format that **ld** recognizes, use the command **mwtomw**.

ldexp – General function (libc)
Combine fraction and exponent
double ldexp(*f***, *e*) double *f*; int *e*;**

ldexp combines the fraction *f* with the binary exponent *e* to return a floating-point value *real* that satisfies the equation *real*=*m**2^*e*.

See Also
atof, ceil, fabs, floor, fraction, frexp, modf

Lexicon – Introduction

The Mark Williams Lexicon is a new approach to documentation of computer software. The Lexicon is designed to improve documentation and eliminate some limitations found in more conventional documentation.

How to use the Lexicon
The Lexicon consists of one large document that contains entries for every aspect of Mark Williams C. You will not have to search through a number of different manuals to find the entry you are looking for.

Every entry in the Lexicon has the same structure. The first line gives the name of the topic being discussed, followed by its type (e.g., **Mathematics function**) and, where appropriate, the file in which it is kept.

The next lines briefly describe the item, then give the item's usage, where applicable. These are followed by a brief discussion of the item, and an example.

Cross-references follow. These can be to other entries or to other texts, notably to *The Art of Computer Programming* and the first edition of *The C Programming Language*. Diagnostics and notes, where applicable, conclude each entry.

Internally, the Lexicon has a tree structure. The "root" entry is the present entry, for **Lexicon**. Below this entry comes the set of *Overview* entries. Each Overview entry introduces a group of entries; for example, the Overview entry for **string** introduces all of the string functions and macros, lists them, and gives a lengthy example of how to use them.

Each entry cross-references other entries. These cross-references point up the documentation tree, toward an overview article and, ultimately, to the entry for

Lexicon itself. They also point down the tree to subordinate entries, and across to entries on related subjects. For example, the entry for **getchar** cross-references **STDIO**, which is its Overview article, plus **putchar** and **getc**, which are related entries of interest to the user. The Lexicon is designed so that you can trace from any one entry to any other, simply by following the chain of cross-references up and down the documentation tree.

Types of entries

There are several types of entries, as follows:

Command
: These describe commands or utilities that run directly under TOS.

Definitions
: These entries define technical terms and provide background information that is useful in C programming.

Library functions
: These present functions or macros included with Mark Williams C. They include **ctype macros** (a macro that checks the type of data being handled); **debugging macros**; **general functions** (non-specialized C functions and macros); **mathematics library** functions; **STDIO functions**; **STDIO macros**; **string functions** (or routines used to manipulate character strings); and **time functions** (routines used to manipulate the time setting rendered by TOS).

Overview
: Each of these entries gives an overview of a group of routines.

Symbols and constants
: Data elements that are used while compiling or running programs; these include **environmental variables** and **manifest constants**.

TOS support
: Entries that give information useful in programming for the Atari ST; these include the following: **TOS devices** (logical devices used by TOS to describe its peripheral devices); **TOS functions**; and **TOS support** (routines designed to support the TOS operating system).

Technical information
: These give detailed information on technical issues. The articles describe **calling conventions**, **data formats**, and others.

UNIX routines
: A function, macro, or data item included to provide compatibility with UNIX, COHERENT, and related operating systems.

The **Overview** entries review an entire topic, and give full cross-references to all of the entries that belong to the category discussed. If you are unfamiliar with a particular variety of routine, be sure to check the Overview entry that discusses it.

At the back of this manual is a list of all entries in the Lexicon, sorted by category. Check there for a complete list of the Overview entries, as well as for lists of all functions sorted by type.

Use the Lexicon

If, while reading an entry, you encounter a technical term that you do not understand, look it up in the Lexicon. You should find an entry for it. For example, if a function is said to return a data type **float** and you do not know exactly what a **float** is, look it up. You will find it described in full. In this way, you should increase your understanding of Mark Williams C, and make your programming easier and more productive.

We wish to hear your comments on the Lexicon; we especially wish to hear if you discover something wrong or if an entry that you looked for is missing.

libaes – Library

GEM AES bindings

libaes is the library that holds the GEM AES binding routines. AES stands for *application environment services*. The routines contained in **libaes** allow you to invoke the elements of the GEM graphics interface, such as icons, windows, and pull-down menus. See the entry for **AES** for a brief description of the routines in this library.

To alter **libaes** or print its table of contents, use the archiver **ar**.

This library can be called on the **cc** command line in one of two ways. First, the **-VGEM** will automatically link it in, plus the library **libvdi** and the runtime startup module **crtsg.o**. Second, it can be included by itself with the *library* option **-laes**. This option must come at the *end* of the **cc** command line, or the library will not be linked in.

See Also

AES, aesbind.h, ar, crtsg.o, gemdefs.h, library, nm, TOS

libc – Library

libc is the archive file that holds the more commonly used C functions, system calls, and compiler run-time support routines. See the entries for **string**, **STDIO**, and **UNIX routines** for information about many of the routines within **libc**. For a complete listing of the modules within **libc**, pass the following command to **msh**:

```
ar t libc.a >foo
```

This writes a list of the library's contents into the file **foo**.

See Also
ar, library, nm

libm — Library

libm is the archive file that holds the mathematics library.

See Also
ar, library, mathematics library, math.h, nm

LIBPATH — Environmental variable

Directories that hold libraries

LIBPATH names the directories that **cc** searches to find the compiler's executable programs and libraries. **make** also searches these directories for the files **mmacros** and **mactions**.

For example, the command

```
setenv LIBPATH=a:\lib,,b:\lib
```

tells **cc** to look for the compiler's executable files first in directory **lib** on drive A:, then in the current directory (as indicated by the two commas with nothing between them), and finally in **lib** on drive B:.

It is set with the **setenv** command.

See Also
cc, make, msh, setenv

library — Overview

A **library** is an archive file of commonly used functions that have been compiled, tested, and stored for inclusion in a program at link time.

Normally, C uses two libraries: **libc.a**, which holds the standard C functions; and **libm.a**, which holds mathematical functions. You can use the archiver **ar** to create your own libraries of functions or edit existing libraries, or you can purchase such libraries from elsewhere. The sizes of the files in an existing library can be listed with the command **size**, and their symbol tables may be listed with the command **nm**.

See the entries for **mathematics library, string, STDIO,** and **UNIX routines** for information about many of the routines within these libraries.

See Also
ar, function, libaes, libc, libm, libvdi

libvdi — Library

GEM VDI bindings

libvdi is the library that holds the GEM VDI routines. VDI stands for *virtual device interface*. These routines perform low-level GEM graphics tasks. For a brief summary of these routines, see the entry for **VDI**.

libvdi's table of contents can be printed and its contents altered with the archiver **ar**.

This library can be called on the **cc** command line in either of two ways. First, the **-VGEM** will automatically link it in, plus the library **libaes** and the runtime startup module **crtsg.o**. Second, it can be included by itself with the **library** option **-lvdi**. This option must come at the *end* of the **cc** command line, or the library will not be linked in.

See Also
AES, ar, crtsg.o, gemdefs.h, library, nm, TOS, vdibind.h

#line — Preprocessor instruction

Reset line numbering
#line *number*
#line *number filename*
#line *manifest constants*

#line is a C preprocessor instruction that resets the line numbering within a file. It takes three forms. The first, **#line** *number*, resets the current line number to *number*. The second, **#line** *number filename*, resets the line number to *number* and resets the name of the file that the compiler thinks is the source file to *filename*. Finally, the form **#line** *manifest constants* contains manifest constants that have been set by earlier preprocessor instructions, such as **#define**; when the constants are interpreted by the **cpp**, the **#line** instruction will then resemble one of the first two forms. Most often, it is used to ensure that error messages point to the correct line in the program's source code.

Note that this instruction normally is not used by a programmer. It is used by a program generator to associate errors in generated C code with the original sources; for example, the program generator **yacc** will use **#line** instructions to link the C code it generates with the yacc code written by the programmer.

See Also
cpp
The C Programming Language, page 208

Notes
The '**#**' of this instruction must appear in the *first*, or leftmost, column on a line, or it will be ignored by **cpp**.

Line A — Technical information

Line A is the interface to the Atari ST's assembly-language-level graphics routines.

If the machine instructions of the 68000 are sorted by their bit patterns, they may be categorized into 16 "lines", according to the value of the high nybble of the instruction word. Lines 1, 2, and 3, for instance, give the **move** instructions. Lines A and F are not used by the 68000 instruction set, so the processor traps when it encounters instructions with these initial bit patterns. Line F is used by the Atari ROM to make GEM AES fit into the ROM. Line A is used to call the low-level graphics routines.

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

Most functions pass their parameters through the structure **la_data**. **la_data** is referenced through a pointer in in the structure **la_init**, which is initialized by function **linea0**. The exceptions are **linea7**, which takes the structure **la_blit**; **lineac**, which takes a pointer; and **linead**, which takes two pointers. All functions and structures are declared in the header file **linea.h**, which also contains a number of macros used to access elements within the Line A structures.

The following briefly summarizes the Line A functions:

linea0	Initialize
linea1	Put pixel
linea2	Get pixel
linea3	Draw a line
linea4	Draw a horizontal line
linea5	Draw a filled rectangle
linea6	Draw a filled polygon
linea7	Bit blit
linea8	Text blit
linea9	Show the mouse's pointer
lineaa	Hide the mouse's pointer
lineab	Transform the mouse's pointer
lineac	Erase a sprite
linead	Draw a sprite
lineae	Copy a raster form
lineaf	Seedfill

Examples

The first example demonstrates **linea3**, **linea5**, and **linea8**. When compiled, it takes four arguments, in decimal: an ASCII character; a column number (0 through 79); a row number (0 through 23); and a mode number (0 through 63). The mode indicates how the character named in the first argument is displayed.


```
#include <stdio.h>
#include <linea.h>
struct la_font *fontp;   /* font pointer for linea interface */
char line[100], *p;
char scr_wrk[1024];      /* area for graphics */
int scr_fat, scr_chi;    /* length and diep for underline */

/*
 * Put a character on the screen.
 */
put_scr(c, x, y, mode)
int c;                          /* character to put out */
int x, y;                       /* x & y coordinates on 80*25 screen */
int mode;                       /* see vst_effects for list of codes */

{
        unsigned int tmp;
        static long patmsk = -1;

        tmp = c - fontp->font_low_ade;
        DELX = fontp->font_char_off[tmp+1] -
                (SRCX = fontp->font_char_off[tmp]);
        DSTX = x << 3;
        DSTY = y << 4;
        WMODE = 0;              /* replace mode */
        STYLE = (mode & 7);

        if(mode & 8) {                          /* reverse */
                X2 = (X1 = DSTX) + scr_fat;
                Y2 = (Y1 = DSTY) + scr_chi;
                PATPTR = &patmsk;
                PATMSK = 1;
                CLIP = 0;
                linea5();                       /* filled rectangle */
                WMODE = 2;                      /* xor mode */
        }

        if(mode & 16) {                         /* underline */
                X2 = (X1 = DSTX) + scr_fat;
                Y2 = Y1 = DSTY + scr_chi;
                linea8();
                LNMASK = -1;
                WMODE = 2;
                linea3();
        }
        else
                linea8();
}
```

```
/* initialize material for screen */
init_scr() {
        linea0();                       /* initialize linea */
        linea9();                       /* hide mouse */
        fontp = la_init.li_a1[2];       /* 8x16 system font */
        FBASE = fontp->font_date;
        FWIDTH = fontp->font_width;
        TEXTFG = 1;                     /* text foreground white */
        SRCY = 0;
        DELY = fontp->font_height;
        scr_fat = fontp->font_fatest;
        scr_chi = fontp->font_height - 1;

        COLBIT0 = 1;
        COLBIT1 = 0;
        COLBIT2 = 0;
        COLBIT3 = 0;
        LITEMSK = 0x5555;
        SKEWMSK = 0x1111;
        SCRTCHP = scr_wrk;
        WEIGHT = 1;
        LSTLIN = -1;
}

init_msg() {
        printf("\033EProgram to demonstrate some linea capabilities\n");
        printf("Each line should have four decimal numbers or 'quit'\n");
        printf("The ASCII value of the char 'A'==65, etc.\n");
        printf("The x and y coordinates relative to a 25X80 screen\n");
        printf("The mode  1=thicken  2=grey 4=italic\n");
        printf("          8=reverse 16=underline\n");
        printf("Combinations work but some are weird\n\n");
}

main() {
        int c, x, y, m;
        init_scr();
        init_msg();

        for(;;) {
                printf("\033A\033K> ");
                fflush(stdout);
                gets(line);
                if(!strcmp(line, "quit"))
                        return(0);
                sscanf(line, "%d %d %d %d", &c, &x, &y, &m);
                put_scr(c, x, y, m);
        }
}
```

The second example uses **linea5** to draw a filled rectangle. Typing any key ends the display.


```
#include <linea.h>
#include <osbind.h>
box(i, j)
{
        long patmsk = -1;               /* pattern all ones */

        WMODE = 2;                      /* xor mode */
        PATPTR = &patmsk;
        PATMSK = 1;                     /* sizeof pattern */
        CLIP = 0;                       /* no clipping */
        X1 = Y1 = i;
        X2 = Y2 = j;
        linea5();                       /* draw box */
}

main(){
        int i;
        linea0();
        linea9();
        Cconws("\033E\033f Any key stops the display");

        for(;Cconis() == 0;)
                for(i = 50; i < 200; i++)
                        box(i, 400-i);

        Cconin();                       /* eat char */
        Cconws("\033e\n");
}
```

See Also

linea.h, TOS, VDI

Notes

Line A is described in chapter 3.4 of *Atari ST Internals*, and in unpublished Atari documentation. These functions are extremely complex, and documentation is not readily available. Programmers who wish to use these routines are well advised to use the above example as a model for testing the Line A functions and studying how they manipulate the screen.

linea.h — Header file

Declare Atari line A routines

linea.h is the header file that declares the the Atari's Line A routines. It also defines all specialized structures used by them.

See Also

header file, Line A, TOS

line feed — Character constant

Mark Williams C recognizes the literal character '\n' for the ASCII line feed character LF (octal 012). This character may be used as a character constant or in a string constant.

See Also

ASCII, character constant

Notes

On many systems, \n both feeds the line and tosses the carriage; however, on the Atari ST \n must be used with \r if the program does not work through STDIO.

Note that to read a file that includes line-feed characters, it must be opened in binary mode. See the entry for **fopen** for more information.

lmalloc – General function (libc)

Allocate dynamic memory

char *lmalloc(*size*) unsigned long *size*;

lmalloc helps to manage an a program's arena. It uses a circular, first-fit algorithm to select an unused block of at least *size* bytes, marks the portion it uses, and returns a pointer to it. The function **free** can be used to return allocated memory to the free memory pool.

Unlike the related function **malloc**, **lmalloc** takes an unsigned long as its *size* argument, which allows allocation of memory blocks larger than 64 kilobytes.

Example

For an example of a related function, see **malloc**.

See Also

arena, calloc, free, lcalloc, lrealloc, malloc, notmem, realloc, setbuf

Diagnostics

lmalloc returns NULL if insufficient memory is available. It prints a message and calls **abort** if it discovers that the arena has been corrupted, which most often occurs by storing past the bounds of an allocated block.

localtime – Time function (libc)

Convert system time to calendar structure

#include <time.h>

tm *localtime(*timep*) time_t *timep;

localtime converts the TOS internal time into the form described in the structure **tm**.

timep points to the system time. It is declared to be of type **time_t**, which is defined in the header file **time.h** as being equivalent to a **long**. The system time, in turn, is returned by the function **time**. Mark Williams C defines the system time to be the number of seconds since January 1, 1970 0h00m00s GMT.

localtime returns a pointer to the structure, **tm**, which is also defined in **time.h**, as follows:


```
struct tm {
        int     tm_sec;         /* current time, second */
        int     tm_min;         /* current time, minute */
        int     tm_hour;        /* current time, hour */
        int     tm_mday;        /* day of the month */
        int     tm_mon;         /* month (0-11) */
        int     tm_year;        /* year */
        int     tm_wday;        /* day of the week */
        int     tm_yday;        /* day of the year */
        int     tm_isdst;       /* daylight savings flag */
};
```

The function **asctime** turns **tm** into an ASCII string.

Unlike its cousin **gmtime**, **localtime** returns the local time, including conversion to daylight saving time, if applicable. The daylight saving time flag indicates whether daylight saving time is now in effect, *not* whether it is in effect during some part of the year. Note, too, that the time zone is set by **localtime** every time the value returned by

```
getenv("TIMEZONE")
```

changes. See the Lexicon entry for **TIMEZONE** for more information on how Mark Williams C handles time zone settings.

Example

The following example recreates the function **asctime**. It builds a string somewhat different from that returned by **asctime** to demonstrate how to manipulate the **tm** structure.

```
#include <time.h>

char *month[] = {
        "January", "February", "March", "April",
        "May", "June", "July", "August", "September",
        "October", "November", "December"
};

char *weekday[] = {
        "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"
};

main()
{
        char buf[20];
        time_t tnum;
        tm *ts;
        int hour = 0;

        time(&tnum); /* get time from system */

        /* convert time to tm struct */
        ts=localtime(&tnum);
```

```
        if(ts->tm_hour==0)
                sprintf(buf,"12:%02d:%02d A.M.",
                        ts->tm_min, ts->tm_sec);

        else
                if(ts->tm_hour>=12) {
                        hour=ts->tm_hour-12;
                        if (hour==0)
                                hour=12;
                        sprintf(buf,"%02d:%02d:%02d P.M.",
                                hour, ts->tm_min,ts->tm_sec);

                } else
                        sprintf(buf,"%02d:%02d:%02d A.M.", ts->tm_hour,
                                ts->tm_min,ts->tm_sec);

        printf("\n%s %d %s 19%d %s\n",
                weekday[ts->tm_wday], ts->tm_mday,
                month[ts->tm_mon], ts->tm_year, buf);

        printf("Today is the %d day of 19%d\n",
                ts->tm_yday, ts->tm_year);

        if(ts->tm_isdst)
                printf("Daylight Saving Time is in effect\n");
        else
                printf("Daylight Saving Time is not in effect\n");
}
```

See Also

gmtime, time (overview), TIMEZONE

Notes

localtime returns a pointer to a statically allocated data area that is overwritten by successive calls.

log — Mathematics function (libm)

Compute natural logarithm
#include <math.h>
double log(*z*) double *z*;

log returns the natural (base e) logarithm of its argument *z*.

Example

For an example of this function, see the entry for **exp**.

See Also

log10, mathematics library

Diagnostics

A domain error in **log** (*z* is less than or equal to 0) sets **errno** to **EDOM** and returns 0.

log10 — Mathematics function (libm)

Compute common logarithm
#include <math.h>
double log10(*z*) double *z*;

log10 returns the common (base 10) logarithm of its argument *z*.

Example

For an example of this function, see the entry for **exp**.

See Also

log, mathematics library

Diagnostics

A domain error in **log10** (*z* is less than or equal to 0) sets **errno** to **EDOM** and returns 0.

Logbase — xbios function 3 (osbind.h)

Read the logical screen's display base
#include <osbind.h>
#include <xbios.h>
char *Logbase()

Logbase reads the screen's logical display base, and returns a pointer to it.

The logical base is where the screen-drawing primitives do their work. This is in contrast to the physical base, which is returned by **Physbase**; the latter is where the display hardware gets the image that is displayed on the monitor. This differentiation allows you to draw one pattern while displaying another.

Example

This example gets the logical and physical screen base addresses. If they are the same, it fills the top of the screen with the pattern 10101010; otherwise, it prints out each address. In the case of this program, they will generally be equal.

```
#include <osbind.h>

main() {
        long *lbase;
        long *pbase;
        int x;

        lbase = (long *) Logbase();          /* Get logical screen */
        pbase = (long *) Physbase();         /* Get physical screen */

        if(pbase == lbase) {
                for(x=0;x<0x1000;x++)
                        *pbase++ = 0xAAAAAAAAL;
        } else {
                printf("The logical screen is at %lx\n", lbase);
                printf("The physical screen is at %lx\n", pbase);
        }
        exit();
}
```

See Also

Physbase, Setscreen, TOS, xbios

long — C keyword

Data type

A **long** is a numeric data type. By definition, a **long** is the largest integer data type; it cannot be smaller than an **int**, although on some machines an **int** and a **long** will be the same size. On most machines, **sizeof long** will equal two machine words, or four **chars** (31 data bits plus a sign bit).

See Also

C keywords, C language, data formats, declarations, int

longjmp — General function (libc)

Return from a non-local goto

#include <setjmp.h>
int longjmp(*env*, *rval***) jmp_buf** *env*; **int** *rval*

The function call is the only mechanism that C provides to transfer control between functions. This mechanism is inadequate for some purposes, such as handling unexpected errors or interrupts at lower levels of a program. To answer this need, **longjmp** provides a non-local *goto*.

longjmp restores an environment that had been saved by a previous **setjmp** call. It returns the value *rval* to the caller of **setjmp**, just as if the **setjmp** call had just returned. Note that **longjmp** must not restore the environment of a routine that has already returned. The type declaration for **jmp_buf** is in the header file **setjmp.h**. The environment saved includes the program counter, stack pointer, and stack frame. These routines do not restore register variables in the environment returned.

See Also

setjmp, setjmp.h

Notes

Programmers should note that many user-level routines cannot be interrupted and reentered safely. For that reason, improper use of **longjmp** and **setjmp** will result in the creation of mysterious and irreproducible bugs. Do not attempt to use **longjmp** within an exception handler.

lrealloc — General function (libc)

Reallocate dynamic memory

char *lrealloc(*ptr*, *size***)**
char ***ptr*; **unsigned long *size*;

lrealloc helps to manage a program's arena. It returns a block of *size* bytes that holds the contents of the old block, up to the smaller of the old and new sizes. **lrealloc** tries to return the same block, truncated or extended; if *size* is smaller than the size of the old block, **lrealloc** will return the same *ptr*.

Unlike the related function **realloc**, **lrealloc** takes an unsigned long as its *size* argument, and therefore can reallocate a memory blocks that is larger than 64 kilobytes.

See Also

arena, calloc, free, lcalloc, lmalloc, malloc, notmem, realloc, setbuf

Diagnostics

lrealloc returns NULL if insufficient memory is available. It prints a message and calls **abort** if it discovers that the arena has been corrupted, which most often occurs by storing past the bounds of an allocated block. **lrealloc** will behave capriciously if handed a fallacious *ptr*.

ls — Command

List directory's contents

ls [-adflrtw] [*file* ...]

ls prints information about each *file*. Normally, **ls** sorts by file name and prints only the name of each *file*. If a directory name is given as an argument, **ls** sorts and lists its contents, not including '.' and '..'. If no *file* is named, **ls** lists the contents of the current directory.

The following options control how **ls** sorts and displays its output.

-a Print all directory entries, including '.', '..', any hidden files, and volume ID's.

-d Treat directories as if they were files.

-f Flag all directories with a trailing backslash '\'.

-l Print information in long format. The fields give mode bits, size in bytes, date of last update, and file name.

-r Reverse the sense of the sort.

-t Sort by time, newest first.

-w Print output in columns; write a backslash '\' after the name of every directory.

The mode field in the long list format consists of four characters. The first character will be one of the following:

-	regular file
d	directory
s	system file
v	volume identifier

The next two characters are **r** or - if the file is read-only, and **w** if the file can be written to. The fourth character is **h** if the file is hidden.

See Also

lc, commands, msh

lseek — UNIX system call (libc)

Set read/write position
long lseek(*fd*, *where*, *how*)
int *fd*, *how*; **long** *where*;

lseek changes the *seek position*, or the point within a file where the next read or write operation is performed. *fd* is the file's file descriptor, which is returned by **open**.

where and *how* describe the new seek position. *where* gives the number of bytes that you wish to move the seek position; it is measured from the beginning of the file if *how* is zero, from the current seek position if *how* is one, or from the end of the file if *how* is two. A successful call to **lseek** returns the new seek position. For example,

```
position = lseek(filename, 100, 0);
```

moves the seek position 100 bytes past the beginning of the file; whereas

```
position = lseek(filename, 0, 1);
```

merely returns the current seek position, and does not change the seek position at all.

lseek differs from its cousin **fseek** in that **lseek** is an TOS call and uses a file descriptor, whereas **fseek** is a C function and uses a FILE pointer.

See Also

STDIO, UNIX routines

Diagnostics

lseek returns -1L on an error, such as seeking to a negative position. If no error occurs, it returns zero.

Notes

Note that if **lseek** goes beyond the end of the file, it will not return an error message until the corresponding read or write is performed.

Note that some operating systems, such as MS-DOS, set the displacement from the file descriptor in bytes; others, such as the VAX VMS, set the displacement in sectors. If you want your programs to be fully portable, you should avoid handing an absolute value to **lseek**.

ltom — Command

Redraw the screen from low to medium resolution

ltom

ltom redraws the screen, moving from low to medium resolution.

See Also

commands, htom, mtoh, mtol, TOS

lvalue — Definition

An **lvalue** is an expression that designates a region of storage. The name comes from the assignment expression **e1=e2;**, in which the left operand must be an lvalue.

An identifier has both an *lvalue* (its address) and an *rvalue* (its contents). Some C operators require lvalue operands; for example, the left operand of an assignment statement must be an lvalue. Some operators give lvalue results; for example, if *e* is a pointer expression, **e* is an lvalue that designates the object to which *e* points.

Note that a *variable* can be used as an lvalue, whereas a constant cannot. For example, you cannot say

```
6 = (foo+bar);
```

A pointer is a variable, and can be manipulated within limits. An array name, however, is a constant and cannot be altered legally. Thus, the code

```
int foo[10];
int *bar;
foo = bar;
```

will generate an error message when you attempt to compile it, whereas the code

```
int foo[10];
int *bar;
bar = foo;
```

will not.

The following example shows the use of both an lvalue and a rvalue:

```
int i, *ip;

ip = &i;          /* ip is an lvalue, i and &i are rvalues */
i = 3;            /* i is an lvalue, 3 is an rvalue */
*ip = 4;          /* *ip is an lvalue, 4 is an rvalue */
```

See Also

rvalue

M

macro — Definition

A **macro** is a body of text that is given a name. When the name is used in a program, it is replaced with the text to which it refers; this is called macro *expansion*. For example, **getchar** is a macro that consists of the function call **getc(stdin)**.

Note that because macros may employ an argument *n* times, any arguments that have side effects will have the side effect repeated *n* times as well, which may be undesirable.

See Also

function

main — Technical information

Introduce program's main function

A C program consists of a set of functions, one of which must be called **main**. This function is called from the runtime startup routine after the runtime environment has been initialized.

Programs can terminate in one of two ways. The easiest is simply to have the **main** routine **return**. Control returns to the runtime startup; it closes all open file streams and otherwise cleans up, and then returns control to the operating system, passing it the value returned by **main** as exit status.

In some situations (errors, for example), it may be necessary to stop a program, and you may not want to return to **main**. Here, you can use **exit**; it cleans up the debris left by the broken program and returns control directly to the operating system.

A second exit routine, called **_exit**, quickly returns control to the operating system without performing any cleanup. This routine should be used with care, because bypassing the cleanup will leave files open and buffers of data in memory.

Programs compiled by Mark Williams C return to the program that called them; if they return from **main** with a value or call **exit** with a value, that value is returned to their caller. Programs that invoke other programs through the **system**, **execve**, or **Pexec** functions check the returned value to see if these secondary programs terminated successfully.

See Also

argc, argv, envp, exit, _exit, runtime startup

make — Command

Program building discipline

make [*option ...*] [*argument ...*] [*target ...*]

make helps you build programs that consist of more than one file of source code.

Complex programs often consist of several *object modules*, each of which is the product of compiling a *source file*. A source file may refer to one or more Include files, which can also be changed. Recompiling and relinking complicated programs can be difficult and tedious.

make regenerates programs automatically. It follows a specification of the structure of the program that you write into a file called **makefile**. **make** also checks the date and time that TOS has recorded for each source file and its corresponding object module; to avoid unnecessary recompilation, **make** will recompile a source file only if it has been altered since its object module was last compiled.

The makefile

A **makefile** consists of three types of instructions: *macro definitions*, *dependency definitions*, and *commands*.

A macro definition simply defines a macro for use throughout the **makefile**; for example, the macro definition

```
FILES=file1.o file2.o file3.o
```

Note the use of the equal sign '='.

A dependency definition names the object modules used to build the target program, and source files used to build each object module . It consists of the *target name*, or name of the program to be created, followed by a colon ':' and the names of the object modules that build it. For example, the statement

```
example:  $(FILES)
```

uses the macro **FILES** to name the object modules used to build the program **example**. Likewise, the dependency definition

```
file1.o:  file1.c macros.h
```

defines the object module **file1.o** as consisting of the source file **file1.c** and the header file **macros.h**.

Finally, a command line details an action that **make** must perform to build the target program. Each command line must begin with a space or tab character. For example, the command line

```
        cc -o example $(FILES)
```

gives the **cc** command needed to build the program **example**. Note that the **cc** command lists the *object modules* to be used, *not* the source files.

Finally, you can embed comments within a **makefile**. **make** recognizes any line that begins with a pound sign '#' as being a comment, and ignores it.

make searches for **makefile** first in directories named in the environmental variable **PATH**, and then in the current directory.

Dependencies

The **makefile** specifies which files depend upon other files, and how to recreate the dependent files. For example, if the target file **test** depends upon the object module **test.o**, the dependency is as follows:

```
test:  test.o
        cc -o test test.o
```

make knows about common dependencies, e.g., that **.o** files depend upon **.c** files with the same base name. The target **.SUFFIXES** contains the suffixes that **make** recognizes.

make also has a set of rules to regenerate dependent files. For example, for a source file with suffix **.c** and a dependent file with the suffix **.o**, the target **.c.o** gives the regeneration rule:

```
.c.o:
        cc -c $<
```

The **-c** option to the **cc** commands tells **cc** not to link or erase the compiled object module. **$<** is a macro that **make** defines; it stands for the name of the file that causes the current action. The default suffixes and rules are kept in the files **mmacros** and **mactions**. The dependencies can be changed by editing these files. Both of these should be kept in one of the directories named in the **LIBPATH** environmental variable.

Macros

To simplify the writing of complex dependencies, **make** provides a *macro* facility. To define a macro, write

NAME = string

The *string* is terminated by the end-of-line character, so it can contain blanks. To refer to the value of the macro, use a dollar sign '**$**' followed by the macro name enclosed in parentheses:

$(NAME)

If the macro name is one character, parentheses are not necessary. **make** uses macros in the definition of default rules:

```
.c.o:
        $(CC) $(CFLAGS) -c $<
```

where the macros are defined as

```
CC=cc
CFLAGS=-V
```

The other built-in macros are:

$*	target name, minus suffix
$@	full target name
$<	list of referred files
$?	referred files newer than target

Each command line *argument* should be a macro definition of the form

```
OBJECT=a.o b.o
```

Arguments that include spaces must be surrounded by quotation marks, because blanks are significant to the micro-shell **msh**.

Note that you can override any built-in macro by resetting its value in the environment.

Options

The following lists the options that can be passed to **make** on its command line.

-d (Debug) Give verbose printout of all decisions and information going into decisions.

-f *file* *file* contains the **make** specification. If this option does not appear, **make** uses the file **makefile**, which is sought first in the directories named in the **PATH** environmental variable, and then in the current directory.

-i Ignore all errors from commands, and continue processing. Normally, **make** exits if a command returns an error.

-n Test only; suppresses actual execution of commands. Note that if **make** will not run due to memory limitations, you can use this option to generate a script whose commands can then be executed under **msh**; for example:

```
make -n > mscript; set verbose; . mscript; unset verbose
```

msh, however, will not pay attention to error status in the same way as **make**.

-p Print all macro definitions and target descriptions.

-q Return a zero exit status if the targets are up to date. Do not execute any commands.

-r Do not use the built-in rules that describe dependencies.

-s Do not print command lines when executing them. Commands preceded by '@' are not printed, except under the **-n** option.

-t (Touch option) Force the dates of targets to be the current time, and bypass actual regeneration.

Invoking make

make can be used either from the micro-shell **msh**, or from the TOS desktop.

To use **make** from the TOS desktop, its suffix must be changed to **TOS** or **TTP**. Once this is done, you can invoke **make** simply by pointing to the appropriate icon with your mouse and clicking it. When the **Open Application** box appears, enter the options and target you want. **make** reads whatever **makefile** is in the current directory, and executes its instructions. It cannot accept options from the desktop, however.

If you wish to use **make** from **msh**, simply invoke **msh** from TOS, then enter the **make** command as you normally would, including options and a path name for the **makefile**, should it be in a directory other than one that you have previously defined in the environmental parameter **PATH**.

See Also

as, cc, commands, msh

Diagnostics

make reports its exit status if it is interrupted or if an executed command returns error status. It replies "Target *name* not defined" or "Don't know how to make target *name*" if it cannot find appropriate rules.

Notes

The order of items in **mmacros\.SUFFIXES** is significant. The consequent of a default rule (e.g., **.o**) must *precede* the antecedent (e.g., **.c**) in the entry **.SUFFIXES**. Otherwise, **make** will not work properly.

malloc — General function (libc)

Allocate dynamic memory
char *malloc(*size***) unsigned** *size*;

malloc helps to manage a program's free-space arenas. It uses a circular, first-fit algorithm to select an unused block of at least *size* bytes, marks the portion it uses, and returns a pointer to it. The function **free** returns allocated memory to the free memory pool.

Each area allocated by **malloc** is rounded up to the nearest even number and preceded by an **unsigned int** that contains the true length. Thus, if you ask for one byte, you will get four, and the **unsigned** that precedes the newly allocated area will be set to four.

When an area is freed, its low order bit is turned on; consolidation occurs when **malloc** passes over an area as it searches for space. The end of each arena contains a block with a length of zero, followed by a pointer to the next arena. Arenas point in a circle.

The most common problem with **malloc** occurs when a program modifies more space than it allocates with **malloc**. This can cause later **mallocs** to go into a loop.

Example

This example reads from the standard input up to *NITEMS* items, each of which is up to *MAXLEN* long, sorts them, and writes the sorted list onto the standard output. It demonstrates the functions **qsort**, **malloc**, **free**, **exit**, and **strcmp**. You may want to use as input what the example for **Random** has output. For an example of how to use **malloc** in a TOS application, see the entry for **Fgetdta**.

```
#include <stdio.h>
#define NITEMS 512
#define MAXLEN 256
char *data[NITEMS];
char string[MAXLEN];

main() {
        register char **cpp;
        register int count;
        extern int compare();
        extern char *malloc();
        extern char *gets();

        for (cpp = &data[0]; cpp < &data[NITEMS]; cpp++) {
                if (gets(string) == NULL)
                        break;
                if ((*cpp = malloc(strlen(string) + 1)) == NULL)
                        exit(1);
                strcpy(*cpp, string);
        }
        count = cpp - &data[0];
        qsort(data, count, sizeof(char *), compare);
        for (cpp = &data[0]; cpp < &data[count]; cpp++) {
                printf("%s\n", *cpp);
                free(*cpp);
        }
        exit(0);
}

compare(p1, p2)
register char **p1, **p2;
{
        extern int strcmp();
        return(strcmp(*p1, *p2));
}
```

See Also

arena, calloc, free, lcalloc, lmalloc, lrealloc, notmem, realloc, setbuf

Diagnostics

malloc returns NULL if insufficient memory is available.

The related function **lmalloc** takes an unsigned long as its *size* argument, and therefore can allocate memory blocks that are larger than 64 kilobytes.

Notes

The commonest error associated with **malloc** is failing to declare it properly. You should always declare **malloc** as returning a pointer to **char**.

Malloc — gemdos function 72 (osbind.h)

Allocate dynamic memory
#include <osbind.h>
long Malloc(*n*) long *n*;

Malloc allocates dynamic memory. *n* contains either the number of bytes to be allocated, or the number -1L (0xFFFFFFFF), which returns all available memory. If *n* contains the number of bytes to be allocated, **Malloc** returns a pointer to the starting address of the memory allocated; if *n* contains -1L, then **Malloc** returns the size of the largest contiguous block of memory. In either case, **Malloc** returns 0 upon failure.

Examples

This example displays the output of **Malloc** when given -1 as its argument.

```
#include <osbind.h>
#define MGRAIN 32768L

main() {
        register long f1, f2, f3, f4;
        register char *p1, *p2;

        f1 = (long)Malloc(-1L);
        p1 = Malloc(MGRAIN);
        f2 = (long)Malloc(-1L);
        p2 = Malloc(MGRAIN);
        f3 = (long)Malloc(-1L);
        Mfree(p1);
        f4 = (long)Malloc(-1L);
        Mfree(p2);

        printf("%lx %lx %lx %lx %lx\n", f1, f2, f3, f4, Malloc(-1L));
        exit(0);
}
```

See Also

gemdos, Mfree, Mshrink, TOS

Notes

As of this writing, **Malloc** appears to have some peculiarities. You should always use **Malloc** to allocate even-sized blocks of memory. Always **Mfree** memory in the *reverse* order of allocation. Finally, try to **Malloc** a few pieces of memory; there appears to be an undocumented limit on the number of times **Malloc** can be called by a given program. Though large, this number is finite; when it is exceeded, **Malloc** will return NULL even though considerable amounts of memory are still available.

manifest constant — Definition

A **manifest constant** is a numeric constant that is given a name so it can be defined differently under different computing environments. An example is **EOF**, the end-of-file marker, which has wildly different representations under different operating systems. Note, too, that numerals are manifest constants by definition.

The use of manifest constants in programs helps to ensure that code is portable by isolating the definition of these elements in a single header file, where they need to be changed only once.

See Also

#define, EOF, header file, NULL, portability

mantissa — Definition

In mathematics, a **mantissa** is the fractional part of a logarithm. In the context of C, "mantissa" often is used to describe the fractional portion of a floating point number; according to Knuth, however, the proper term is *fraction*.

See Also

data formats, double, float, frexp

math.h — Header file

Declare mathematics functions
#include <math.h>

math.h is the header file to be included with programs that use any of Mark Williams C's mathematics routines. It includes the following: definitions for mathematical functions; error return values, as used by the **errno** function; definitions of mathematical constants, e.g., **HUGE_VAL**; the definition of structure **cpx**, which describes complex variables; definitions of internal compiler functions; and, finally, declarations of all mathematical functions.

See Also

library, libm, mathematics library

mathematics library — Overview

The following mathematics routines are available with Mark Williams C:

acos	calculate inverse cosine
asin	calculate inverse sine
atan	calculate inverse tangent
atan2	calculate inverse tangent of quotient
cabs	calculate complex absolute value
cos	calculate cosine
cosh	calculate hyperbolic cosine
exp	calculate exponent
fabs	calculate absolute value function
floor	calculate floor function
hypot	calculate hypotenuse
j0	calculate Bessel function, order 0
j1	calculate Bessel function, order 1
jn	calculate Bessel function, order *n*
log	calculate natural logarithm
log10	calculate common logarithm
pow	calculate power
sin	calculate sine
sinh	calculate hyperbolic sine
sqrt	calculate square root
tan	calculate tangent
tanh	calculate hyperbolic tangent

See Also

libm.a, Lexicon, math.h

Notes

When programs that contain mathematics routines are compiled, the mathematics libraries must be called specifically on the **cc** command line. For example, to compile the example presented under the entry for **acos**, use the following **cc** command line:

```
cc -f -o acos.prg acos.c -lm
```

The **-f** option links in the floating point routines for **printf**, while the **-lm** option links in the mathematics libraries. Note that the **-lm** option must come *last* on the **cc** command line, or the library will not be searched properly.

maxmem — External data

extern unsigned int maxmem;

maxmem is an external variable that sets the maximum size of the program's data area. You can set **maxmem** in your program to protect a portion of memory from the memory allocation routine **sbrk**; otherwise, **maxmem** is set to the end of physical memory by the C runtime startup routine.

See Also

_end, malloc, sbrk

me — Command

MicroEMACS screen editor
me [**-e**] [*file ...*]

me is the command for MicroEMACS, the screen editor for Mark Williams C. With MicroEMACS, you can insert text, delete text, move text, search for a string

and replace it, and perform many other editing tasks. MicroEMACS reads text from files and writes edited text to files; it can edit several files simultaneously, while displaying the contents of each file in its own screen window.

Screen layout

If the command **me** is used without arguments, MicroEMACS opens an empty buffer. If used with one or more file name arguments, MicroEMACS will open each of the files named, and display its contents in a window. If a file cannot be found, MicroEMACS will assume that you are creating it for the first time, and create an appropriately named buffer and file descriptor for it.

The last line of the screen is used to print messages and inquiries. The rest of the screen is portioned into one or more *windows* in which text is displayed. The last line of each window shows whether the text has been changed, the name of the buffer, and the name of the file associated with the window.

MicroEMACS notes its *current position*. It is important to remember that the current position is always to the *left* of the cursor, and lies *between* two letters, rather than at one letter or another. For example, if the cursor is positioned at the letter 'k' of the phrase "Mark Williams", then the current position lies *between* the letters 'r' and 'k'.

Commands and text

The printable ASCII characters, from ' ' to '~', can be inserted at the current position. Control characters and escape sequences are recognized as *commands*, described below. A control character can be inserted into the text by prefixing it with **<ctrl-Q>** (that is, hold down the **<control>** key and type the letter 'Q').

There are two types of commands to remove text. *Delete* commands remove text and throw it away, whereas *kill* commands remove text but save it in the *kill buffer*. Successive kill commands append text to the previous kill buffer. Moving the cursor before you kill a line will empty the kill buffer, and write the line just killed into it.

Search commands prompt for a search string terminated by **<return>** and then search for it. Case sensitivity for searching can be toggled with the command **<esc>@**. Typing **<return>** instead of a search string tells MicroEMACS to use the previous search string.

Some commands manipulate words rather than characters. MicroEMACS defines a word as consisting of all alphabetic characters, plus '_' and '$'. Usually, a character command is a control character and the corresponding word command is an escape sequence. For example, **<ctrl-F>** moves forward one character and **<esc>F** moves forward one word. Note that the MicroEMACS commands are not case sensitive; for example, **<ctrl-F>** and **<ctrl-f>** are identical.

Text can also be handled in blocks. MicroEMACS defines a block of text as all the text that lies between the *mark* and the current position of the cursor. For example, typing **<ctrl-W>** kills all text from the mark to the current position of the cursor; this is useful when moving text from one file to another. When you invoke MicroEMACS, the mark is set at the beginning of the file; you can reset the mark to the cursor's current position by typing **<ctrl-@>**.

Using MicroEMACS with the compiler

MicroEMACS can be invoked automatically by the compiler command **cc** to help you repair all errors that occur during compilation. The **-A** option to **cc** causes MicroEMACS to be invoked automatically when an error occurs. The compiler error messages are displayed in one window, the source code in the other, and the cursor is at the line on which the first error occurred. When the text is altered, exiting from MicroEMACS automatically recompiles the file.

This cycle will continue either until the file compiles without error, or until you break the cycle by typing **<ctrl-U> <ctrl-X> <ctrl-C>**.

The option **-e** to the **me** command allows you to invoke the error buffer by hand.

The MicroEMACS help facility

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

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

```
Topic:
```

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

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

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

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

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

Options

The following list gives the MicroEMACS commands. They are grouped by function, e.g., *Moving the cursor*. Some commands can take an *argument*, which specifies how often the command is to be executed. The default argument is 1. The command **<ctrl-U>** introduces an argument. By default, it sets the argument

to four. Typing <ctrl-U> followed by a number sets the argument to that number. Typing <ctrl-U> followed by one or more <ctrl-U>s multiplies the argument by four.

Moving the cursor

<ctrl-A> Move to start of line.

<ctrl-B> (Back) Move backward by characters.

<esc>B Move backward by words.

<ctrl-E> (End) Move to end of line.

<ctrl-F> (Forward) Move forward by characters.

<esc>F (Forward) Move forward by words.

<esc>G Go to an absolute line number in a file. Same as **<ctrl-X>G**.

<ctrl-N> (Next) Move to next line.

<ctrl-P> (Previous) Move to previous line.

<ctrl-V> Move forward by pages.

<esc>V Move backward by pages.

<ctrl-X>= Print the current position.

<ctrl-X>G Go to an absolute line number in a file. Can be used with an argument; otherwise, it will prompt for a line number. Same as **<esc>G**.

<esc>! Move the current line to the line within the window given by *argument*; the position is in lines from the top if positive, in lines from the bottom if negative, and the center of the window if zero.

<esc>< Move to the beginning of the current buffer.

<esc>> Move to the end of the current buffer.

Killing and deleting

<ctrl-D> (Delete) Delete next character.

<esc>D Kill the next word.

<ctrl-H> If no argument, delete previous character. Otherwise, kill *argument* previous characters.

<ctrl-K> (Kill) With no argument, kill from current position to end of line; if at the end, kill the newline. With argument set to one, kill from beginning of line to current position. Otherwise, kill *argument* lines forward (if positive) or backward (if negative).

previous = arrant

<ctrl-W> Kill text from current position to mark.

<ctrl-X><ctrl-O>
Kill blank lines at current position.

<ctrl-Y> (Yank back) Copy the kill buffer into text at the current position; set current position to the end of the new text.

<esc><ctrl-H>
Kill the previous word.

<esc><DEL>
Kill the previous word.

<DEL> If no argument, delete the previous character. Otherwise, kill *argument* previous characters.

Windows

<ctrl-X>1 Display only the current window.

<ctrl-X>2 Split the current window into two windows. This command is usually followed by **<ctrl-X>B** or **<ctrl-X><ctrl-V>**.

<ctrl-X>N (Next) Move to next window.

<ctrl-X>P (Previous) Move to previous window.

<ctrl-X>Z Enlarge the current window by *argument* lines.

<ctrl-X><ctrl-N>
Move text in current window down by *argument* lines.

<ctrl-X><ctrl-P>
Move text in current window up by *argument* lines.

<ctrl-X><ctrl-Z>
Shrink current window by *argument* lines.

Buffers

<ctrl-X>B (Buffer) Prompt for a buffer name, and display the buffer in the current window.

<ctrl-X>K (Kill) Prompt for a buffer name and delete it.

<ctrl-X><ctrl-B>
Display a window showing the change flag, size, buffer name, and file name of each buffer.

<ctrl-X><ctrl-F>
(File name) Prompt for a file name for current buffer.

<ctrl-X><ctrl-R>
(Read) Prompt for a file name, delete current buffer, and read the file.

<ctrl-X><ctrl-V>
(Visit) Prompt for a file name and display the file in the current window.

Saving text and exiting

<ctrl-X><ctrl-C>
Exit without saving text.

<ctrl-X><ctrl-S>
(Save) Save current buffer to the associated file.

<ctrl-X><ctrl-W>
(Write) Prompt for a file name and write the current buffer to it.

<ctrl-Z> Save current buffer to associated file and exit.

Compilation error handling

<ctrl-X>> Move to next error.

<ctrl-X>< Move to previous error.

Search and replace

<ctrl-R> (Reverse) Incremental search backward; a pattern is sought as each character is typed.

<esc>R (Reverse) Search toward the beginning of the file. Waits for entire pattern before search begins.

<ctrl-S> (Search) Incremental search forward; a pattern is sought as each character is typed.

<esc>S (Search) Search toward the end of the file. Waits for entire pattern before search begins.

<esc>% Search and replace. Prompt for two strings; then search for the first string and replace it with the second.

<esc>/ Search for next occurrence of a string entered with the **<esc>S** or **<esc>R** commands; this remembers whether the previous search had been forward or backward.

<esc>@ Toggle case sensitivity for searches. By default, searches are case insensitive.

Keyboard macros

<ctrl-X>(Begin a macro definition. MicroEMACS collects everything typed until the next **<ctrl-X>)** for subsequent repeated execution. **<ctrl-G>** breaks the definition.

<ctrl-X>) End a macro definition.

<ctrl-X>E (Execute) Execute the keyboard macro.

Change case of text

<esc>C (Capitalize) Capitalize the next word.

<ctrl-X><ctrl-L>
(Lower) Convert all text from current position to mark into lower case.

<esc>L (Lower) Convert the next word to lower case.

<ctrl-X><ctrl-U>
(Upper) Convert all text from current position to mark into upper case.

<esc>U (Upper) Convert the next word to upper case.

White space

<ctrl-I> Insert a tab.

<ctrl-J> Insert a new line and indent to current level. This is often used in C programs to preserve the current level of indentation.

<ctrl-M> (Return) If the following line is not empty, insert a new line; if empty, move to next line.

<ctrl-O> Open a blank line; that is, insert newline after the current position.

<tab> With argument, set tab fields at every *argument* characters. An argument of zero restores the default of eight characters. Note that setting the tab to any character other than eight causes space characters to be set in your file instead of tab characters.

Send commands to operating system

<ctrl-C> Suspend MicroEMACS and invoke a new copy of **msh**. Typing **exit** returns you to MicroEMACS and allows you to resume editing.

<ctrl-X>! Prompt for an **msh** command and execute it.

Setting the mark

<ctrl-@> Set mark at current position.

<esc>. Set mark at current position.

<ctrl><space>
Set mark at current position.

Help window

<ctrl-X>? Prompt for word for which information is needed.

<esc>? Search for word over which cursor is positioned.

<esc>2 Erase help window.

Miscellaneous

<ctrl-G> Abort a command.

<ctrl-L> Redraw the screen.

<ctrl-Q> (Quote) Insert the next character into text; used to insert control characters.

<esc>Q (Quote) Insert the next control character into the text. Same as **<ctrl-Q>**.

<ctrl-T> Transpose the characters before and after the current position.

<ctrl-U> Specify a numeric argument, as described above.

<ctrl-U><ctrl-X><ctrl-C>
Abort editing and re-compilation. Use this command to abort editing and return to TOS when you are using the **-A** option to the **cc** command.

<ctrl-X>F Set word wrap to *argument* column. If argument is one, set word wrap to cursor's current position.

<ctrl-X><ctrl-X>
Mark the current position, then jump to the previous setting of the mark. This is useful when moving text from one place in a file to another.

Diagnostics

MicroEMACS prints error messages on the bottom line of the screen. It prints informational messages (enclosed in square brackets '[' and ']' to distinguish them from error messages) in the same place.

MicroEMACS manipulates text in memory rather than in a file. The file on disk is not changed until you save the edited text. MicroEMACS prints a warning and prompts you whenever a command would cause it to lose changed text.

See Also

commands

Notes

Because MicroEMACS keeps text in memory, it does not work for extremely large files. It prints an error message if a file is too large to edit. If this happens when you first invoke a file, you should exit from the editor immediately. Otherwise, your file on disk will be truncated. If this happens in the middle of an editing session, however, delete text until the message disappears, then save your file and exit. Due to the way MicroEMACS works, saving a file after this error message has appeared will take more time than usual.

This version of MicroEMACS does not include many facilities available in the original EMACS display editor, which was written by Richard Stallman at M.I.T. In particular, it does not include user-defined commands or pattern search commands.

Note that the current version of MicroEMACS, including source code, is proprietary to Mark Williams Company. The code may be altered or otherwise changed for your personal use, but it may *not* be used for commercial purposes, and it may not be distributed without prior written consent by Mark Williams Company.

MicroEMACS is based upon the public domain editor by David G. Conroy.

me.a — Archive

me.a is an archive that holds the source files for the Mark Williams proprietary version of the MicroEMACS screen editor. If you wish to recompile MicroEMACS, you must first extract the source files from the archive. Use the command **cd** to move to the directory where you have stored this archive, then give **msh** the following command:

```
ar xv me.a
```

See Also

ar, me

Mediach — bios function 9 (osbind.h)

Check whether disk has been changed
#include <osbind.h>
#include <bios.h>
long Mediach(*drive***) int** *drive*;

Mediach checks whether a new disk has been inserted into a floppy-disk drive. *drive* is a number from zero to 15, and indicates which drive to check: zero indicates drive A, one indicates drive B, etc. **Mediach** returns zero if the medium has not been changed, one if it may have been changed, and two if it was changed.

Example

This example discovers whether the floppy disks have been changed.

```
#include <osbind.h>
main()
{
        int d, ds;
        char *status[3] = { "not", "possibly", "definitely" };

        for (d = 0; d < 2; d += 1) {
                ds = Mediach(d);
                printf("drive %c has ", d+'a');

                if (ds < 0 || ds > 2)
                        printf("bad status: %d\n", ds);
                else
                        printf("%s changed\n", status[ds]);
        }
}
```

See Also

bios, TOS

memchr — String function (libc)

Search a region of memory for a character

char *memchr(*region, character, n***)**
char ****region*;**
unsigned int *character, n*;

memchr searches the first *n* characters in *region* for *character*. It returns a pointer to *character* if it is found, or NULL if it is not.

Unlike the string-search function **strchr**, **memchr** searches a region of memory. Therefore, it does not stop when it encounters a null character.

See Also

strchr, string

memcmp — String function (libc)

Compare two regions

int memcmp(*region1, region2, count***)**
char ****region1, *region2*;**
unsigned int *count*;

memcmp compares *region1* with *region2* character by character for *count* characters.

If every character in *region1* is identical to its corresponding character in *region2*, then **memcmp** returns zero. If it finds that a character in *region1* has a numeric value greater than that of the corresponding character in *region2*, then it returns a number greater than zero. If it finds that a character in *region1* has a numeric value less than that of the corresponding character in *region2*, then it returns a number less than zero.

For example, consider the following code:

```
char region1[13], region2[13];
strcpy(region1, "Hello, world");
strcpy(region2, "Hello, World");
memcmp(region1, region2, 12);
```

memcmp scans through the two regions of memory, comparing **region1[0]** with **region2[0]**, and so on, until it finds two corresponding "slots" in the arrays whose contents differ. In the above example, this will occur when it compares **region1[7]** (which contains 'w') with **region2[7]** (which contains 'W'). It then compares the two letters to see which stands first in the character table used in this implementation, and returns the appropriate value.

See Also

strcmp, string, strncmp, strstr

Notes

memcmp compares regions of memory rather than strings; therefore, it does not stop when it encounters a null character.

memcpy — String function (libc)

Copy one region of memory into another

char *memcpy(*region1, region2, n***)**
char ****region1, *region2*;**
unsigned int *n*;

memcpy copies *n* characters from *region2* into *region1*. Unlike the routines **strcpy** and **strncpy**, **memcpy** copies from one region to another; therefore, it will not halt automatically when it encounters a null character.

memcpy returns *region1*.

See Also

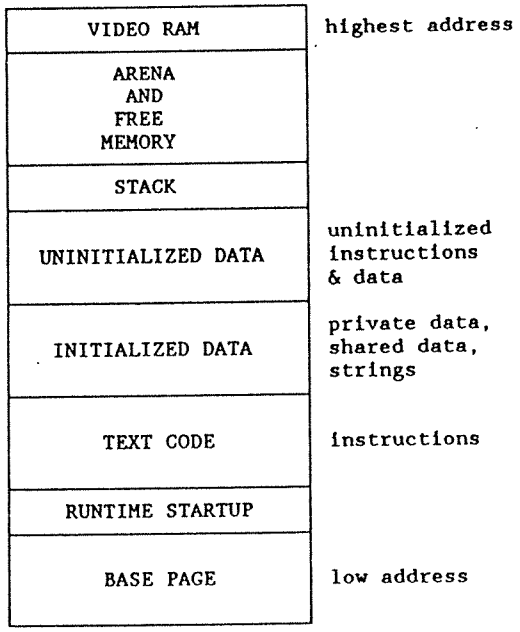
memmove, strcpy, string, strncpy

Notes

If *region1* and *region2* overlap, the behavior of **memcpy** is undefined. *region1* should point to enough reserved memory to hold *n* bytes of data; otherwise, code or data will be overwritten.

memory allocation — Technical information

The following diagram shows how Mark Williams C allocates memory.



The stack *descends* from the highest address in its space toward the static data area; new arguments are placed on the stack in its *lowest* address. Everything from the top of the stack space to the end of the data segment is free to accept dynamically allocated data.

The size of the stack cannot be altered while a program is running. The amount of stack is set by the global variable **_stksize**. By default, the runtime startup sets the stack size to two kilobytes (2,048 bytes) Note, however, that a highly recursive function may cause the stack to grow larger than two kilobytes so that it overwrites other data areas. This will cause your program to work incorrectly.

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

```
long _stksize = nL;
```

where *n* is an *even* constant that specifies the number of bytes to allocate.

Example

The example in the entry for **Physbase** displays system memory graphically.

The following example displays the "memory map" of a GEM-DOS process. It demonstrates **argc**, **argv**, **envp**, **environ**, **end**, **etext**, **edata**, and **_stksize**, as well as how to use the header file **basepage.h**.

```
#include <basepage.h>
dodisplay(value, name)
long value; char *name;
{
      printf("0x%08lx %s\n", value, name);
}

#define display(x) dodisplay((long)(x), #x)

main(argc, argv, envp)
int argc; char *argv[], *envp[];
{
      extern long _stksize;
      extern char **environ;
      extern char etext[], edata[], end[];

      display(BP->p_env);
      display(envp[0]);
      display(environ[0]);
      display(argv[0]);

      if (argv[1] != 0)
            display(argv[1]);
      if (argc > 2)
            display(argv[argc-1]);

      display(BP);
      display(BP->p_lowtpa);
      display(BP->p_cmdlin+1);
      display(_start);
      display(BP->p_tbase);

      display(etext);
      display(BP->p_tbase+BP->p_tlen);
      display(edata);
      display(BP->p_dbase+BP->p_dlen);
      display(end);

      display(BP->p_bbase+BP->p_blen);
      display(envp);
      display(environ);
      display(argv);
      display(argv+argc);

      display(_stksize);
      display(&argc);
      display(&argv);
      display(&envp);
      display(BP->p_hitpa);
}
```

See Also

C language, calling conventions, data format

memset — String function (libc)

Fill an area with a character

char *memset(*buffer, character, n***);**
char ****buffer*; int** *character***; unsigned int** *n***;**

memset fills the first *n* bytes of the area pointed to by *buffer* with copies of *character*. It casts *character* to an **unsigned char** before filling *buffer* with copies of it.

memset returns the pointer *buffer*.

See Also

memchr, memcmp, memcpy, memmove, string

menu — Technical information

A **menu** is a graphics form that is used extensively by GEM. It is a specialized form of an AES object, which is defined by the structure **OBJECT** described in the header file **obdefs.h**. For more information on this structure, see the entry for **object**.

Each menu's object tree must be built in a special way. The **root** object is a **G_IBOX** that is sized to dimensions of the screen. In high resolution, the screen is 640 rasters wide by 400 high; in medium resolution, it is 640 rasters wide by 200 high; and in low resolution, it is 320 rasters wide by 200 high. The **root** has two children: the **bar** object and the **screen** object.

The bar object

The *root* object's first child is the **bar** object. It describes the menu bar, and is of object type **G_BOX**. Its length is that of the screen, and its width is that of a normal character plus two rasters for gutter. In high resolution, a character is 16 rasters high; in medium and low resolutions, it is eight rasters high. Thus, in high resolution the **bar** object is 18 rasters high; in medium and low resolutions, it is ten rasters high.

The **bar** object has one child: an **active** object, whose type is **G_IBOX**. The **active** box is sized to hold all of the titles that appear in the bar along the top of the screen.

The **active** box, in turn, has one or more children: the **title** strings, which are the titles of the menus. These strings are of the type **G_TITLE**. This type of object is used only with menus. By design, the first (leftmost) title controls the drop-down menu that names the available GEM desk accessories.

The screen object

The **screen** object is the **root** object's other child. It is of type **G_IBOX**, and it is sized to cover the portion of the screen that is used by the drop-down menus. Thus, it should be as wide as the screen and as high as the longest drop-down menu.

The **screen** object has one or more children; each child is a **box** that displays a drop-down menu. There should be one **box** for each drop-down menu; i.e., the number of **boxes** must equal the number of **titles**. Each **box** is of type **G_BOX**.

Each box must be wide enough and high enough to hold all of the text that will be written into it. For example, if the longest string to go into it is ten characters wide, then the box must be at least 64 rasters wide (in high resolution) or the string will splash over its edge. Each **box** should be aligned on the left with its corresponding **title**. There is no need, however, to keep the various **boxes** from overlapping. GEM stores in a buffer the portion of the screen that is overwritten by a drop-down menu, so that it can be restored when the menu is erased. This buffer can hold up to one quarter of the screen, or 64,000 bits. No **box** should exceed this limit, or debris will be left on the screen when the menu is erased.

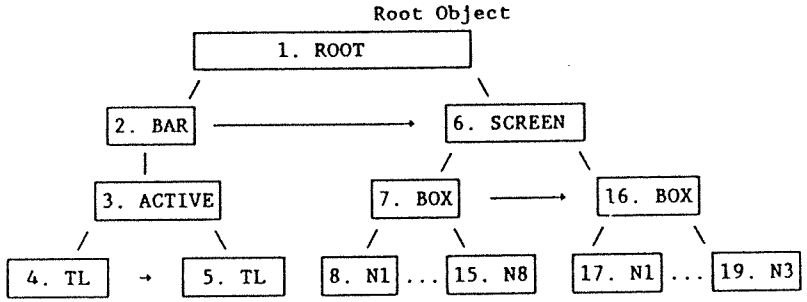
The name objects

Each **box** can have one or more children, called **names**. Each **name** is of type **G_STRING**, and names the particular option that you are offering the user. All the **names** must be as wide as the box; otherwise, the **box** will "leak", and cause more than one selection to be illuminated when the mouse pointer is moved into that **box**. The Y coordinate for each **name** must be increased by one line's height. For example, if a **box** has three **names**, the Y coordinate of the first should be zero, that of the second should be 16 (in high resolution, eight in medium or low resolution), and that of the third should be 32. This will keep the **names** from overlapping, which could possibly have disastrous results. As always, the X and Y coordinates of an object are relative to those of its parent.

The first (leftmost) **box** is special in that the AES can manipulate its **name** objects. By design, the first **box** must have eight **name** children. The first **name** can be defined by the user. The second **name** consists of a row of hyphens; its state is set to **DISABLED**, which causes it to be written in gray, rather than solid, letters. The next six **names** should point to empty strings. These will be filled in by the AES with the names of the available desk accessories. The AES will alter the size of the leftmost **box** if fewer than six have been loaded.

Genealogical table

The following "genealogical table" shows the object tree for a menu that has two drop-down menus, the latter with three entries. The numbers indicate each element's place in the object tree, and are used to set the parent-child-sibling pointers. These are set by the order in which the elements are loaded into the object's array:



You must invoke the menu with the function **menu_bar**; the AES will handle the rest. Note that, as shown in the above example, **menu_bar** regards as significant the *order* in which the elements of a menu are loaded into the object array. The order should be as follows:

root
bar
active
title(s)
screen
first menu **box**
first **items**
...
last menu **box**
last **items**

When the mouse is used to select a menu entry, the AES generates a message that contains that object's index number within the menu tree; use **evnt_mesag** to receive the message and initiate the proper response. The AES will automatically handle all invocation of desk elements; you do not need to write code for them.

Example

This example clears the screen and displays a menu that lists all of the GEM desk accessories.

Note that the objects in this example are sized by hand to fit with a screen that is 640 rasters across by 400 rasters high. If your screen does not match these dimensions, this example may not work. It will, however, show you how the elements of the menu object fit together.

```
#include <aesbind.h>
#include <gemdefs.h>
#include <obdefs.h>
```



```
OBJECT mask[] = { -1,-1,-1,G_BOX,LASTOB,NORMAL, 0x11C1L, 0, 0,
      640, 400 };

#define MDESK 7
#define MCOW 16
#define MHORSE      17
#define MMOUSE      18
#define MPIG 19
#define MPARROT     20
#define MQUIT 22

OBJECT menu[] = {
{ -1, 1, 5, G_IBOX, NONE, NORMAL, 0, 0, 0, 80, 25 },
{ 5, 2, 2, G_BOX, NONE, NORMAL, (BLACK<<12)|(BLACK<<8), 0, 0, 80, 1+(2<<8) },
{ 1, 3, 4, G_IBOX, NONE, NORMAL, 0, 2, 0, 12, 1+(3<<8) },
{ 4, -1, -1, G_TITLE, NONE, NORMAL, (long)" Desk ", 0, 0, 6, 1+(3<<8) },
{ 2, -1, -1, G_TITLE, NONE, NORMAL, (long)" Menu ", 6, 0, 6, 1+(3<<8) },

{ 0, 6, 15, G_IBOX, NONE, NORMAL, 0, 0, 1+(3<<8), 80, 19 },
{ 15, 7, 14, G_BOX, NONE, NORMAL, ((-1L&0xFF)<<16)|(BLACK<<12)|(BLACK<<8),
 2, 0, 22, 8 },
{ 8, -1, -1, G_STRING, NONE, NORMAL, (long)"  About menu", 0, 0, 22, 1 },
{ 9, -1, -1, G_STRING, NONE, DISABLED, (long)"----------", 0, 1, 22, 1 },

{ 10, -1, -1, G_STRING, NONE, NORMAL, (long)"  Desk 1", 0, 2, 22, 1 },
{ 11, -1, -1, G_STRING, NONE, NORMAL, (long)"  Desk 2", 0, 3, 22, 1 },
{ 12, -1, -1, G_STRING, NONE, NORMAL, (long)"  Desk 3", 0, 4, 22, 1 },
{ 13, -1, -1, G_STRING, NONE, NORMAL, (long)"  Desk 4", 0, 5, 22, 1 },
{ 14, -1, -1, G_STRING, NONE, NORMAL, (long)"  Desk 5", 0, 6, 22, 1 },

{ 6, -1, -1, G_STRING, NONE, NORMAL, (long)"  Desk 6", 0, 7, 22, 1 },
{ 5, 16, 22, G_BOX, NONE, NORMAL, ((-1L&0xFF)<<16)|(BLACK<<12)|(BLACK<<8),
 8, 0, 11, 7 },
{ 17, -1, -1, G_STRING, NONE, NORMAL, (long)"  Cow", 0, 0, 11, 1 },
{ 18, -1, -1, G_STRING, NONE, NORMAL, (long)"  Horse", 0, 1, 11, 1 },

{ 19, -1, -1, G_STRING, NONE, NORMAL, (long)"  Mouse", 0, 2, 11, 1 },
{ 20, -1, -1, G_STRING, NONE, NORMAL, (long)"  Pig", 0, 3, 11, 1 },
{ 21, -1, -1, G_STRING, NONE, NORMAL, (long)"  Parrot", 0, 4, 11, 1 },
{ 22, -1, -1, G_STRING, NONE, DISABLED, (long)"----------", 0, 5, 11, 1 },
{ 15, -1, -1, G_STRING, LASTOB, NORMAL, (long)"  Quit", 0, 6, 11, 1 },
};
#define NMENU (sizeof menu / sizeof menu[0])

alertf(n, p) int n; char *p;
{
      static char buffer[512];
      sprintf(buffer, "%r", &p);
      return form_alert(n, buffer);
}

main()
{
      int b[8], n, x, y, w, h;
```

```
/* open application; set pointer to arrow */
appl_init();
graf_mouse(ARROW, &n);

for (n = 0; n < NMENU; n += 1)
      rsrc_obfix(menu, n);

/* build window, draw object, open menu */
wind_get(0, WF_FULLXYWH, &x, &y, &w, &h);
objc_draw(mask, ROOT, MAX_DEPTH, x, y, w, h);
menu_bar(menu, 1);

/* wait for a message from the user */
for (;;) {
      evnt_mesag(b);
      /* b[0] holds the type of message */
      switch (b[0]) {

      /* if menu is clicked ... */
      case MN_SELECTED:
            /* ... b[4] holds entry clicked */
            switch(b[4]) {
            case MDESK:
                  alertf(1, "[0][Menu |menu  ][Ok]");
                  break;

            case MCOW:
                  alertf(1, "[0][MOO!  ][Ok]");
                  break;

            case MHORSE:
                  alertf(1, "[0][NEIGH!  ][Ok]");
                  break;

            case MMOUSE:
                  alertf(1, "[0][SQUEAK!  ][Ok]");
                  break;

            case MPIG:
                  alertf(1, "[0][OINK!  ][Ok]");
                  break;

            case MPARROT:
                  alertf(1, "[0][SQUAWK!  ][Ok]");
                  break;

            case MQUIT:
                  menu_bar(menu, 0);
                  appl_exit();
                  exit(0);

            default:
                  alertf(1, "[0][item %d?  ][Ok]", b[4]);
                  break;
            }

            menu_tnormal(menu, b[3], 1);
            break;
```



```
            default:
                  alertf(1, "[0][message %d?  ][Ok]", b[0]);
                  break;
            }
      }
}
```

See Also

AES, object, TOS, window

menu_bar — AES function (libaes)

Show or erase the menu bar

#include <aesbind.h>
#include <obdefs.h>
int menu_bar(*tree, eraseshow***) OBJECT *tree; int** *eraseshow***;**

menu_bar is an AES routine that shows or erases the menu bar; the menu bar is the bar that appears at the top of the screen and names the menus that are available to the user. *tree* is the name of the object tree being used. *eraseshow* indicates whether you want to show or erase the menu bar: zero indicates erase, and one indicates show. **menu_bar** returns zero if an error occurred, and a number greater than zero if one did not.

Example

For an example of how to use this routine, see the entry for **menu**.

See Also

AES, menu, object, TOS

menu_icheck — AES function (libaes)

Write or erase a check mark next to a menu item

#include <aesbind.h>
#include <obdefs.h>
int menu_icheck(*tree, item, eraseshow***) OBJECT *tree; int** *item, eraseshow***;**

menu_icheck is an AES routine that draws or erases a check mark next to a selected menu entry. *tree* points to the object tree that holds the menu, and *object* is the object within the tree that is being handled. *eraseshow* indicates whether you want to show the check mark or erase it: zero indicates erase, and one indicates show. **menu_icheck** returns zero if an error occurred, and a number greater than zero if one did not.

See Also

AES, menu, object, TOS

menu_ienable — AES function (libaes)

Enable or disable a menu item

#include <aesbind.h>

#include <obdefs.h>
int menu_ienable(*tree, object, disable*)
OBJECT *tree; int *object, disable*;

menu_ienable is an AES routine that enables or disables a menu item. A disabled item is displayed in faint letters and cannot be clicked by the user. *tree* points to the object tree that contains the menu, and *object* is the number of the object within the tree. *disable* indicates whether the item should be enabled or disabled: zero indicates disable, and one indicates enable. **menu_ienable** returns zero if an error occurred, and a number greater than zero if one did not.

See Also

AES, menu, object, TOS

menu_register — AES function (libaes)

Add a name to the desk accessory menu list
#include <aesbind.h>
#include <obdefs.h>
int menu_register(*accessory, textstring*) int *accessory*; char *textstring*;

menu_register is an AES routine that adds a name to the desk accessory menu list. *accessory* is the ID of the desk accessory. *textstring* points to the string of text inserted into the desk accessory menu. For more information about the desk accessory menu, see the entry for **menu**.

menu_register returns the desk accessory's identifier, from zero through five.

Example

For an example of this function, see the entry for **desk accessory**.

See Also

AES, desk accessory, menu, object, TOS

Notes

Because only six desk accessories can be used at any one time, only six items can be displayed on the desk accessory menu.

menu_text — AES function (libaes)

Replace text of a menu item
#include <aesbind.h>
#include <obdefs.h>
int menu_text(*tree, object, text*) OBJECT *tree; char *text; int *object*;

menu_text is an AES routine that changes the text for a menu item. *tree* points to the object tree for the menu, and *object* is the number of the object within the tree that holds that particular menu entry. *text* points to the text string to be plugged into the menu. **menu_text** returns zero if an error occurred, and a number greater than zero if one did not.

See Also

AES, menu, object, TOS

menu_tnormal — AES function (libaes)

Display menu title in normal or reverse video
#include <aesbind.h>
#include <obdefs.h>
int menu_tnormal(*tree, object, video*) OBJECT *tree; int *object, video*;

menu_tnormal is an AES routine that displays the menu title in normal or reverse video. *tree* points to the object tree that encodes the menu, and *object* is the number of menu title within the tree. *video* indicates whether you want the title to be in normal or reverse video: zero indicates reverse video, and one indicates normal. **menu_tnormal** returns zero if an error occurred, and a number greater than zero if one did not.

See Also

AES, menu, object, TOS

metafile — Technical information

A **metafile** is a file of VDI instructions that can be stored on disk and incorporated into other programs. This allows you to create "boiler-plate" images that are transferred easily.

Note that a metafile consists of a set of VDI instructions, rather than device-dependent bits. This allows you to edit such a file easily to alter how the program works. More importantly, because the elements of an image are described logically rather than absolutely, it allows each element to be manipulated easily, and the image as a whole to be maneuvered. This lets you create images independent of the type or resolution of the device on which they are displayed.

Consider, for example, the example of the bouncing colored ball used in the Atari demonstration program. At present, that program has a set of "snapshots" of the ball in different positions; to animate the ball, the program simply cycles through the snapshots. If this program were stored in a VDI metafile, however, a programmer could describe how each plane on the surface of the ball is logically connected to its neighbors; by setting parameters, then, the entire ball in all of its aspects could be resized easily or moved about the screen. This, in turn, would allow the programmer to create a user interface, in which the user could "zoom in" toward the ball, "zoom out", move the ball around the screen, change its rate or direction of rotation, etc.

Metafile structure

For a full description of the VDI metafile structure, see Appendix C to volume 1 of the *GEM Programmer's Guide*. The following briefly summarizes the metafile format.

Each metafile begins with a 16-integer header, structured as follows:

1 Always set to 0xFFFF.

2 VDI version number: 100 times the major version number, plus the minor version number.

3 Type of coordinates: zero indicates normalized device coordinates (NDC); two indicates raster coordinates (RC). One is reserved by TOS.

4-7 Respectively, minimum width and height, and maximum width and height required to display image in the file. These are set with the function **v_extent_meta**; otherwise, they are set to zero.

8-16 Reserved; always set to zeroes.

The header is followed by a series of VDI entries; each consists of an array of **int**s, in the following order:

0 The VDI function's opcode. See the list below for the appropriate opcode for each legal VDI routine.

1 The number of vertices (i.e., endpoints or corners) in the figure being drawn.

2 The number of integer parameters passed to the VDI routine.

3 The VDI routine's sub-opcode; see the table below for each routine's appropriate sub-opcode.

4-*n* The settings for each vertex. The number of vertices described corresponds to the value in **1**.

n*+4-*m The values for each integer parameter. The number of parameters described corresponds to the value in **2**.

Finally, each metafile closes with an integer set to 0xFFFF.

Customized routines can be inserted into a metafile with the function **v_meta_write**.

Metafile routines

The following VDI library routines can be incorporated into metafiles. The first column gives the routine's opcode, the second gives its sub-opcode, the third gives its name, and the fourth gives a brief description.

3	0	**v_clrwk**	clear a virtual device
4	0	**v_updwk**	update workstation (flush buffers)
5	2	**v_exit_cur**	exit from alphabetic mode
5	3	**v_enter_cur**	enter alphabetic mode
5	20	**v_form_adv**	advance page on hard-copy device

5	21	**v_output_window**	print portion of a virtual device
5	22	**v_clear_disp_list**	clear a printer's display list
5	23	**v_bit_image**	print a bit-image file
6	0	**v_pline**	draw a polyline
7	0	**v_pmarker**	draw a polymarker
8	0	**v_gtext**	output graphics text
9	0	**v_fillarea**	flood enclosed area with fill pattern
11	1	**v_bar**	draw an outlined, filled rectangle
11	2	**v_arc**	draw a circular arc
11	3	**v_pieslice**	draw a circular pie segment
11	4	**v_circle**	draw a circle
11	5	**v_ellipse**	draw an ellipse
11	6	**v_ellarc**	draw an elliptical arc
11	7	**v_ellpie**	draw an elliptical pie segment
11	8	**v_rbox**	draw rounded rectangle
11	9	**v_rfbox**	draw rounded rectangular fill area
12	0	**vst_height**	set graphics text height, in pixels
13	0	**vst_rotation**	set angle of graphics text
14	0	**vs_color**	set mix for a color
15	0	**vsl_type**	set polyline's pattern
16	0	**vsl_width**	set polyline width
17	0	**vsl_color**	set polyline color
18	0	**vsm_type**	set polymarker type
19	0	**vsm_height**	set polymarker height
20	0	**vsm_color**	set polymarker color
21	0	**vst_font**	set graphics text font
22	0	**vst_color**	set graphics text color
23	0	**vsf_interior**	set fill type
24	0	**vsf_style**	set fill style
25	0	**vsf_color**	set fill color
32	0	**vswr_mode**	set writing mode
39	0	**vst_alignment**	set graphics text alignment
104	0	**vsf_perimeter**	set drawing of perimeter
106	0	**vst_effects**	set graphics text special effects
107	0	**vst_point**	set graphics text height, in points
108	0	**vsl_ends**	set polyline end types
112	0	**vsf_udpat**	set user-defined fill pattern
113	0	**vsl_udsty**	set user-defined polyline style
114	0	**vr_recfl**	draw a rectangular fill area
129	0	**vs_clip**	clip an area of the virtual device

See Also

TOS, v_meta_extents, v_write_meta, VDI, vm_filename

Notes

Metafiles need the VDI's GDOS in their operation. They should not be used if the GDOS is not present in your edition of VDI.

mf — Command

Measure space left in RAM

mf

mf is a command that measures the amount of free space left in RAM for program execution. It takes no arguments.

See Also

commands, df, msh

Mfpint — xbios function 13 (osbind.h)

Initialize the MFP interrupt

#include <osbind.h>
#include <xbios.h>
void Mfpint(*interrupt, vector***) int** *interrupt*; **char ****vector*;

Mfpint initializes the multi-function peripheral (MFP) interrupt, and returns nothing. This routine allows a programmer to trap a hardware interrupt in her program. *interrupt* is the number of the interrupt to be set, 0 through 15, as follows, going from lowest to highest priority:

MFP_BIT0	0	I/O port bit 0
MFP_BIT1	1	undefined
MFP_BIT2	2	undefined
MFP_BIT3	3	undefined
MFP_TIMD	4	timer D, RS-232 baud rate generator
MFP_TIMC	5	timer C, system 200-hz clock
MFP_BIT4	6	I/O port bit 4
MFP_BIT5	7	undefined
MFP_TIMB	8	timer B
MFP_XERR	9	RS-232 transmit error
MFP_EMPT	10	RS-232 transmit buffer empty
MFP_RERR	11	RS-232 receive error
MFP_FULL	12	RS-232 receive buffer full
MFP_TIMA	13	timer A, user programmable
MFP_BIT6	14	I/O port bit 6
MFP_BIT7	15	I/O port bit 7

vector points to the interrupt routine to be set.

See Also

Jdisint, Jenabit, TOS, xbios

Mfree — gemdos function 73 (osbind.h)

Free allocated memory

#include <osbind.h>
long Mfree(*memory***) long** *memory*;

Mfree frees memory allocated by the function **Malloc**. *memory* points to the address of the memory to free. **Mfree** returns 0 if memory could be freed, and non-zero if it could not.

Example

The following example prints the number of bytes currently free and the number allocated.

```
#include <osbind.h>

main() {
        unsigned long memleft;
        unsigned long memhere;
        char *almem;

        /*
         * This first 'printf' is needed to make the numbers
         * look right, because printf malloc's memory for the
         * FILE buffer
         */

        printf("Test of Malloc(), Mfree() and Mshrink()\n");

        printf("%8lx bytes free, %8lx bytes allocated\n",
                (memleft = Malloc(-1L)), 0L);

        memhere = memleft>>1;
        almem = (char *) Malloc(memhere);
        printf("%8lx bytes free, %8lx bytes allocated (%8lx)\n",
                Malloc(-1L), memleft-Malloc(-1L), memhere);

        Mshrink(almem,0x1000L);
        printf("%8lx bytes free, %8lx bytes allocated (%8lx)\n",
                Malloc(-1L), memleft-Malloc(-1L), 0x1000L);
        Mfree(almem);
        printf("%8lx bytes free, %8lx bytes allocated (%8lx)\n",
                Malloc(-1L), memleft-Malloc(-1L), 0L);
}
```

See Also

gemdos, Malloc, Mshrink, TOS

Notes

Do not attempt to **Mfree** blocks of memory not directly allocated by **Malloc**. Memory freed by **Mfree** is not inserted into the arena used by **malloc**, but is returned to the system.

Midiws — xbios function 12 (osbind.h)

Write a string to the MIDI port

#include <osbind.h>
#include <xbios.h>
void Midiws(*count, pointer***) int** *count*; **char** **pointer*;

Midiws writes a string to the musical instrument device interface (MIDI) port, and returns nothing. *count* gives the number of characters that will be sent, minus one; and *buffer* points to where the characters are stored. Note that this routine will transmit *count* characters; NUL characters will be used like any other character.

Example

This example plays some notes on a MIDI instrument connected to the ST through the MIDI-OUT plug.

```
#include <osbind.h>

/* MIDI status byte values */

#define NOTE_OFF (0x80)                 /* Key off command */
#define NOTE_ON (0x90)                  /* Key on command */

/* Some useful things to know... */

#define MIDDLE_C (60)

#define C_OFFSET (0)
#define D_OFFSET (2)
#define E_OFFSET (4)
#define F_OFFSET (5)
#define G_OFFSET (7)
#define A_OFFSET (9)
#define B_OFFSET (11)
#define FLAT (-1)
#define SHARP (1)
#define OCTAVE_STEP (12)

unsigned char notes[128];               /* Note counters... */

key_down(note_offset)
int note_offset; {                      /* Note relative... */
                                        /* ...to middle C */
      int midi_note;
      char midi_buf[4];

      if ((midi_note=MIDDLE_C+note_offset) < 0 || midi_note >127)
            return;                     /* Return if out of range */
      notes[midi_note]++;               /* Mark as key-down... */
      midi_buf[0]=NOTE_ON;              /* Note on... */
      midi_buf[1]=midi_note;            /* This one... */
      midi_buf[2]=0x40;                 /* this fast... */
      Midiws(2, midi_buf);              /* Send message out */
}
```



```
key_up(note_offset)                     /* Note */
int note_offset; {
      int midi_note;
      char midi_buf[4];

      if ((midi_note=MIDDLE_C+note_offset) < 0 || midi_note > 127)
            return;                     /* Return if out of range */
      if (notes[midi_note]-- < 0)
            notes[midi_note] = 0;       /* Decrement down count */
      midi_buf[0]=NOTE_OFF;             /* Note off... */
      midi_buf[1]=midi_note;            /* This one... */
      midi_buf[2]=0x40;                 /* this fast... */
      Midiws(2, midi_buf);              /* send message out */
}

clean_up() {                            /* buffer for commands */
      char midi_buf[258];               /* And a pointer. */
      char *mp;                         /* A counter. */
      int  i=0;                         /* Another counter */
      int  c=0;

      mp = midi_buf;
      *mp++ = NOTE_OFF;
      while (i < 128) {
            while(notes[i] != 0) {
                  notes[i]--;
                  *mp++ = i;
                  *mp++ = 0x40;
                  c++;
            }
            i++;
      }
      if(c > 0)
            Midiws(c<<1, midi_buf);
}

/* Delay for a little while -- Use the vertical sync for timing.*/

delay(n)
int n; {
      int i;
      while(n-- > 0) {
            for(i=35 ; i>0 ; i--)
                  Vsync();
      }
}
```

```
main() {
        int i;
        int n;

        key_down(C_OFFSET);
        delay(2);
        key_down(E_OFFSET);
        delay(2);
        key_down(G_OFFSET);
        delay(2);
        key_down(C_OFFSET+OCTAVE_STEP);
        delay(5);
        key_up(E_OFFSET);
        key_up(G_OFFSET);
        key_down(F_OFFSET);
        key_down(A_OFFSET);
        delay(20);
        clean_up();
}
```

See Also

TOS, xbios

mkdir – Command

Create a directory
mkdir *directory*

mkdir creates *directory*. Files or directories with the same name as *directory* must not already exist. **directory** will be empty except for the entries '.', the directory's link to itself, and '..', its link to its parent directory.

See Also

commands, msh, rm, rmdir

mktemp – General function (libc)

Generate a temporary file name
char *mktemp(*pattern***) char ****pattern***;**

mktemp generates a unique file name. It can be used, for example, to name intermediate data files.

Note that the functions **tmpnam** and **tempnam** each assemble a temporary file name and then call **mktemp**. These routines ease the difficulty in creating a proper name for a temporary file.

See Also

msh, tempnam, tmpnam

modf – General function (libc)

Separate integral part and fraction
double modf(*real, ip***) double** *real, *ip***;**

modf is the floating-point modulus function. It returns the fractional part of its argument *real*, which is a value *f* in the range 0 <= *f* < 1. It also stores the integral part in the **double** location referenced by *ip*. These numbers satisfy the equation *real* = *f* + **ip*.

Example

This example prompts for a number from the keyboard, then uses **modf** to calculate the number's fractional portion.

```
#include <stdio.h>

main()
{
        extern char *gets();
        extern double modf(), atof();
        double real, fp, *ip;
        char string[64];

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

                real = atof(string);
                fp = modf(real,ip);
                printf("%lf is the integral part of %lf\n",
                        *ip, real);
                printf("%lf is the fractional part of %lf\n",
                        fp, real);

        }
}
```

See Also

atof, ceil, fabs, floor, frexp, ldexp

modulus – Definition

Modulus is the operation that returns the remainder derived from a division operation. For example, 12 modulus four equals zero, because when 12 is divided by four it leaves no remainder. The term "modulo" also refers to the product of a modulus operation; in the above example, the modulo is zero. In C, the modulus operation is indicated with a percent sign '%'; therefore, 12 modulus 4 is written 12%4.

The modulus operation often is used to trim numbers to a preset range. For example, if you wanted to create a list of single-digit random numbers, you would use the command:

```
rand()%10
```

This is demonstrated by the following example.

Example

This example prints a list of 20 single-digit random numbers. The random-number table is seeded with a portion of the current system time.

```
main()
{
        long nowhere; /* place to put unused pointer */
        int counter;

        srand((int)time(&nowhere));
        for (counter = 0; counter <20; counter++)
                printf("%d\n", rand()%10);
}
```

See Also

operator

mousehidden — Command

Return how often mouse pointer has been hidden
mousehidden

mousehidden is a command that returns the number of times the mouse pointer has been hidden. Under GEM, if the mouse pointer has been hidden more than once, it must be "restored" as many times as it has been hidden before it reappears on the screen. **mousehidden** will tell you how many times you must restore the mouse pointer before it reappears.

See Also

commands, hidemouse, Line A, showmouse, TOS

msh — Command

msh is the Mark Williams micro-shell, which is designed for use under TOS. It combines aspects of the Bourne shell and the Berkeley C shell into one command that is powerful and easy to use.

msh is a *command processor*. It finds commands and executes them either singly or in batches; and it allows the user to direct the output of a command to a device, into a new file, or to another command for further processing. It can replace text with symbols defined by the user, or with wildcards that are expanded according to carefully defined rules.

The simplest command consists of a list of words; the words are separated from each other by spaces or tab characters, and the list is terminated by a **<newline>** sequence. Each word may contain *history substitutions*, *variable substitutions*, *file name substitutions*, *quoted characters*, *quoted strings*, or *file redirection*. **msh** also supports aliasing, for use in batch files and scripts. These are discussed below.

Several commands may be placed on the same line; the commands are then separated with semicolons or other *command separators*; these are outlined below. A list of commands may be grouped into a single command by enclosing the list within parentheses.

Both simple commands and lists of commands be made to extend over more than one line by typing a slash '/' before pressing the **<return>** key.

History command

msh also comes equipped with a history command. This command allows you to re-execute a command without having to retype it.

msh automatically records your previously typed commands into a buffer. The number of commands it "remembers" is set by the **history** variable: for example, typing

```
set history=8
```

tells **msh** to remember the last eight commands you have issued.

The history command is invoked with the exclamation point '!'. The command **!!** re-executes your last command. Therefore, the commands

```
ls -w
!!
```

will give you two columnar listings of the contents of your current directory. The command *!name* re-executes the last command with *name* that is in your history directory. For example, when you type the following list of commands:

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

the history command **!ls** reaches back and executes the command **ls -w**. To execute a previous command by its number, subtract its value from the current command. For example, **!-1** re-executes the previous command; it is a synonym for **!!**. To execute a command issued three commands ago, type **!-3**; this will execute **echo foo >stuff**.

Variable substitution

A *variable* is a symbol defined by the user with the **set** command; for example, the command

```
set X="echo foo"
```

declares that **X** is a symbol equivalent to the string **echo foo**. When a variable is used in a **msh** command line, it must be preceded by a dollar sign '**$**' or an exclamation point '**!**'. For example, to call the variable set in the above example, type **$X** or **!X**. When it sees a token that begins with either of these punctuation marks, **msh** searches for it first on the list of variables that have been assigned with the **set** command, then on the list of those assigned with the **setenv** command, and finally on the list of tokens that it received from TOS or from the parent shell. For example, if you type

```
set esc="^[
set cls="echo $(esc)E"
```

(where <esc> indicates the escape character) and then type

```
$cls
```

msh will expand this variable into

```
echo ^[E
```

and then execute the **echo** command with the argument <esc>E, which in turn clears the screen.

The difference between *$name* and *!name* is that the latter may include command separators because it is rescanned as input, whereas the former is not rescanned. For example, the variable set with the command

```
set X="echo foo ; echo bar"
```

should be reference with the token !X rather than $X. Command separators are described in detail below.

Directories .bin and .cmd

msh has two built-in directories: **.bin** and **.cmd**. **.bin** holds **msh's** built-in commands. It is searched automatically by **msh**, and so it does not need to be listed in the **PATH** environment parameter. **.cmd** holds user-defined commands. You can create a new command and load it into **.cmd** with the **set** command. For example, the following command to **msh** creates a new list command:

```
set !n .cmd lc="ls -w"
```

This tells **msh** to equate the command **lc** with the string **ls -w**, which prints the contents of a directory in columnar format. **.cmd** must be included in the **PATH** environment, or it will not be searched by **msh** when you issue a command.

File name substitutions

File name substitutions contain the punctuation marks [ ] ? * { }. The following notes what each punctuation mark does:

[list], [a-z]
: Match any of the characters *l*, *i*, *s*, or *t*, or any character in the range **a-z**.

?
: Match any one character or no character.

*
: Match any character, any string of characters, or no character.

{list}
: Braces enclose a list of words that are each combined with the remainder of the word.

Command substitutions

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

```
pr `ls -l`
```

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

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

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

resets the GEM time after a warm boot.

Note that the grave mark may be passed to **msh** as a literal character if it is preceded with a slash '/'.

Character quotations

A *quotation* is used when you want **msh** to disregard the special meaning of a character and read it merely as a literal character. In general, preceding a character with a slash will remove the special meaning of a character, except under the following circumstances:

1. A slash followed by an end-of-file indicator is always an error.
2. A slash followed by a **<newline>** becomes a space and continues input on the next line.
3. When set between " "'s or ' ''s, a slash followed by a **<newline>** translates into **<newline>**, and /" becomes a literal quotation mark. All other characters quoted with '/' are left untouched.
4. Within literal quotations, '/' is literal.

Quoted strings

Strings may be quoted by enclosing them in apostrophes or quotation marks. Quoting a string means that **msh** or a command is to accept it literally. Note that quoting a string with apostrophes prevents any further expansion; all wildcards and variables will be treated as literal characters. Quoting a string with quotation marks, however, tells **msh** to treat white space as part of the string, but allows further expansion of variables. The following exercise will demonstrate how these forms of quotation differ:

```
set A="123"
set B="XYZ"
echo $A             $B
echo "$A            $B"
echo '$A            $B'
```

File redirection

The term *file redirection* means redirecting the input or output of a command into a file. The following redirection operators are recognized by **msh**:

> *file*	Redirect output of a file into *file*. If *file* already exists, replace its contents with the output of the command.
>> *file*	Append the output of a command onto *file*. If *file* does not exist, create it and fill it with the output of the command.
2> *file*	Redirect material normally sent to the standard error device into *file*.
2>> *file*	Append material normally sent to the standard error device onto the end of *file*.
3> *file*	Redirect material normally sent to the printer into *file*.
3>> *file*	Append material normally sent to the printer onto the end of *file*.
>& *file*	Redirect the output of a command and any diagnostic messages it produces into *file*.
>>& *file*	Append the output of a command and all of the diagnostic messages it generates onto *file*. If *file* does not exist, create it and fill it with the output and diagnostic messages generated by the command.
< *file*	Use the contents of *file* to control the execution of a command.

Separating and joining commands

Commands can be separated or joined on the same command line by using the following marks:

;	Execute commands sequentially.
&&	Execute commands sequentially until one terminates with non-zero exit status (i.e., until an error occurs in one).
|	Form a *pipe* between the commands: feed the standard output of the command on the left of the '|' into the standard input of the command on the right.
|&	Form a pipe that passes both the output of the command on the left and any diagnostic messages it produces as input to the command on the right.
||	Commands separated by '||' are run sequentially until one terminates with zero exit status (i.e., executed without error).

Commands

Mark Williams C includes a number of commands that are designed to be used with **msh**. For a list of these commands and a brief description of each, see the entry for **commands**. If you need help with **msh** or any of its built-in commands, type **help** and the name of the command for which you need help. **msh** will print on the screen a summary of how to use that command.

Setting the environment

msh allows you to set a number of *environmental variables*. **msh** uses some of these variables, and makes all of them available to programs that run under it. A program can read these variables by using the function **getenv**. Environmental variables can be set or changed with the command **setenv**, and erased with the command **unsetenv**. Typing **setenv** without an argument will display the list of environmental variables plus their settings.

For Mark Williams C to work properly, the following *environmental parameters* must be set:

HOME	The default directory: where **msh** performs a task when no other directory is named.
INCDIR	Name the directory in which **cc** searches for header files and other text files to be included in compilation.
LIBPATH	Name the path along which **cc** searches for the executable files for the compiler and the linker i.e., **cc0.prg**, **cc1.prg**, **cc2.prg**, **cc3.prg**, **crts0.o**, **ld.prg**, and the libraries.
PATH	This environmental variable consists of a list of directory prefixes that are separated by commas. These prefixes name the directories that are searched in order for commands or batch files to be run. For example, typing `PATH = ,\bin,\lib` will ensure that **msh** will search the the current directory, then the directories **\bin** and **\lib**, in that order, to find the executable file named in a command.
SUFF	This consists of a list of file-name suffixes that are separated by commas. These suffixes are appended to the given command name when searching the directories named in **$(PATH)**.
TMPDIR	Name the directory into which temporary files are written.

See the Lexicon entry **environment** for more information.

Shell variables

The following variables control the operation of **msh**. Some can be set with the **set** command. Typing **set** without an argument will display a list of all current variables, both those set by the user and those set by **msh**:

history Set the length of the history list. For example, to set the **history** variable to eight, type the following:

```
set history=8
```

This allows you to invoke any of the last eight commands by using the form !-*n*.

cwd The current working directory.

cwdisk The current working device.

prompt This variable holds the prompt string. The default is '$'.

status This variable holds the exit status returned by the last command executed. It should not be reset by the user.

Command files

msh reserves the variables **$0** through **$9** for arguments passed on a command line. This allows you to write shell scripts whose variables can be set when you run the script.

For example, the following commands could be typed into the file **foo**:

```
cc -V -f $1 $2 $3 -lm
```

Thereafter, typing **foo** followed by the names of up to three C source files will compile the files with the floating point **printf** routines, and link in the mathematics library.

msh has three aliases built into it, which extends the range of your command files. The alias **$*** represents all of the arguments to the current command. For example, the command

```
cc -V $*
```

when placed into a file, compiles all of the files listed as arguments to that file.

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

```
echo $#
```

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

Finally, the alias **$<** represents any line received from the standard input device, up to the newline character. For example, the following command gives a very slow version of the concatenation command, **cat**:


```
set in .cmd slowcat=(
while (set foo="$<" && not (equal "$foo" "")) (echo $foo)
)
```

Loops and conditional statements

msh supports conditional statements and loops. The basic conditional command is **if**. Its syntax is as follows:

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

If **word1** executes successfully, then **word2** is executed; otherwise, the **word3**, if present, is executed. Commands can be grouped into statements by enclosing them within parentheses. The text within the parentheses can extend across as many lines of text without needing to precede newlines with backslashes. For example, the command

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

echoes the strings **foo**, **bar**, and **baz**, and then prints the contents of the current directory.

msh also contains a **while** command, which can control a conditional loop. Its syntax is as follows:

```
while word1 word2
```

As long as **word1** executes successfully, **word2** will also be executed. Each word may be a list of commands enclosed within parentheses.

msh contains two test commands, **equal** and **not**. **equal** compares two strings; it succeeds if the strings are identical, and fails if they are not. Its syntax is as follows:

```
equal argument1 argument2
```

Note that either argument can be a literal string, an integer, or an embedded command. **not** inverts the logical result of its argument.

The command **is_set** is also useful in building loops and conditional statements. This command takes the name of an environmental variable as its argument. It returns zero if the variable is set, and a number greater than zero if it is not. Its syntax is as follows:

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

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

The **profile** *file*

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

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

See Also

commands, environment, set, setenv, wildcard, unset, unsetenv

Mshrink — gemdos function 74 (osbind.h)

Shrink amount of allocated memory
#include <osbind.h>
long Mshrink(*begin, length***) long** *begin, length*;

Mshrink shrinks the amount of memory allocated by a program, and returns dynamic memory to the free memory pool. *begin* points to the beginning of the space to be kept, and *length* indicates the amount of memory to be kept. **Mshrink** returns zero if memory could be de-allocated, and non-zero if it could not.

Example

For an example of how to use this function, see the entry for **Mfree**.

See Also

gemdos, Malloc, Mfree, TOS

Notes

The **gemdos** call has a third parameter that is always zero; the **Mshrink** macro inserts this parameter automatically.

mshversion — Command

Print current version of **msh**
mshversion

The command **mshversion** prints the version of the microshell **msh** that you are using. Typing **mshversion** returns a string of the form:

```
Mark Williams Micro Shell, version 3.0
```

See Also

commands, msh

msleep — Command

Stop executing for a specified time
msleep *milliseconds*

msleep suspends processing for a set time. *milliseconds* is the amount of time to suspend processing, in milliseconds.

See Also

commands, sleep, TOS

mtoh — Command

Redraw the screen from medium to high resolution
mtoh

mtoh redraws the screen, moving from medium to high resolution.

See Also

commands, htom, ltom, mtol, TOS

mtol — Command

Redraw the screen from medium to low resolution
mtol

mtol is a command that redraws the screen, moving from medium to low resolution.

See Also

commands, htom, ltom, mtoh, TOS

mtype.h — Header file

List processor code numbers

The header file **mtype.h** assigns a code number to each of the processors supported by Mark Williams C compilers. These include the Intel i8086, i8088, i80186, and i80286; the Zilog Z8001 and Z8002; the DEC PDP-11 and VAX; the IBM 370, and the Motorola 68000.

See Also

header file, portability

mv — Command

Rename files or directories
mv *oldfile newfile*
mv *file ... directory*

mv renames files. In the first form above, it changes the name of *oldfile* to *newfile*. If *newfile* previously existed, **mv** deletes its former contents; if not, **mv** creates it. If *newfile* is a directory, **mv** places *oldfile* under that directory.

In the second form, **mv** moves each file argument into the directory argument. If the source and destination files are on different disk drives, **mv** copies the source to the destination and removes the source.

mv will not copy directories between devices and will not remove directories that occupy the destination of the command.

See Also
commands, cp, msh

mwtomw — Command

Convert old Mark Williams format to Mark Williams 3.0 format
mwtomw *filename*

The command **mwtomw** takes an executable file that had been compiled with Mark Williams C version 2.1.7 or earlier, and converts it to the format used with version 3.0. *filename* is the name of the executable file to convert.

See Also
commands, drtomw, ld

N

nested comments — Definition

Both *The C Programming Language* and the draft ANSI standard declare that comments cannot be nested. Earlier versions of Mark Williams C included a switch, called -VCNEST, that allowed a programmer to nest comments. This switch has been removed. Current and future versions of Mark Williams C abort compilation when they detect nested comments.

See Also
C language

newline — Character constant

Mark Williams C recognizes the literal character '\n' for the ASCII newline character LF (octal 012). This normally feeds the line and returns the carriage. This character may be used as a character constant or in a string constant.

See Also
ASCII, character constants

Notes
On the Atari ST, '\n' must be used with the carriage return character '\r' if the program does not go through STDIO.

nm — Command

Print a program's symbol table
nm [**-adgnopru**] *file* ...

nm prints the symbol table of each *file*. Each *file* argument must be a Mark Williams C object module or an object library built with the archiver **ar**. If an argument is a library, **nm** prints the symbol table for each member of the library.

The first argument selects one of several options. It is optional; if present, it must begin with '-'. The options are as follows:

-a Print all symbols. Normally, **nm** prints names that are in C-style format and ignores symbols with names inaccessible from C programs.

-d Print only defined symbol.

-g Print only global symbols.

-n Sort numerically rather than alphabetically. **nm** uses unsigned compares when sorting symbols with this option.

-o Append the file name to the beginning of each output line.

-p Print symbols in the order in which they appear within the symbol table.

-r Sort in reverse-alphabetical order.

-u Print only undefined symbols.

By default, **nm** sorts symbol names alphabetically. Each symbol is followed by its value and its segment.

See Also

cc, commands, ld, size, strip

Notes

Because version 3.0 changes the format of executable files, the edition of **nm** shipped with version 3.0 does not work with executables linked with Mark Williams C version 2.1.7 or earlier. To convert such files to a format that **nm** recognizes, use the command **mwtomw**.

nm now works with symbol tables larger than 64 kilobytes. It uses a balanced-tree to sort symbols, and will absorb as much memory as the system has available. When memory runs out, **nm** prints the symbols already sorted and continues with the remainder of the symbol table.

not — Command

Invert logical value of an argument
not *argument*

not is a test command that is built into the microshell **msh**. It inverts the logical value of *argument*; that is, it changes zero to one and any value other than zero to zero. *argument* may be either an absolute value or a value returned by another command.

Example

The following command prints the string **Not high res** if the monitor is not in high resolution, and it prints **High res** if the monitor is in high resolution.

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

Note that the command **getrez** returns two if the monitor is in high resolution.

See Also

commands, equal, if, is_set, msh, while

notmem — General function (libc)

Check if memory is allocated
int notmem(*ptr***) char *****ptr;**

notmem checks if a memory block has been allocated by **calloc, malloc, lcalloc, lmalloc,** or **realloc**. *ptr* points to the block to be checked. **notmem** searches the arena for *ptr*; it returns one if *ptr* is not a memory block obtained from **malloc, calloc,** or **realloc**, and zero if it is.

See Also

arena, calloc, free, malloc, realloc, setbuf

n.out — Definition

n.out is the format used by the Mark Williams C compiler, assembler and linker to generate their output.

n.out first gives global information and information about the size of each segment. Segments of the indicated size follow the header in a fixed order. **n.out** defines the header structure for the 68000 as follows:

```
struct ldheader {
        short  l_magic;
        short  l_flag;
        short  l_machine;
        short  l_tbase;
        size_t l_ssize[NLSEG];
        long   l_entry;
};
```

All elements of the **nout** header are stored in canonical byte order. **l_magic** is the "magic number" that identifies a load module; it always contains **0407**. **l_flag** contains flags that indicate the type of the object module. **l_machine** is the processor identifier. **l_tbase** is the start of the text segment. **l_entry** contains the machine address where execution of the module commences. **l_ssize** gives the size of each segment.

size prints the segment sizes of the **n.out** format header, **nm** lists the symbols, and **strip** will remove the symbols.

See Also

as, ld, nm, size, strip

nout.h — Header file

Describe output format **n.out**
#include <nout.h>

The header file **nout.h** contains the description of the output format **n.out**, which is created by Mark Williams C. It is used by the compiler, the assembler, and by the linker **ld** to create correctly formatted output files.

See Also
header file, n.out

NUL — Character constant

NUL is the character ASCII 0 and, in C, signals the end of a string. It is represented as '\0'. Note that NUL is defined as part of the string it is terminating; therefore, a string that is defined to be 50 characters long can, in fact, hold 49 printable characters plus NUL.

See Also
ASCII, character constant, NULL, string

NULL — Manifest constant

NULL is defined in the header file **stdio.h**. It is the null pointer **(char *)0**, which is a pointer filled with zeros. Numerous routines return this value to indicate failure; it is useful as a return value because it points nowhere, and so removes the possibility of accidentally destroying a section of memory after failure.

See Also
manifest constant, NUL, pointer, stdio.h

Notes
References through **NULL** on the Atari ST cause a *bus error*, i.e., two cherry bombs appear on the screen.

nybble — Definition

A **nybble** is four bits, or half of an eight-bit byte. The term is generally used to refer to the low four bits or the high four bits of a byte; thus, a byte may be said to have a "low nybble" and a "high nybble". One nybble encodes one hexadecimal digit.

See Also
bit, byte

O

obdefs.h — Header file

Declare TOS objects and structures
#include <obdefs.h>

obdefs.h is a header file that contains TOS common object definitions and structures. It defines numerous elements used in programs written for the Atari ST, such as definitions of color settings, editable fields, and fonts.

See Also
header file, object, TOS

objc_add — AES function (libaes)

Redefine a child object within an object tree
#include <aesbind.h>
#include <obdefs.h>
int objc_add(*tree, parent, child***) OBJECT ***tree*; **int** *parent, child*;

objc_add is an AES routine that redefines a child object within an object tree; specifically, it redefines an object as being the offspring of a specified parent. *tree* points to the object tree being modified. *child* is the number of the object being redefined, and *parent* is the number of the object being made *child*'s parent.

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

See Also
AES, object, TOS

objc_change — AES function (libaes)

Change object's state
#include <aesbind.h>
#include <obdefs.h>
int objc_change(*tree, object, junk, x, y, w, h, newstate, redraw***)**
OBJECT ***tree*; **int *object, junk, x, y, w, h, newstate, redraw*;

objc_change is an AES routine that changes the state of an object within a named clipping rectangle. This is done by altering the member **ob_state** within the **OBJECT** structure. For more information on object states, see the entry for **object**.

objc_change is a simple extension to **objc_draw**, which allows you to reset **ob_state** and optionally skip redrawing the object.

tree points to the object tree being modified, and *object* is the number of the object within the object tree. *junk* is reserved, and must be zero.

x, *y*, *w*, and *h* set, respectively, the X coordinate of the clipping rectangle, its Y coordinate, its width, and its height. **objc_change** will alter only the portion of the object that falls within this clipping rectangle.

newstate indicates the new state for the object, as follows:

0x00	**NORMAL**	Object is normal
0x01	**SELECTED**	Shown in reverse video
0x02	**CROSSED**	Object has 'X' drawn next to it
0x04	**CHECKED**	Check mark drawn next to object
0x08	**DISABLED**	Object redrawn in gray; unselectable
0x10	**OUTLINED**	Object is outlined
0x20	**SHADOWED**	Object has shadow drawn beneath it

Finally, *redraw* indicates whether or not to redraw the object being modified: zero indicates not to redraw, and one indicates redraw.

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

See Also

AES, object, TOS

objc_delete — AES function (libaes)

Delete an object from an object tree
#include <aesbind.h>
#include <obdefs.h>
int objc_delete(*tree, object***) OBJECT ****tree***; int** ***object***;

objc_delete is an AES routine that deletes an object from an object tree. *tree* points to the object tree being modified, and *object* is the number of the object within the object tree. **objc_delete** returns zero if an error occurred, and a number greater than zero if one did not.

See Also

AES, object, TOS

objc_draw — AES function (libaes)

Draw an object
#include <aesbind.h>
#include <obdefs.h>
int objc_draw(*tree, object, depth, x, y, w, h***)**
OBJECT ****tree*; int** ***object, depth, x, y, w, h***;**

objc_draw is an AES routine that draws an object. *tree* points to the object tree that contains the object in question. *object* is the number of the object within the object tree. *depth* indicates the number of levels to which the object should be drawn, as follows: zero, draw only the object itself; one, draw the object plus its children; two, draw the object and its children and grandchildren; through eight (which is called **MAX_DEPTH** in **obdefs.h**), which draws the object and all of its descendents. Thus, setting *object* to zero (the root object within the tree) and setting *depth* to **MAX_DEPTH** will draw the entire object tree.

x, *y*, *w*, and *h* set, respectively, the X coordinate of the clipping rectangle, its Y coordinate, its width, and its height.

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

Example

For an example of this routine, see the entry for **object**.

See Also

AES, object, TOS

objc_edit — AES function (libaes)

Edit a text object
#include <aesbind.h>
#include <obdefs.h>
int objc_edit(*tree, object, character, oldindex, kind, newindex***)**
OBJECT ****tree*; int** ***object, character, oldindex, kind, *newindex***;**

objc_edit is an AES routine that edits a text object within an object tree. The object being edited must be either of type **G_TEXT** or **G_BOXTEXT**. *tree* points to the object tree that contains the object being edited, and *object* is the number of that object within the tree. *character* is the character to be inserted into the text. *oldindex* is the index of the character being replaced. *kind* is the type of replacement you want performed, as follows:

0	Reserved
1	Move input text into template; turn on cursor
2	Compare input with validation string; update text; display string
3	Turn off cursor

newindex is the index of the character that follows the one being edited. This value is set by the AES.

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

See Also

AES, TOS

objc_find — AES function (libaes)

Find if mouse pointer is over particular object
#include <aesbind.h>
#include <obdefs.h>

int objc_find(*tree, object, depth, mousex, mousey*)
OBJECT **tree*; **int** *object, depth, mousex, mousey*;

objc_find is an AES routine that finds whether the mouse pointer is positioned over a particular object. *tree* points to the object tree that holds the object in question, and *object* is its number within the object tree.

depth is the depth to which the object tree should be searched, as follows: zero, search only for *object*; one, search for *object* and its children; two, search for the object plus its children and grandchildren; through eight (which is called **MAX_DEPTH** in **obdefs.h**), which searches for the object and all of its descendents.

Finally, *mousex* and *mousey* give the coordinates of the mouse pointer.

objc_find returns the number of the object over which the mouse pointer was found to be positioned, or -1 if it was found not to be positioned over any requested object.

See Also
AES, object, TOS

objc_offset — AES function (libaes)

Calculate an object's absolute screen position
#include <aesbind.h>
int objc_offset(*tree, object, x, y*)
OBJECT **tree*; **int** *object*, **x*, **y*;

objc_offset is an AES routine that returns the absolute position on the screen of a given object. *tree* points to the object tree that holds the object in question, and *object* is its number within the tree. *x* and *y* give, respectively, the X and Y coordinates of the object. These are set by AES.

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

See Also
AES, object, TOS

objc_order — AES function (libaes)

Reorder a child object within the object tree
#include <aesbind.h>
#include <obdefs.h>
int objc_order(*tree, object, newposition*)
OBJECT **tree*; **int** *object, newposition*;

objc_order is an AES routine that moves a child object to a new position within the object tree. *tree* points to the object tree that holds the object to be moved, and *object* is its number within the object tree. *newposition* gives the new position for this object in the list of its siblings: zero indicates the bottom of the list, one indicates one from the bottom, and so on; -1 indicates the top of the list.

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

See Also
AES, object, TOS

object — Technical information

An **object** is an AES data form that encodes an element to be displayed on the screen. An object can be a rectangle, a text string, a box, a bit-mapped picture, a combination of any of these, or (most importantly) a number of such elements linked together in the form of an object tree.

The object tree

An *object tree* is a group of visual elements that are linked together in a tree structure. One object is the tree's *root object*. It can have one or more *child objects* and each child object can have one or more *siblings* and children.

Consider the following example, for the object tree **example**. Like all object trees, **example** has a root object, **example[0]**. This object, in turn, has three children: **example[1]**, **example[2]**, and **example[3]**. Each of these three children has two siblings. For example, **example[2]**'s siblings are **example[1]** and **example[3]**. Each of these children can, in turn, have its own children, each of which can have siblings and children of its own.

As you can see, **example** names an array of objects. Each object's subscript depends on the order in which it is loaded into memory. If you wish to write an object tree by hand, it is up to you to know each object's subscript in order to write the tree correctly.

Each object within the tree contains three "pointers" in its description. These are not true C pointers (i.e., memory addresses), but integers that are used by the AES to orient each object within its tree. The first pointer, **next**, points to the object's next sibling. For example, the **next** pointer for **example[1]** is **2**, which points to **example[2]**. If an object is the last of its siblings or if it has no siblings, then **next** must point to the object's *parent* object. The only exception is the root object, which has no sibling and no parent; its **next** pointer is always set to -1.

The second pointer and third pointers, **head** and **tail**, point respectively to the object's first child and its last child. For example, **example[0]** has a head pointer of **1**, which indicates that **example[1]** is the first of its children, and a tail pointer of **3**, which indicates that **example[3]** is the last of its children. If an object has only one child, then the head and tail pointers must both point to it; and if an object has no children, then both pointers must be set to -1.

Note that if object 1's **head** is set to two and its **tail** is set to seven, this does *not*

mean that objects 3 through 6 are all children of object 1. It only means that the first of its chain of children is object 2 and the last is object 7. The members of object 1's "family" are indicated by the **next** pointers of the children themselves.

The OBJECT structure

Each object in an object tree must be described with the **OBJECT** structure that is declared in the header file **obdefs.h**. This structure is declared as follows:

```
typedef struct object
{
        int ob_next;                    /* Object's next sibling */
        int ob_head;                    /* First child */
        int ob_tail;                    /* Last child */
        unsigned int ob_type;           /* Type of object */
        unsigned int ob_flags;          /* Flags */
        unsigned int ob_state;          /* Status */
        long ob_spec;                   /* Object's specification */
        int ob_x;                       /* X coordinate of object */
        int ob_y;                       /* Y coordinate of object */
        int ob_width;                   /* Width */
        int ob_height;                  /* Height */
} OBJECT;
```

An object, as can be seen, is built out of the following 11 elements:

ob_next The **next** pointer.

ob_head The **head** pointer.

ob_tail The **tail** pointer.

ob_type This indicates the object's type. The different types of object will be discussed below.

ob_flags This field encodes one of a set of flags for the object. The allowable flags are as follows:

0x000	NONE	No flags selected
0x001	SELECTABLE	Selectable by user
0x002	DEFAULT	Default (e.g., for buttons)
0x004	EXIT	If selected, ends dialogue
0x008	EDITABLE	Editable by user (e.g., string)
0x010	RBUTTON	Radio button
0x020	LASTOB	Last object in tree
0x040	TOUCHEXIT	Exit when button is pressed
0x080	HIDETREE	Hide object from searches
0x100	INDIRECT	Redirect to another object

Not every flag applies to every type of object. Some flags are mutually exclusive, e.g., EXIT and TOUCHEXIT. The former must be a selectable object that the user must touch with the mouse pointer and then click and release the button. The latter must *not* be a selectable object and exits when the user merely clicks the mouse button, rather than when button is released.

ob_state This indicates the object's status, i.e., how the object is to be displayed. The status codes are as follows:

0x00	NORMAL	Normal display
0x01	SELECTED	Displayed in reverse video
0x02	CROSSED	'X' drawn in object
0x04	CHECKED	Check mark drawn next to object
0x08	DISABLED	Draw in shading rather than solid
0x10	OUTLINED	Draw border around object
0x20	SHADOWED	Draw shadow beneath object

The SELECTED specification is often used to show that an object has been selected, such as happens when you click an icon on the GEM desktop. The specifications CROSSED, OUTLINED, and SHADOWED are used only with boxes.

The specification can be changed as the program runs; for example, the specification in a menu object can change to indicate that the item is disabled or has been selected. You can either change an object's specification by hand, or you can use an AES library routine to do so. For example, **menu_tnormal** will change a menu entry's specification from DISABLED to NORMAL and vice versa, without your having to address that object directly within its tree. See **objc_change** for more information.

ob_spec The object's specification. This field, which is the only **long** field in the **OBJECT** structure, can hold a pointer to a string, a pointer to a structure, or a bit map, depending on the type of object being described. Which specification belongs with which object will be described below.

ob_x X coordinate of the object. In the root object, this value is an absolute value, in rasters; for each subordinate object, this value is relative to the X value of its parent. This allows the entire object tree to be repositioned on the screen simply by changing the X coordinate of the root object.

ob_y Y coordinate of the object. In the root object, this value is an absolute value, in rasters; for each subordinate object, this value is relative to the Y value of its parent.

ob_width The object's width. This is always an absolute value.

ob_height The object's height. This is always an absolute value.

Types of objects

The following table lists the available types of objects. As noted above, each type of object uses the field **ob_spec** in a different way; the specification is also given:

G_BOX Draw a rectangle on the screen. The field **ob_spec** holds a bit map that describes the box's color and the thickness of its border, as follows:

bits 0-3	interior color
bits 4-6	interior pattern (0=empty, 7=solid)
bit 7	1=transparent, 0=opaque
bits 8-11	text color
bits 12-15	border color
bits 16-23	border thickness (-127 through 127)
bits 24-31	character index

Negative numbers draw the border outwards from the edge of the rectangle, whereas positive numbers draw the border inwards.

The codes for text and interior color are as follows:

0	WHITE
1	BLACK
2	RED
3	GREEN
4	BLUE
5	CYAN
6	YELLOW
7	MAGENTA
8	LWHITE
9	LBLACK
10	LRED
11	LGREEN
12	LBLUE
13	LCYAN
14	LYELLOW
15	LMAGENTA

The names in capital letters are mnemonics that are defined in the header file **obdefs.h**. This means that you can use these mnenomics in your program without having to remember the numeric code of each color. *Example:* To set a figure with a border width of one raster, a border color of black, a text color of black, the transparent bit off, the fill pattern of solid, and an interior color of white, use the following C code:

```
(1<<16)|(BLACK<<12)|(BLACK<<8)|(1<<7)|(7<<4)|WHITE)
```

This translates into the hexadecimal number 0x111F0L.

G_BOXCHAR

This draws a rectangle with a single character inside it. It is used for elements like the "fuller" button on GEM windows. **ob_spec** is the same as for **G_BOX**, except that bits 24-31 encode the character to be displayed within the box.

G_BOXTEXT

This draws a box and writes text inside it. **ob_spec** points to the structure **TEDINFO**, which is described below.

G_BUTTON

This draws a button, which AES handles in its usual manner. **ob_spec** points to the string that is written inside the button.

G_FTEXT

This draws a string on the screen that can be edited by the user in the form of a dialogue. This is demonstrated in the second example, below. **ob_spec** points to the structure **TEDINFO**, which is described below.

G_FBOXTEXT

This draws an editable string, like **G_FTEXT**, but surrounds it with a box as well. **ob_spec** points to the structure **TEDINFO**, which is described below.

G_IBOX

This draws an "invisible box" on the screen. This box is used to connect a number of elements without changing the appearance of the object. For example, if you wished to reverse a large section of the screen when an icon is clicked, you would overlay the icon with an invisible box sized to the dimensions of the area you wished to reverse; when the icon was clicked, the entire area within the invisible box would be reversed, not just the icon itself. **ob_spec** is the same as for **G_BOX**.

G_ICON

This draws an icon on the screen. **ob_spec** points to the structure **ICONBLK**, which is described below.

G_IMAGE

This draws a user-defined shape on the screen. **ob_spec** points to the structure **BITBLK**, which is described below.

G_STRING

This writes a string. **ob_spec** points to the string being written.

G_TEXT

This writes formatted text. **ob_spec** points to the structure **TEDINFO**, which is described below.

G_TITLE

This creates a title on the menu bar. **ob_spec** points to the string to be written. This object is used only in a menu.

G_USERDEF

This is an object defined by the programmer. **ob_spec** points to the structure **USERBLK**, which is described below.

As indicated above, four specialized structures are used by the set of objects: **BITBLK**, **ICONBLK**, **TEDINFO**, and **USERBLK**.

The BITBLK structure

The **BITBLK** structure is defined in the header file **obdefs.h** as follows:

```
typedef struct bit_block
{
        int *bi_pdata;      /* Points to bit map */
        int bi_wb;          /* Width of bit map in bytes */
        int bi_hl;          /* Height in lines */
        int bi_x;           /* Source X in bit form */
        int bi_y;           /* Source Y in bit form */
        int bi_color;       /* Color of bit */
} BITBLK;
```

bi_pdata points to an array of integers that encode the object's bit map. **bi_wb** gives the width of the bit map, in bytes. Note that the value of this variable must be even, to align along word boundaries. **bi_hl** gives the height of the bit map, in rasters. **bi_x** and **bi_y** give, respectively, the X and Y coordinates of the bit map. Finally, **bi_color** gives the object's color, encoded as above.

The ICONBLK structure

The structure **ICONBLK** is defined in the header file **obdefs.h** as follows:

```
typedef struct icon_block
{
        int *ib_pmask;          /* Points to icon mask */
        int *ib_pdata;          /* Points to icon description */
        char *ib_ptext;         /* String to appear in icon */
        int ib_char;            /* Character to appear in icon */
        int ib_xchar;           /* X location of character */
        int ib_ychar;           /* Y location of character */
        int ib_xicon;           /* X location of icon */
        int ib_yicon;           /* Y location of icon */
        int ib_wicon;           /* Width of icon */
        int ib_hicon;           /* Height of icon */
        int ib_xtext;           /* X location of text */
        int ib_ytext;           /* Y location of text */
        int ib_wtext;           /* Width of text */
        int ib_htext;           /* Height of text */
} ICONBLK;
```

ib_pmask points to an array of integers that describe the icon mask. **ib_pdata** points to an array of integers that describe the icon itself. **ib_text** points to a string to be written into the icon. **ib_char** holds the index of the character for the icon in its low byte. The foreground (data) color is stored in bits 12-15, and the background (mask) color is stored in bits 8-11. The color codes are the same as those listed for **G_BOX**, above. **ib_xchar** and **ib_ychar** give, respectively, the X and Y coordinates of the character. **ib_xicon**, **ib_yicon**, **ib_wicon**, and **ib_yicon** give, respectively, the X coordinate, the Y coordinate, the width, and the height of the icon. **ib_xtext**, **ib_ytext**, **ib_wtext**, and **ib_htext** give, respectively, the X coordinate, the Y coordinate, the width, and the height of the text string at the bottom of the icon.

The TEDINFO structure

This structure is used to create an editable dialogue. It is defined in the header file **obdefs.h** as follows:

```
typedef struct text_edinfo
{
        long te_ptext;          /* Points to text */
        long te_ptmplt;         /* Points to template */
        long te_pvalid;         /* Points to validation chars */
        int te_font;            /* Font */
        int te_junk1;           /* Junk word */
        int te_just;            /* Justification */
        int te_color;           /* Color */
        int te_junk2;           /* Junk word */
        int te_thickness;       /* Border thickness */
        int te_txtlen;          /* Length of text string */
        int te_tmplen;          /* Length of template string */
} TEDINFO;
```

te_ptext points to a string to be displayed within the object. The text typed by the user will be written over this string. If you do not want text to be displayed, replace it with a string of '@' characters as long as the maximum length of the string to be input.

te_ptmplt points to a template that will be used to input data. The template consists of a prompt, plus a string of underbar characters that is as long as the maximum length of the string that the user can input. The following is an example of a template string:

```
ENTER FILE NAME: _______.___
```

te_pvalid points to a string of validation characters. This string must be as long as the string that the user can input. Each character input by the user is checked against its corresponding validation character to ensure that it is of the right type. The validation characters are as follows:

9	All numerals, zero through nine
a	All alphabetic characters plus space
n	Alphabetic characters, numerals, space
p	Valid TOS path name characters
A	Upper-case alphabetic characters plus space
N	Upper-case alphabetic characters, space, numerals
F	TOS file name characters, question mark, asterisk, colon
P	TOS path name characters, question mark, asterisk, colon
X	Anything

In some versions of the AES, entry of an underscore to any validation character besides **F** *or* **X** *will cause a catastrophic system error.*

te_font indicates which font you want. **te_junk1** and **te_junk2** are reserved; they can be set to any value. **te_just** indicates how you want the text to be justified: **TE_LEFT** indicates left justification; **TE_RIGHT**, right justification; and

TE_CNTR, centering. **te_color** indicates the color of the object; the color codes are the same as for **G_BOX**.

te_thickness is the thickness of the border; it uses the same values as **G_BOX**. Finally, **te_txtlen** and **te_tmplen** give, respectively, the length of the user input string and the length of the template, each in bytes. The length of each should be one byte longer than the strings pointed to by **te_ptext** and **te_ptmplt**, to allow the addition of the NUL character at the end of each.

The USERBLK structure

The **USERBLK** structure is called the **APPLBLK** or **APPL_BLK** structure in other bindings. It is defined in the header file **obdefs.h** as follows:

```
typedef struct user_blk
{
        long ub_code;       /* points to user's code */
        long ub_parm;       /* points to parameter */
} USERBLK;
```

This structure allows you to define your own object or routine; **ub_code** points to the routine in question, which can be specialized code written in C or assembly language to do tasks beyond the scope of the normal AES routines. **ub_parm** points to the parameter to be passed to the routine named in **ub_code**. To use this structure, a programmer must have a sophisticated grasp of the AES.

Designing objects

Designing an object by hand is difficult. Whenever possible, you should use **resource**, the Mark Williams resource editor to design screen elements, or prepare a resource-description file that can be compiled with **rescomp**, the Mark Williams resource compiler. The following describes how to build an object by hand in C, to help you grasp the structure of objects and object trees.

Before beginning, you should do the following: First, draw a picture of the object on graph paper. For text, each cell on the graph paper can considered equivalent to one character cell, i.e., the space taken up by one standard character on the screen (in high resolution, a character is eight rasters wide by 16 high; in medium resolution, it is eight rasters wide by eight high; and in low resolution, it is four wide by eight high). Otherwise, each cell can be considered equivalent to a pixel. Drawing the picture is tedious, but it will save you time over trying to draw it "on the fly" on the screen.

Second, draw a "genealogical table" of all the objects within the object tree. This will ensure that you set the **next**, **head**, and **tail** pointers for each object correctly. An example of such a table appears in the entry for **menu**.

Example

This example draws a set of seven nested rectangles on the screen. Typing any key returns you to **msh**.


```
#include <aesbind.h>
#include <gemdefs.h>
#include <obdefs.h>

#define SPEC1 0x100F1L
/*
 * i.e.: (1 << 16) |       [Border 1 raster thick]
 *       (WHITE << 12) |   [Border color; WHITE = 0]
 *       (WHITE << 8) |    [Text color]
 *       (1 << 7) |        [Turn on replace]
 *       (7 << 4) |        [Fill pattern to solid]
 *        BLACK            [Fill color; BLACK = 1]
 */

#define SPEC2 0x111F0L
/*
 * i.e.: (1 << 16) |       [Border 1 raster thick]
 *       (BLACK << 12) |   [Border color]
 *       (BLACK << 8) |    [Text color]
 *       (1 << 7) |        [Turn on replace]
 *       (7 << 4) |        [Fill pattern to solid]
 *        WHITE            [Fill color]
 */

/* define object; widths and heights will be set elsewhere */
OBJECT fill[] = {
/* next/head/tail/type/  flags / state /spec./  X / Y/  W /  H */
    -1,  1,  1, G_BOX, DEFAULT, NORMAL, SPEC1,  0,  0,  0,   0,
     0,  2,  2, G_BOX, DEFAULT, NORMAL, SPEC2,  0,  0,  0,   0,
     1,  3,  3, G_BOX, DEFAULT, NORMAL, SPEC1,  0,  0,  0,   0,
     2,  4,  4, G_BOX, DEFAULT, NORMAL, SPEC2,  0,  0,  0,   0,
     3,  5,  5, G_BOX, DEFAULT, NORMAL, SPEC1,  0,  0,  0,   0,
     4, -1, -1, G_BOX, DEFAULT, NORMAL, SPEC2,  0,  0,  0,   0
};

main()
{
        int nowhere = 0;    /* For unused pointers */
        int i;
        int x, y, w, h;     /* Dimensions of screen */

        appl_init(); /* Begin application */
        /*
         * get size of screen; set object dimensions.
         * "0" is desktop window, which always fills screen
         */
        wind_get(0, WF_FULLXYWH, &x, &y, &w, &h);
        fill[0].ob_width =  w;
        fill[0].ob_height = h;

        for (i = 1; i < 6; i++) {
                fill[i].ob_x = w/12;
                fill[i].ob_y = h/12;
                fill[i].ob_width  = w - ((w/6)*i);
                fill[i].ob_height = h - ((h/6)*i);
        }
```

```
        /* Turn off mouse pointer */
        graf_mouse(M_OFF, &nowhere);

        /* Draw object */
        objc_draw(fill, ROOT, MAX_DEPTH, 0, 0, w, h);

        /* Wait for keybd event */
        evnt_keybd();

        /* Turn on mouse ptr */
        graf_mouse(M_ON, &nowhere);

        appl_exit();
        exit(0);
}
```

See Also

AES, menu, obdefs.h, rescomp, resdecom, resource, TOS, window

object format – Definition

An **object format** describes the form of compiled program that still contains relocation information. The linker **ld** reads file in object format to create executable files.

Mark Williams C creates object modules that are in the format **n.out**, which differs somewhat from other formats used on the Atari ST.

See Also

ld, n.out

od – Command

Print a hexadecimal dump of a file

od [-bcdox] [*file***] [[+] *offset*[.][b]]**

od prints the specified *file* as a sequence of octal numbers, or machine words. If no *file* is specified, **od** dumps the standard input.

The following options allow the user to select the output format:

-b	bytes in hexadecimal
-c	bytes in ASCII characters
-d	words in decimal
-o	words in octal

Dumping can start at *offset* into the file. The specified *offset* is octal unless the '.' suffix is present to signify decimal. The *offset* is in bytes unless the **b** suffix is present to signify 512-byte blocks.

See Also

ASCII, commands, db, msh

Offgibit – xbios function 29 (osbind.h)

Clear a bit in the sound chip's A port

#include <osbind.h>
#include <xbios.h>
void Offgibit(*mask*) char *mask*;

Offgibit manipulates the sound chip's register A (also called the "A port"). This port controls the disk drives.

Offgibit reads the contents of register A; it then ANDs this value with *mask*; and it writes the result back into register A. The bits in this register are bound to various control lines within the Atari ST. For a table of which bits bind which lines, see the entry for **Ongibit**.

Example

The following example demonstrates **Ongibit** and **Offgibit**:

```
#include <osbind.h>

main() {
        unsigned char a;

        Cconws("Wait for both floppy drives to stop and type a key\r\n");
        Cnecin();

        a = Giaccess(0, 14);                    /* save the original value... */

        Offgibit(0xF9);                         /* turn off bits 1 and 2 */
        Cconws("Both floppy drive lights on...\n\r");
        Cnecin();

        Ongibit(0x02);                          /* turn on bit 1 */
        Cconws("Drive A light off...\n\r");
        Cnecin();

        Ongibit(0x04);                          /* turn on bit 2 */
        Cconws("Drive B light off...\n\r");
        Cnecin();

        Giaccess(a,0x80|14);                    /* restore original contents */
        Pterm0();
}
```

See Also

Giaccess, Ongibit, TOS, xbios

Ongibit – xbios function 30 (osbind.h)

Turn on a bit in the sound chip's A port

#include <osbind.h>
#include <xbios.h>
void Ongibit(*mask*) char *mask*;

Ongibit manipulates the sound chip's register A (also called the "A port").

Ongibit first reads the contents of register A; it then ORs with *mask*; and finally it writes the result back into register A.

The bits in register A are bound to various control lines within the Atari ST, as follows:

0	side of the floppy disk (0/1)
1	drive A (selected when clear)
2	drive B (selected when clear)
3	RS-232 request-to-send (RTS) line
4	RS-232 data-terminal-ready (DTR) line
5	Centronics data strobe
6	general purpose output (GPO) on video connector
7	unused

number should be set the bit that corresponds to the desired line.

Example

For an example of this function, see the entry for **Offgibit**.

See Also

Giaccess, Offgibit, TOS, xbios

open — UNIX system call (libc)

Open a file

int open(*file*, *type*) **char** **file*; **int** *type*;

open prepares a *file* to receive data, or to have its data read. When it can open *file*, **open** returns a file descriptor, which is a small, positive integer that identifies the open *file* for subsequent calls to **read, write, close, dup,** or **dup2**. *type* determines how the file is opened, as follows:

0	read only
1	write
2	read and write

After *file* is opened, reading or writing begins at byte 0.

Example

This example copies the file named in **argv[1]** to the one named in **argv[2]** by using UNIX-style routines. It demonstrates the functions **open, close, read, write,** and **creat**.


```
#include <stdio.h>
#define BUFSIZE (20*512)
char buf[BUFSIZE];

main(argc, argv) int argc; char *argv[]; {
      register int ifd, ofd;
      register unsigned int n;

      if (argc != 3)
            fatal("Usage: copy source destination");
      if ((ifd = open(argv[1], 0)) == -1)
            fatal("cannot open input file");
      if ((ofd = creat(argv[2], 0)) == -1)
            fatal("cannot open output file");

      while ((n = read(ifd, buf, BUFSIZE)) != 0) {
            if (n == -1)
                  fatal("read error");
            if (write(ofd, buf, n) != n)
                  fatal("write error");

      }
      if (close(ifd) == -1 || close(ofd) == -1)
            fatal("cannot close");
      exit(0);
}
fatal(s) char *s;
{
      fprintf(stderr, "copy: %s\n", s);
      exit(1);
}
```

See Also

fdopen, file descriptor, fopen, STDIO, UNIX routines

Diagnostics

open returns -1 if the file is nonexistent, or if a system resource is exhausted.

Notes

open is a low-level call that passes data directly to TOS. It should not be mixed with high-level calls, such as **fread, fwrite,** or **fopen**.

operator — Definition

An **operator** relates one operand to another. For example, the statement

```
1+2
```

relates the operands **1** and **2** through the operation of addition; on the other hand, the statement

```
A>B
```

relates the operands **A** and **B** logically, by asserting that the former is greater than the latter; whereas

```
A=B
```

relates the operands **A** and **B** by assigning the value of the latter to the former. The following is a table of the C operators:

*	multiplication
/	division
%	remainder
+	addition
-	subtraction
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
&&	logical AND
!=	inequality
!	logical negation
||	logical OR
=	assign
+=	increment and assign
-=	decrement and assign
*=	multiply and assign
/=	divide and assign
%=	modulo and assign
++	increment
--	decrement
==	equivalence
&	bitwise AND
^	bitwise exclusive OR
~	bitwise complement
|	bitwise inclusive OR
<<	shift left
>>	shift right
*	indirection
&	render an address
()	function indicator
[]	array indicator
->	structure pointer
.	structure member
? :	conditional expression
sizeof	size of an object

For a table of the precedence of operators, see the entry for **precedence**.

See Also

precedence, sizeof

The C Programming Language, page 48.

osbind.h — Header file

Declare TOS functions

#include <osbind.h>

osbind.h is a header file that declares the functions **bios()**, **gemdos()**, and **xbios()**. It also defines numerous macros that ease the use of these functions. The text of **osbind.h** is included with your copy of Mark Williams C.

See Also

bios, gemdos, header file, xbios, TOS

P

path — Definition

A **path** is a sequence of names that are separated by a fixed character, '/' under the COHERENT or UNIX operating systems, or '\' under TOS or MS-DOS. Each name is a directory that contains the next-named directory; the only exception is the last name in the path, which may be the name of a file instead of a directory. For example, the COHERENT path

```
doc/letters/fred
```

names first the directory **doc**; then the directory **letters**, which is a sub-directory of **doc**; and finally **fred**, which can name either a directory or a file.

If a path begins with the separator character, then it is called *absolute*, and the first directory is a sub-directory of the root directory of the current physical device. Otherwise, the path is assumed to begin with the current directory. For example, the above example is a relative path that begins with the current directory; however, the following path names an absolute path for MS-DOS or TOS:

```
A:\doc\letters\fred
```

See Also

msh, PATH, path.h, setenv

path — General function (libc)

Build a path name for a file

#include <path.h>
#include <stdio.h>
char *path(*path, filename, mode***) char ****path***, ****filename***; int** ***mode***;

path is a general function that builds a path name for a file. *path* points to the list of directories to be searched for the file. You can use the function **getenv** to obtain the current definition of the environmental variable **PATH**; or use the default setting of **PATH** found in the header file **path.h**; or, you can define *path* by hand. *filename* is the name of the file for which **path** is to search. *mode* is the mode in which you wish to access the file, as follows:

1	execute the file
2	write to the file
4	read the file

path uses the function **access** to check the access status of *filename*. If **path** finds the file you requested and the file is available in the mode that you requested, it returns a pointer to a static area in which it has built the appropriate path name. It returns NULL if either *path* or *filename* are NULL, if the search failed, or if the requested file is not available in the correct mode.

Example

This example accepts a file name and a search mode. It then tries to find the file in one of the directories named in the **PATH** environmental variable.

```
#include <stdio.h>
#include <path.h>

main(argc, argv)
int argc; char *argv[];
{
        char *env, *pathname;
        extern char *getenv(), *path();
        int mode;

        if (argc != 3)
        {
                printf("Usage: findpath filename mode\n");
                exit(0);
        }

        if(((mode=atoi(argv[2]))>4) || (mode==3) || (mode<1))
        {
                printf("modes: 1=execute, 2=write, 3=read\n");
                exit(0);
        }

        env = getenv("PATH");
        if ((pathname = path(env, argv[1], mode)) != NULL)
        {
                printf("PATH = %s\n", env);
                printf("pathname = %s\n", pathname);
        }
        else
                printf("search failed\n");
}
```

See Also

access, access.h, PATH, path.h, stdio

path.h — Header file

Declare path()

#include <path.h>

path.h is a header file that declares the function **path**. It also contains a number of default definitions for variables, including **PATH** and **LIBPATH**.

See Also

path, PATH

PATH — Environmental variable

Directories that hold executable files

PATH names a default set of directories that are searched by **msh** when it seeks an executable file. You can set **PATH** with the command **setenv**. For example,

typing

```
setenv PATH=.bin,\bin,,\lib
```

tells **msh** to search for executable files first in its set of built-in commands (as indicated by **.bin**), then in the directory **\bin**, then in the current directory (as indicated by the two commas with nothing between them), and finally in the directory **lib**.

See Also

msh, path, path.h, setenv

pattern — Definition

A **pattern** is any combination of ASCII characters and wildcard characters that can be interpreted by a command.

The function **pnmatch** compares two patterns and signals if they match.

See Also

egrep, pnmatch, wildcard

peekb — General function (libc)

Extract a byte from memory
int peekb(*bp*) char *bp;

peekb examines an arbitrary location in memory. It reads a byte located at the address *bp*. **peekb** circumvents the system's memory protection by temporarily entering supervisor mode.

See Also

peekl, peekw, pokeb, pokel, pokew

Notes

peekb is supplied for use in user-mode programs. Programs that run in supervisor mode, i.e., interrupt handlers, trap handlers, and boot sector programs, should access the memory locations directly with the following macro:

```
#define peekb(cp)   (*((char *)cp))
```

peekb does not work correctly in supervisor mode, which allows you to access memory locations directly.

peekl — General function (libc)

Extract a long from memory
long peekl(*lp*) long *lp;

peekl returns the **long** (four bytes) at *lp*. **peekl** circumvents the system's memory protection by temporarily entering supervisor mode.

See Also

peekb, peekw, pokeb, pokel, pokew

Notes

peekl does not test for odd addresses, and will generate a bus error if given such an address. In general, be careful about what you **peek** and **poke**.

peekl is supplied for use in user-mode programs. Programs that run in supervisor mode, i.e., interrupt handlers, trap handlers, and boot sector programs, should access the memory locations directly with the following macro:

```
#define peekl(lp)   (*((long *)lp))
```

peekl does not work correctly in supervisor mode, which allows you to access memory locations directly.

peekw — General function (libc)

Extract a word from memory
int peekw(*wp*) int *wp;

peekw returns the **word** (two bytes) at *wp*. **peekw** circumvents the system's memory protection by temporarily entering supervisor mode.

See Also

peekb, peekl, pokeb, pokel, pokew

Notes

peekw does not test for odd addresses, and will generate a bus error if given such an address. In general, be careful about what you **peek** and **poke**.

peekw is supplied for use in user-mode programs. Programs that run in supervisor mode, i.e., interrupt handlers, trap handlers, and boot sector programs, should access the memory locations directly with the following macro:

```
#define peekw(wp)   (*((int *)wp))
```

peekw does not work correctly in supervisor mode, which allows you to access memory locations directly.

perror — General function (libc)

System call error messages
#include <errno.h>
perror(*string*)
char *string; extern int *sys_nerr*; extern char *sys_errlist[];

perror prints an error message on the standard error device. The message consists of the argument *string*, followed by a brief description of the last system call that failed. The external variable **errno** contains the last error number. Normally, *string* is the perror of the command that failed or a file perror.

The external array **sys_errlist** gives the list of messages used by **perror**. The external **sys_nerr** gives the number of messages in the list.

See Also

errno, errno.h, error codes

Pexec — gemdos function 75 (osbind.h)

Load or execute a process

#include <osbind.h>
long Pexec(*mode, path, tail, env***)**
int *mode* **; char ****path***, ****tail***, ****env***;**

Pexec loads or executes a process. *mode* equals zero if the process is to be loaded and executed, or three if the process is to be loaded but not executed; the latter mode is used with overlays. *path* points to the path name of the file to be loaded; it must be a NUL-terminated string. *tail* points to the command tail, which included redirection information. *env* points to a block of strings that define the environment. Each string must terminate with a NUL character, and the block as a whole must terminate in **NULL**.

If *mode* equals zero, **Pexec** returns the child process's exit status when the child process exits; if *mode* equals three, it returns the address of the base page of the loaded process. In either instance, it returns a negative error code if it cannot load the process.

Example

This example times the execution speed of a program. It also demonstrates the time function **clock**.

```
#include <osbind.h>
#include <time.h>

main(argc, argv)
int argc; char *argv[];
{
      char program[80];
      char command[256];
      int x;
      clock_t timer;
      int status;

      if (argc < 2) {
            printf("usage: time command [ args ... ]\n");
            exit(1);
      }

      strcpy(program,argv[1]);
      strcat(program,".PRG");
      command[0] = 0;
```



```
      for (x=2 ; x < argc ; x++) {
            strcat( command, " ");
            strcat( command, argv[x]);
      }

      timer = clock();
      status = Pexec(0, program, command, "PATH=\0");
      timer = clock() - timer;

      printf("%ld.%03ld seconds\n",
            timer/CLK_TCK, (timer%CLK_TCK) * (1000/CLK_TCK));
      return status;
}
```

See Also

argv, gemdos, TOS

Physbase — xbios function 2 (osbind.h)

Read the physical screen's display base

#include <osbind.h>
#include <xbios.h>
char *Physbase()

Physbase reads the physical screen's display base, and returns a pointer to the display base. The physical screen base is the location in memory currently displayed.

Example

The following example uses **Physbase** and **Setscreen** to allow you to display graphically the entire system memory. Typing the up-arrow key scrolls the screen up the equivalent of one character row; typing the down-arrow key scrolls the screen down; typing **<Help>** lets you move the display to an absolute memory address; and typing **<Undo>** tells you the current memory base. Typing **<return>** exits. Moving the memory base to zero allows you to observe the operation of TOS, including stack, clocks, and peripheral devices; you are invited to manipulate the peripheral devices to observe them in action.

```
#include <aesbind.h>
#include <gemdefs.h>
#include <osbind.h>
#include <stdio.h>

/* manifest constants */
#define UP_ARROW 0x4800
#define DN_ARROW 0x5000
#define HELP 0x6200
#define RETURN 0x1C00
#define UNDO 0x6100

/* externs */
extern long atol();
```

```
main()
{
        char *oldphys;      /* base of default video display */
        char *setting;      /* base of maneuverable video display */
        char holder[50];
        unsigned long tmp;

        /* initialize base of video display, and begin */
        setting = oldphys = Physbase();
        appl_init();

        for (;;)
        {
                switch(evnt_keybd())
                {

                        /* move up one row */
                        case UP_ARROW:
                                /* round to row boundary */
                                setting -= 1280;
                                if (setting < 0)
                                        setting = 0;
                                /* reset base of physical display */
                                Setscreen(-1L, setting, -1);
                                break;

                        /* move down one row */
                        case DN_ARROW:
                                setting += 1280;
                                Setscreen(-1L, setting, -1);
                                break;

                        /* move to absolute address */
                        case HELP:
                                Setscreen(-1L, oldphys, -1);
                                printf("Enter memory setting, decimal: ");
                                fflush(stdout);
                                tmp = atol(gets(holder));
                                setting = (char *)tmp;

                                if (setting < 0)
                                        setting = 0;
                                Setscreen(-1L, setting, -1);
                                break;

                        /* reset original video base, and exit */
                        case RETURN:
                                Setscreen(-1L, oldphys, -1);
                                appl_exit();
                                exit(0);
```



```
                        /* show current video memory base */
                        case UNDO:
                                Setscreen(-1L, oldphys, -1);
                                printf("Screen base is %lu\n", setting);
                                evnt_keybd();
                                Setscreen(-1L, setting, -1);
                                break;

                        default:
                                break;
                }
        }
}
```

See Also

Logbase, Setscreen, TOS, xbios

picture — Example

Format numbers under mask

double picture(*number, mask, output***)**
double *number*; **char** **mask*, **output*;

picture uses a mask to format a double-precision number. It is designed to be used with programs that require precise formatting of printed numbers.

picture formats a given number by using a mask string. The mask may contain any characters; however, only a few have special significance. Non-special characters in the mask body are printed if, during execution, they are preceded by one or more numerals. Trailing non-special characters print if the displayed number is negative.

The following lists the special characters that control formatting within a mask:

9 Provides a slot for a number. For example, 5 with mask **999 CR** gives **005<sp><sp><sp>**, whereas printing -5 with mask **999 CR** gives **005 CR**. Note that 'C' and 'R' are not special characters, but are taken literally.

Z Provide a slot for a number but supress leading zeroes. For example, printing **1034** with mask **ZZZ,ZZZ** gives **<sp><sp>1,034**. Note that the comma is not a special character, but is printed literally.

J Provide a slot for a number but shrink out leading zeroes. For example, printing **1034** with mask **JJJ,JJJ** gives **1,034**.

K Provide a slot for a number but shrink out all zeroes. For example, printing **070884** with mask **K9/K9/K9** gives **7/8/84**.

$ Print a dollar sign to the front of the displayed number. For example, printing **105** with mask **$Z,ZZZ** gives **<sp><sp>$105**.

. Separate the number between decimal and integer portions. For example, printing **105.67** with mask **ZZZ.999** gives **105.670**.

T Provide a slot for a number, but supress trailing zeroes. For example, printing **105.670** with mask **ZZ9.9TT** gives **105.67<sp>**.

S Provide a slot for a number, but shrink out trailing zeroes. For example, printing **105.600** with mask **ZZ9.9SS** gives **105.6**.

- If you place a hyphen to the left of the mask, it is printed at the beginning of the number, but only if it is negative. For example, printing **105** with mask **-Z,ZZZ** yields **<sp><sp>105**, whereas printing **-105** yields **<sp><sp>-105**.

(This character acts like the minus sign '-', but prints a '('. For example, printing **105** with mask **(ZZZ)** gives **<sp>105<sp>**, whereas printing **-5** gives **<sp><sp>(5)**.

+ If placed to the left of the mask, this character floats to the front like the minus sign '-', but is replaced by a '-' if the number is minus. For example, printing **5** with mask **+ZZZ** gives **<sp><sp>+5**, whereas printing **-5** gives **<sp><sp>-5**. Placed behind the mask, it is printed if the number is positive, but is replaced by a minus sign '-' if the number is negative. For example, printing **5** with mask **ZZZ+** gives **<sp><sp>5+**, whereas printing **-5** gives **<sp><sp>5-**.

***** When placed to the left of the mask, this character fills all leading spaces to its right. For example, printing **104.10** with mask ***ZZZ,ZZZ.99** gives *******104.10**, and printing **104.10** with mask ***$ZZ,ZZZ.99** gives ******$104.10**.

Example

For an example of **picture**, compile the source program **picture.c** with the option **-DTEST**.

See Also

commands, STDIO

Diagnostics

picture returns all overflow as a **double**. For example, attempting to print **-1234** with mask **(ZZZ)** gives **(234)** and returns **-1**.

Notes

For the source code of **picture**, see the file **picture.c**, which is included with Mark Williams C. Note that **picture** is not included in a library.

pnmatch — String function (libc)

Match string pattern

int pnmatch(*string, pattern, flag***)**

char ****string*, ******pattern*; **int *flag*;

pnmatch matches *string* with *pattern*, which is a regular expression. **pnmatch** returns one if *pattern* matches *string*, and zero if it does not. Each character in *pattern* must exactly match a character in *string*; however, the wildcards '*', '?', '[', and ']' can be used in *pattern* to expand the range of matching. The *flag* argument must be either zero or one: zero means that *pattern* must match *string* exactly, whereas one means that *pattern* can match any part of *string*. In the latter case, the wildcards '^' and '$' can also be used in *pattern*.

Example

For an example of this function, see the entry for **fgets**.

See Also

egrep, msh, string

Notes

flag must be zero or one for **pnmatch** to yield predictable results.

pointer — Definition

A **pointer** is a data type that consists of the address of another item of data; therefore, it is said to "point" to that item of data.

The physical size of the pointer data type is determined entirely by the microprocessor. Pointers are 16 bits long on the i8086, SMALL model, Z8001, and on the PDP-11; they are 32 bits long on the i8086, LARGE model, Z8002, the 68000, and the VAX.

Note that failure to declare a function that returns a pointer will result in that function being implicitly declared as an **int**. This will not cause an error on microprocessors in which an **int** and a pointer have the same size; however, transporting this code to a microprocessor in which an **int** consists of 16 bits and a pointer consists of 32 bits will result in the pointers being truncated to 16 bits and the program probably failing.

C allows pointers and integers to be compared or converted to each other without restriction. Mark Williams C flags such conversions with the strict message

```
integer pointer pun
```

and comparisons with the strict message

```
integer pointer comparison
```

These problems should be corrected if you want your code to be portable to other computing environments.

Casting a pointer from one data type to another may result in the loss of precision when alignment restrictions are taken into account. These sorts of data transformations should be done with great care to ensure that code remains portable.

See Also

data formats, declarations, portability, pun

pokeb — General function (libc)

Insert a byte into memory
int pokeb(*bp, b***) char *****bp; int** *b*;

pokeb writes the character *b* at an arbitrary location *bp* in memory. **pokeb** circumvents the system's memory protection by temporarily entering supervisor mode. **pokeb** returns its argument **b**.

See Also

peekb, peekl, peekw, pokel, pokew

Notes

pokeb is supplied for use in user-mode programs. Programs that run in supervisor mode, i.e., interrupt handlers, trap handlers, and boot sector programs, should access the memory locations directly with the following macro:

```
#define pokeb(cp,c) (*((char *)cp) = c)
```

pokeb does not work correctly in supervisor mode, which allows you to access memory locations directly.

pokel — General function (libc)

Insert a long into memory
long pokel(*lp, l***) long *****lp, l**;

pokel writes the **long** *l* (four bytes) at an arbitrary location *lp* in memory. **pokel** circumvents the system's memory protection by temporarily entering supervisor mode.

See Also

peekb, peekl, peekw, pokeb, pokew

Notes

pokel does not test for odd addresses, and will generate a bus error if given such an address. In general, be careful about what you **peek** and **poke**.

pokel is supplied for use in user-mode programs. Programs that run in supervisor mode, i.e., interrupt handlers, trap handlers, and boot sector programs, should access the memory locations directly with the following macro:

```
#define pokel(lp,l) (*((long *)lp) = l)
```

pokel does not work correctly in supervisor mode, which allows you to access memory locations directly.

pokew — General function (libc)

Insert a long into memory
int pokew(*wp, l***) int *****wp, w**;

pokew writes the word *w* (two bytes) at an arbitrary location *wp* in memory. **pokew** circumvents the system's memory protection by temporarily entering supervisor mode.

See Also

peekb, peekl, peekw, pokeb, pokel

Notes

pokew does not test for odd addresses, and will generate a bus error if given such an address. In general, be careful about what you **peek** and **poke**.

pokew is supplied for use in user-mode programs. Programs that run in supervisor mode, i.e., interrupt handlers, trap handlers, and boot sector programs, should access the memory locations directly with the following macro:

```
#define pokew(wp,w) (*((int *)wp) = w)
```

pokew does not work correctly in supervisor mode, which allows you to access memory locations directly.

port — Definition

A **port** passes data to and receives data from a remote device.

See Also

aux:, fclose, FILE, fopen, prn:, stream

portability — Technical information

Portability means that code can be recompiled and run under different computing environments without modification. Although true portability is an ideal that is difficult to realize, you can take a number of practical steps to ensure that your code is portable:

1. Do not assume that an integer and a pointer have the same size. Remember that undeclared functions are assumed to return an **int**. If a function returns a pointer, declare it so.
2. Do not write routines that depend on a particular order of code evaluation, particular byte ordering, or particular length of data types.
3. Do not write routines that play tricks with a machine's "magic characters"; for example, writing a routine that depends on a file's ending with <**ctrl-Z**> instead of **EOF** ensures that that code can run only under operating systems that recognize this magic character.

4. Always use manifest constants, such as **EOF**, and make full use of **#define** statements.
5. Use header files to hold all machine-dependent declarations and definitions.
6. Declare everything explicitly. In particular, be sure to declare functions as **void** if they do not return a value; this avoids unforeseen problems with undefined return values.
7. Do not assume that integers and pointers have the same size or even the same kind of structure. Do not assume that pointers are all the same or can point anywhere. On the i8086, in SMALL model a pointer to a function addresses relative to the code segment, whereas a pointer to data addresses relative to the data segment. On some machines, character pointers are of a different size or structure than word pointers.
8. The constant NULL is defined as being different from any valid pointer. Use it and nothing else for that purpose.

See Also

#define, header file, manifest constant, pointer, pun, void

pow — Mathematics function (libm)

Compute a power of a number
#include <math.h>
double pow(*z*, *x*) double *z*, *x*;

pow returns *z* raised to the power of *x*, or *z*^*x*.

Example

For an example of this function, see the entry for **exp**.

See Also

mathematics library

Diagnostics

pow indicates overflow by an **errno** of **ERANGE** and a huge returned value.

pr — Command

Paginate and print files
pr [*options*] [*file* ...]

pr paginates each named *file* and sends it to the standard output. The file name '-' means standard input. If no *file* is specified, **pr** reads the standard input.

Each page has a header that gives the date, file name, and page and line numbers. **pr** may be used with the following options.

+*n* Skip the first *n* pages of each input file.

-*n* Print the text in *n* columns. This is used to print out material that was typed in one or more columns.

-h *header* Use *header* in place of the text name in the title. If *header* is more than one word long, it must be enclosed within quotation marks.

-e*ck* Reset spacing represented by tab character. On input, expand tab character *c* to positions *k* plus one, two times *k* plus one, three times *k* plus one, etc. The default *c* is '\t'. The default value of *k* is eight.

-i*ck* Replacing spacing with the tab character. On input, replace spaces with the tab character *c* at positions *k* plus one, two times *k* plus one, three times *k* plus one, etc. The default *c* is '\t'. The default value of *k* is eight.

-l*n* Set the page length to *n* lines (default, 66).

-m Print the texts simultaneously in separate columns. Each text will be assigned an equal amount of width on the page; any lines longer than that will be truncated. This is used to print several similar texts or listings simultaneously.

-n Number each line as it is printed.

-s*c* Separate each column by the character *c*. You can separate columns with a letter of the alphabet, a period, or an asterisk. Normally, each column is left justified in a fixed-width field.

-t Suppress the printing of the header on each page, as well as the header and footer space.

-w*n* Set the page width to *n* columns (default, 80). Text lines are truncated to fit the column width. The maximum width is 256 columns.

Example

To print a numbered listing of a text file, do the following: First, plug a printer into your Atari ST and turn it on. Second, type this command:

```
pr -n filename >prn:
```

where *filename* is the name of the file you wish to print.

See Also

commands, prn:

precedence — Definition

Precedence refers to the property of each C operator that determines priority of execution; operators are executed in order of their degree of precedence, from

highest to lowest.

The following table summarizes the precedence of C operators. The are listed in descending order of precedence: those listed higher in the table are executed before those lower in the table. Operators listed on the same line have the same level of precedence.

Operator	*Associativity*
() [] -> .	Left to right
! ~ ++ -- - (type) * & sizeof	Right to left
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= *= /= %=	Right to left
,	Left to right

See Also

operators
The C Programming Language, page 48

printf — STDIO function (libc)

Format output
int printf(*format*, [*arg1*, ... *argN*])
char **format*; [**data type**] *arg1*, ... *argN*;

printf uses the *format* string to specify an output format for each *arg*, which it then writes on the standard output. **printf** reads characters from *format* one at a time; any character other than a percent sign '%' or a string that is introduced with a percent sign is copied to the output directly. '%' tells **printf** that what follows specifies how the corresponding *arg* is to be formatted; the characters that follow

'%' can set the output width and the type of conversion desired. The following modifiers, in this order, may precede the conversion type:

1. A minus sign '-' will left-justify the output field, instead of the default right justify.

2. A string of digits gives the *width* of the output field. Normally, the field is padded with spaces to the field width; it is padded on the left unless left justification is specified with a '-'. If the field width begins with '0', the field is padded with '0' characters instead of spaces; the '0' does not cause the field width to be taken as an octal number. If the width specification is an asterisk '*', the routine uses the next *arg* as an integer that gives the width of the field.

3. A period '.' followed by one or more digits gives the *precision*. For floating point (**e**, **f**, and **g**) conversions, the precision sets the number of digits printed after the decimal point. For string (**s**) conversions, the precision sets the maximum number of characters that can be used from the string. If the precision specification is given as an asterisk '*', the routine uses the next *arg* as an integer that gives the precision.

4. The letter 'l' before any integer conversion (**d**, **o**, **x**, or **u**) indicates that the argument is a **long** rather than an **int**. Capitalizing the conversion type has the same effect; note, however, that capitalized conversion types are *not* compatible with all C compiler libraries, or with the draft ANSI standard.

The following format conversions are recognized:

% Output a '%' character. No arguments are processed.

c Convert the **int** argument to a character.

d Convert the **int** argument to signed decimal.

D Convert the **long** argument to signed decimal.

e Convert the **float** or **double** argument to exponential form. The format is: *d.ddddddesdd*, where there is always one digit before the decimal point and as many as the *precision* after it (the default is six). The exponent sign *s* may be either '+' or '-'.

f Convert the **float** or **double** argument to a string with an optional leading minus sign '-', at least one decimal digit, a decimal point ('.'), and optional decimal digits after the decimal point. The number of digits after the decimal point is the *precision* (default, six).

g Convert the **float** or **double** argument to whichever of the formats **d**, **e**, or **f** loses no significant precision and takes the least space.

o Convert the **int** argument to unsigned octal.

O Convert the **long** argument to unsigned octal.

r The next argument points to an array of new arguments that may be used recursively. The first argument of the list is a **char *** that contains a new format string. When the list is exhausted, the routine continues from where it left off in the original format string.

s Print the string to which the **char *** argument points. Reaching either the end of the string, indicated by a NUL character, or the specified *precision*, will terminate output. If no *precision* is given, only the end of the string will terminate.

u Convert the **int** argument to unsigned decimal.

U Convert the **long** argument to unsigned decimal.

x Convert the **int** argument to unsigned hexadecimal.

X Convert the **long** argument to unsigned hexadecimal.

Example

The following example demonstrates many **printf** statements.

```
main()
{
        extern void demo_r();
        int precision = 1;
        int integer = 10;
        float decimal = 2.75;
        double bigdec = 27590.21;
        char letter = 'K';
        char buffer[20];

        strcpy (buffer, "This is a string.\n");

        printf("This is an int:   %d\n", integer);
        printf("This is a float:  %f\n", decimal);
        printf("Another float:    %3.*f\n", precision, decimal);
        printf("This is a double: %lf\n", bigdec);
        printf("This is a char:   %c\n", letter);
        printf("%s", buffer);
        printf("%s\n", "This is also a string.");

        demo_r("Print everything: %d %f %lf %c",
                integer, decimal, bigdec, letter);
        exit(0);
}

void demo_r(string)
char *string;
{
        printf("%r\n", (char **)&string);
}
```

The following example uses **printf** to print the location of the mouse pointer on the screen. The code \033H tells **printf** to output an **<esc>** character and the letter 'H', which tells TOS to home the cursor.


```
#include <gemdefs.h>
#include <aesbind.h>

#define CLICKS 1                        /* no. of clicks expected on mouse button */
#define BUTTON 1                        /* which button; 1 = leftmost */
#define DOWN 1                          /* i.e., the mouse button is down */

/* throw-away declarations, to keep system from scribbling over itself */
int nowhere = 0;
Rect norect = { 0, 0, 0, 0 };

main() {
/* declarations used by evnt_multi() */
        int selection;                  /* code for event that occurred */
        unsigned int which = (MU_KEYBD | MU_BUTTON);
        int buffer[11];                 /* place to write AES messages */
        int mousex;                     /* mouse X coordinate */
        int mousey;                     /* mouse Y coordinate */

 /* OK, here we go ... */
        appl_init();
        graf_mouse(ARROW, &nowhere);

        for(;;) {
                selection = evnt_multi(which, CLICKS, BUTTON, DOWN,
                        0, norect, 0, norect, buffer, 0, 0, &mousex, &mousey,
                        &nowhere, &nowhere, &nowhere, &nowhere);

                switch(selection) {
                case MU_KEYBD:
                        appl_exit();
                        exit(0);

                case MU_BUTTON:
                        graf_mouse(M_OFF, &nowhere);
                        printf("\033HX: %03d Y: %03d\n", mousex, mousey);
                        graf_mouse(M_ON, &nowhere);
                        break;

                default:
                        break;
                }
        }
}
```

See Also

fprintf, putc, puts, scanf, screen control, sprintf, write

Notes

Because C does not perform type checking, it is essential that each argument match its specification in the format string.

The use of upper-case format characters to specify long arguments is not standard, and will be phased out to conform with the ANSI standard. Use the 'l' modifier.

At present, **printf** does not return a meaningful value.

prn: — Operating system device

TOS logical device for parallel port

TOS gives names to its logical devices. Mark Williams C uses these names, to allow the STDIO library routines to access these devices via TOS. **prn:** is the logical device for the the parallel port.

Example

```
#include <stdio.h>
main(){
        FILE *fp, *fopen();
        if ((fp = fopen("prn:","w")) != NULL)
                fprintf(fp,"prn: enabled.\n");
        else printf("prn: cannot open.\n");
}
```

See Also

aux:, con:

process — Definition

A process is a program in the state of execution.

See Also

daemon, file

Protobt — xbios function 18 (osbind.h)

Generate a prototype boot sector
#include <osbind.h>
#include <xbios.h>
void Protobt(*buffer, serialno, type, flag*)
char **buffer*; **long** *serialno*; **int** *type, flag*;

Protobt generates a prototype boot sector, and returns nothing. *buffer* points to a 512-byte buffer; this buffer may already contain an image of a boot sector, but whether it does or not is irrelevant. *serialno* is a serial number that will be stamped into the boot sector; setting *serialno* to -1 leaves the boot sector's serial number unchanged, whereas setting it to any number higher than 0x01000000 creates a random serial number that will be stamped into the boot sector. *type* is an integer that encodes the type of disk being worked with, as follows:

0	40 tracks, single sided
1	40 tracks, double sided
2	80 tracks, single sided
3	80 tracks, double sided

Setting *type* to -1 retains the current disk type.

Finally, *flag* indicates whether the boot sector is executable or non-executable: zero indicates executable; one, non-executable; and -1, retain the current type.

Example

For an example of how to use this macro, see the entry for **Flopfmt**.

See Also

TOS, xbios

Prtblk — xbios function 36 (osbind.h)

Print a dump of the screen
#include <osbind.h>
#include <xbios.h>
int Prtblk(*p*) **struct prtblk** **p*;

Prtblk is a macro that uses the TOS function **xbios**. It prints out a block of memory; it returns 0 if the printing was successful, and nonzero if it was not. *p* points to a specialized structure, which is defined in the header file **xbios.h**, as follows:

```
struct prtblk {
        char *pb_blkptr;        /* Address of bit block or text string */
        int pb_offset;          /* Bit offset into block */
        int pb_width;           /* Width of area to dump, in pixels */
        int pb_height;          /* Height of block to dump, or zero for text print */
        int pb_left;            /* Pixels to left of block */
        int pb_right;           /* Pixels to right of block:
                                 * enclosing bitmap is
                                 * pb_right+pb_width+pb_left wide */
        int pb_srcres;          /* Source resolution, a la Getrez() */
        int pb_dstres;          /* Output resolution */
        int *pb_colpal;         /* Color palette, ala Setpallete() [sic] */
        int pb_type;            /* Printer type */
        int pb_port;            /* Printer port */
        int *pb_masks;          /* Halftone dithers */
};
```

Prtblk can also be used to print text strings.

Example

This example demonstrates the functions **Prtblk**, **Setprt**, **Physbase**, **Getrez**, and **Setcolor**.

```
#include <osbind.h>
#include <xbios.h>
struct prtblk pb;

main() {
        int palette[16];
        register int i;
```

```
        /* Determine printer characteristics */
        i = Setprt(-1);
        if (i & PR_DAISY)
              pb.pb_type = PB_DAISY;
        else if (i & PR_MONO)
              pb.pb_type = PB_MONO160;

        else if (i & PR_EPSON)
              pb.pb_type = PB_MONO120;
        else
              pb.pb_type = PB_COLOR160;

        pb.pb_port = (i & PR_SERIAL) ? PB_AUX : PB_PRT;
        pb.pb_dstres = (i & PR_FINAL) ? PB_FINAL : PB_DRAFT;

        /* Print the screen */
        if (pb.pb_type != PB_DAISY) {
              pb.pb_blkptr = Physbase();

              switch (pb.pb_srcres = Getrez()) {
              case 0:       pb.pb_width = 320;
                    pb.pb_height = 200;
                    break;

              case 1:       pb.pb_width = 640;
                    pb.pb_height = 200;
                    break;

              case 2:       pb.pb_width = 640;
                    pb.pb_height = 400;
                    break;
              }

              pb.pb_colpal = &palette[0];
              for (i = 0; i < 16; i += 1)
                    palette[i] = Setcolor(i, -1);

              pokew(0x4EEL,1);            /* Set prtcnt, locks out Scrdmp() */
              if (Prtblk(&pb) != 0)
                                          Cconws("Screen print failed.\r\n");

        } else
              Cconws("Cannot print graphics on daisy wheel printer.\r\n");

        /* Print a text string */
        pb.pb_blkptr = "\r\nThis is a string.\r\n";
        pb.pb_width = strlen(pb.pb_blkptr);
        pb.pb_height = 0;
        pokew(0x4EEL, 1);

        if (Prtblk(&pb) != 0)
              Cconws("Text print failed.\r\n");
        return 0;
}
```


See Also

TOS, xbios, xbios.h

Pterm — gemdos function 76 (osbind.h)

Terminate a process

#include <osbind.h>

void Pterm(*status***) int** *status***;**

Pterm terminates the current process, and returns control to the parent process. *status* can be a status code that can be interpreted by the parent process. **Pterm** returns non-zero in the unlikely event that the process could not be terminated.

Example

This program exits with a non-zero status.

```
#include <osbind.h>

main() {
    Pterm(2);                   /* Exit with return code set to 2 */
}
```

See Also

gemdos, Pexec, Pterm, Ptermres, TOS

Pterm0 — gemdos function 0 (osbind.h)

Terminate an TOS process

#include <osbind.h>

void Pterm0()

Pterm0 terminates a TOS process, and should never return.

Example

For an example of this function, see the entry for **Bconin**.

See Also

gemdos, Pterm, Ptermres, TOS

Ptermres — gemdos function 49 (osbind.h)

Terminate a process but keep it in memory

#include <osbind.h>

void Ptermres(*n, code***) long** *n***; int** *code***;**

Ptermres terminates a process in TOS, but retains *n* bytes of the process in memory. *code* is the exit code for the process being terminated; it is returned to the process that invoked the current process.

Example

For an example of this function, see the entry for **\auto**.

See Also

gemdos, Pexec, Pterm, Pterm0, TOS

Notes

Programs that use this macro may not be portable to future versions of TOS, but they are interesting to work with in the meantime.

pun – Definition

In the context of C, a **pun** occurs when a programmer uses one data form interchangeably with another. Puns are supported by C's willingness to apply implicit conversion rules.

A pun most often occurs unintentionally when the programmer fails to declare a function as returning a pointer; by default, what the function returns is assumed to be an **int**, and is handled as such. No trouble will arise if the program is run on a machine that defines an **int** and a pointer to have the same length (e.g., i8086 SMALL model); however, such code cannot be transported to an environment in which this is not the case (e.g., i8086 LARGE model).

See Also

pointer, portability

Puntaes – xbios function 39 (osbind.h)

Disable AES

#include <osbind.h.h>
#include <xbios.h>
void Puntaes()

Puntaes disables the AES. This function may not do anything when the AES is in ROM.

See Also

TOS, xbios

putc – STDIO macro (stdio.h)

Write character to stream

#include <stdio.h>
int putc(*c*, *fp*) char *c*; FILE **fp*;

putc is a macro that writes a character *c* onto file stream *fp*, and returns that character upon success.

Example

The following example demonstrates **putc**. It opens an ASCII file and prints its contents on the screen. For another example of **putc**, see the entry for **getc**.


```
#include <stdio.h>
main()
{
        FILE *fp;
        int ch;
        int filename[20];
        printf("Enter file name: ");
        gets(filename);

        if ((fp = fopen(filename,"r")) != NULL)
        {
                while ((ch = fgetc(fp)) != EOF)
                        putc(ch, stdout);
        }
        else
                printf("Cannot open %s.\n", filename);
        fclose(fp);
}
```

See Also

fputc, getc, putchar, STDIO
The C Programming Language, pages 152, 166

Diagnostics

EOF is returned when a write error occurs.

Notes

Because **putc** is a macro, arguments with side effects may not work as expected.

putchar – STDIO macro (stdio.h)

Write a character to standard output

#include <stdio.h>
int putchar(*c*) char *c*;

putchar is a macro that expands to **putc(*c*, stdout)**; it writes a character onto the standard output.

Example

For an example of this routine, see the entry for **getchar**.

See Also

fputc, putc, STDIO
The C Programming Language, pages 144, 152

Diagnostics

EOF is returned when a write error occurs.

Notes

Because **putchar** is a macro, arguments with side effects may not work as expected.

puts — STDIO function (libc)

Write string to standard output
#include <stdio.h>
void puts(*string*) char *string

puts appends a newline character to the argument ***string*** and writes the result on the standard output.

Example

The following uses **puts** to write a string on the screen.

```
#include <stdio.h>

main()
{
        puts("This is a string.\n");
}
```

See Also

fputs, STDIO

putw — STDIO macro (stdio.h)

Write word to stream
#include <stdio.h>
int putw(*word*, *fp*) int *word*; FILE *fp;

The macro **putw** writes *word* onto the file stream *fp*. It returns the value written.

putw differs from **putc** in that **putw** writes an **int**, whereas **putc** writes a **char** that is promoted to an **int**.

See Also

ferror, STDIO

Diagnostics

putw returns **EOF** when an error occurs. A call to **ferror** may be needed to distinguish this value from a genuine end-of-file flag.

Notes

Because **putw** is a macro, arguments with side effects may not work as expected. The bytes of *word* are written in the natural byte order of the machine.

pwd — Command

Print the name of the current directory
pwd

pwd prints the name of the current working directory.

See Also

cd, commands, msh

Q

qsort — General function (libc)

Sort arrays in memory
void qsort(*data, n, size, comp***) char ****data***; int** *n*, *size*; **int (****comp***)();**

qsort is a generalized algorithm for sorting arrays of data in primary memory. It uses C. A. R. Hoare's "quicksort" algorithm. **qsort** works with a sequential array of memory called *data*, which is divided into *n* parts of *size* bytes each. In practice, *data* is usually an array of pointers or structures, and *size* is the **sizeof** the pointer or structure. Each routine compares pairs of items and exchanges them as required. The user-supplied routine to which *comp* points performs the comparison. It is called repeatedly, as follows:

```
(*comp)(p1, p2)
char *p1, *p2;
```

Here, *p1* and *p2* each point to a block of *size* bytes in the *data* array. In practice, they are usually pointers to pointers or pointers to structures. The comparison routine must return a negative, zero, or positive result, depending on whether *p1* is logically less than, equal to, or greater than *p2*, respectively.

Example

For an example of this function, see the entry for **malloc**.

See Also

shellsort, strcmp, strncmp
The Art of Computer Programming, vol. 3

Notes

qsort differs from the other sorting function, **shellsort**, in that it uses a recursive algorithm that makes heavy use of the stack.

R

rand — General function (libc)

Generate pseudo-random numbers
int rand()

rand generates a set of pseudo-random numbers. It returns integers in the range 0 to 32,767, and purportedly has a period of 2^32. **rand** will always return the same series of random numbers unless you change its *seed*, or beginning-point, with **srand**.

Example

This example demonstrates the functions **rand** and **srand**. It uses a threshold level that is passed in **argv[1]** (default, MAXVAL/2), the number of trials passed in **argv[2]** (default, 1,000), and a seed passed in **argv[3]** (default, no seeding).

```
#define MAXVAL 32767              /* range of rand: [0,2^15-1] */
main(argc, argv)
int argc; char *argv[];
{
        register int i, hits, threshold, ntrials;

        hits = 0;
        threshold = (argc > 1) ? atoi(argv[1]) : MAXVAL/2;
        ntrials = (argc > 2) ? atoi(argv[2]) : 1000;
        if (argc > 3)
                srand(atoi(argv[3]));

        for (i = 1; i <= ntrials; i++)
                if (rand() > threshold)
                        ++hits;

        printf("%d values above %d in %d trials (%0%%).\n",
                hits, threshold, ntrials, (100L*hits)/ntrials);
}
```

See Also

srand
The Art of Computer Programming, vol. 2

Random — xbios function 17 (osbind.h)

Generate a 24-bit pseudo-random number
#include <osbind.h>
#include <xbios.h>
long Random()

Random generates and returns a 24-bit pseudo-random number. The generator is seeded from the frame-counter, and is likely to be different every time the computer is turned on.

Example

The following example generates an array of random numbers. You may wish to use this as input for the example in **malloc**, which demonstrates sorting.

```
#include <osbind.h>
main() {
        int i;
        for (i=100;i>0;i--) {
                printf("%8ld ", Random());
                if ( i%4 == 0 )
                        printf( "\n" );
        }
}
```

See Also

TOS, xbios

The Art of Computer Programming, vol. 2

Notes

The lowest bit has a distribution of exactly 50%.

random access — Definition

In the context of computing, **random access** means that an entity, such as memory, can be accessed at any point, not just at the beginning. This means that all points within memory can be accessed equally quickly. This contrasts with *sequential access*, in which entities must be accessed in a particular order, so that some entities take longer to access than do others.

A tape drive is an example of a sequential access device, i.e., the order in which data are read is dictated by the order in which they stream past the tape head. Random-access memory (RAM) is an example of random access. Hard disks and floppy disks combine elements of random access and sequential access.

RAM, which usually consists of semiconductor integrated circuits, is also strictly random access. In this regard, the term "RAM" is slightly misleading; a more accurate name would be "read/write memory", to contrast RAM with read-only memory (ROM), which is also *random access* memory.

See Also

read-only memory

ranlib — Definition

The **ranlib** is a "directory" that appears at the beginning of each library. It contains the name of each global symbol (i.e., function name) that appears within the library, and a pointer to the module in which that symbol is defined. Thus, the ranlib eliminates the need for the linker to search the entire library sequentially to find a given global symbol, which speeds up linking noticeably.

If the date on the library file is later than that in the ranlib header, the linker will ignore the ranlib and perform a sequential search through the library; the linker will also send the warning message

```
Outdated ranlib
```

to the standard error device. This is done to prevent the accidental use of an outdated ranlib, which could be disastrous. When you use the archiver **ar** to update a library or to create a new library, be sure to employ the options that update the ranlib as well as modify or create the library.

See Also

ar, date, ld, touch

Notes

Under certain circumstances, it was possible to generate the **Outdated ranlib** error message even though the ranlib was in fact up to date. In previous releases of Mark Williams C, this occurred when it was installed on a system with the date set to the current date, rather than not set, as requested in the installation procedures. Installing Mark Williams C with the date set on the system had the effect of updating the date stamp on the library files, which put the date on the ranlib header and that of its library file out of synch. The linker thus thought that the ranlib was outdated, when it was in fact correct. This problem was fixed on a previous release.

rational number — Definition

A **rational number** is the quotient of two integers.

See Also

integer, real numbers

rc_copy — AES function (libaes)

Copy a rectangle

#include <aesbind.h>

int rc_copy(*oldrect, newrect*) **int** *oldrect*[4], *newrect*[4];

rc_copy is an AES routine that copies a rectangle from one part of the screen to another. *oldrect* and *newrect* hold, respectively, the rectangle being copied and the area to which it is being copied. Each array holds the following information:

rectangle[0]	X point (upper left corner)
rectangle[1]	Y point (upper left corner)
rectangle[2]	width
rectangle[3]	height

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

See Also
AES, TOS

Notes
A clipping rectangle should be set using the VDI function **vs_clip** before this routine is used. If you do not, you may inadvertently copy a rectangle over an element in low memory, such as a RAM disk.

rc_equal – AES function (libaes)

Compare two rectangles
#include <aesbind.h>
int rc_equal(*rect1, rect2*) **int** *rect1*[4], *rect2*[4];

rc_equal is an AES routine that compares two rectangles. *rect1* and *rect2* hold the two rectangles being compared. Each array holds the following information:

rectangle[0]	X point (upper left corner)
rectangle[1]	Y point (upper left corner)
rectangle[2]	width
rectangle[3]	height

rc_equal returns zero if the rectangles are not identical, and one if they are.

See Also
AES, TOS

rc_intersect – AES function (libaes)

Check if two rectangles intersect
#include <aesbind.h>
int rc_intersect(*rect1, rect2*) **int** *rect1*[4], *rect2*[4];

rc_intersect is an AES routine that check to see if two rectangles intersect. *rect1* and *rect2* point to the two rectangles being compared. Each array holds the following information:

rectangle[0]	X point (upper left corner)
rectangle[1]	Y point (upper left corner)
rectangle[2]	width
rectangle[3]	height

The values within the array *rect2* will be changed to the coordinates of the area common to both rectangles, or to meaningless values if they do not intersect. **rc_intersect** returns zero if the rectangles do not intersect, and one if they do.

See Also
AES, TOS

rc_union – AES function (libaes)

Calculate overlap between two rectangles
#include <aesbind.h>
void rc_union(*rect1, rect2*) **int** *rect1*[4], *rect2*[4];

rc_union is an AES routine that computes a rectangle that encloses two overlapping rectangles. *rect1* and *rect2* point to the two overlapping rectangles. Each array holds the following information:

rectangle[0]	X point (upper left corner)
rectangle[1]	Y point (upper left corner)
rectangle[2]	width
rectangle[3]	height

The values within the array *rect2* will be changed to the coordinates of the rectangle that encloses the overlapping rectangles. These variables are set to meaningless values if the rectangles do not intersect. **rc_union** returns nothing.

See Also
AES, TOS

Notes
This routine should be used only if you are certain that the rectangles in question do overlap. The routine **rc_intersect** returns a value that indicates if the rectangles do in fact overlap.

rdy – Command

Create, save, and load rebootable RAM disk
gem rdy
rdy [CMD=*command* **FILE=***filename* **...]**

rdy is the Mark Williams C utility that creates, saves, and loads a RAM disk. The RAM disk that **rdy** makes has, among others, the following properties:

- The RAM disk will survive system resets. Pressing the reset button on the back of the computer will not cause the RAM disk to be erased.
- The RAM disk can be made the system's boot disk. This allows you to reboot your system and load all desk accessories from the RAM disk.
- The RAM disk can be set to substitute for any physical drive from C through P, and to any size that fits into the memory available on your machine.
- The RAM disk can be copied, contents and all, into an executable file; this file can then be loaded directly into memory. This simplifies the task of recreating the RAM disk after your computer has been turned off.

rdy is designed to work either under **msh** by using a command-line interface, or from the GEM desktop by using a graphics interface.

All source code for **rdy** including its resource files and header file, is included in the archive **rdy.a**. See the entry for the archiver **ar** for information on how to extract the contents of **rdy.a** for alteration and compilation.

How rdy *works*

rdy goes through the following steps when it builds a RAM disk.

1. It writes a prototype RAM disk and copies it into a file. The size of the RAM disk, its device name (e.g,. "C"), whether it should be the boot device, and the name of the file into which the prototype should be copied, are all set either according to variables supplied by the user or, if no such variables are set, according to built-in prototypes.
2. It loads the RAM disk into memory. After it does so, the system automatically warm boots. **rdy** automatically updates various system tables, so that TOS knows that the RAM disk is present, but it is up to the user to install the icon for the RAM disk on the GEM desktop.
3. **rdy** can then be told to back up the installed RAM disk, plus whatever files you have copied into it, into an executable file.
4. If you wish, **rdy** will remove one of its RAM disks. Note that the only certain way to remove a RAM disk is either through **rdy** itself or by cycling power on your computer.
5. Finally, **rdy** can read an installed RAM disk, a back-up file, or a prototype file, and print its parameters on the screen.

To operate **rdy**, you must tell it the *command* that you want it to execute (e.g., create a prototype file or load a RAM disk into memory), and then supply the necessary variables the command needs (e.g., if you are getting information about a prototype file, the name of the file you want **rdy** to read).

As noted above, you can pass this information to **rdy** either through a command-line interface under the shell **msh**, or through a graphics interface directly from the GEM desktop. **rdy** reads the environment and looks for the environmental variable **CMD**; if it is not set, or if it set to NULL (which is always the case when running from the GEM desktop), **rdy** reads its associated resource file and runs through the graphics interface; otherwise, it ignores its graphics routines and operates through an ordinary command-line interface. Each interface is described in detail below.

Using the command-line interface

As noted above, **rdy** checks its environment for the environmental variable **CMD**. If **CMD** is found, **rdy** invokes the command-line interface; otherwise, it invokes the graphics interface.

The set of environmental variables that **rdy** uses is as follows:

BOOT	boot flag for the RAM disk
CMD	choose one of: **DROP** - remove a RAM disk from memory **HELP** - give information on **rdy** **LIST** - list RAM disks active in memory or saved in file **LOAD** - load a RAM disk into memory from a file **MAKE** - create a new RAM disk file **SAVE** - save a RAM disk from memory into a file
DISK	drive identifier of the RAM disk
FILE	file name for RAM-disk prototype or backup
ROOT	size of the RAM disk root directory, in 512-byte sectors
SIZE	size of the RAM disk data area, in kilobytes

CMD and the envionmental variables that **rdy** uses can be set either by using the **setenv** command to implant them into the **msh** environment, or by setting them on the **rdy** command line itself. As far as **rdy** is concerned, typing

```
setenv CMD=LOAD
setenv FILE="a:\bin\ramdisk.rdy"
rdy
```

is equivalent to typing

```
rdy CMD=LOAD FILE="a:\bin\ramdisk.rdy"
```

The only difference is the command **setenv** fixes the variables **CMD** and **FILE** within the environment, where they can be read by **rdy** and other programs, whereas passing them on the command line means that they disappear when **rdy** has finished its work.

To build a new RAM disk on your machine, use the following script:

1. Decide how large a RAM disk you want. To perform compiles, a RAM-disk must be at least 100 kilobytes. A 512-kilobyte machine can support a 100- to 200-kilobyte RAM disk, where a 1,024-kilobyte machine can support a RAM disk of up to 512 kilobytes. The rest of this example will demonstrate building a 100-kilobyte RAM disk.
2. Enter the microshell **msh**. Now, type the command:

   ```
   rdy CMD=MAKE DISK=C SIZE=100 FILE="a:\ramdisk.rdy
   ```

   Change the **SIZE** and **DISK** parameters to suit your preferences. **rdy** will create a description of the RAM disk, and write it into the file **ramdisk.rdy**, and then return you to **msh**.
3. Reinvoke **rdy**, as follows:

   ```
   rdy CMD=LOAD FILE=rdydisk.ram
   ```

   rdy will now load the prototype RAM disk into memory. Note that during the loading process, your system will warm boot and return you to the GEM

desktop.

4. Use the desktop's **install** utility to install the RAM disk on the desktop. You may wish to rename the icon, and otherwise modify the desktop. When you have done so, save the desktop settings.

5. Re-enter **msh**. Now, use **cd** to move to the RAM disk. Configure the disk as you wish: create directories and copy files into it that you use often. On a small RAM disk, the MicroEMACS editor performs very well out of the RAM disk; on a large RAM disk, you may wish to move the entire compiler and linker onto it.

6. Now, place a newly formatted disk into drive A, and invoke **rdy** as follows:

```
rdy CMD=SAVE FILE="a:\ramdisk.dta"
```

   There is nothing magical about the file name in the above example; you may call the file whatever you wish. **rdy** will copy an image of the entire RAM disk into the file **ramdisk.dta** on drive A. The next time you need to cold boot the system, you can use **rdy** to copy the contents of this file back into the RAM disk; this should save you considerable amounts of time.

Your RAM disk is now ready.

Using the graphics interface

As noted above, if **rdy** does not find the parameter **CMD** defined either in its environment or on its command line, it will automatically invoke its graphics interface. The graphics interface is easier to use than the command-line interface, especially by users unfamiliar with **rdy**, but offers a narrower range of options.

To see how the graphics interface works, type **exit** to leave **msh**, and then click the icon labelled **rdy.prg**. The screen will clear, and in a moment, a new menu bar will appear at the top of the screen. The title at the left of the menu bar, called **Desk**, gives you access to all desk accessories. The title at the right, **Read Me**, describes how **rdy** works. You can use this feature to refresh your memory. The title in the center, **Options**, lets you command **rdy** to perform one or more tasks for you.

If you pass the mouse pointer over the **Options** title, a menu will drop down. This menu has six entries, as follows:

Create a RAM disk
: Write a prototype RAM disk into a file named by the user (default, **a:\ramdisk.rdy**).

Load a RAM disk
: This loads a prototype or backup RAM disk into memory. Note that the system will warm boot automatically as soon as the disk is installed.

Back up a RAM disk
: Copy a RAM disk and its contents, into a file. This file can later be loaded back into memory.

Remove a RAM disk
: Erase a RAM disk from memory. Note that the only sure way to remove a RAM disk is either with this command, or by cycling power on your computer. Note that the system will warm boot automatically as soon as the disk is removed.

Get data on a RAM disk
: Read a RAM disk, a prototype file, or a backup file, then print information about it on the standard output device.

Quit
: Exit from **rdy**.

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

The first dialogue box asks you how much RAM your system has, either 512 kilobytes or 1024 kilobytes. Click the appropriate button.

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

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

rdy asks the name of the file in which to store the prototype RAM disk. Then, it asks you if you want this RAM disk to be your system's boot disk. When you have answered these questions, **rdy** displays the configuration of the new RAM disk, and ask you if it is correct. If you answer "No', you will return to the **rdy** desktop; otherwise, the new prototype RAM disk will be written.

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

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

desktop will return with the new icon displayed.

Finally, you should create a new directory (or "folder", in Atari jargon) named **tmp**. Do so by clicking the "New Folder" entry on the Desktop's **File** menu, and following its directions.

Working with a RAM disk

A RAM disk improves the speed with which you work by reducing the amount of time the compiler needs to read a file. Even a small RAM disk will speed your work greatly if it is used properly. For example, the compiler writes a temporary files to pass information between its four phases; writing the temporary files onto the RAM disk eliminates the time taken by writing these files onto a disk and reading them back again. Test compiles have shown that this change alone will reduce the time needed to compile and link a large program by more than half.

To take full advantage of your RAM disk, you will need to tell the Mark Williams microshell **msh** that it exists and how you want it to be used. To do so, you must edit the file **profile**, which **msh** reads when you invoke it; make the following two changes. First, the line that begins **TMPDIR=** indicates where you wish to store temporary files; this line should be changed to the name of your RAM disk. For example, if your RAM disk is named **D**, this line should read as follows:

```
TMPDIR=D:\
```

Then, the line that begins **PATH=** lists for **msh** all the directories where it should look for executable files. Your RAM disk should go near the beginning of that list. For example, this line may read as follows:

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

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

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

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

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

The bootable RAM disk

As noted above, **rdy** can create a RAM disk that is defined to the system as its boot disk. TOS will look for its boot file in directory **\auto** on that device, and look for its desk accessories in its root directory. The following describes how to use **rdy** in order to take advantage of this and other more advanced features.

1. Create a RAM disk using the steps listed above. Load it into memory and create its icon. Then, double-click the RAM disk's icon to open it. Create two new folders for it, called **tmp** and **auto**. If you are running a hard disk, put the hard disk driver into the **auto** folder; then press the reset button to warm boot. This allows you to access the hard disk throughout the rest of this routine. Then double-click the RAM disk's icon to reopen it.

2. Now, drag into the RAM disk the programs and utilities you want to store there. Some programmers prefer to keep **msh** and MicroEMACS (or another preferred editor) in a folder called **bin** on the RAM disk, because they are used constantly. Be sure to leave enough room on the RAM disk to hold the compiler's temporary files.

3. Save the desktop and copy the file **desktop.inf** onto the RAM disk if it was not written there. Note that if you have a hard disk with drive C, TOS will insist on writing **desktop.inf** there.

4. The next step is to test the RAM disk by warm booting. Press the reset button. The floppy disk drive should run for a second as the system looks for a boot block; then, the programs in the RAM disk's **auto** folder will run. Finally, the desktop you saved should appear as the desktop initializes.

   If anything is missing or wrong, go back, fix it, and test again until everything is right.

5. The next step is to back up your configured RAM disk. Put a blank floppy disk into drive A. Format it with the volume name **coldstrt.dsk**, copy the files **rdy.prg** and **rdy.rsc** onto it, and make an **auto** folder on it.

   Then click **rdy.prg**, select **Back up a RAM disk**, and follow its directions to save the contents of the loaded RAM disk into file **a:\auto\coldstrt.prg**. Note that a backed-up RAM disk is an executable file in its own right; you do not need to invoke **rdy** to load it into memory.

6. Now that the back up is finished, you have a disk from which you can cold start your system easily. To test whether all will work correctly, turn off your computer for a few moments, then turn it on again. Drive A should be selected for several seconds while TOS loads the program **coldstrt.prg** into memory. The screen will flash as the RAM disk warm boots to install itself. Then, the programs installed in the **auto** folder of the RAM disk will run and take effect. If you installed a hard disk driver, you should see the hard disk initialize. Finally, the desktop that you saved on the RAM disk will be displayed as the desktop initializes.

   The screen should not flash after the initial reset, but it often does. This could be due to any number of reasons. The most easily fixed is a loose cable: make sure that the cables that plug into the back of your ST are pressed all the way into their sockets.

You now have a RAM disk that you can use either to warm boot or cold boot your system. When ever you need to cold boot your system, simply place the boot disk you just created into floppy disk drive A, turn on the computer, let it boot, then put the boot disk away.

When an error occurs, you have reason to believe that TOS or the AES has corrupted itself, or a program enters an infinite loop, just push the reset button. The machine will reboot off the RAM disk in a few seconds, and all the contents of the RAM disk will be exactly as you left them.

Problems can occur from a number of sources. If a program had a file open on the RAM disk when you reset the system, the file may not have been completely written to the RAM disk. Removing the file name may not recover all the clusters allocated to the file. These lost clusters may become a problem if you continue to reset out of the program, because the RAM disk will eventually run out of data clusters for files. This can be fixed by saving your work to a floppy or hard disk, removing the RAM disk, and reloading it from your cold boot disk.

The RAM disk occupies high memory, beyond the value of the system variable **phystop**, which is normally 32 kilobytes past the start of the video display. If a program writes into this memory, it will destroy your RAM disk and all its contents will be lost. This is most easily done by not clipping graphics to the screen correctly: then, if you scribble past the border of the screen, the virtual image you create will overwrite the RAM disk.

You probably will want to update your cold boot RAM disk from time to time. If you are replacing an old version of a file with a new version, you can simply copy the new version in and replace the old back-up RAM disk file. However, if you are changing the structure or contents of the RAM disk, then do it in the following order.

First, remove the old files from your loaded RAM disk. Remove the old folders from your loaded ramdisk. Create the new folders on your loaded RAM disk; then copy the new files onto your loaded RAM disk.

Then, place **coldstrt.dsk** into the floppy drive, run **rdy.prg**, and save the RAM disk image to **a:\auto\coldstrt.prg**. If you change the contents of your RAM disk in another order, you may get a much larger RAM disk image than necessary.

A saved RAM disk contains all the data clusters up to the last one allocated. You can eliminate fragmentation of your RAM disk data clusters by simply copying all the files and folders from your RAM disk onto a floppy disk, deleting all the files and folders from your RAM disk, and copying the files and folders back onto the RAM disk from the floppy disk.

You can also use **rdy.prg** from a shell command file; for example:

```
setenv CMD=SAVE FILE=a:\auto\coldstrt.prg DISK=c; rdy
```


You may wish to store this command in a file on your cold boot disk. Be sure to set the variable **DISK** to the correct drive identifier for your RAM disk!

Finally, enjoy the speed and convenience of your new RAM disk. You may wish to spend some of this time studying the sources for **rdy**; they are stored in the archive file **rdy.a**. See the entry for the archiver **ar** for information on how to extract files from the archive.

See Also

commands, TOS

Notes

The Supra hard disk autoboot can sometimes interfere with RAM disks built with **rdy**. To use RAM disks with the Supra hard disk autoboot, do the following:

1. Create a new RAM disk of the desired size and drive; this must not conflict with a hard-drive identifier. Make it non-bootable. Give this file a name like **ram***XXXD***.prg**, where *XXX* is the size of the RAM disk, in kilobytes, and *D* is the letter of the RAM disk's identifier.
2. If you already have an **\auto** fold on partition **C**, move all of its files into another folder, and delete the **\auto** folder.
3. Create a new **\auto** folder.
4. Put the RAM disk into the new **\auto** folder *first*.
5. Move all of the files that were moved out of the **\auto** folder into the new **\auto** folder.

When the system boots for the first time after a power-up, the RAM disk will load and then cause the system to warm-boot. On warm-boot, the RAM disk loader portion of the RAM disk file sees that a drive is already loaded with the given drive specifier and exits without loading the RAM disk again and without rebooting the system. The rest of the programs in the **\auto** folder are then executed.

rdy.a — Archive

rdy.a is an archive that holds the source files for **rdy**, the Mark Williams utility that creates rebootable RAM disks on the Atari ST.

If you wish to recompile **rdy**, you must first extract the source files from the archive. Use the command **cd** to move to the directory where you have stored this archive, then give **msh** the following command:

```
ar xv rdy.a
```

See Also

ar, rdy

read — UNIX system call (libc)

Read from a file
int read(*fd*, *buffer*, *n*) int *fd*; char **buffer*; int *n*;

read reads up to *n* bytes of data from the file descriptor *fd* and writes them into *buffer*. The amount of data actually read may be less than that requested if **read** detects EOF. The data are read beginning at the current seek position in the file, which was set by the most recently executed **read** or **lseek** routine. **read** advances the seek pointer by the number of characters read.

Example

For an example of how to use this function, see the entry for **open**.

See Also

UNIX routines, STDIO

Diagnostics

With a successful call, **read** returns the number of bytes read; thus, zero bytes signals the end of the file. It returns -1 if an error occurs, such as bad file descriptor, bad *buffer* address, or physical read error.

Notes

read is a low-level call that passes data directly to TOS. It should not be intermixed with high-level calls, such as **fread**, **fwrite**, or **fopen**.

readonly — C keyword

Storage class

readonly is a C keyword that modifies data declarations. As its name implies, the **readonly** modifier declares that data are to be read only; this helps protect key data against casual modification by the user or another programmer.

See Also

C keywords, C language, keyword

Notes

The draft ANSI standard for the C language eliminates this keyword.

read-only memory — Definition

As its name suggests, **read-only memory**, or ROM, is memory that can be read but not overwritten. It most often is used to store material that is used frequently or in key situations, such as a language interpreter or a boot routine.

See Also

random access

realloc — General function (libc)

Reallocate dynamic memory
char *realloc(*ptr*, *size*) char **ptr*; unsigned *size*;

realloc helps you manage a program's arena. It returns a block of *size* bytes that holds the contents of the old block, up to the smaller of the old and new sizes. **realloc** tries to return the same block, truncated or extended; if *size* is smaller than the size of the old block, **realloc** will return the same *ptr*.

Example

For an example of this function, see the entry for **calloc**.

See Also

arena, calloc, free, lcalloc, lmalloc, lrealloc, malloc, notmem, setbuf

Diagnostics

realloc returns **NULL** if insufficient memory is available. It prints a message and calls **abort** if it discovers that the arena has been corrupted, which most often occurs by storing past the bounds of an allocated block. **realloc** will behave unpredictably if handed an incorrect *ptr*.

The related function **lrealloc** takes an unsigned long as its *size* argument, and therefore can reallocate memory blocks that are larger than 64 kilobytes.

real number — Definition

A **real number** is any number of the set of rational numbers or irrational numbers.

See Also

float, rational number, integer, irrational number

record — Definition

A **record** is a set of data of a fixed length that has been given a unique identifier, and whose structure conforms to an exact description. An example of a record is an entry in a file of names and addresses: each entry has a fixed length, is marked by a unique identifier, and has a fixed number of bytes set aside in fixed order to record name, address, city, state, and ZIP code.

Note, too, that what is called a "record" in Pascal is called a "structure" in C.

See Also

field, structure

register — C keyword

Storage class

register is a C keyword that declares a class of data storage. A variable so declared will be stored in a register, which may increase the speed with which it is read by a program.

See also

auto, C keywords, C language, extern, register variable, static

register – Definition

A **register** is special high-speed memory within a microprocessor that can be addressed concisely and within which data can be stored and modified. The size and the configuration of a microprocessor's registers affect its computing potential. Registers can be manipulated much faster than RAM.

The routines in the Mark Williams C libraries generally assume that they have been called from C programs; thus, they may freely overwrite any registers that the compiler overwrites in its generated code.

See Also

register variable

register variable – Definition

register is a C storage class. A **register** declaration tells the compiler to try to keep the defined local data item in a machine register. Under Mark Williams C, the **int foo** can be declared to be a register variable with the following statement:

```
register int foo;
```

On the i8086, two registers are available to accept register variables; if more than two are declared, all after the first two will be treated as ordinary **autos**. On the 68000, eight registers are available to accept register variables: three address registers and five data registers.

By definition of the C language, registers have no addresses, so pointers to registers cannot be passed as function arguments. Placing heavily-used local variables into registers often improves performance, but in some cases declaring **register** variables can degrade performance somewhat.

See Also

auto, extern, static, storage class
The C Programming Language, page 81

rescomp – Command

Resource compiler
rescomp [-v] [-o *outfile*] *infile*[.ext]

The command **rescomp** compiles *infile*, which must be a file of resource-description text into a GEM resource.

The option -v tells **rescomp** to report statistics as it compiles.

The option -o changes the name of the three files it produces to *outfile*. By default, **rescomp** names its output files after *infile*. When the compiler creates these files, it gives them the extensions **.rsc** for the resource file, **.rsd** for the compiled resource description, and **.h** for the C header file.

See the section in the introduction on the **Resource compiler and decompiler** for a summary of the resource description language.

See Also

resdecom, resource

resdecom – Command

Resource decompiler
resdecom [-o *outfile*[.ext]] [-d *defile*[.ext]] [-mv] [-] *infile*[.ext]

The command **resdecom** decompiles a GEM resource into a file of resource-description language. This can be useful when you want to change a line of text within a resource, globally change a string which appears in more than one place within a resource, or if you want to create a simple resource without using the Resource Editor. It is easy to track changes between versions of your resource by comparing decompiled resource files.

Decompiled resources often take much more room than their compiled counterparts. The text descriptions of some objects are more compact, but images, icons, buttons and compound objects take up more space.

To decompile an existing resource set into a resource description file, use the program **resdecom.prg**. Its options are as follows:

-d *defile*[.ext]
: Specify the name of the definition file.

-o *outfile*[.ext]
: Rename the output file that the decompiler creates. The default is the name of the resource you are decompiling, with the extension **.rdl**.

-m Force menus to be treated as forms.

-v Verbose option: decompile with messages.

- Send output to the standard output.

resdecom looks for two files that are labeled with the extensions **.rsc** and **.rsd**. It reads them and writes a resource description file that takes the names of the resource files, adding the extension **.rdl**.

The file produced by **resdecom** can be edited with MicroEMACS or most other text editors. You can then recompile it into a resource by using the resource com-

piler **rescomp**.

See Also

rescomp, resource

resource — Command

Invoke the resource editor

resource

The command **resource** invokes the Mark Williams Resource Editor. A resource editor simplifies the creation of icons, menus, dialogue boxes, forms, and alerts. In general, it helps you to design and implement graphics interfaces for your programs.

resource encodes objects that you display and manipulate on the editor's desktop. With **resource**, you can move objects around the screen, and edit each until it is as you want it to appear with your GEM application program. It then fills in the X, Y, height, and width coordinates, and records the relative position of each object within its object tree. **resource** also allows you to name each object. It then produces a header file that contains the names and their "handles," so you can refer to each object easily from within your program.

In addition to the C header file, **resource** produces two other files that contain information that your application program will use to reproduce the interface you have created. One file, with the suffix **.rsc**, is the resource file called by your application program. The other is a "name and type" definition file with the suffix **.rsd**. This definition file is used only by **resource** and by the resource decompiler **resdecom**. It is not used by the application program.

To use the editor, you must have an Atari ST system with TOS in ROM, at least one disk drive, a monochrome or color monitor in medium or high resolution, and **Mark Williams C for the Atari ST**.

The **Mark Williams Resource Editor** is designed to work in medium or high resolution. Many of the dialogues in the Resource Editor contain large amounts of information and will not work correctly in low resolution.

The editor also has the following limitations:

- The structure of a resource file limits it to 64 kilobytes.
- A text string cannot be longer than 65 bytes.
- The colors in an object are limited to white, black, red, and green, and the thickness of its border to four rasters (inside and out).

To run **resource**, you must install the files **resource.prg** and **resource.rsc** into the same directory; the directory should be one of those named in the environmental variable **PATH**.

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

```
resource
```

at the prompt. If you want to invoke the Resource Editor from the GEM desktop, double-click the mouse on **resource.prg**.

See Also

rescom, resdecom, object, menu

return — C keyword

Return a value and control to calling function

return is a C statement that returns a value from a function to the function that called it. **return** can be used without a value, to return control of the program to the calling function; also, the calling function is free to ignore the value **return** hands it. Note that it is good programming practice to declare functions that return nothing to be of type **void**.

Note that a function can return only one value to the function that called it. Most often, this value is used to signal whether the function performed successfully or not.

See Also

C keywords, C language
The C Programming Language, page 68

rewind — STDIO function (libc)

Reset file pointer

#include <stdio.h>
int rewind(*fp*) FILE *fp;

rewind resets the file pointer to the beginning of stream *fp*. It is a synonym for **fseek(*fp*, 0L, 0)**.

Example

For an example of this routine, see the entry for **fscanf**.

See Also

fseek, STDIO

Diagnostics

rewind returns EOF if an error occurs; otherwise, it returns zero.

rindex — String function (libc)

Find a character in a string

char *rindex(*string*, *c*) char *string; char *c*;

rindex scans *string* for the last occurrence of character *c*. If *c* is found, **rindex** returns a pointer to it. If it is not found, **rindex** returns NULL.

Example

This example uses **rindex** to help strip a sample file name of the path information.

```
#include <stdio.h>
#define PATHSEP '\\'            /* path name separator */
extern char *rindex();
extern char *basename();

main()
{
        char *testpath = "A:\\foo\\bar\\baz";

        printf("Before messaging: %s\n", testpath);
        printf("After messaging: %s\n", basename(testpath));
}

char *basename(path)
char *path;
{
        char *cp;
        return (((cp = rindex(path, PATHSEP)) == NULL)
                ? path : ++cp);
}
```

See Also

index, memchr, string, strrchr

Notes

This function is identical to the function **strrchr**, which is described in the ANSI standard. Mark Williams C includes **strrchr** in its libraries. It is recommended that it be used instead of **rindex** so that programs more closely approach strict conformity with the ANSI standard.

rm – Command

Remove files

rm *file ...*

rm removes each *file*, and frees data blocks associated with it.

See Also

commands, msh, rmdir

rmdir – Command

Remove directories

rmdir *directory ...*

rmdir removes each *directory*. This will not be allowed if a *directory* is the current working directory or is not empty.

rmdir will not allow you to remove the current working directory.

See Also

commands, mkdir, msh, rm

Rsconf – xbios function 15 (osbind.h)

Configure the serial port

#include <osbind.h>
#include <xbios.h>
long Rsconf(*speed, flow, UCR, RSR, TSR, SCR*)
int *speed, flow, UCR, RSR, TSR, SCR*;

Rsconf configures the serial port. *speed* is an integer that sets the baud, as follows:

0	19,200	**8**	600
1	9600	**9**	300
2	4800	**10**	200
3	3600	**11**	150
4	2400	**12**	134
5	2000	**13**	110
6	1800	**14**	75
7	1200	**15**	50

flow is an integer that sets the flow control, as follows:

0	None (the default)
1	XON/XOFF (**<ctrl-S>/<ctrl-Q>**)
2	Request to send/clear to send (RTS/CTS)
3	XON/XOFF and RTS/CTS

UCR stands for USART control register. (USART, in turn, means universal synchronous-asynchronous receiver-transmitter). This variable is a byte-length bit map that controls the operation of the serial port. Its bits encode the following information:

Bit 0	unused
Bit 1	0 indicates odd parity; 1, even parity
Bit 2	0 indicates no parity; 1, parity as set in bit 1
Bits 3,4	Start/stop bits and format: 00 synchronous; start=0; stop=0 10 asynchronous; start=1; stop=1 01 asynchronous; start=1; stop=1.5 11 asynchronous; start=1; stop=2
Bits 5,6	Word length: 00 8 bits 10 7 bits 01 6 bits 11 5 bits
Bit 7	0=Use frequency from transmit control and receive control directly 1=Divide frequency by 16

RSR is a byte-length bit map that controls the receive status register; setting the bits sets the following conditions:

Bit 0	Enable reception
Bit 1	In synchronous mode, enable comparison of character in SCR with character in receive buffer
Bit 2	In synchronous mode, signal that character identical to character in SCR may be received; in asynchronous mode, signal reception of start bit
Bit 3	In synchronous mode, signal that character identical to character in SCR has been received; in asynchronous mode, signal reception of BREAK
Bit 4	Signal frame error: stop bit is a NUL, but byte received is not
Bit 5	Signal parity error
Bit 6	Signal buffer overrun
Bit 7	Signal buffer full

TSR is a byte-length bit map that controls the transmitter status register. The bits in this map indicate the following:

Bit 1	Enable transmission
Bits 2,3	High or low output mode: 00 High 10 High 01 Low 11 Loop-back mode
Bit 3	In synchronous mode, not used; in asynchronous, sends break condition
Bit 4	Send end-of-transmission character after current character
Bit 5	Switch to reception immediately after end of transmission
Bit 6	Send character in sender floating register before writing new character into send buffer
Bit 7	Buffer empty

Finally, *SCR* initializes the synchronous character register; this variable should be set to zero.

Note that setting *UCR*, *RSR*, *TSR*, or *SCR* to **-1** will cause it to be ignored by TOS.

Rsconf returns a **long** that holds the old UCR, RSR, TSR, and SCR, in that order.

Example

This example sets the serial port to 4800 baud with XON/XOFF flow control. For an example of using this function from the **\auto** directory, see the entry for **\auto**.

```
#include <osbind.h>

#define BR_4800 (2)                    /* 4800 baud */
#define FC_XON (1)                     /* XON/XOFF */

main() {
      Rsconf(BR_4800, FC_XON, -1, -1, -1, -1);
      Cconws("Serial port set to 4800 baud, XON/XOFF\n\r");
}
```

See Also

TOS, xbios

Notes

Resetting the speed, even if there is no change, will transmit an ASCII DEL across the serial line. This may be intended to help remote systems or modems to determine line speed.

rsconf — Command

Configure the serial port

rsconf *speed flow UCR RSR TSR SCR*

rsconf is a command that uses the **xbios** function **Rsconf** to reconfigure the serial port. *speed* is the baud rate to which the port will be set, as follows:

0	19,200	**8**	600
1	9600	**9**	300
2	4800	**10**	200
3	3600	**11**	150
4	2400	**12**	134
5	2000	**13**	110
6	1800	**14**	75
7	1200	**15**	50

flow sets the flow control, as follows:

0	None (the default)
1	XON/XOFF (**<ctrl-S>/<ctrl-Q>**)
2	Request to send/clear to send (RTS/CTS)
3	XON/XOFF and RTS/CTS

UCR, *RSR*, *TSR*, and *SCR* set, respectively, the control register, the receive status, the transmission status, and the synchronous character register. See **Rsconf** for more information on the values to which these arguments can be set. Setting each to -1 will cause them to be ignored by TOS.

See Also

commands, Rsconf, TOS

rsrc_free — AES function (libaes)

Free memory allocated to a set of resources
#include <aesbind.h>
int rsrc_free()

rsrc_free is an AES routine that frees the random-access memory that had been allocated to a set of resources by the routine **rsrc_load**. Because the contents of only one resource file can be kept in memory at any given time, you should use this routine before loading a second resource file. **rsrc_free** returns zero if an error occurred, and a number greater than zero if one did not.

See Also

AES, TOS

rsrc_gaddr — AES function (libaes)

Get the address of a resource object
#include <aesbind.h>
int rsrc_gaddr(*type, index, address***) int** *type, index*; **OBJECT *****address*;

rsrc_gaddr is an AES routine that gets the address of a given resource object. *type* indicates the type of object being sought, as follows:

0	object tree
1	object within a tree
2	text (**TEDINFO**)
3	icon (**ICONBLK**)
4	predefined bit pattern (**BITBLK**)
5	string
6	image data
7	object specification
8	pointer to text (**TEDINFO**)
9	pointer to text template (**TEDINFO**)
10	pointer to text validation string (**TEDINFO**)
11	pointer to mask for icon image (**ICONBLK**)
12	pointer to data for icon image (**ICONBLK**)
13	pointer to icon text (**ICONBLK**)
14	pointer to bit image (**BITBLK**)
15	address of pointer to free string
16	address of pointer to free image

index gives the index number of the object within the resource file. *address* points to the address of the data sought; this value is set by the routine. **rsrc_gaddr** returns zero if an error occurred, and a number greater than zero if one did not.

See Also

AES, TOS

rsrc_load — AES function (libaes)

Load a resource file into memory
#include <aesbind.h>
int rsrc_load(*filename***) char ****filename*;

rsrc_load is an AES routine that loads a resource file into memory. *filename* points to the name of the file to be loaded. Note that by convention, the name of the file must have the suffix **.rsc**.

Note that only one resource file can be loaded into memory at any given time; **rsrc_load** automatically calls **rsrc_free** to free the memory allocated to any previously loaded resource file.

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

See Also

AES, TOS

rsrc_obfix — AES function (libaes)

Change the form of an object's coordinates
#include <aesbind.h>

#include <obdefs.h>
int rsrc_obfix(*tree, object***) char ****tree***; int** *object*;

rsrc_obfix is an AES routine that changes the form the coordinates for an object that is stored in a resource file. A resource file encodes an object's coordinates in the form of character coordinates, not pixel coordinates. These character coordinates are transformed into pixel coordinates when the resource file is loaded, because only then is the resolution of the screen known. *tree* points to the address of the tree that contains the object in question, and *object* is the number of the object within the tree. **rsrc_obfix** always returns one.

Example

For an example of this function, see **menu**.

See Also

AES, TOS

rsrc_saddr — AES function (libaes)

Store address of a free string or a bit image
#include <aesbind.h>
int rsrc_saddr(*type, index, address***) int** *type, index*; **char ****address*;

rsrc_saddr is an AES routine that copies into an object the address of a pointer to either the free string or the free image of another object within the object tree. *type* denotes the type of pointer whose address is being stored: 15 indicates a pointer to a free string, and 16 indicates a pointer to a bit image. **rsrc_saddr** returns zero if an error occurred, and a number greater than zero if one did not.

See Also

AES, TOS

runtime startup — Overview

The C runtime startup is a routine that is linked with a C program as the first part of an executable program. It performs the functions needed to start and terminate the C environment. To begin the program, it initializes the stack and calls **main**; to conclude the program, it calls **exit** with the return value from **main**.

Three C runtime startup routines are available on Mark Williams C for the Atari ST: **crts0.o**, the normal runtime startup; **crtsg.o**, the runtime startup for the GEM environment; and **crtsd.o**, which is used to create a GEM desktop application. The default is **crts0.o**, which is appropriate for most uses. You can call **crtsg.o** on the **cc** command line in either of two ways: with the switch **-VGEM**, or with the **name** option **Ncrtsg.o**. The **crtsd.o** start-up routine can be called with the option **-VGEMACC** or with the **name** option **Ncrtsd.o**.

See Also

calling conventions, cc, crts0.o, crtsd.o, crtsg.o, stack, _stksize

rvalue — Definition

An **rvalue** is the value of an expression. The name comes from the assignment expression **e1=e2;**, in which the right operand is an rvalue.

Unlike an lvalue, an rvalue can be either a variable or a constant.

See Also

lvalue

Rwabs — bios function 4 (osbind.h)

Read or write data on a disk drive
#include <osbind.h>
#include <bios.h>
long Rwabs(*r_or_w, buffer, n, rec, drive***) int** *r_or_w, n, rec, drive*; **char ****buffer*;

Rwabs reads from or writes data to a disk drive. *r_or_w* indicates the task to perform, as follows:

0	read
1	write
2	read, no medium change
3	write, no medium change

n is the number of sectors to transfer; *rec* is the number of the first record to transfer; and *drive* is the name of the disk drive to use: zero indicates drive A, one indicates drive B, etc.

buffer points to the area to which the data are to be written, or from which they are to be read. If *buffer* is set to zero, then the status set in the argument *r_or_w* is used to set the drive's medium change status.

Rwabs returns zero if all went well, and a number less than zero if an error occurred.

See Also

bios, TOS

S

sbrk – General function (libc)

Increase a program's data space
char *sbrk(*increment*)
unsigned int *increment*;

sbrk increases a program's data space by *increment* bytes. It increments the variable **p_hitpa** of the base page, which points to the end of the program's data space. See **basepage.h** for more information on **p_hitpa**. Note that the memory allocation routine **malloc** calls **sbrk** should you attempt to allocate more space than is available in the program's data space.

sbrk returns a pointer to the previous setting of **p_hitpa** if the requested memory is available, or **((char *)-1)** if it is not.

See Also

basepage.h, malloc, maxmem

Notes

sbrk will not increase the size of the program data area if the physical memory requested exceeds the physical memory allocated by TOS, or if the requested memory exceeds the limit set in the user-defined variable **maxmem**. **sbrk** does not keep track of how space is used; therefore, memory seized with **sbrk** cannot be freed. *Caveat utilitor.*

scanf – STDIO function (libc)

Accept and format input
#include <stdio.h>
int scanf(*format, arg1, ... argN*)
char ***format*; **[data type] ***arg1*, ... ***argN*;

scanf reads the standard input, and uses the string *format* to specify a format for each *arg1* through *argN*, each of which must be a pointer.

scanf reads one character at a time from *format*; white space characters are ignored. The percent sign character '%' marks the beginning of a conversion specification. '%' may be followed by characters that indicate the width of the input field and the type of conversion to be done.

scanf reads the standard input until the return key is pressed. Inappropriate characters are thrown away; e.g., it will not try to write an alphabetic character into an **int**.

The following modifiers can be used within the conversion string:

1. The asterisk '*', which indicates that the next input field should be skipped rather than assigned to the next *arg*.
2. A string of decimal digits, which specifies a maximum field width.
3. An **l**, which specifies that the next input item is a **long** object rather than an **int** object. Capitalizing the conversion character has the same effect.

The following conversion characters are recognized:

c Assign the next input character to the next *arg*, which should be of type **char ***.

d Assign the decimal integer from the next input field to the next *arg*, which should be of type **int ***.

D Assign the decimal integer from the next input field to the next *arg*, which should be of type **long ***.

e Assign the floating point number from the next input field to the next *arg*, which should be of type **float ***.

E Assign the floating point number from the next input field to the next *arg*, which should be of type **double ***.

f Same as **e**.

F Same as **E**.

o Assign the octal integer from the next input field to the next *arg*, which should be of type **int ***.

O Assign the octal integer from the next input field to the next *arg*, which should be of type **long ***.

s Assign the string from the next input field to the next *arg*, which should be of type **char ***. The array to which the **char *** points should be long enough to accept the string and a terminating NUL character.

x Assign the hexadecimal integer from the next input field to the next *arg*, which should be of type **int ***.

X Assign the hexadecimal integer from the next input field to the next *arg*, which should be of type **long ***.

It is important to remember that **scanf** reads up, but not through, the newline character; the newline remains in the standard input device's buffer until you dispose of it somehow. Programmers have been known to forget to empty out the buffer before calling **scanf** a second time, which leads to unexpected results.

Example

The following example uses **scanf** in a brief dialogue with the user.

```
#include <stdio.h>

main()
{
        int left, right;

        printf("No. of fingers on your left hand:  ");
        fflush(stdout);
        scanf("%d", &left);
        while(getchar() != '\n');  /* eat newline char */

        printf("No. of fingers on your right hand:  ");
        fflush(stdout);
        scanf("%d", &right);
        while(getchar() != '\n');

        printf("You've %d left fingers, %d right, & %d total\n",
                left, right, left+right);
}
```

See Also

fscanf, sscanf, STDIO
The C Programming Language, page 147

Diagnostics

scanf returns the number of arguments filled. It returns EOF if no arguments can be filled or if an error occurs.

Notes

Because C does not perform type checking, it is essential that an argument match its specification; for that reason, **scanf** is best used to process only data that you are certain are in the correct data format. The use of upper-case format characters to specify long arguments is not standard; use the 'l' modifier for portability.

It is not recommended that **scanf** be used to obtain a string from the keyboard: use **gets** to obtain the string, and **sscanf** to format it.

Scrdmp — xbios function 20 (osbind.h)

Print a dump of the screen
#include <osbind.h>
#include <xbios.h>
void Scrdmp()

Scrdmp dumps the screen to the printer port, and returns nothing. Note that at present this routine works only with the monochrome monitor.

Example

This example dumps the screen to a printer. Be sure that before you use this example, your printer is plugged into your computer, properly described to TOS, and turned on.

```
#include <osbind.h>
#include <bios.h>

main() {
        if(Bcostat(BC_PRT) == 0 )
                Cconws( "The printer is not ready.\n\r" );

        else {
                Cconws( "The screen is being printed... Please wait.\n\r" );
                Scrdmp();
                Cconws( "The screen is printed.\n\r" );
        }
        return(0);
}
```

See Also

TOS, xbios

screen control — Technical information

The Atari ST uses the following escape sequences to control the terminal screen. These can be passed by the macro **Cconout**, as well as by numerous other output routines, to manipulate the Atari ST's screen:

Note that **<esc>** represents the escape character, ASCII 033.

<esc>A	Cursor up
<esc>B	Cursor down
<esc>C	Cursor forward
<esc>D	Cursor backward
<esc>E	Clear screen, home cursor
<esc>H	Home cursor
<esc>I	Return to same position on previous line
<esc>J	Erase to the end of the page
<esc>K	Clear to the end of the line
<esc>L	Insert line
<esc>M	Delete line
<esc>Y *row col*	Position cursor at *row*, *col*, which are row/column numbers plus 040 (space character)
<esc>bc	Set foreground color to *c*
<esc>cc	Set background color to *c*
<esc>d	Erase beginning of display
<esc>e	Make cursor visible
<esc>f	Make cursor invisible
<esc>j	Save cursor position

<esc>k	Restore cursor position
<esc>l	Erase a line
<esc>o	Erase from beginning of line to cursor
<esc>p	Enter reverse video mode
<esc>q	Exit reverse video mode
<esc>v	Wrap text at end of line
<esc>w	Discard text at end of line

For the sequences **<esc>b** and **<esc>c**, the variable *c* is the color index plus 040. In monochrome mode, the color index can be zero or one; in medium resolution, it can be zero through three; and in low resolution, it can be one through 15.

Example

The following example clears the screen and homes the cursor, then moves the cursor to row 12, column 6 on the screen.

```
main() {
      char row = 12+'\040';
      char column = 6+'\040';

      printf("\033E");
      printf("\033Y%c%c", row, column);
}
```

See Also

Cconout, gemdos, TOS

scrp_read — AES function (libaes)

Read the scrap directory
#include <aesbind.h>
int scrp_read(*buffer***) char ***buffer***;**

The "scrap" feature provides a way for applications to pass information among themselves.

The information to be passed is written into a file, which is always called **scrap.***xxx*. The suffix indicates what type of information the file contains: text (**.txt**), a GEM metafile (**.gem**), a bit image (**.img**), or spreadsheet data (**.dif**).

The name of the directory that holds the scrap file is written into a static buffer, or *clipboard*. The clipboard contains only the name of the directory in which the information is kept, not the information itself. The clipboard is overwritten each time it is used, so in effect only one scrap file can be used at any given time. AES provides routines for reading and writing to the clipboard; it is up to you to see to it that the scrap file is correctly written and read.

scrp_read is an AES routine that reads the clipboard. *buffer* points to the name of a buffer into which the contents of the clipboard will be written. **scrp_read** returns zero if an error occurred, and a number greater than zero if one did not.

See Also

AES, TOS

scrp_write — AES function (libaes)

Write to the scrap directory
#include <aesbind.h>
int scrp_write(*directory***) char ***directory***;**

scrp_write is an AES routine that writes the name of the scrap directory onto the clipboard. *directory* is the name of the scrap directory. **scrp_write** returns zero if an error occurred, and a number greater than zero if one did not. For more information on using the clipboard, see the entry for **scrp_read**.

See Also

AES, scrp_read, TOS

set — Command

Set an **msh** variable
set [*VARIABLE***=***value***]**

set sets sets the **msh** *VARIABLE* to *value*. For example, the command

```
set b="b:\bin"
```

tells **msh** that the variable **b** is equivalent to **b:\bin**; thus, typing

```
cd $b
```

is equivalent to typing

```
cd b:\bin
```

Typing **set** without an argument displays all the variables that have been set. Typing

```
set in history
```

lists the contents of the shell's history buffer. Typing

```
set in .bin
```

lists the installed built-in functions; **.bin** is **msh**'s internal directory, which points to areas in absolute memory where commands are stored.

A second internal directory, **.cmd**, set aside for the user to install functions with the **set** command. For example, the command

```
set in .cmd off="cursconf 3"
```

installs the command **off** into **.cmd**, and declares it to be equivalent to the command **cursconf 3**. **cursconf** is a command that is built into the micro-shell, and uses the TOS function **Cursconf** to manipulate the system cursor. This command turns off the cursor blink.

See Also

commands, msh, unset

setbuf — STDIO function (libc)

Set alternative stream buffers

#include <stdio.h>

setbuf(*fp, buffer***) FILE *****fp*****; char *****buffer*****;**

The standard I/O library STDIO automatically buffers all data read and written in streams, with the exception of streams to terminal devices. STDIO normally uses **malloc** to allocate the buffer, which is a **char** array **BUFSIZ** characters long; **BUFSIZ** is defined in the header file **stdio.h**.

setbuf's arguments are the file stream *fp* and the *buffer* to be associated with the stream. The call should be issued after the stream has been opened, but before any input or output request has been issued. The *buffer* passed to **setbuf** may be **NULL**, in which case the stream will be unbuffered, or contains at least **BUFSIZ** bytes.

See Also

STDIO

setcol — Command

Reset a color

setcol *entry, color*

setcol is a command that uses the **xbios** function **Setcolor** to reset a color. *entry* is the entry in the color palette that you wish to reset, from zero through 15. *color* is the three-digit number that indicates the color to which you wish to set *entry*.

See Also

commands, getcol, TOS

Setcolor — xbios function 7 (osbind.h)

Set one color

#include <osbind.h>

#include <xbios.h>

int Setcolor(*number, value***) int** *number, value***;**

Setcolor sets one color. *number* is the element on the color palette that is being redefined; it can be any number from zero to 15. *value* is the color value to which *number* is being reset; setting any *number* to a negative value ensures that no change is made.

On monochrome monitors,

```
Setcolor(0, 0);
```


gives a black background and white letters, whereas

```
Setcolor(0, 1);
```

switches the screen to a white background and black letters.

Setcolor returns the old value of *number*. The change will be made during the next vertical blank.

Examples

The first example reads and prints out the values of the color map.

```
#include <osbind.h>

color_disp(indx, val)
int indx;
int val;
{
        int red, green, blue;

        red = (val>>8) & 7;                     /* Red value in bits 8-10 */
        green = (val>>4) & 7;                   /* Green value in bits 4-6 */
        blue = val & 7;                         /* Blue value in bits 0-2 */
        printf( " %2d  : %1d %1d %1d\n", indx, red, green, blue );
}

main() {
        int i;
        printf( "Entry  R G B\n" );
        for ( i=0; i<16 ; i++ )
        color_disp( i, Setcolor( i, -1 ));
}
```

The second example works with a monochromatic monitor. It reverses the colors of the characters and background.

```
#include <osbind.h>
main() {
        int color = Setcolor(0, -1);
        Setcolor(0, ++color%2);
}
```

See Also

TOS, xbios

setenv — Command

Set an environmental variable

setenv [*VARIABLE=value*]

setenv sets an environmental variable. Environmental variables are those that are *exported*, or handed to other programs for their use at run time. For example, the environmental variable **TIMEZONE** is read by the C routine **ctime** as part of its time-handling work; whereas the environmental variable **LIBPATH** is read by the linker **ld** to locate its libraries.

You are free to define new environmental variables within your programs, and use **setenv** to define them on your system. Note that it is traditional to spell environmental variable with capital letters.

Typing **setenv** without any arguments displays all of the environmental variables that have been set so far.

See Also

commands, msh, unsetenv

Setexc — bios function 5 (osbind.h)

Get or set an exception vector
#include <osbind.h>
#include <bios.h>
long Setexc(*number, address***) int** *number*; **char** **address*;

Setexc gets or sets an exception vector. Vectors 0x00 through 0xFF are defined by the 68000 hardware; the extended vectors are defined in the header file **signal.h**, as follows:

0x100	timer tick
0x101	critical error handler
0x102	terminate handler
0x103-0x1FF	reserved for future use by TOS
0x200-0x2FF	reserved for future use by users

number is the number of the exception vector to be read or set. *address* is the address to be set into the exception table; -1 indicates that the vector is to be read rather than set. **Setexc** returns either the previous address if it is setting the vector, or the current address if is reading the vector.

Example

This example shows how to use **Setexc** to trap divide-by-zero errors. Note that this program calls the routine **setrte**, which is included with Mark Williams C in the file **setrte.s**. To compile, use the command line

```
cc -o Setexc.prg Setexc.c setrte.s
```

The following gives the text of **Setexc.c**:

```
#include <osbind.h>
#define DIV0 (5)                    /* Divide by 0 vector number */

diverr() {
        setrte();                   /* Make this an exception routine */

        Cconws("\r\nDivision by 0\r\n");
}
```



```
main() {
        register unsigned long oldvec;
        int a = 0;
        int b;

        oldvec = (unsigned long)Setexc(DIV0, diverr);
                                    /* Set the exception */
        printf("This is a test of divide by 0...\n");
        b = 133/a;                  /* Generate error */
        printf("The result of 133/%d is %d\n", a, b);
        Setexc(DIV0, oldvec);       /* Set vector back */
        exit(0);                    /* Return to system */
}
```

See Also

bios, signal.h, TOS

Notes

TOS does not reset exception vectors on process termination; therefore, you must reset them yourself or face the consequences.

setjmp — General function (libc)

Perform non-local goto
#include <setjmp.h>
int setjmp(*env***) jmp_buf** *env*;

The function call is the only mechanism that C provides to transfer control between functions. This mechanism, however, is inadequate for some purposes, such as handling unexpected errors or interrupts at lower levels of a program. To answer this need, **setjmp** helps to provide a non-local *goto* facility. **setjmp** saves a stack context in *env*, and returns value zero. The stack context can be restored with the function **longjmp**. The type declaration for **jmp_buf** is in the header file **setjmp.h**. The context saved includes the program counter, stack pointer, and stack frame. This routine does not restore register variables, but other variables are not affected.

See Also

getenv, longjmp, setjmp.h

Notes

Programmers should note that many user-level routines cannot be interrupted and reentered safely. For that reason, improper use of **setjmp** and **longjmp** will result in the creation of mysterious and irreproducible bugs. The use of **longjmp** to exit interrupt exception or signal handlers is particularly hazardous.

setjmp.h — Header file

Define setjmp() and longjmp()
#include <setjmp.h>

setjmp.h defines the structure **jmp_buf** for a **setjmp** environment.

See Also

header file, longjmp, setjmp

setpal — Command

Reset the color palette
setpal *entry1 ... entry16*

setpal is a command that uses the **xbios** function **Setpallete** (*sic*) to reset the system's color palette. The arguments *entry1* through *entry16* each is a three-digit number that specifies the color code for the corresponding entry in the Atari color palette. If fewer than 16 arguments are given, only that many entries in the palette will be changed.

To alter a specific entry in the color palette, use the command **setcol**.

See Also

commands, getpal, setcol, Setpallete, TOS

Setpallete — xbios function 6 (osbind.h)

Set the screen's color palette
#include <osbind.h>
#include <xbios.h>
void Setpallete(*palette***) int** *palette*[16];

Setpallete (*sic*) sets the screen's color palette, and returns nothing. *palette* points to an array of 16 hexadecimal integers, each of which indicates a different color. The palette is implemented at the next vertical blank interval.

Example

This example sets the color palette. A palette is a table of 16 words containing the definitions for 16 colors as indexed by set bits in the "planes".

```
#include <osbind.h>

short ugly[] = {
        0x000, 0x111, 0x222, 0x333,
        0x444, 0x555, 0x666, 0x777,
        0x007, 0x070, 0x700, 0x707,
        0x770, 0x077, 0x737, 0x337
        };

main() {
        Setpallete( ugly );
}
```

See Also

TOS, xbios

setphys — Command

Reset physical screen's display space
setphys *address*

setphys is a command that resets the physical screen's display base. It can be used to display any part of the ST's memory as a bit map. *address* is the address of the new display base.

See Also

commands, getphys, TOS

setprt — Command

Reset the printer port
setprt *configuration*

setprt is a command that uses the **xbios** function **Setprt** to reconfigure the printer port. *configuration* is an integer that indicates the port's new configuration. For a table of the configuration codes, see the entry for **Setprt**.

See Also

commands, Setprt, TOS

Setprt — xbios function 33 (osbind.h)

Get or set the printer's configuration
#include <osbind.h>
#include <xbios.h>
int Setprt(*configuration***) int** *configuration*;

Setprt gets or sets the configuration of the printer port. *configuration* is a 16-bit map that configures the port. If it is set to 0xFFFF (-1), the port's current configuration is read; otherwise, its value is used to set the port, as follows:

0x01	daisywheel printer
0x02	monochrome printer
0x04	if set, Epson-type dot-matrix printer; if not, Atari printer
0x08	if set, final mode; if not, draft mode
0x10	if set, printer uses serial port; if not, printer port
0x20	if set, uses single sheets; if not, uses fanfold paper

Bits 6 through 14 are reserved, and bit 15 must be zero. These values are defined in the header file **xbios.h**.

Setprt returns the printer port's current configuration when *configuration* is set to -1; otherwise, it returns a meaningless value.

Example

For examples of this function, see the entries for **\auto** and **prtblk**.

See Also

Prtblk, TOS, xbios, xbios.h

setrez — Command

Reset the screen resolution
setrez *resolution*

setrez is a command that resets the screen's resolution. *resolution* indicates the new screen resolution, as follows: zero, high resolution; one, medium resolution; and two, low resolution. Note that changing from a color resolution to a monochrome resolution will warm start the machine and put you back to a correct resolution for your monitor. Changing from low to medium resolution, or vice versa, will create a distorted image that can be corrected with, respectively, the commands **ltom** and **mtol**.

See Also

commands, getrez, Getrez, TOS

Notes

If you enter **msh** or run a GEM program without restoring the resolution, unpredictable results will be evident.

Setscreen — xbios function 5 (osbind.h)

Set the video parameters
#include <osbind.h>
#include <xbios.h>
void Setscreen(*log, phys, res***) char ****log, ****phys***; int** *res*;

Setscreen sets the video parameters, and returns nothing. *log* and *phys* are the bases of the logical and physical screen displays. *res* is the new screen resolution:

0	low resolution
1	medium resolution
2	high resolution

Setting any variable to a negative number ensures that that variable will be ignored.

Example

This example demonstrates **Setscreen**. For another example, see the entry for **Physbase**.

```
#include <osbind.h>
#include <bios.h>
```



```
main() {
        char *newscr, *oldscr, *memblk;
        int x, y;
        Cconws("Working...\n");
        oldscr = (char *) Physbase();

        if((memblk = (char *)Malloc(32*1024L)) == 0) {
                printf("Malloc of %ld bytes failed.\n", 32*1024L);
                Pterm(1);
        }

        newscr = (char *) (((long) memblk + 0xFFL) & ~(0xFFL));
        Setscreen(newscr,-1L,-1);           /* Change logical base */
        Cconws("\033H\033J");               /* Clear logical screen */

        for (y=0; y<24; y++) {               /* for 20 rows... */
                for (x=0; x<39; x++) {       /*  39 times each... */
                        Bconout(BC_RAW, 0x0E);
                        Bconout(BC_RAW, 0x0F);
                }
                Cconws("\r\n");
        }

        Setscreen(-1L,newscr,-1);           /* Move physical base... */
        Cconin();
        Setscreen(oldscr,oldscr,-1);        /* Restore addresses... */
        return 0;
}
```

See Also

Getrez, Logbase, Physbase, TOS, xbios

Notes

If you change the resolution of the screen with this routine, the screen will be cleared. Under some circumstances, the previous screen base will also be cleared. Therefore, it is best to switch screen bases and then change resolutions, rather than doing both with one call.

Settime — xbios function 22 (osbind.h)

Set the current time
#include <osbind.h>
#include <xbios.h>
void Settime(*datetime***) long** *datetime*;

Settime sets the current time and date for the intelligent keyboard (IKBD), and returns nothing. *datetime* is a 32-bit mask whose bits indicate the following:

0-4	no. of two-second increments (0-29)
5-8	no. of minutes (0-59)
9-15	no. of hours (0-23)
16-20	day of the month (1-31)
21-26	month (1-12)
27-31	year (0-119, 0 indicates 1980)

Example

This examples sets the IKBD time. Note that this does not affect the current GEM-DOS time.

```
#include <osbind.h>

main() {
        register unsigned long time;
        int seconds;
        int minutes;
        int hours;
        int day;
        int month;
        int year;

        printf("Enter the date and time (MM/DD/YYYY HH:MM): ");
        scanf("%d/%d/%d %d:%d", &month, &day, &year, &hours, &minutes );
        seconds = 0;
        if(year < 100)
                year += 1900;
        time =  ((unsigned long)(year-1980)<<25)
                |((unsigned long)month<<21)
                |((unsigned long)day<<16)
                |((unsigned long)hours<<11)
                |((unsigned long)minutes<<5)
                |((unsigned long)seconds>>1);
        timeprint("We are setting the time to", time );
        Settime(time);
```



```
/* Verify what we did. */

        time = Gettime();
        timeprint("What we get is", time);
}

void fixdig(buf, onumber, size)
char *buf;
int onumber;
int size;
{
        register long limit;
        register long number;
        int o;

        number = onumber;

        limit = 10;
        for (o = 1; o < size ; o++)
                limit *= 10;

        if ((number >= limit)||(number <0)) {
                for (o = 0; o < size; o++)
                        *buf++ = '*';
                *buf = 0;
                return;
        }
        for (o = 0; o < size; o++) {
                limit /= 10;
                *buf++ = '0'+number/limit;
                number = number%limit;
        }
        *buf = '\0';
}

timeprint(string, time)
char *string;
register unsigned long time;
{
        int seconds;
        int minutes;
        int hours;
        int month;
        int day;
        int year;
        char mins[3];
        char secs[3];
```

```
        seconds = (time & 0x001F) << 1;          /* Bits 0:4 */
        minutes = (time >> 5)  & 0x3F;           /* Bits 5:10 */
        hours = (time >> 11) & 0x1F;             /* Bits 11:15 */

        day = (time >> 16) & 0x1F;               /* Bits 16:20 */
        month = (time >> 21) & 0x0F;             /* Bits 21:24 */
        year = ((time >> 25) & 0x7F)+1980;       /* Bits 25:31 */

        fixdig(mins, minutes, 2);
        fixdig(secs, seconds, 2);
        printf("%s %d:%s:%s on %d/%d/%d\n", string, hours, mins,
              secs, month, day, year);
}
```

For another example of this function, see the entry for **time**.

See Also

Gettime, Ksettime, time, TOS, xbios

Notes

The time data in the bit map used by **Settime** is in exactly the reverse order of the data used by the **gemdos** functions.

Sgettime — Time function (libc)

Read time from intelligent keyboard's clock

#include <time.h>
tm *Sgettime();

Sgettime is a function that reads the time from the intelligent keyboard's clock. This clock is maintained apart from the other clocks on the Atari ST. **Sgettime** returns a pointer to the structure **tm**, which it initializes. **tm** is defined in the header file **time.h**. For more information about it, see the entry for **time**.

See Also

Kgettime, Ssettime, time (overview), time.h, tm

Notes

Unlike the function **Gettime**, which deals in two-second increments, **Sgettime** allows the programmer to work with clock ticks.

Unlike the related function **Kgettime**, **Sgettime** works on the Mega ST.

shel_envrn — AES function (libaes)

Search for an environmental variable

#include <aesbind.h>
int shel_envrn(*parameter, name***) char *****parameter, *name;**

shel_envrn is an AES routine that searches for a particular environmental variable in the desktop's environment. *name* points to the name of the variable whose value you want; note that the name must end with an equal sign '='. *parameter* points to the byte immediately *following* the value of the variable. **shel_envrn** always returns one.

Example

The following example uses the **shel** library to exchange information with the environment.

```
#include <aesbind.h>

alertf(n, p) int n; char *p;
{
        static char buffer[512];
        sprintf(buffer, "%r", &p);
        return form_alert(n, buffer);
}

main()
{
        char *cp;
        char cmd[128], tail[128];
        int retval;

        appl_init();

        retval = shel_envrn(&cp, "PATH=");
        alertf(1, "[0][shel_envrn  |returns |%d ][Ok]", retval);
        alertf(1, "[0][PATH= is |%s  ][Ok]", cp);

        retval = shel_read(cmd, tail);
        alertf(1, "[0][shel_read  |returns |%d ][Ok]", retval);
        alertf(1, "[0][command is  |%s  ][Ok]", cmd);
        alertf(1, "[0][tail is  |%s   ][Ok]", tail);

        retval = appl_find("SHEL    ");
        alertf(1, "[0][appl_find SHEL is  |%d  ][Ok]", retval);

        retval = appl_find("shel    ");
        alertf(1, "[0][appl_find shel is  |%d  ][Ok]", retval);

        retval = appl_find("MSH     ");
        alertf(1, "[0][appl_find MSH is  |%d  ][Ok]", retval);

        if (alertf(1, "[2][ invoke shel ][No|Yes]") == 2) {
              retval = shel_write(1, 1, 1, "shel.prg", "sock it to me");
              alertf(1, "[0][ shel_write | returns | %d ][Ok]", retval);
        }

        appl_exit();
        return 0;
}
```

See Also

AES, TOS

Notes

shel_envrn can find a variable only in the desktop environment, *not* the environment of the current process. Due to the design of the AES, it can return only that part of the environment which fits into a small buffer. The sixth character of the

desktop environment is always set to ';' by the desktop because the **PATH** environment passed by the ROM always has a null character in that position.

shel_find — AES function (libaes)

Search PATH for file name
#include <aesbind.h>
int shel_find(*pathname***) char ***pathname***;**

shel_find is an AES routine that does searches for a file in the directories named in the **PATH** environmental variable. *pathname* points to the name of the file being sought; **shel_find** changes this name to the full path name of the file if it is found. **shel_find** returns zero if an error occurred, and a number greater than zero if one did not.

See Also

AES, PATH, TOS

shel_read — AES function (libaes)

Let an application identify the program that called it
#include <aesbind.h>
int shel_read(*command, tail***) char ***command, *tail***;**

shel_read is an AES routine that returns the name of the command that invoked the current AES application. *command* points to the name of the command, and *tail* points to its tail; the values of both are set by this routine. **shel_read** returns zero if an error occurred, and a number greater than zero if one did not.

Example

For an example of this function, see **shel_envrn**.

See Also

AES, TOS

Notes

Even after a command and a tail is passed successfully through **shel_read**, **shel_write** returns two copies of the command, instead of the command and its tail.

shel_write — AES function (libaes)

Tell desktop which application to run next
#include <aesbind.h>
int shel_write(*flag, graphic, gem, command, tail***)**
int *flag, graphic, gem***; char ***command, *tail***;**

shel_write is an AES routine that tells AES whether to run another application, and, if necessary, which application to run. In GEM terms, it combines **Pterm** and optionally **Pexec**. In UNIX terms, it combines **exit** and optionally **exec**. In effect, it tells the desktop to continue.

flag indicates whether to run another application: zero, exit to the operating system; one, run another application. *graphic* indicates if the application to be run is a graphics application: zero indicates no, and one indicates yes. *gem* indicates if the application to be run is an AES application: zero indicates no, and one indicates yes.

Finally, *command* and *tail* point, respectively, to the command's name and tail. **shel_write** returns zero if an error occurred, and a number greater than zero if one did not.

Example

For an example of this function, see **shel_envrn**.

See Also

AES, TOS

shellsort — General function (libc)

Sort arrays in memory
void shellsort(*data, n, size, comp***)**
char ***data*; int** *n, size***; int (***comp***)();**

shellsort is a generalized algorithm for sorting arrays of data in primary memory. It uses D. L. Shell's sorting method. **shellsort** works with a sequential array of memory called *data*, which is divided into *n* parts of *size* bytes each. In practice, *data* is usually an array of pointers or structures, and *size* is the **sizeof** the pointer or structure.

Each routine compares pairs of items and exchanges them as required. The user-supplied routine to which *comp* points performs the comparison. It is called repeatedly, as follows:

```
(*comp)(p1, p2)
char *p1, *p2;
```

Here, *p1* and *p2* each point to a block of *size* bytes in the *data* array. In practice, they are usually pointers to pointers or pointers to structures. The comparison routine must return a negative, zero, or positive result, depending on whether *p1* is less than, equal to, or greater than *p2*, respectively.

Example

For an example of how to use this routine, see the entry for **string**.

See Also

ctype, qsort
The Art of Computer Programming, vol. 3, pp. 84*ff*, 114*ff*

Notes

shellsort differs from the sort function **qsort** in that it uses an iterative algorithm that does not require much stack.

short — C keyword

Data type

A short is a numeric data type. By definition, it cannot be longer than an **int** or a **long**. For Mark Williams C, a **short** is equal to an **int**; that is, **sizeof short** equals two **chars**, or 15 bits plus a sign. A **short** normally is sign extended when cast to a larger data type; however, an **unsigned short** will be zero extended when cast.

See Also

C keywords, C language, data format, data type, declarations

show — Command

Display a stored screen image
show *screenfile*

show displays a screen image that has been stored either with the command **snap**, or with one of several graphics editors. *screenfile* is the name of the file in which the screen image is stored. *screenfile* can be in any of the following formats, as indicated by its suffix:

.PI1	DEGAS uncompressed screen images, low resolution
.PI2	DEGAS uncompressed screen images, medium resolution
.PI3	DEGAS uncompressed screen images, high resolution
.NEO	Neochrome uncompressed screen images
.MUR	COLR editor screen murals
.PIC	Atari Logo SAVEPIC files

show checks the size of each file to confirm that it is of the correct type; if it is of the wrong size for its type, **show** exits silently.

Note that you may need to alter the image's resolution to resolve it on your current device. This can be done with the battery of commands **htom, ltom, mtoh**, and **mtol**. For example, to display the low-resolution DEGAS file **foo.pi1** on a high-resolution monitor, use the following command line:

```
show foo.pi1 ; ltom ; mtoh
```

show can also be used with the command **snap** to convert an image from one format to another. For example, to convert the high-resolution DEGAS file **foo.pi3** to Neochrome format, use the following command line:

```
show foo.pi1 ; htom ; mtol ; snap foo.neo
```


See Also

commands, htom, ltom, mtoh, mtol, snap, TOS

showmouse — Command

Redisplay the mouse pointer
showmouse

showmouse uses the function **linea9** to redisplay the mouse pointer.

See Also

commands, hidemouse, Line A, mousehidden, TOS

signal.h — Header file

Define Atari ST signals
#include <signal.h>

signal.h is a header file that defines signals used on the Atari ST. These include 68000 machine exceptions, trap instructions, and GEM-DOS aliases.

See Also

bombs, header file, TOS

sin — Mathematics function (libm)

Calculate sine
#include <math.h>
double sin(*radian***) double** *radian***;**

sin calculates the sine of its argument *radian*, which must be in radian measure.

Example

For an example of this function, see the entry for **acos**.

See Also

mathematics library

sinh — Mathematics function (libm)

Calculate hyperbolic sine
#include <math.h>
double sinh(*radian***) double** *radian***;**

sinh calculates the hyperbolic sine of *radian*, which is in radian measure.

Example

For an example of this function, see the entry for **cosh**.

See Also

mathematics library

size — Command

Print the size of an object module
size [-act] *file...*

size prints the size of each segment of each given *file*, which must be a relocatable object module. The total size is given in decimal, and the size of each segment is given in both decimal and hexadecimal. All sizes are in bytes.

The options are as follows:

-a Print the size of debug, symbol, and relocation segments as well.

-c Print the total size of all common areas in each relocatable object module.

-t At the end, print the total size of each segment summed over all the files; no total is printed if only one *file* is specified.

size prints out the size of each segment, as follows:

c	code
d	data
e	extra
s	stack
a1	auxiliary 1
a2	auxiliary 2
a3	auxiliary 3
a4	auxiliary 4

The suffix **x** (for extension) on a segment identifier indicates the difference between the initialized size in the file and the minimum size in memory. For programs compiled under Mark Williams C, this only appears as **dx** and indicates the size of the arena.

See Also

cc, commands, cpp, nm, strip

Notes

Because version 3.0 changes the object format, the edition of **size** shipped with version 3.0 does not work with objects compiled with Mark Williams C version 2.1.7 or earlier. To convert such objects to a format that **size** recognizes, use the command **mwtomw**.

sizeof — C keyword

Return size of a data element

sizeof is a C operator that returns a constant **int** that is the size of any given data element. The element examined can be a data object, a portion of a data object, or a type cast. **sizeof** returns the size of the element in **chars**; for example


```
long foo;
sizeof(foo);
```

returns four, because a **long** is as long as four **chars**.

Note that **sizeof** is especially useful in **malloc** routines, and when you need to specify byte counts to I/O routines. Using it to set the size of data types instead of using a predetermined value will increase the portability of your code.

See Also

C keywords, C language, data types, operators
The C Programming Language, page 188

sleep — Command

Stop executing for a specified time
sleep *seconds*

sleep suspends execution for a specified number of *seconds*. This routine is especially useful with other commands to the shell **msh**. For example, typing

```
sleep 3600; echo coffee break time
```

will execute the **echo** command in one hour (3,600 seconds) to indicate an important appointment. **sleep** operates in two-second increments under TOS.

See Also

commands, msh, msleep

snap — Command

Save a screen image
snap *scrfile*

snap takes a "snapshot" of the screen's image, and writes it into *scrfile*. **snap** stores a screen image in any of the following formats, as indicated by the suffix to *scrfile*:

.PI1	Degas uncompressed screen images, low resolution
.PI2	Degas uncompressed screen images, medium resolution
.PI3	Degas uncompressed screen images, high resolution
.NEO	Neochrome images
.MUR	COLR editor screen murals
.PIC	Atari Logo SAVEPIC files

For example, typing

```
snap foo.mur
```

saves the current screen image into file **foo.mur**. It can then be redisplayed with the **show** command.

See Also
commands, show, TOS

sort — Command

Sort lines of text
sort [-bcdflmnru] [-t *c*] [-o *outfile*] [-T *dir*] [+*beg*[-*end*]][*file* ...]

sort reads lines from each *file* specified, or the standard input if none. It writes to the standard output in sorted order. The order into which the output is sorted is determined by comparing a *key* from each line; the key is all or part of an input line, depending upon options are selected. By default, the key is the entire input record (line) and ordering is by the ASCII collating sequence, i.e., lower-valued ASCII characters sorted before higher-valued.

The following options affect how the key is constructed or how the output is ordered.

-b Ignore leading white space (blanks or tabs) in key comparisons.

-d Dictionary ordering; only letters, blanks, and digits are considered in key comparisons. This is essentially the ordering used to sort telephone directories.

-f Fold upper-case letters to lower case for comparison purposes.

-i Ignore all characters outside of the printable ASCII range (octal 040-0176).

-n This option tells **sort** that the key is a numeric string, which consists of optional leading blanks and optional minus sign followed by any number of digits with an optional decimal point. The ordering is by the numeric, as opposed to alphabetic, value of the string.

-r Reverse the ordering, i.e., **sort** from largest to smallest.

As noted above, the key compared from each line need not be the entire input line. The option *+beg* indicates the beginning position of the key field in the input line, and the optional *-end* indicates that the key field ends just before the *end* position. If no *-end* is given, the key field ends at the end of the line. Each of these positional indicators has the form *+m.nf* or *-m.nf*, where *m* is the number of fields to skip in the input line and *n* is the number of characters to skip after skipping fields. Optional flags *f* are chosen from the above key flags (**bdfinr**) and are local to the specified field.

The following additional options control how **sort** works.

-c Check the input to see if it is sorted. Print the first out of order line found.

-m Merge the input files. **sort** assumes each *file* to be sorted already. For large files, it runs much faster with this option.

-o *outfile*
Put the output into *outfile* rather than on the standard output. This allows **sort** to work correctly if the output file is one of the input files.

-t*c* Use the character *c* to separate fields rather than the default blanks and tabs.

-u Suppress multiple copies of lines with key fields that compare equally.

See Also
commands

sprintf — STDIO function (libc)

Format output
#include <stdio.h>
int sprintf(*string*, *format* [, *arg*] ...)
char **string*, **format*;

sprintf uses the string *format* to specify an output format for each *arg*; it then writes every *arg* into *string*, which it ends with NUL. For a detailed discussion of **sprintf**'s formatting codes, see **printf**.

Example
For an example of this function, see the entry for **sscanf**.

See Also
printf, sprintf, STDIO
The C Programming Language, page 150

Notes
The output *string* passed to **sprintf** must be large enough to hold all output characters. Because C does not perform type checking, it is essential that each argument match its format specification.

At present, **sprintf** does not return a meaningful value.

sqrt — Mathematics function (libm)

Compute square root
#include <math.h>
double sqrt(*z*) double *z*;

sqrt returns the square root of *z*.

Example
For an example of this function, see the entry for **ceil**.

See Also
mathematics library

Diagnostics
When a domain error occurs (i.e., when *z* is negative), **sqrt** sets **errno** to EDOM and returns zero.

srand — General function (libc)

Seed random number generator
void srand(*seed***) int** *seed***;**

srand uses *seed* to initialize the sequence of pseudo-random numbers returned by **rand**. Different values of *seed* initialize different sequences.

Example
For an example of this function, see the entry for **rand**.

See Also
rand
The Art of Computer Programming, vol. 2

sscanf — STDIO function (libc)

Format input
#include <stdio.h>
int sscanf(*string***,** *format* **[,** *arg* **] ...)**
char ****string*; char ****format***;**

sscanf reads the argument *string*, and uses *format* to specify a format for each *arg*, each of which must be a pointer. For more information on **sscanf's** conversion codes, see **scanf**.

Example
This example uses **sprintf** to create a string, and then reads it with **sscanf**. It also illustrates a common problem with this routine.

```
#include <stdio.h>

main()
{
        char string[80];
        char s1[10], s2[10];

        sprintf(string,"1234567890123456789012345678901234567890");
        sscanf(string,"%9c",s1);
        sscanf(string,"%10c",s2);

        printf("27s is the string\n", string);
        printf("%s: first 9 characters in string\n", s1);
        printf("%s\n", s2);
        printf("comes from not leaving space for terminator\n");
}
```


See Also
fscanf, scanf, STDIO
The C Programming Language, page 150

Diagnostics
sscanf returns the number of arguments filled. It returns zero if no arguments can be filled or if an error occurs.

Notes
Because C does not perform type checking, an argument must match its format specification. **sscanf** is best used only to process data that you are certain are in the correct data format, such as data that were written with **sprintf**.

stack — Definition

The **stack** is the segment of memory that holds function arguments, local variables, function return addresses, and stack frame linkage information. Neither the 68000 nor the Atari ST support dynamic stack resizing, so programs run on the ST have a fixed segment allocated to the stack at run time.

The Mark Williams C runtime startup routine allocates **_stksize** bytes of stack when a program is executed, and sets the 68000 stack pointer register, **a7**, to point at the highest address in this segment. **_stksize** is then assigned a pointer to the lowest address that the stack pointer may reach before the stack begins to overwrite program data. **_stksize** is set to two kilobytes by the Mark Williams C library. It may be set to another value by including an initialized declaration for it in your program; for example

```
long _stksize = 16000L;
```

sets the stack size to 16,000 bytes.

The value of **_stksize** must be even. The size of the stack cannot change once your program has begun to execute because the allocation must be made before the stack is used and your program uses stack as soon as it begins to execute.

If your program uses recursive algorithms, or declares large amounts of automatic data, or simply contains many levels of functions calls, the stack may "overflow", and overwrite the program data. You can check for stack overflow very simply. The runtime startup reinitializes the **long _stksize** to point to an address that the stack should not reach. You can compare **_stksize** to the address of the last automatic variable in any function; as long as **_stksize** is *less* than the address of that automatic function, you are safe.

Example
This example checks for stack overflow; it aborts the program and prints a message when overflow occurs. The **main** routine prints the location of its arguments, calls the stack overflow routine, and then calls itself recursively. For another example, see the entry for **Fgetdta**.

```
_stktest(){
        int i;
        if ((long)&i <= _stksize) {
                puts ("Stack overflow!");
                exit(1);
        }
}

main(argc)
int argc; {
        extern long _stksize;
        printf("argc at %lx\n", &argc);
        _stktest();
        main(argc);
}
```

See Also

\auto, _stksize

Notes

TOS pushes data onto the user stack; therefore, you should make sure that your stack has a cushion of at least 128 bytes to hold these data when your program enters the system.

standard error — Definition

The **standard error** is the peripheral device or file where programs write error messages by default. It is defined in the header file **stdio.h** under the abbreviation **stderr**, and by default is the computer's monitor.

See Also

freopen, header file, msh, standard input, standard output, stdio.h

standard input — Definition

The **standard input** is the device or file from which data are accepted by default. It is defined in the header file **stdio.h** under the abbreviation **stdin**, and will be the computer's keyboard unless redirected by the operating system, a shell, or **freopen**.

See Also

freopen, header file, msh, standard error, standard output, stdio.h

standard output — Definition

The **standard output** is the device or file where programs write output by default. It is defined in the header file **stdio.h** under the abbreviation **stdout**, and in most instances is defined to be the computer's monitor.

See Also

freopen, header file, msh, standard error, standard input, stdio.h

stat — General function (libc)

Find file attributes

#include <stat.h>
int stat(*file, statptr*)
char **file*; struct stat **statptr*;

stat returns a structure that contains the GEM-DOS attributes of a file. This function is included to maintain compatibility with the UNIX and COHERENT operating systems.

file points to the path name of file, and *statptr* points to a structure of the type **stat**, as defined in the header file **stat.h**.

The following summarizes the structure **stat**:

```
struct stat {
        short st_dev;           /* set only under COHERENT */
        short st_ino;           /* set only under COHERENT */
        short st_mode;          /* attributes              */
        short st_nlink;         /* set only under COHERENT */
        short st_uid;           /* set only under COHERENT */
        short st_gid;           /* set only under COHERENT */
        short st_rdev;          /* set only under COHERENT */
        long st_size;           /* file size (in bytes)    */
        time_t st_atime;        /* time last accessed      */
        time_t st_mtime;        /* time last modified      */
        time_t st_ctime;        /* time created            */
};
```

The following summarizes the legal settings for **st_mode**, which sets the file's attributes:

S_IJRON	0x01	read-only file
S_IJHID	0x02	hidden from search
S_IJSYS	0x04	system, hidden from search
S_IJVOL	0x08	volume label in first 11 bytes
S_IJDIR	0x10	directory
S_IJWAC	0x20	written to and closed

The entry **st_size** gives the size of the file, in bytes.

Entries in the structure **stat** are there to preserve compatibility with the COHERENT operating system. Most return meaningless values when used on the Atari ST, with the following exceptions: **st_atime**, **st_mtime**, and **st_ctime** all return the time that the file or directory was last modified; **st_size** gives the size of the file, in bytes; and **st_mode** gives the mode of the file.

See Also

fstat, ls, msh, open, stat.h

Diagnostics

stat returns -1 if an error occurs, e.g., the file cannot be found. Otherwise, it returns zero.

stat.h — Header file

Definitions and declarations used to obtain file status
#include <stat.h>

stat.h is a header file that contains the declarations of several structures used by the routines **fstat** and **stat**, which return information about a file's status.

See Also

header file, stat

static — C keyword

Declare storage class

static is a C storage class. A **static** variable resembles an **extern** in that it does not disappear when its calling function exits. Unlike an **extern**, however, a **static** variable is "private": when used within a function, it can be accessed only by that function; when used outside a function, it can be accessed only by functions that are defined within the same source file as the variable. This helps to avoid name conflicts; for example, if a program consists of two files, each of which has a variable named **foo**, declaring each **foo** to be **static** keeps them from overwriting each other.

Functions that are used locally can also be declared to be **static**; this helps to prevent name conflicts when assembling programs from a number of different sources, such as libraries from a variety of vendors and modules written by different programmers.

See Also

auto, C keywords, C language, extern, register variable, storage class
The C Programming Language, page 80

stderr — Definition

stderr is an abbreviation for *standard error*. It is defined in the header file **stdio.h**.

See Also

stdin, stdio.h, stdout, standard error

stdin — Definition

stdin is an abbreviation for *standard input*. It is defined in the header file **stdio.h**.

See Also

standard input, stderr, stdio.h, stdout

STDIO — Overview

STDIO is an abbreviation for *standard input and output*. It refers to a set of standard library functions that accompany all C compilers and that govern input and output with peripheral devices.

Mark Williams C includes the following STDIO routines:

clearerr	present status stream
exit	leave a program gracefully
fclose	close a file stream
fdopen	open a file stream for I/O
feof	discover a file stream's status
ferror	discover a file stream's status
fflush	flush an output buffer
fgetc	get a character
fgets	get a string
fgetw	get a word
fileno	get a file descriptor
fopen	open a file stream
fprintf	format and print to a file stream
fputc	output a character
fputs	output a string
fputw	output a word
fread	read a file stream
freopen	open a file stream
fscanf	format and read from a file stream
fseek	seek in a file stream
ftell	return file pointer position
fwrite	write to a file stream
getc	get a character
getchar	get a character
gets	get a string
getw	get a word
printf	print a formatted string
putc	output a character
putchar	output a character
puts	output a string
putw	output a word
rewind	reset a file pointer
scanf	format and input from standard input
setbuf	set alternative file-stream buffers

sprintf	format and print to a string
sscanf	format and read from a string
ungetc	return character to file stream

STDIO routines are buffered by default.

See Also

buffer, FILE, Lexicon, stdio.h, stream, UNIX routines
The C Programming Language, page 166

stdio.h — Header file

Declarations and definitions for I/O

stdio.h is a header file that defines several manifest constants used in standard I/O, such as **NULL** and **FILE**, declares the STDIO functions, and defines numerous I/O macros.

See Also

header file, manifest constant, STDIO

stdout — Definition

stdout is an abbreviation for *standard output*; it is defined in the header file **stdio.h**.

Example

For an example of how to redirect **stdout** from within a program, see the entry for **system**.

See Also

standard output, stderr, stdin, stdio.h

stime — Time function (libc)

Set the operating system time
#include <time.h>
int stime(*timep***) time_t *****timep**;

stime sets the operating system time, which Mark Williams C defines as being the number of seconds since midnight of January 1, 1970, 0h00m00s GMT. The argument *timep* points to the new system time, which is of the type **time_t**; this is defined in the header file **time.h** as being equivalent to a **long**.

Example

For an example of using this function from the **\auto** directory, see the entry for **\auto**.

Example

The following example prints the time, then uses **stime** to reset the time by one hour.

```
#include <time.h>
main()
{
        long nowhere; /* buffer to put unwanted things */

        /* print current time */
        printf("%s\n", ctime);

        /* subtract one hour (3600 seconds) from current time */
        if (stime((time(&nowhere) - 3600)) == -1)
        {
                printf("Cannot reset time.\n");
                exit(1);
        }

        /* print altered time */
        printf("%s\n", ctime);

        /* add one hour to current time, to correct above */
        if (stime((time(&nowhere) + 3600)) == -1)
        {
                printf("Cannot re-reset time.\n");
                exit(1);
        }

        /* print fixed time, to confirm correction */
        printf("%s\n", ctime);
}
```

See Also

date, time (overview)

Diagnostics

stime returns -1 on error, zero otherwise.

_stksize — External data

_stksize is an external symbol that sets the size of the stack. It is defined in the Mark Williams Company libraries as being equal to two kilobytes, which is more than enough stack for most applications.

If you wish to have more stack, insert into **main** the declaration

```
long _stksize = n;
```

where *n* is the number of bytes required. *n* must be even.

Example

For an example of how to use this variable in a program, see the entry for **memory allocation**. For an example of a program that uses **_stksize** to check for stack overflow, see the entry for **Fgetdta**.

See Also
ld, stack

storage class — Technical information

Storage class refers to the part of a declaration that indicates how data are to be stored. The legal storage classes are as follows:

auto
extern
register
static

typedef is technically defined as a storage class as well, but it does not actually indicate how data are stored. The default class is **auto**.

See Also
auto, extern, register, static, typedef
The C Programming Language, page 192

strcat — String function (libc)

Append one string to another
char *strcat(*string1***, *string2***) char ****string1***, ****string2***;**

strcat appends all characters in *string2* onto the end of *string1*. It returns the modified *string1*.

Example
For an example of this function, see the entry for **string**. For an example of this function in a TOS application, see the entry for **Fgetdta**.

See Also
string, strncat
The C Programming Language, page 44

Notes
string1 must point to enough space to hold itself and *string2*; otherwise, another portion of the program or operating system may be overwritten.

strchr — String function (libc)

Find a character in a string
char *strchr(*string, character***);**
char ****string*; int** *character***;**

strchr searches for *character* within *string*. The null character at the end of *string* is included within the search.

strchr returns a pointer to the first occurrence of *character* within *string*. If *character* is not found, it returns NULL.

See Also
memchr, strcspn, string, strpbrk, strrchr, strspn, strstr, strtok

Notes
This is equivalent to the function **index**, which is also included with Mark Williams C.

strcmp — String function (libc)

Compare two strings
int strcmp(*string1***, *string2***) char ****string1***, ****string2***;**

strcmp compares *string1* with *string2* lexicographically. It returns zero if the strings are identical, returns a number less than zero if *string1* occurs earlier alphabetically than *string2*, and returns a number greater than zero if it occurs later. This routine is compatible with the ordering routine needed by **qsort**.

Example
For examples of this function, see the entries for **string** and **malloc**.

See Also
memcmp, qsort, shellsort, string, strncmp, strcspn, strspn, strstr
The C Programming Language, page 101

strcpy — String function (libc)

Copy one string into another
char *strcpy(*string1***, *string2***) char ****string1***, ****string2***;**

strcpy copies the contents of *string2*, up to the NUL character, into *string1* and returns *string1*.

Example
See **string**. For an example of using this function in a TOS application, see the entry for **Fgetdta**.

See Also
memcpy, string, strncpy
The C Programming Language, page 100

Notes
string1 must point to enough space to hold *string2*, or another portion of the program or operating system may be overwritten.

strcspn — String function (libc)

Length one string excludes characters in another
unsigned int strcspn(*string1***, *string2***)**
char ****string1*, ****string2***;**

strcspn compares *string1* with the characters in *string2*. It then returns the length, in characters, for which *string1* consists of characters *not* found in *string2*.

See Also

memchr, strchr, string, strpbrk, strrchr, strspn, strstr, strtok

stream — Definition

The term **stream** applies to any entity that can be named and from which bits can flow, such as a device or a file. The name "stream" reflects the fact that the C programming environment does not depend upon record descriptors and other devices that predetermine what form data can assume; rather data, from whatever source, are seen to be a flow of bytes whose significance is set entirely by the program that reads them.

For example, whether 16 bits forms an **int**, two **chars**, and should be used as an absolute value or a bit map, is entirely up to the program that receives it. It is also irrelevant to the program that processes these 16 bits whether they come from the keyboard, from a file on disk, or from a peripheral device.

See Also

bit, byte, data formats, file

strerror — String function

Translate an error number into a string
char *__strerror__(*error*); **int** *error*;
char *__strerror__(**int** *error*);

strerror helps to generate an error message. It takes the argument *error*, which presumably is an error code generated by an error condition in a program, and may return a pointer to the corresponding error message.

See Also

perror, string

Notes

strerror returns a pointer to a static array that may be overwritten by a subsequent call to **strerror**.

strerror differs from the related function **perror** in the following ways: **strerror** receives the error number through its argument *error*, whereas **perror** reads the global constant **errno**. Also, **strerror** returns a pointer to the error message, whereas **perror** writes the message directly into the standard error stream.

strerror and **perror** must return the same error message when handed the same error number.

string — Overview

The character string is a common formation in C programs. The runtime representation of a string is an array of ASCII characters that is terminated by a NUL character ('\0'). Mark Williams C uses this representation when a program contains a string constant; for example:

```
"I am a string constant"
```

The address of the first character in the string normally is used as the starting point of the string; note that a pointer to a string holds only this address. Note, too, that an array of 20 characters can hold a string of 19 (*not* 20) non-NUL characters; the 20th character is the NUL that terminates the string.

The following routines are available to help manipulate strings:

index	search string for a character
memchr	search buffer for a character
memcmp	compare two buffers
memcpy	copy one buffer into another
memset	initialize a buffer
pnmatch	match a string pattern
rindex	search string for a character
strcat	concatenate two strings
strchr	find a character in a string
strcmp	compare two strings
strcpy	copy one string into another
strcspn	return length for which strings do not match
strerror	translate error number into string
strlen	measure a string
strncat	concatenate two strings
strncmp	compare two strings
strncpy	copy one string into another
strpbrk	find first occurrence of any character in string
strrchr	find rightmost occurrence of character
strspn	return length for which strings match
strstr	find one string within another
strtok	break a string into tokens

Example

This example reads from **stdin** up to *NNAMES* names, each of which is no more than *MAXLEN* characters long. It then removes duplicate names, sorts the names, and writes the sorted list to the standard output. It demonstrates the functions **shellsort, strcat, strcmp, strcpy,** and **strlen**.

```
#include <stdio.h>

#define NNAMES 512
#define MAXLEN 60

char *array[NNAMES];
char first[MAXLEN], mid[MAXLEN], last[MAXLEN];
char *space = " ";

extern int strcomp();
extern char *strcat();

main() {
        register int index, count, inflag;
        register char *name;

        count = 0;
        while (scanf("%s %s %s\n", first, mid, last) == 3)
        {
                strcat(first, space);
                strcat(mid, space);
                name = strcat(first, (strcat(mid, last)));
                inflag = 0;
                for (index=0; index < count; index++)
                        if (strcmp(array[index], name) == 0)
                                inflag = 1;
                if (inflag == 0)
                {
                        array[count] = malloc(strlen(name) + 1);
                        strcpy(array[count], name);
                        count++;
                }
        }

        shellsort(array, count-1, sizeof(char *), strcomp);
        for (index=0; index < count; index++)
                printf("%s\n", array[index]);
        exit(0);
}

strcomp(s1, s2)
register char **s1, **s2;
{
        extern int strcmp();
        return(strcmp(*s1, *s2));
}
```

See Also

ASCII, Lexicon

Notes

The draft ANSI standard for the C language allows adjacent string literals, e.g.:

```
"hello" "world"
```

Mark Williams C now supports this standard. Note, however, that this feature may not be portable to all other compilers. Also, because it departs from the Kernighan and Ritchie description of C, it will generate a warning message if you use the compiler's **-VSBOOK** option.

strip — Command

Strip debug, relocation, and symbol tables from executable file

strip -drs *file ...*

strip removes the symbol table, relocation information, and debug tables from a file. It makes the executable file or object module noticeably smaller.

strip recognizes the following options:

-d Keep debug information. If this option is *not* used, all debug information used by the debuggers **csd** and **db** is removed.

-r Keep relocation information. Note that this is not the GEM-DOS relocation information.

-s Keep the symbol table.

See Also

Notes

Because version 3.0 changes the object format, the edition of **strip** shipped with version 3.0 does not work with objects compiled with Mark Williams C version 2.1.7 or earlier. To convert such objects to a format that **strip** recognizes, use the command **mwtomw**.

strlen — String function (libc)

Measure the length of a string

int strlen(*string***) char *****string;**

strlen measures *string*, and returns its length in bytes, *not* including the NUL terminator. This is useful in determining how much storage to allocate for a string.

Example

For an example of how to use this function, see the entry for **string**. For an example of using this function in a TOS application, see the entry for **Fgetdta**.

See Also

string

The C Programming Language, page 95

strncat — String function (libc)

Append one string onto another

char *strncat(*string1, string2, n***)**

char ***string1*, ***string2*; **unsigned *n*;

strncat copies up to *n* characters from *string2* onto the end of *string1*. It stops when *n* characters have been copied or it encounters a NUL character in *string2*, whichever occurs first, and returns the modified *string1*.

Example

For an example of this function, see the entry for **strncpy**.

See Also

strcat, string

Notes

string1 should point to enough space to hold itself and *n* characters of *string2*. If it does not, a portion of the program or operating system may be overwritten.

strncmp – String function (libc)

Compare two strings

int strncmp(*string1*, *string2*, *n***)**
char ****string1*, ******string2*; **unsigned *n*;

strncmp compares lexicographically the first *n* bytes of *string1* with *string2*. Comparison ends when *n* bytes have been compared, or a NUL character encountered, whichever occurs first. **strncmp** returns zero if the strings are identical, returns a number less than zero if *string1* occurs earlier alphabetically than *string2*, and returns a number greater than zero if it occurs later. This routine is compatible with the ordering routine needed by **qsort**.

Example

For an example of this function, see the entry for **strncpy**.

See Also

memcmp, strcmp, strcspn, string, strspn, strstr

strncpy – String function (libc)

Copy one string into another

char *strncpy(*string1*, *string2*, *n***)**
char ****string1*, ******string2*; **unsigned *n*;

strncpy copies up to *n* bytes of *string2* into *string1*, and returns *string1*. Copying ends when *n* bytes have been copied or a NUL character has been encountered, whichever comes first. If *string2* is less than *n* characters in length, *string2* is padded to length *n* with one or more NUL bytes.

Example

This example, called **swap.c**, reads a file of names, and changes them from the format

```
first_name  [middle_initial] last_name
```


to the format

```
last_name, first_name [middle_initial]
```

It demonstrates **strncpy, strncat, strncmp,** and **index**.

```
#include <stdio.h>
#define NNAMES 512
#define MAXLEN 60

char *array[NNAMES];
char gname[MAXLEN], lname[MAXLEN];
extern int strncmp(), strcomp();
extern char *strcpy(), *strncpy(), *strncat(), *index();

main(argc, argv)
int argc; char *argv[];
{
        FILE *fp;
        register int count, num;
        register char  *name, string[60], *cptr, *eptr;
        unsigned glength, length;

        if (--argc != 1)
        {
                fprintf (stderr, "Usage: swap filename\n");
                exit(1);
        }

        if ((fp = fopen(argv[1], "r")) == NULL)
                printf("Cannot open %s\n", argv[1]);
        count = 0;

        while (fgets(string, 60, fp) != NULL)
        {
                if ((cptr = index(string, '.')) != NULL)
                {
                        cptr++;
                        cptr++;
                } else if ((cptr = index(string,' ')) != NULL)
                        cptr++;

                strcpy(lname, cptr);
                eptr = index(lname, '0');
                *eptr = ',';

                strcat(lname," ");
                glength = (unsigned)(strlen(string) - strlen(cptr));
                strncpy(gname, string, glength);

                name = strncat(lname, gname, glength);
                length = (unsigned)strlen(name);
                array[count] = malloc(length + 1);

                strcpy(array[count],name);
                count++;
        }
```

```
        for (num = 0; num < count; num++)
                printf("%s\n", array[num]);
        exit(0);
}
```

See Also

memcpy, strcpy, string

Notes

string1 must point to enough space to hold itself and *string2*; otherwise, a portion of the program or operating system may be overwritten.

strpbrk — String function

Find first occurrence in a string of any character from another string
char *strpbrk(*string1, string2***)**
char ****string1*, ****string2***;**

strpbrk returns a pointer to the first character in *string1* that matches any character in *string2*. It returns NULL if no character in *string1* matches a character in *string2*. The set of characters that *string2* points to is sometimes called the "break string". For example,

```
char *string = "To be, or not to be: that is the question.";
char *brkset = ",;";
strpbrk(string, brkset);
```

returns the value of the pointer **string** plus six; this points to the comma, which is the first character in the area pointed to by **string** to match any character in the string pointed to by **brkset**.

See Also

strchr, strcspn, string, strpbrk, strrchr, strspn, strstr, strtok

Notes

strpbrk resembles the function **strtok** in functionality, but unlike **strtok**, it preserves the contents of the strings being compared. It also resembles the function **strchr**, but lets you search for any one of a group of characters, rather than for one character alone.

strrchr — String function

Search for rightmost occurrence of a character in a string
char *strrchr(*string, character***)**
char ****string*; int** *character***;**

strrchr looks for the last, or rightmost, occurrence of *character* within *string*. *character* is declared to be an **int**, but is handled within the function as a **char**. Another way to describe this function is to say that it performs a reverse search for a character in a string.

strrchr returns a pointer to the rightmost occurrence of *character*, or NULL if *character* could not be found within *string*.

See Also

memchr, strchr, strcspn, string, strpbrk, strspn, strstr, strtok

Notes

strrchr is identical to the function **rindex**, which is included with Mark Williams C.

strspn — String function

Return length for which one string includes characters in another
unsigned int strspn(*string1, string2***)**
char ****string1*, ****string2***;**

strspn returns the length for which *string1* initially consists only of characters that are found in *string2*. For example,

```
char *s1 = "hello, world";
char *s2 = "kernighan & ritchie";
strcspn(s1, s2);
```

returns two, which is the length for which the first string initially consists of characters found in the second.

See Also

memchr, strchr, strcspn, string, strpbrk, strrchr, strstr, strtok

strstr — String function

Find one string within another
char *strstr(*string1, string2***)**
char ****string1*, ****string2***;**

strstr looks for *string2* within *string1*. The terminating null character is not considered part of *string2*.

strstr returns a pointer to where *string2* begins within *string1*, or NULL if *string2* does not occur within *string1*.

For example,

```
char *string1 = "Hello, world";
char *string2 = "world";
strstr(string1, string2);
```

returns **string1** plus seven, which points to the beginning of **world** within **Hello, world**. On the other hand,

```
char *string1 = "Hello, world";
char *string2 = "worlds";
strstr(string1, string2);
```

returns NULL because **worlds** does not occur within **Hello, world**.

See Also

memchr, strchr, strcspn, string, strpbrk, strrchr, strspn, strtok

struct — C keyword

Data type

struct is a C keyword that introduces a structure. The following is an example of how **struct** can be used in the description of a name and address file:

```
struct address {
        char firstname[10];
        char lastname[15];
        char street[25];
        char city[10];
        char state[2];
        char zip[5];
        int  salescode;
};
```

The C Programming Language prohibits the assignment of structures, the passing of structures to functions, and the returning of structures by functions. Mark Williams C allows one structure to be assigned to another, provided the two structures are of the same type. It also allows structures to be passed by and returned by functions. These features are supported by most compilers, but users should be aware that their use can cause problems in porting code to some compilers.

See Also

array, C keywords, C language, field, structure
The C Programming Language, page 119

structure — Definition

A **structure** is a set of variables that has been given a name and can be manipulated as a single entity. The variables may be of different data types. Structures are a convenient way to deal with data elements that belong together, such as names and addresses, employee descriptions, or sales and inventory information.

See Also

field, record, struct
The C Programming Language, page 119

structure assignment — Technical information

The C Programming Language forbids structure assignment, the passing of structures to functions, and returning structures from functions (as opposed to the passing or returning of *pointers* to structures). Mark Williams C lifts these restrictions.

Some C compilers transform structure arguments and structure returns into structure pointers. Note that the use of structure assignment, structure arguments, or structure returns may create problems when porting the code to another C compiler.

See Also

portability, struct, structure

Notes

Note that because this feature deviates from the description of the C language found in *The C Programming Language*, compiling with the **-VSBOOK** option will flag all points where it occurs in your program.

SUFF — Environmental variable

SUFF names a set of suffixes that **msh** will automatically append to command names. The suffixes are appended to the given command name when searching the directories named in the **PATH** environmental variable. For example, typing

```
setenv PATH=\bin,,\lib
setenv SUFF=,.prg,.tos,.ttp
```

means that when you give **msh** the command

```
foo
```

it will look for a file with one of the following names:

```
\bin\foo
\bin\foo.prg
\bin\foo.tos
\bin\foo.ttp
foo
foo.prg
foo.tos
foo.ttp
\lib\foo
\lib\foo.prg
\lib\foo.tos
\lib\foo.ttp
```

The file names are searched for in the order given above, and **msh** stops searching after finding the first file that matches the requested pattern.

It is set the with **setenv** command.

See Also

msh, setenv

Super — gemdos function 32 (osbind.h)

Enter privilege mode
long Super(*stack***) char ****stack***;**

Super manipulates the Atari ST's privilege mode. *stack* points to a new supervisor stack. If the machine is presently set in user mode, it switches to supervisor mode; if in supervisor mode, it returns to user mode.

Example

This example changes the floppy write verify flag so floppy writes are not automatically verified. This speeds up processing, but can be dangerous, and is not recommended.

```
#include <osbind.h>
#define FVERIFY ((short *) 0x0444L)

main()
{
      long save_ssp;

      save_ssp = Super(0L);      /* Switch to system mode */
      *FVERIFY = 0; /* Clear the word. */
      Super(save_ssp);     /* Restore system */
}
```

See Also

gemdos, TOS

Notes

Super has been documented elsewhere as returning the supervisor/user mode flag if *stack* is set to -1L; however, it crashes the system instead. With systems that have TOS in ROMs, *stack* should be set to one to perform this task.

Supexec — xbios function 38 (osbind.h)

Run a function under supervisor mode
#include <osbind.h>
#include <xbios.h>
unsigned long Supexec(*address***)**
int ***address*;**

Supexec invokes supervisor mode, and allows you to run a routine under it. *address* is the address of the function to be run.

The **Supexec** function has two features that are not widely known but could prove useful in your programs.

The first is that any value returned by function run under under **Supexec** is returned untouched by the **xbios** trap.

Example

The following example uses the return value of a function run under **Supexec** to time execution speeds:


```
/* Redefine Supexec() function to get long return value */
#include <osbind.h>
#undef Supexec
#define Supexec(a) xbios(38,a)

/* Return the system 200 hz timer tick count */
long read_ticks() { return *((long *)0x4ba); }

/* Return microseconds that (*f)() takes to execute */
long time_function(f) int (*f)();
{
      register int ntimes = 4*5*1000;
      long tstart = Supexec(read_ticks);
      while (--ntimes >= 0) (*f)();
      return (Supexec(read_ticks) - tstart + 2) >> 2;
}

/* Some functions to time */
null_function() { return; }
int ia = 0x0123, ib = 0x3210;
int iret_function() { return ia,ib; }
int iadd_function() { return ia+ib; }
int isub_function() { return ia-ib; }
int imul_function() { return ia*ib; }
int idiv_function() { return ia/ib; }

long la = 0x01234567L, lb = 0x76543210L;
long lret_function() { return la,lb; }
long ladd_function() { return la+lb; }
long lsub_function() { return la-lb; }
long lmul_function() { return la*lb; }
long ldiv_function() { return la/lb; }

double da = 12340.0, db = 4321.0;
double dret_function() { return da,db; }
double dadd_function() { return da+db; }
double dsub_function() { return da-db; }
double dmul_function() { return da*db; }
double ddiv_function() { return da/db; }

/* Report the times for the functions */
main() {
      printf("null %ld microseconds\n", time_function(null_function));
      printf("iret %ld microseconds\n", time_function(iret_function));
      printf("iadd %ld microseconds\n", time_function(iadd_function));
      printf("isub %ld microseconds\n", time_function(isub_function));
      printf("imul %ld microseconds\n", time_function(imul_function));
      printf("idiv %ld microseconds\n", time_function(idiv_function));

      printf("lret %ld microseconds\n", time_function(lret_function));
      printf("ladd %ld microseconds\n", time_function(ladd_function));
      printf("lsub %ld microseconds\n", time_function(lsub_function));
      printf("lmul %ld microseconds\n", time_function(lmul_function));
      printf("ldiv %ld microseconds\n", time_function(ldiv_function));
```

```
        printf("dret %ld microseconds\n", time_function(dret_function));
        printf("dadd %ld microseconds\n", time_function(dadd_function));
        printf("dsub %ld microseconds\n", time_function(dsub_function));
        printf("dmul %ld microseconds\n", time_function(dmul_function));
        printf("ddiv %ld microseconds\n", time_function(ddiv_function));
        return 0;
}
```

The second feature is that a function run under **Supexec** can be passed parameters by including them in the call to the **xbios** trap. The first parameter to the function will always be a long pointer to itself. Any subsequent parameters will be available if they are declared in normal C style.

Example

The following example passes three arguments to a function run under **Supexec** to copy a block of low memory to a user-supplied buffer.

```
/* Redefine Supexec() to pass 3 arguments */
#include <osbind.h>
#undef Supexec
#define Supexec(a,b,c,d) xbios(38,a,b,c,d)

/* Word copy function with dummy parameter */
supercopy(self,destp,srcp,nwds) register int (*self)(), *destp, *srcp, nwds;
{
        while (--nwds >= 0) *destp++ = *srcp++;
}

/* Copy the process dump area to our data space and print it */
main() {
        int proc[64];/* More or less */
        Supexec(supercopy,proc,0x380L,64);
        for (i = 0; i < 64; i += 4)
                printf("%04x %04x %04x %04x\n", proc[i], proc[i+1], proc[i+2],
                        proc[i+3]);
        return 0;
}
```

See Also

TOS, xbios

Sversion — gemdos function 48 (osbind.h)

Get the version number of TOS

#include <osbind.h>
int Sversion()

Sversion gets and returns the current TOS version number.

Example

This example prints the TOS version number on the standard output.


```
#include <osbind.h>

main() {
        union {
                struct {
                unsigned minor:8;
                unsigned major:8;
                } braker;
                int all;
        } versn;

        versn.all = Sversion();
        printf("TOS/GEMDOS version %2X revision %2x.\n",
                versn.braker.major, versn.braker.minor);
}
```

See Also

gemdos, TOS

swab — General function (libc)

Swap a pair of bytes

void swab(*src***, *dest*, *nb*) char *src, *dest; unsigned** *nb*;

The ordering of bytes within a word differs from machine to machine. This may cause problems when moving binary data between machines. **swab** interchanges each pair of bytes in the array *src* that is *n* bytes long, and places the result into the array *dest*. The length *nb* should be an even number, or the last byte will not be touched. *src* and *dest* may be the same place.

Example

This example prompts for an integer; it then prints the integer both as you entered it, and as it appears with its bytes swapped.

```
#include <stdio.h>

main() {
        int word;

        printf("Enter an integer: \n");
        scanf("%d", &word);
        printf("The word is 0x%x\n", word);
        swab(&word, &word, 2);
        printf("The word with bytes swapped is 0x%x\n", word);
}
```

See Also

byte ordering

switch — C keyword

Test a variable against a table

switch is a C keyword that lets you perform a number of tests on a variable in a convenient manner. For example,

```
while(foo < 10)
        switch(foo) {
        case 1:
                dosomething();
                break;
        case 2:
                somethingelse();
                break;
        case 3:
                anotherthing();
                break;
        default:
                break;
        }
}
```

is equivalent to

```
while(foo < 10) {
        if(foo == 1) {
                dosomething();
                continue;
        } else if(foo == 2) {
                somethingelse();
                anotherthing();
                continue;
        } else if(foo == 3) {
         /* Note: compiler eliminate duplicate code */
                anotherthing();
                continue;
        } else
                break;
}
```

Note that **switch** is always used with the **case** statement, and nearly always with the **default** statement.

See Also

break, C keywords, C language, case, default, keyword, while
The C Programming Language, page 54

system — General function (libc)

Pass a command to TOS for execution
int system(*commandline***) char ***commandline***;**

system passes *commandline* to the Mark Williams microshell, which loads it into memory and executes it. **system** executes commands exactly as if they had been typed directly into the shell.

system uses the environmental variables **SHELL** and **PATH** to find the command line processor. **SHELL** defaults to **msh.prg**, but you can substitute any other command processor that can evalutate the command lines passed through **system** by **make**, **me**, or **db**.

Example

This example uses **system** to call the **msh** command **ls** to list all C programs in the present directory. It redirects its output into the file **list.fil**.

```
#include <stdio.h>

FILE *newfp;
int oldstdout;

main()
{
        extern int system();

        ropen("list.fil");
        system("dir *.c");
        rclose();
}

/*
 * Redirect stdout prior to system() call.
 * You can't redirect child process's I/O
 * but you can redirect main()'s and let the child inherit it.
 */

ropen(tofile)
char *tofile;
{
        if ((newfp = fopen(tofile, "wr")) == NULL)
                fatal("cannot open output file \"%s\"", tofile);

        /* Duplicate stdout so it can be restored later */
        if ((oldstdout = dup(fileno(stdout))) == -1)
                fatal("dup failed");

        /* Force duplication of new file handle as stdout */
        if (dup2(fileno(newfp), fileno(stdout)) == -1)
                fatal("dup2 failed");

}

/*
 * Terminate redirection
 */

rclose()
{
        /* Restore old stdout */
        if (dup2(oldstdout, fileno(stdout)) == -1)
                fatal("dup2 failed");
        /* Close the extra handle */
        if (close(oldstdout) != 0)
                fatal("cannot close old stdout");
        fclose(newfp);
}
```

```
/*
 * Fatal error
 */
fatal(p)
char *p;
{
        fprintf(stderr, "redirect: %r\n", &p);
        exit(1);
}
```

See Also

execve, exit, msh, Pexec

Notes

No shell variable that has been set with the **set** command is duplicated.

system variables — Technical information

The TOS operating system uses a number of "magic locations" where it stores key system variables. By using the **peek** and **poke** routines included with Mark Williams C, you can alter these variables directly, to customize TOS more closely to your needs and tastes.

You can safely manipulate the address 0x0 to 0x800 only when your program is in supervisor mode; you can enter supervisor mode by calling the **gemdos** function **super**.

The following table gives each "magic location", the common Atari mnemonic for it (should you wish to build a header file to work with these locations), the length of the system variable, and a brief description.

0x400/etv_timer/long
Points to the timer event handler.

0x404/etv_critic/long
Points to the critical error handler.

0x408/etv_term/long
Points to routine that ends a program.

0x04C/etv_xtra/long

0x420/memvalid/int
Check if the memory controller's configuration is valid.

0x424/memcntrl/int
Copy of configuration value in memory controller.

0x426/resvalid/long
If proper value given, jump is made to reset routine pointed to by address 0x42A.

0x42A/resvector/long
Address of reset routine.

0x42E/phystop/long
Top of RAM.

0x432/membot/long
Points to beginning of transient program area.

0x436/memtop/long
Points to end of transient program area.

0x43A/memval2/long
This if set properly, declares memory configuration to be valid.

43E/flock/int
If set to a value other than zero, disk access is in progress.

0x440/seekrate/int
Set disk drive seek rate, as follows: zero, six milliseconds; one, 12 milliseconds; two, two milliseconds; and three, three milliseconds.

0x442/timr_ms/int
Clock rate, in microseconds.

0x444/fverify/int
If set to a value other than zero, every disk write access is verified.

0x446/bootdev/int
Number of disk drive from which operating system was loaded.

0x448/palmode/int
If set to a value other than zero, system is in PAL mode (50 Hz); otherwise, system is in NTSC mode.

0x44A/defshiftmd/int
If Atari shifted from monochrome to color, new resolution is set here: zero indicates low resolution; one, medium resolution.

0x44C/sshiftmd/int
Screen resolution, as follows: zero, low resolution; one, medium resolution; two, high resolution.

0x44E/v_bas_ad/long
Points to logical screen base. Address always begins on a 256-byte boundary.

0x452/vblsem/int
If set to zero, vertical blank routines are not executed.

0x454/nvbls/int
Number of vertical blank routines queued for execution.

0x456/vblqueue/long
Points to the list of routines queued to be executed during vertical blanking.

0x45A/colorptr/long
If other than zero, holds pointer to color palette to be executed during next vertical blank.

0x45E/screenpt/long
Points to beginning of video RAM.

0x462/vbclock/long
Number of vertical blank interrupt routines.

0x466/frclock/long
Number of vertical blank routines executed.

0x46A/hdv_init/long
Points to hard-disk initialization.

0x46E/swv_vec/long
Points to routine to change screen resolution.

0x472/hdv_bpb/long
Points to fetch BIOS parameter block for hard disk.

0x476/hdv_rw/long
Points to read/write routine for hard disk.

0x47A/hdv_boot/long
Points to routine to reboot hard disk.

0x47E/hdv_mediach/long
Points to routine to handle medium change for hard disk.

0x482/cmdload/int
If set to a value other than zero, system will attempt to load file **command.prg** after TOS has been loaded.

0x484/conterm/char
Set console attributes. This is a byte-length bit map, whose first four bits signify the following: bit 0, toggle key click; bit 1, toggle key repeat; bit 2, toggle bell when **<ctrl-G>** is typed; and bit 3, toggle returning **Kbshift** in bits 24-31 for the function **Conin**.

0x48E/themd/four longs
Memory descriptor filled by function **Getmpb**.

0x4A2/savptr/long
Pointer to save area for process registers after a BIOS call.

0x4A6/nflops/int
Number of floppy disk drives.

0x4AE/sav_context/long
Points to temporary areas used by exception-handling routines.

0x4B2/bufl0/long
Points to head of data sector list.

0x4B2/bufl1/long
Points to head of file allocation table (FAT).

0x4BA/hz_200/long
Counter for 200-Hz system clock.

0x4BE/the_env/four chars
Default environment string, four NULs.

0x4C2/drvbits/long
Bit map indicating connected drives: bit zero indicates drive A, bit one indicates drive B, etc.

0x4C6/dskbufp/long
Pointer to 1,024-byte disk buffer.

0x4EE/prt_cnt/int
If set to one, a dump of the current screen is sent to the printer port. Dump can be aborted by typing **help** and **alt** keys simultaneously.

0x4F2/sysbase/long
Pointer to beginning of operating system.

0x4F6/shell_p/long
Pointer to global shell information.

0x4FA/end_os/long
Pointer to end of operating system.

0x4FE/exec_os/long
Pointer to start of AES.

Example

The following example **pokes** address **0x484** to turn off the key click:

```
main()
{
        pokeb(0x484L, peekb(0x484L) & ~1);
}
```

See Also

memory allocation, peekb, peekl, peekw, pokeb, pokel, pokew, TOS