B

backspace — Character constant

Mark Williams C recognizes the literal character '\b' for the ASCII space character BS (octal 010). This character may be used as a character constant or in a string constant.

Example

The following example prints a string, then backspaces over it and prints another message.

```
main()
{
        printf("BLINK!\b\b\b\b\bhello, world\n");
}
```

See Also

ASCII, character constant

basepage.h — Header file

Define TOS base page structure
#include <basepage.h>

basepage.h is a header file that defines the TOS base page structure. Its text is as follows:

```
#ifndef BASEPAGE_H
#define BASEPAGE_H
typedef struct {
        long        p_lowtpa;       /* Low transient program area */
        long        p_hitpa;        /* High transient program area */
        long        p_tbase;        /* Text segment base */
        long        p_tlen;         /* Text segment length */
        long        p_dbase;        /* Data length base */
        long        p_dlen;         /* Data length length */

        long        p_bbase;        /* Bss segment base */
        long        p_blen;         /* Bss segment length */
        long        p_fxx0[3];      /* Fill area one */
        long        p_env;          /* Environment string pointer */
        long        p_fxx1[20];     /* Fill block two */
        char        p_cmdlin[128];  /* Command line */
} BASEPAGE;
extern BASEPAGE _start[];
#define BP (&_start[-1])
#endif
```

See Also

header file, TOS

Bconin — bios function 2 (osbind.h)

Receive a character
#include <osbind.h>
#include <bios.h>
long Bconin(*handle***) int** *handle*;

Bconin receives a character from a peripheral device. *handle* is an integer that indicates which device is being read, as follows:

0	**prn:** (the line printer)
1	**aux:** (the auxiliary serial port)
2	**con:** (the console)
3	the MIDI port
4	the intelligent keyboard (output only)
5	the raw screen (output only)

When **Bconin** reads from **con:**, it returns the key's raw scan code in the low byte of the high word and either an ASCII character or zero in the low byte of the low word, depending upon whether the key typed generates an ASCII character or not; when it is reading from **aux:**, it returns the character in the low byte of the low word.

For a table of keyboard scan codes, see the entry for **keyboard**. Note, too, that this function is unaffected by redirection of either **con:** or **aux:**.

Example

This example emulates a simple dumb terminal. It demonstrates the functions **Bconin**, **Bconout**, **Bconstat**, **Bcostat**, and **Ptermo**.

```
#include <osbind.h>
#include <bios.h>

main()
{
        register long c;

        for (;;) {
                if (Bconstat(BC_CON)) {
                        c = Bconin(BC_CON);

                        if ((int)c == 0) {
                                c >>= 16;
                                if (c == KC_UNDO)
                                        break;
                                else
                                        Bconout(BC_CON, '\a');
```

```
                } else {
                        while (Bcostat(BC_AUX) == 0)
                                ;
                        Bconout(BC_AUX, (int)c);
                }
        }

        if (Bconstat(BC_AUX)) {
                c = Bconin(BC_AUX);
                Bconout(BC_CON, (int)c);
        }
    }
    Pterm0();
}
```

See Also

aux:, Bconout, Bconstat, Bcostat, bios, con:, keyboard, TOS

Bconout – bios function 3 (osbind.h)

Send a character to a peripheral device

#include <osbind.h>
#include <bios.h>
void Bconout(*handle, character***) int** *handle, character***;**

Bconout sends characters to an output device. *handle* is an integer that indicates to which device to send characters, as follows:

0	**prn:** (the line printer)
1	**aux:** (the auxiliary serial port)
2	**con:** (the console)
3	the MIDI port
4	the intelligent keyboard (output only)
5	the raw screen (output only)

character is the character being output, which is encoded in the lower eight bits of the integer. **Bconout** returns nothing. This function is unaffected by redirection of the logical devices **con:** or **aux:**.

If *handle* is set to five, characters are displayed on the screen as with device number 2, but control characters are not interpreted. This allows the display of graphics characters from the Atari character set, in the range of one through 31.

Example

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

See Also

Bconin, Bconstat, Bcostat, bios, TOS

Bconstat – bios function 1 (osbind.h)

Return the input status of a peripheral device

#include <osbind.h>
#include <bios.h>
long Bconstat(*device***) int** *device***;**

Bconstat reads the input status of the specified peripheral device. *device* is an integer that encodes the the desired device, as follows:

0	**prn:** (the line printer)
1	**aux:** (the auxiliary serial port)
2	**con:** (the console)
3	the MIDI port
4	the intelligent keyboard (output only)
5	the raw screen (output only)

Bconstat returns -1 if at least one character is ready to be handled, and zero if no characters are ready. This function is unaffected by redirection.

Example

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

See Also

Bconin, Bconout, Bcostat, bios, TOS

Bcostat – bios function 8 (osbind.h)

Read the output status of a peripheral device

#include <osbind.h>
#include <bios.h>
long Bcostat(*handle***) int** *handle***;**

Bcostat reads the output status of a peripheral device. *handle* is a number that indicates the device to be checked, as follows:

0	**prn:** (the line printer)
1	**aux:** (the auxiliary serial port)
2	**con:** (the console)
3	the MIDI port
4	the intelligent keyboard (output only)
5	the raw screen (output only)

Bcostat returns -1 if the device is ready, 0 if it is not. This function is unaffected by redirection.

Example

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

See Also

Bconin, Bconout, Bconstat, bios, TOS

BIOS – Definition

BIOS is an acronym for *basic input/output system*. In most machines, the BIOS consists of a group of routines carried in the read-only memory (ROM). These routines contain basic instructions for accessing the various aspects of the hardware.

See Also

bios, STDIO

bios – TOS function

Call an input/output routine in the TOS BIOS
#include <osbind.h>
extern long bios(*n, f1, f2 ... fn*);

bios allows you to call an input/output function directly in the Atari BIOS. It works by building a stack frame and executing trap no. 13. Unless the **-VNOTRAP** option is used when compiling a program, the instruction **jsr bios_** is replaced by a trap no. 13 instruction.

n is the number of the function, and *f1* through *fn* are the parameters to be used with the routine. In most circumstances, it is unnecessary to call **bios**, for the header file **osbind.h** defines a number of functions for it. All structures and constants used by these functions are contained in the header file **bios.h**.

The following functions call **bios** to deal with the peripheral devices. The first column gives its function number, the second its name, and the third a brief description:

2	**Bconin**	receive a character
3	**Bconout**	output a character
1	**Bconstat**	return input status of device
8	**Bcostat**	return output status of device
10	**Drvmap**	return map of logical drives
7	**Getbpb**	return pointer to BIOS parameter block
0	**Getmpb**	copy memory parameter block
11	**Getshift**	get/set status for shift/alt/control keys
9	**Mediach**	check if medium has been changed
4	**Rwabs**	read/write a disk drive
5	**Setexc**	set an exception vector
6	**Tickcal**	return system timer's calibration

See Also

osbind.h, TOS

Notes

No **bios** function checks for incorrect device numbers. Passing an invalid device number to a routine may crash the system.

bios and **xbios** traps can be nested to a level of three deep. This occurs either when an interrupt-level routine calls a **bios** or **xbios** function while a **bios** or **xbios** function is executing, or when a **bios** or **xbios** function itself traps to the **bios** or **xbios**. A dangerous situation may occur if a **bios** or **xbios** routine in a routine that is executed by an interrupt handler or can be invoked asynchronously; in these situations, the level of nesting can quickly exceed the limit of three.

All **bios** functions are unbuffered. Combining them with buffered routines, such as those in the STDIO, **gemdos**, or GEM AES libraries, will lead at best to unpredictable results.

bios.h – Header file

Declare bios constants and structures
#include <bios.h>

bios.h is a header file that includes all constants and structures used by the GEMDOS **bios** functions. For a list of these functions, see the entry for **bios**.

See Also

bios, header file, TOS, xbios.h

Bioskeys – xbios function 24 (osbind.h)

Reset the keyboard to its default
#include <osbind.h>
#include <xbios.h>
void Bioskeys()

Bioskeys resets the keyboard to its default settings, and returns nothing. It undoes whatever changes were made with the function **Keytbl**.

Example

```
#include <osbind.h>
main() {
      Bioskeys();
}
```

See Also

Keytbl, TOS, xbios

bit – Definition

bit is an abbreviation for **b**inary digi**t**. It is the basic unit of data processing. A bit can have a value of either zero or one. Bits can be concatenated to form bytes.

A bit can be used either as a placeholder to construct a number with an absolute

value, or as a flag whose value has a particular meaning under specially defined circumstances. In the former use, a string of bits builds an integer. In the latter use, a string of bits forms a **map**, in which each bit has a meaning other than its numeric value.

See Also

bit map, byte, integer, nybble

bit map — Definition

A **bit map** is a string of bits in which each bit has a symbolic, rather than numeric, value. For example, the **Drvmap** function returns a 16-bit map of the active drives on the Atari ST. The bits indicate which of 16 possible disk drives is available, with bit 0 (i.e., 1<<0) corresponding to drive A, bit 1 to drive B, etc.

See Also

bit, byte
The C Programming Language, page 136

Notes

C permits the manipulation of bits within a byte through the use of bit field routines. These generate code rather than calls to routines. Bit fields are generally less efficient than masking because they always generate masking and shifting.

Blitmode — xbios function 64 (osbind.h)

Get/set blitter configuration
int Blitmode(*flag***) int** *flag***;**

Blitmode gets or sets the configuration of the blitter chip. The bits of *flag* encode the new setting for the blitter chip, as follows:

0	zero, set blit mode to software; one, set to hardware
1-14	reserved, may be anything
15	reserved, must be zero

If *flag* is set to -1, the blitter mode is not reset and **Blitmode** returns an **int** whose bits encode the current configuration of the blitter, as follows:

0	zero, in software; one, in hardware
1	zero, no blit chip installed; one, blit chip installed
2-14	reserved, may be anything
15	always zero

Example

This example returns the current blitter mode.


```
#include <osbind.h>
main()
{
        int setting = Blitmode(-1);
        /* check if blitter chip is present */
        if (setting & 0x02)
        {
                printf("A blitter chip is present.\n");
                /* check if mode is hardware or software */
                if (setting & 0x01)
                        printf("The blit mode is in hardware.\n");
                else
                        printf("The blit mode is in software.\n");
        }
        else
                printf("No blitter chip is present.\n");
}
```

See Also

TOS, xbios

Notes

The reserved bits will be used in future models of the Atari ST.

If you attempt to set the blitter mode on a machine that does not have the blitter chip, the mode will always be set to zero (in software). This function works only on machines that have the Atari blitter ROMs. On machines with earlier ROMs, the function returns 0x40 (the **xbios** function number).

bombs — Technical information

68000 processor exceptions

When a program goes seriously wrong on the Atari ST, TOS takes the following actions:

1. It stores a description of the program's state in a buffer in low memory.
2. It displays one or more "cherry bombs" on the screen; persons with older versions of the operating system may see little "mushroom clouds" instead. The number of bombs seen is equal to the number of the processor exception, as follows:

2	Bus error
3	Address error
4	Illegal instruction
5	Zero divide
6	CHK instruction
7	TRAPV instruction
8	Privilege violation
9	Trace
10	Line A emulator

11 Line F emulator
12 Reserved
13 Reserved
14 Reserved (000, 008), format error (010)

3. TOS attempts to terminate the program and continue processing.

You use the debugger **db** to display the program state saved in low memory by TOS. Use the following commands:

db -k	enter **db**
:r	display contents of registers
:f	print type of fault
:q	quit

This prints the processor registers at the time of the fault and identifies the fault. The exceptions that occur on the 68000 processor are listed in the header file **signal.h**.

See Also

db, signal.h, TOS

boot — Definition

Boot is an abbreviation for *bootstrapping procedure*. This refers to the procedure by which a computer loads the operating system, organizes and tests memory, and initializes peripheral devices.

Some operating systems use the term *warm boot* to refer to a second-stage bootstrapping procedure. An operating system may execute a warm boot to restore portions of the operating system that may have been overlaid by user code during the operation of a program, or to reinitialize the system without going through the entire boot procedure.

See Also

exit

Notes

TOS does not warm boot on program termination.

break — C keyword

Exit from loop or switch statement

break is a C statement that causes an immediate exit from a **switch** sequence, or from a **while**, **for**, or **do** loop.

Example

For an example of this instruction, see the entry for **VDI**.

See Also

C keywords, C language
The C Programming Language, page 56

buffer — Definition

A **buffer** is a portion of memory reserved for a particular purpose. In the context of C, a buffer most often is an area set aside to hold data read from or to be written to a file stream. Often, although not always, this involves setting aside a portion of the arena with **malloc** or its related functions.

Many operating systems automatically place data from a peripheral device into a buffer. Buffers normally can be cleared with **fflush**, by pressing the carriage return key on routines that perform input, or by sending a newline character on routines that perform output. The function **close**, which closes a file, will flush all buffers associated with that file. **exit** calls **close**.

Combining unbuffered and buffered I/O functions on the same file or device within one program will produce results that are at best unpredictable.

On the Atari ST, all STDIO routines use buffering by default. **stdin** and **stdout** are buffered, but **stderr** is not. Buffering can be turned off with the function **setbuf**. All Atari BIOS functions that perform I/O are not buffered.

Example

The following example demonstrates what does and does not happen when you use **fflush** with the output buffer.

```
#include <stdio.h>
main()
{
        extern char *malloc();      /* declare malloc & what it returns */
        char *buffer;

        /* use malloc() to create a 120-char buffer */
        if ((buffer = malloc(120)) == NULL)
        {
                /* if malloc() fails, bail out */
                fprintf(stderr, "malloc failed\n");
                exit(1);
        }

        printf("Type your name:  ");
        fflush(stdout);      /* flush stdout buffer */
        gets(buffer);/* copy string into malloc'd buffer */
        printf("Your name is %s\n", buffer);
}
```

See Also

arena, array, Cconrs, Cconws, close, exit, fflush, malloc, setbuf, STDIO
The C Programming Language, page 173

byte — Definition

A **byte** is a group of eight bits, which often is used to encode a character or a small integer quantity. Note that for C, the term "byte" has no meaning. C defines data types as being multiples of the data type **char**, and what a **char** is depends on the hardware. Although a **char** is often defined as being eight bits long, the same as a byte, this definition is not universal.

See Also

bit, char, data formats, nybble

byte ordering — Technical information

Byte ordering is the order in which a given machine stores successive bytes of a multibyte data item. Note that different machines order bytes differently.

The following example displays a few simple examples of byte ordering:

```
main()
{
        union
        {
                char b[4];
                int i[2];
                long l;
        } u;
        u.l = 0x12345678L;

        printf("%x %x %x %x\n",
                u.b[0], u.b[1], u.b[2], u.b[3]);
        printf("%x %x\n", u.i[0], u.i[1]);
        printf("%lx\n", u.l);
}
```

When run on the 68000 or the Z8000, the program gives the following results:

```
12 34 56 78
1234 5678
12345678
```

As you can see, the order of bytes and words from low to high memory is the same as is represented on the screen.

When run on a PDP-11, however, the program gives these results:

```
34 12 78 56
1234 5678
12345678
```


As you can see, the PDP-11 inverts the order of words in memory.

Finally, when the program is run on the i8086 you see these results:

```
78 56 34 12
5678 1234
12345678
```

The i8086 inverts both words and long words.

See Also

C language, canon.h, data formats

C

C keywords — Overview

A **keyword** is a word that is reserved within C, and may not be used to name variables, functions, or macros. Mark Williams C recognizes the following keywords:

alien	**extern**	**signed**
auto	**float**	**sizeof**
break	**for**	**static**
case	**goto**	**struct**
char	**if**	**switch**
const	**int**	**typedef**
continue	**long**	**union**
default	**readonly**	**unsigned**
do	**register**	**void**
double	**return**	**volatile**
else	**short**	**while**
enum		

In conformity with the proposed ANSI standard, the keyword **entry** is no longer recognized. The keywords **const** and **volatile** are now recognized, but not implemented. Mark Williams C recognizes the keywords **readonly** and **alien**, but these are not implemented on the 68000.

See Also

C language

C language — Overview

The following summarizes how Mark Williams C implements the C language.

Identifiers:

Characters allowed: **A-Z**, **a-z**, **_**, **0-9**
Case sensitive.
Number of significant characters in a variable name:
at compile time: **128**
at link time: **16**
C appends '**_**' to end of external identifiers

Reserved identifiers (keywords):

alien	**extern**	**signed**
auto	**float**	**sizeof**
break	**for**	**static**
case	**goto**	**struct**
char	**if**	**switch**
continue	**int**	**typedef**
const	**long**	**union**
default	**readonly**	**unsigned**
do	**register**	**void**
double	**return**	**volatile**
else	**short**	**while**
enum		

In conformity with the proposed ANSI standard, the keyword **entry** is no longer recognized. The keywords **const** and **volatile** are now recognized, but not implemented. The compiler will produce a warning message if the keyword **volatile** is used with the peephole optimizer. Mark Williams C reserves the keywords **readonly** and **alien**, but these are not implemented on the 68000.

Data formats (in bits):

char	8
unsigned char	8
double	64
float	32
int	16
unsigned int	16
long	32
unsigned long	32
pointer	32
short	16
unsigned short	16

float *format:*

DECVAX floating point format:
1 sign bit
8-bit exponent
24-bit normalized fraction with hidden bit
DECVAX double format:
Same as **float**, but with 56 bits of fraction
Reserved values:
+- infinity, -0
All floating-point operations are done as **doubles**

Limits:
Maximum bitfield size: 16 bits
Maximum number of **cases** in a **switch**: no formal limit
Maximum block nesting depth: no formal limit
Maximum parentheses nesting depth: no formal limit
Maximum structure size: no formal limit
Maximum auto array size: 32 kilobytes
Maximum static array size: no formal limit

Preprocessor instructions:

#assert	**#if**
#define	**#ifdef**
#else	**#ifndef**
#elif	**#include**
#endif	**#line**
#file	**#undef**

Structure name-spaces:
Supports both Berkeley, and Kernighan and Ritchie conventions for structure in union.

Register variables:
Five available for **ints** or **longs**
Three available for pointers

Function linkage:
Return values for **ints**, **longs**, or pointers in d0
Return values for **doubles** in d0 and d1
Pointers to returned structures in a0, copied to destination by caller
Parameters pushed on stack in reverse order, **chars** and **shorts** pushed as words, **longs** and pointers pushed as **longs**, structures copied onto stack
Caller must clear parameters off stack
Stack frame linkage is done through a6

Register usage:
d0, d1: Scratch data and function return values
d2: Scratch data
d3, d4, d5, d6, d7: Register variables for **longs** and **ints**
a0, a1, a2: Scratch addresses and function structure return
a3, a4, a5: Register pointers for any type or structure
a6: Call frame linkage pointer
a7: Stack pointer

Special features and optimizations:

- By default, the compiler makes the following substitutions:

```
jsr gemdos_      trap $1
jsr micrortx     trap $5
jsr bios_        trap $13
jsr xbios_       trap $14
```

  This reduces the overhead for system calls and makes the code reentrant (although the system itself may not be). Turn off this feature with the option **-VNOTRAP**.

- Branch optimization is performed: this uses the smallest branch instruction for the required range.
- Unreached code is eliminated.
- Duplicate instruction sequences are removed.
- Jumps to jumps are eliminated.
- Multiplication and division by constant powers of two are changed to shifts when the results are the same.
- Sequences that can be resolved at compile time are identified and resolved.
- Peephole optimization remembers register contents.
- Cross-jumps are eliminated. This changes code like this:

```
        move a, b
        bra  LABEL1
LABEL0: move c, b
        bra  LABEL2
LABEL1:move b, d
        bra  LABEL3
LABEL2:move f, d
        bra  LABEL3
```

to:

```
        move a, b
        move b, d
        bra  LABEL3
LABEL0:move c, b
        move f, d
        bra  LABEL3
```

See Also
byte ordering, calling conventions, data formats, data types, declarations, keywords, Lexicon, memory allocation

cabs — Mathematics function (libm)

Complex absolute value function
#include <math.h>
double cabs(z) struct { double *r, i*; **}** *z*;

cabs computes the absolute value, or modulus, of its complex argument *z*. The absolute value of a complex number is the length of the hypotenuse of a right triangle whose sides are given by the real part *r* and the imaginary part *i*. The result is the square root of the sum of the parts.

Example

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

See Also

hypot, mathematics library

calling conventions — Technical information

This entry discusses the Mark Williams C function calling conventions. This information is helpful to users who wish to interface C programs with assembly language routines or with object code generated by other language processors. Programs that depend upon specific details of these calling conventions may not be portable to other processors or other C compilers.

In general, Mark Williams C pushes arguments from right to left. Mark Williams C pushes function arguments as follows:

char	as a word
short	as a word
int	as a word
long	as a long word
float	as a pair of long words
double	as a pair of long words
pointer	as a long word

"Word" in this instance means a 68000 (16-bit) word.

An underbar '_' is appended to the beginning of the function's name. Assembly-language programmers must append '_' to the beginning of the name of each C-callable function.

An **add**, **lea**, or **addq** instruction after the call removes the arguments from the stack.

The C prologue executes a **link** to allocate space for automatics and saved registers. Because C functions may use registers a3 through a5 and d3 through d7 for register variables, the C prologue saves used registers, and the C epilogue restores them. The C epilogue executes an **unlk** before returning.

Parameters and local variables in the called function are referenced as offsets from the (frame pointer) register. The stack-pointer register points below the local variable with the lowest address.

Functions return values as follows:

char	in d0.W
int	in d0.W
long	in d0.L
float	in d0 and d1 (returned as **double**)
double	in d0 and d1
pointer	in d0.L

Functions that return **struct** or **union** actually return a pointer to the **struct** or **union** in register a0. The code generated for the function call will move the result to its destination.

C does not require that the number of arguments passed to a function be the same as the number of arguments specified in the function's declaration. Routines with a variable number of arguments are not uncommon. The two formatted I/O routines in the standard library (**printf** and **scanf**) are, in fact, routines that take a variable number of arguments.

Consider the following program as an example:

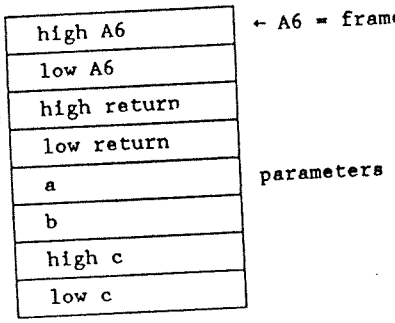
```
long f(a, b, c)
char a;
int b;
long c;
{
    return ((a * b) + c);
}
main() {
        char a = 1;
        int b = 2;
        long c = 3;

        f(a,b,c);
}
```

When compiled with the -S option, it produces the following code:

```
        .shri

        .globl f_
f_:
        link        a6, $0
        move        10(a6), d0
        muls        8(a6), d0
        ext.l       d0
        add.l       12(a6), d0
        unlk        a6
        rts
        .globl main_

main_:
        link        a6, $-8
        moveq       $1, d0
        move.b      d0, -2(a6)
        moveq       $2, d0
        move        d0, -4(a6)

        moveq       $3, d0
        move.l      d0, -8(a6)
        move.l      -8(a6), -(a7)
        move        -4(a6), -(a7)
        move.b      -2(a6), d0
        ext         d0
        move        d0, -(a7)

        jsr         f_
        addq        $8, a7
        unlk        a6
        rts
```

The symbols **main** and **f** have become **main_** and **f_**. The automatic variables in **main** are addressed at negative offsets from a6: **char a** is located at -2(**a6**), **int b** at -4(a6), and **long c** at -8(a6). A byte of unused storage follows **a** so that **b** occurs on an even address. **main** pushes **c**, then **b**, then sign extends **a** and pushes the resulting word. The arguments in **f** are addressed at positive offsets from a6: **char a** is located at 8(**a6**), **int b** at 10(a6), and **long c** at 12(a6). **char c** is treated as an **int**. The result expression is computed into d0.L. When **f** returns, **main** pops the arguments with an **addq** instruction.

In **f** after execution of the **link**, the stack appears as follows:



The following function returns a structure:

```
struct date {
        int month, day, year;
} today;

struct date
mkda(m, d, y)
{
        struct date tmp;

        tmp.month=m;
        tmp.day  =d;
        tmp.year =y;
        return(tmp);
}

main()
{
        today = mkda(3, 20, 85);
}
```

When this program is translated into assembly language by compiling it with the -S option, the result is as follows:

```
        .comm  today_, 6

        .shri

        .globl mkda_
```

```
mkda_:
        link    a6, $-6
        move    8(a6), -6(a6)
        move    10(a6), -4(a6)
        move    12(a6), -2(a6)
        lea     -6(a6), a1
        lea     8(a6), a0
        movea.l             a0, a2
        move    (a1)+, (a2)+
        move.l (a1)+, (a2)+
        unlk    a6
        rts
        .globl main_

main_:
        link    a6, $0
        moveq   $85, d0
        move    d0, -(a7)
        moveq   $20, d0
        move    d0, -(a7)
        moveq   $3, d0
        move    d0, -(a7)
        jsr     mkda_
        addq    $6, a7
        lea     6(a0), a1
        movea.l             $today_+6, a0
        move    -(a1), -(a0)
        move.l -(a1), -(a0)
        unlk    a6
        rts
```

See Also

C language, memory allocation

calloc — General function (libc)

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

calloc is one of a set of routines that helps manage a program's arena. **calloc** calls **malloc** to obtain a block large enough to contain *count* items of *size* bytes each; it then initializes the block to zeroes and returns a pointer to it. When this memory is no longer needed, you can returned it to the free pool by using the function **free**.

Example

This example attempts to **calloc** a small portion of memory; it then reallocates it to demonstrate **realloc**.


```
#include <stdio.h>

main()
{
        register char *ptr, *ptr2;
        extern char *calloc(), *realloc();
        unsigned count, size;

        count = 4;
        size = sizeof(char *);

        if ((ptr = calloc(count, size)) != NULL)
                printf("%u blocks of size %u calloced\n",
                        count, size);
        else
                printf("Insuff. memory for %u blocks of size %u\n",
                        count, size);

        if ((ptr2 = realloc(ptr,(count*size) + 1)) != NULL)
                printf("1 block of size %u realloced\n",
                        (count*size)+1);
}
```

See Also

arena, free, lcalloc, lmalloc, lrealloc, malloc, notmem, realloc

Diagnostics

calloc returns **NULL** if insufficient memory is available.

Notes

The related function **lcalloc** takes unsigned long arguments, and therefore can allocate memory blocks that are larger than 64 kilobytes.

canon.h — Header file

Canonical conversion for the 68000
#include <canon.h>

canon.h defines canonical conversion routines used for the 68000, to ensure that byte ordering is correct.

See Also

byte ordering

carriage return — Character constant

Mark Williams C recognizes the literal character '\r' for the ASCII carriage return character CR (octal 015). This character "tosses the carriage", i.e., it returns the cursor to the beginning of the line. The newline character '\n' drops the cursor down to the next line. With the routines in **libc** and **libm**, '\n' is a synonym for '\n' plus '\r'. TOS routines, such as **Cconws**, need both characters explicitly.

See Also
ASCII, character constant

Notes
Note that files that contain the carriage return character must be opened in binary mode. The default mode for opening a file recognizes only alphanumeric characters, plus **<space>**, the tab character, and '\n'. See the entry for **fopen** for more information on how to open a file in binary mode.

case — C keyword

Introduce entry in switch statement

case is a prefix that is used to introduce the individual entries in a **switch** statement. For example,

```
while ((int = getchar()) != EOF)
      switch (foo) {
      case 'q':
      case 'Q':
            exit(0);
      case ' ':
            n++;
      default:
            break;
      }
```

case introduces the three possibilities recognized by the **switch** statement: a space, 'q', and 'Q'. The statements that follow a **case** statement behave as if they were enclosed within braces. Note that if a **case** statement is not specifically concluded with **exit**, **break**, **return**, or a similar statement, the **switch** statement will continue to search its list for variables that satisfy its condition.

See Also
break, C keywords, C language, switch
The C Programming Language, page 55

cast — Definition

The **cast** operation is when you "coerce" a variable from one data type to another.

There are two reasons to cast a variable. The first is to convert a variable's data into a form acceptable to a given function. For example, the function **hypot** takes two **doubles**; if the variables **leg_x** and **leg_y** are **ints**, then you would pass them to **hypot** as follows:

```
hypot((double)leg_x, (double)leg_y);
```

If you do not do this, **hypot** will still grab a **double**'s worth of memory: the two bytes of your **int**, plus two bytes of whatever happens to be sitting in memory.

The other reason to cast a variable is when you cast one type of pointer to another. For example,


```
char *foo;
int *bar;
bar = (int *)foo;
```

Although **foo** and **bar** are of the same length, you would cast **foo** in this instance to stop the C compiler from complaining about a type mismatch.

See Also
data formats, data types

cat — Command

Concatenate files
cat [*file* ...]

cat copies each *file* to the standard output. A *file* specified by '-' indicates the standard input. If no *file* is specified, **cat** reads the standard input.

<ctrl-S> stops the printing of text, and **<ctrl-Q>** resumes printing.

See Also
commands, msh

Cauxin — gemdos function 3 (osbind.h)

Read a character from the serial port
#include <osbind.h>
long Cauxin()

Cauxin reads a character from the serial port **aux:**, and returns the character read. It is affected by redirection.

Example
The following example creates a dumb terminal emulator that operates through the serial port. It demonstrates the macros **Cauxin**, **Cauxis**, **Cauxos**, **Cauxout**, **Cconis**, **Cconout**, and **Crawcin**. You can exit from the program by typing **<ctrl-Z>**. Run the example either from the GEM desktop, or with the **tos** command.

```
#include <osbind.h>

main() {
        char c;

        for (;;) {
                if (Cauxis())
                        Cconout(c = Cauxin());
                if (Cconis()) {
                        if ((c = Crawcin()) == 26) {
                                break;
                        } else {
                                if (Cauxos())           /* If ready */
                                        Cauxout(c);     /* send char */
                                else                    /* Otherwise */
                                        Cconout('\07');  /* ring bell */
                        }
                }
        }
}
```

See Also

crtsg.0, gemdos, tos, TOS

Notes

TOS defines handle 2 as being **aux:**, the serial port. The microshell **msh** normally redirects handle 2 to another device; because **Cauxin** and its related functions can be redirected, any program that uses **Cauxin**, **Cauxis**, **Cauxos**, or **Cauxout** must be run directly from the GEM desktop, or run under the shell with the **tos** command, which re-redirects handle 2 to the **aux:** device.

An alternative is to use **Bconin** and its relatives instead of the **Cauxin** family when writing programs to be run under **msh**.

Cauxis — gemdos function 18 (osbind.h)

Check if characters are waiting at serial port
#include <osbind.h>
long Cauxis()

Cauxis checks to see if characters are waiting to be read at the serial port. It returns -1 if there are characters waiting, and 0 if there are not.

Example

For an example of how to use this macro, see the entry for **Cauxin**.

See Also

gemdos, tos, TOS

Notes

This function must be compiled with the **-VGEM** option, and run either from the GEM desktop or with the **tos** command.

Cauxos — gemdos function 19 (osbind.h)

Check if serial port is ready to receive characters
#include <osbind.h>
long Cauxos()

Cauxos checks the output status of the serial port. **Cauxos** returns -1 if the serial port is ready to send a character, and 0 if it is not.

Example

For an example of how to use this macro, see the entry for **Cauxin**.

See Also

gemdos, tos, TOS

Notes

Programs that use this function must be compiled with the -VGEM option, and run either from the GEM desktop or with the **tos** command.

Cauxout — gemdos function 4 (osbind.h)

Write a char to the serial port
#include <osbind.h>
void Cauxout(c) int c;

Cauxout writes the character *c* to the serial port, and returns nothing.

Example

For an example of how to use this macro, see the entry for **Cauxin**.

See Also

gemdos, tos, TOS

Notes

Programs that use this function must be compiled with the -VGEM option, and run either from the GEM desktop or with the **tos** command.

cc — Command

Compiler controller
cc [*options*] *file ...*

cc is the program that controls compilation. It guides files of source and object code through each phase of compilation and linking. **cc** has many options to assist in the compilation of C programs; in essence, however, all you need to do to produce an executable file from your C program is type **cc** followed by the name of the file or files that hold your program. It checks whether the file names you give

it are reasonable, selects the right phase for each file, and performs other tasks that ease the compilation of your programs.

File names

cc assumes that each *file* name that ends in **.c** or **.h** is a C program and passes it to the C compiler for compilation.

cc assumes that each *file* argument that ends in **.s** is in Mark Williams assembly language and processes it with the assembler **as**.

cc also passes all files with the suffixes **.o** or **.a** unchanged to the linker **ld**.

How cc works

cc normally works as follows: First, it compiles or assembles the source files, naming the resulting object files by replacing the **.c** or **.s** suffixes with the suffix **.o**. Then, it links the object files with the C runtime startup routine and the standard C library, and leaves the result in file *file*.**prg**. If only one object file is created during compilation, it is deleted after linking; however, if more than one object file is created, or if an object file of the same name existed before you began to compile, then the object file or files are not deleted.

Setting the environment

cc looks for the compiler and its other tools in directories that the user names. The names of these directories together compose **cc**'s *environment*, and each name comprises an *environmental variable*. An environmental variable is set through the micro-shell **msh**, by using the command **setenv**. The user must set the following environmental variables for **cc** to work correctly:

LIBPATH This names the directories that hold the phases of the compiler, the libraries, and the C run-time start-up routines. If you have more than one version of a file, **cc** will use the first one that it finds along the **LIBPATH**.

INCDIR This names the "default" directory within which the C preprocessor **cpp.prg** will look for files that are called with a **#include** statement. This default directory is searched along with the directory of the source file and the directories specified with **-I** options.

PATH This sets where **cc** finds the executable files it uses to compile and link your program.

TMPDIR This names the directory into which temporary files should be written. The default if this variable is not set is the directory in which the source files are kept. Note that this variable need be set only if space is a problem on any of your storage devices.

These environmental variables should be set in your **profile** file. See the entry for **msh** for more information about **profile**.

Options

The following lists all of **cc**'s command-line options. **cc** passes some options through to the linker **ld** unchanged, and correctly interprets to it the options **-o** and **-u**.

Note that a number of the options are esoteric and normally are not used when compiling a C program. The following are the most commonly used options:

-A	invoke editor when errors occur
-c	compile only; do not link
-f	include floating-point **printf**
-l*name*	pass library **lib***name*.**a** to linker
-o *name*	call output file *name*
-V	print details of compiler's actions
-VASM	generate assembly-language output

-A MicroEMACS option. If an error occurs during compilation, **cc** automatically invokes the MicroEMACS screen editor. The error or errors are displayed in one window and the source code file in the other, with the cursor set to the line number indicated by the first error message. Typing **<ctrl-X>>** moves to the next error, **<ctrl-X><** moves to the previous error. To recompile, close the edited file with **<ctrl-Z>**. Compilation will continue either until the program compiles without error, or until you exit from the editor by typing **<ctrl-U>** followed by **<ctrl-X><ctrl-C>**.

-c Compile option. Suppress linking and the removal of the object files.

-D*name*[=*value*]
Define *name* to the preprocessor, as if set by a **#define** directive. If *value* is present, it is used to initialize the definition.

-E Expand option. Run the C preprocessor **cpp** and write its output onto the standard output.

-f Floating point option. Include library routines that perform floating-point arithmetic. Because the floating-point routines require approximately five kilobytes of memory, the standard C library does not include them; the **-f** option tells the compiler to include them. If a program is compiled without the **-f** option but attempts to print a floating point number during execution by using the **e**, **f**, or **g** format specifications to **printf**, the message

```
You must compile with -f option for floating point
```

will be printed and the program will exit.

-I*directory*
Include option. Specify the directory the preprocessor should search for files given in **#include** directives, using the following criteria: If the **#include** statement reads

```
#include "file.h"
```

cc searches for **file.h** first in the source directory, then in the directory named in the -I*directory* option, and finally in the system's default directories. If the **#include** statement reads

```
#include <file.h>
```

cc searches for **file.h** first in the directory named in the -I*directory* option, and then in the system's default directories. Multiple -I*directory* options are executed in their order of appearance.

-K Keep option. Do not erase the intermediate files generated during compilation. Temporary files will be written into the current directory.

-l *name*
library option. Pass the name of a library to the linker. **cc** expands -l*name* into **lib***name***.a** and searches **LIBPATH**.

-N[p0123sdlrt]*string*
Name option. Rename a specified pass to *string*. The letters **p0123sdlrt** refer, respectively, to **cpp**, **cc0**, **cc1**, **cc2**, **cc3**, the assembler, the linker, the libraries, the run-time start-up, and the temporary files. For example, the **-VGEM** option described below implicitly executes the option **-Nrcrtsg.o** to change the name of the run-time start-up module.

-NOV*string*
No variant option. Turn off a variant option that is turned on by default. See the table of variant options, below, for more information.

-o *name*
Output option. Rename the executable file from the default *file***.prg** to *name*.

-Q Quiet option. Suppress all messages.

-S Suppress the object-writing and link phases, and invoke the disassembler **cc3**. This option produces an assembly-language version of a C program for examination, for example if a compiler problem is suspected. The assembly-language output file name replaces the **.c** suffix with **.s**. This is equivalent to the **-VASM** option. The option **-VLINES** can be used with **-S** to generate line numbers as comments in the assembly-language output.

-U *name*
Undefine symbol *name*. Use this option to undefine symbols that the preprocessor defines implicitly, such as the name of the native system or machine.

-V Verbose option. **cc** prints onto the standard output a step-by-step description of each action it takes.

V*string*
Variant option. Variants that are marked **on** are turned on by default. To turn them off, use the appropriate form of the option -NOV*string*. For example, to turn off the option **-VSTRICT**, use the option **-NOVSTRICT**. Most options are turned off by default. To turn them on, enter their names as given below. For example, to turn on the option **-VPEEP**, which turns on the peephole optimizer, simply include it in the **cc** command line. Options marked **Strict:** generate messages that warn of the conditions in question. **cc** recognizes the following variants:

-VASM Output assembly-language code. Identical to -S option, above. It can be used with the **-VLINES** option, described below, to generate a line-numbered file of assembly language. Default is **off**.

-VCOMPAC
Similar to **-VSMALL**, except that PC-relative addressing is used only for code reference.

-VCSD Generate debugging information for **csd**, the Mark Williams C Source Debugger.

-VFLOAT Include floating point **printf** routines. Same as -f option, above.

-VGEM Use routines designed for GEM environment. This uses run-time startup routine **crtsg.o** and links in the libraries **libaes.a** and **libvdi.a**. Default is **off**.

-VGEMACC
Use routines designed for a GEM desk accessory. This uses runtime startup routine **crtsd.o** and links in the libraries **libaes.a** and **libvdi.a**. Default is **off**.

-VGEMAPP
Use routines designed for a GEM application. This is a synonym for **-VGEM**. Default is **off**.

-VLINES Generate line number information. Can be used with the option described above to generate assembly language output that uses line numbers. Default is **off**.

-VMOASM This switch is for an unsupported feature. It is similar to the -S switch, except that it produces Motorola-style assembly language. **as**, the Mark Williams assembler, does not recognize this syntax. Also, the output of this feature may not be a valid source for the Motorola 68000 assembler. Although this is not a supported feature, please contact Mark Williams Company if you discover any problems with it.

-VNOOPT Turn off optimization. Default is **off**, i.e., optimization is on.

-VNOTRAPS
Turn off trap substitution. By default, all **gemdos**, **bios**, **xbios**, and **micro_rtx** calls are traps. By setting this option, subroutine calls will be generated instead of traps. A trap is a single-word instruction, analogous to an interrupt; it is faster and takes up less space than an ordinary subroutine call. This option allows the user to test or use routines that have any of the aforementioned names. Default is **off**.

-VPEEP Peephole optimization. Perform additional optimization on executable. This should not be used when device registers are accessed repeatedly, because the peephole optimizer attempts to reduce memory accesses when values are known to be in registers already.

-VPSTR Put strings into the shared segment, if possible. Used to generate ROMable code. Default is **off**.

-VQUIET Suppress all messages. Identical to **-Q** option. Default is **off**.

-VSBOOK Strict: note deviations from *The C Programming Language*. Default is **off**.

-VSCCON Strict: note constant conditional. Default is **off**.

-VSINU Implement struct-in-union rules instead of Berkeley-member resolution rules. Default is **off**, i.e., Berkeley rules are the default.

-VSLCON Strict: **int** constant promoted to **long** because value is too big. Default is **on**.

-VSMALL Enable PC-relative addressing for global data and function references. This can only be used when the program has no global references that are more than 32 kilobytes away from where they are referenced. The linker will detect if a span is not reached and report an error. The size of pointers does not change, and there is no problem mixing modules compiled with and without **-VSMALL**. The advantage of using **-VSMALL** is that the code generated is smaller and tends to be faster.

-VSMEMB Strict: check use of structure/union members for adherence to standard rules of C. Default is **on**.

-VSNREG Strict: register declaration reduced to auto. Default is **on**.

-VSPVAL Strict: pointer value truncated. Default is **off**.

-VSRTVC Strict: risky types in truth contexts. Default is **off**.

-VSTAT Give statistics on optimization.

-VSTRICT Turn on all strict checking. Default is **on**.

-VSUREG Strict: note unused registers. Default is **off**.

-VSUVAR Strict: note unused variables. Default is **on**.

-V3GRAPH
Translate ANSI trigraphs. Default is **off**.

-Z Pause between passes and prompt for disk change. Used with the compiler using single-sided disks.

See Also

as, cc0, cc1, cc2, cc3, commands, cpp, ld

cc0 — Definition

cc0 is the Mark Williams C *parser*. It parses C programs using the method of recursive descent and translates the program into a logical-tree format.

See Also

cc, cc1, cc2, cc3, cpp

cc1 — Definition

cc1 is the Mark Williams C code generator. This phase generates code from the trees created by the parser, **cc0**. The code generation is table driven, with entries for each operator and addressing mode.

See Also

cc, cc0, cc2, cc3, cpp

cc2 — Definition

cc2 is the optimizer/object generator phase of Mark Williams C. It optimizes the code generated by **cc1**, and writes the object code. Mark Williams C uses multiple optimization algorithms. One optimizes jump sequences: it eliminates common code, optimizes span-dependent jumps, and removes jumps to jumps. The other function scans the generated code repeatedly to eliminate unnecessary instructions.

See Also

cc, cc0, cc1, cc3, cpp

cc3 — Definition

cc3 is the output phase of Mark Williams C that writes a file of assembly language rather than a relocatable object module. This phase is optional; it allows you to examine the code generated by the compiler. To produce an assembly-language output of a C program, use the **-S** option on the **cc** command line. For example,

```
cc -S foo.c
```

tells **cc** to produce a file of assembly language called **foo.s**, instead of an object module.

See Also

cc, cc0, cc1, cc2, cpp

Cconin – gemdos function 1 (osbind.h)

Read a character from the standard input
#include <osbind.h>
long Cconin()

Cconin reads a character from the standard input and echoes it to the standard output. It returns the character read.

Example

This example gets characters from the keyboard and displays them on the screen until a **<ctrl-Z>** is typed.

```
#include <osbind.h>

main() {
        int c = 0;

        while (c != 0x1A)
                Cconout((int)(c = Cconin()));
}
```

See Also

gemdos, TOS

Notes

<ctrl-C> aborts a program if typed in response to **Cconin**.

Cconis – gemdos function 11 (osbind.h)

Find if a character is waiting at standard input
#include <osbind.h>
int Cconis()

Cconis checks to see if characters are waiting at the standard input. It returns -1 if a character is waiting, and zero if no character is waiting.

Example

This example displays a moving asterisk until any non-shift key is typed. **Cconis** is also demonstrated in the example for **Cauxin**.

```
#include <osbind.h>

main() {
        int x=0;
        int dir=0;

        Cconws("\033H\033f");                   /* Home, cursor disabled */

        while (Cconis() == 0) {                 /* Until a key is typed */
                if(dir == 0) {                  /* if left to right */
                        Cconws("\010 *");
                        if(++x > 78)
                                dir++;
                } else {                        /* if right to left */
                        Cconws("\010\010\033K*");  /* Back up, clear to end */
                        if (--x <= 0)
                                dir=0;
                }
        }
        x = Cconin();                           /* Eat the character */
        Cconws("\033e");                        /* Turn cursor on. */
}
```

See Also

gemdos, screen control, TOS

Cconos – gemdos function 16 (osbind.h)

Check if console is ready to receive characters
#include <osbind.h>
long Cconos()

Cconos checks to see if the console is ready to receive characters. It returns -1 if the console is ready, and 0 if it is not.

Example

This program exits with a status of 1 if the console cannot be written to; otherwise, it displays a message and exits with a status of 0.

```
#include <osbind.h>
main() {
        if (Cconos() == 0) {
                exit(1);
        }
        Cconws("The console is ready...\n\r");
        exit(0);
}
```

See Also
gemdos, screen control, TOS

Notes
As of this writing, **Cconos** always returns -1, and does no checking.

Cconout – gemdos function 2 (osbind.h)

Write a character onto standard output
#include <osbind.h>
void Cconout(*c***) int***c***;**

Cconout writes character *c* onto the standard output. It returns nothing.

For information on the screen handling escape sequences used by this routine, see the entry for **screen control**.

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

See Also
gemdos, screen control, TOS

Notes
<ctrl-C> aborts a program if used with **Cconout**.

Cconrs – gemdos function 10 (osbind.h)

Read and edit a string from the standard input
#include <osbind.h>
void Cconrs(*string***) char ****string***;**

Cconrs reads and edits *string*, which it receives from the standard input. The first byte of *string* holds the length of the data portion of the buffer; the second byte holds the actual number of characters read; and the remainder holds the characters read, with a NUL character appended to the end.

Example
This example reads an edited string from **stdin** and writes it and its length to **stdout**. **buff[0]** is the size of the data portion of the buffer, and **buff[1]** is the length read.

```
#include <osbind.h>
main() {
      unsigned char buff[130];

      buff[0] = 128;
      Cconrs(buff);
      printf("String '%s' is %d bytes long\n", &buff[2], buff[1]);
}
```


See Also
gemdos, TOS

Notes
<ctrl-C> aborts a program if typed in response to a **Cconrs**.

Cconws – gemdos function 9 (osbind.h)

Write a string onto standard output
#include <osbind.h>
void Cconws(*string***) char ****string***;**

Cconws writes *string* onto the standard output. It stops writing when it reads the NUL. **Cconws** returns nothing.

Example
This example writes a NUL-terminated string to **stdout**. Note the '\r' used with the '\n'.

```
#include <osbind.h>
main() {
      Cconws("This is a NUL-terminated string.\r\n");
}
```

See Also
gemdos, screen control, TOS

Notes
Note that **<ctrl-S>**, **<ctrl-Q>**, and **<ctrl-C>** act, respectively, as **XON**, **XOFF**, and **abort** while **Cconws** is acting.

cd – Command

Change directory
cd *directory*

The micro-shell **msh** keeps track of the directory in which the user is currently working. If a command is not specified by a complete path name beginning with the name of the storage device on which it is kept, **msh** prefixes it with the name of the current working directory. **cd** changes the current working directory to *directory*. If no *directory* is specified, the directory named in the **$HOME** environmental variable becomes the current working directory.

For example, consider a disk on drive B that has two directories: **foo** and **bar**. By definition, the *root* directory is **B:**, and **foo** and **bar** each are sub-directories of **B:**. To change to the sub-directory **foo**, you would type:

```
cd foo
```

To move from **foo** to **bar**, type the full path name of **bar**:

```
cd b:\bar
```

Note that the symbol '..' stands for a directory's *parent* directory; in this example, both **foo** and **bar** have **B:** as their parent directory. So, to move back from **bar** to **foo**, you could type:

```
cd ..\foo
```

This first moves you from **bar** to **bar**'s parent directory, **B:**; then from the parent directory into **foo**. By definition, a root directory has no parent.

See Also

commands, msh, pwd

ceil — Mathematics function (libm)

Set numeric ceiling
#include <math.h>
double ceil(z) double z;

ceil returns a double-precision floating point number whose value is the smallest integer greater than or equal to *z*.

Example

The following example demonstrates how to use **ceil**:

```
#include <math.h>
dodisplay(value, name)
double value; char *name;
{
        if (errno)
                perror(name);
        else
                printf("%10g %s\n", value, name);
        errno = 0;
}

#define display(x) dodisplay((double)(x), #x)
main() {
        extern char *gets();
        double x;
        char string[64];

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

                display(x);
                display(ceil(x));
                display(floor(x));
                display(fabs(x));
                display(sqrt(x));
        }
}
```


See Also

abs, fabs, floor, frexp

char — C keyword

Data type

char is a C data type. It is the smallest addressable unit of data, and it usually consists of eight bits (one byte) of storage. **sizeof(char)** returns one by definition, with all other data types defined as multiples thereof. All Mark Williams compilers sign-extend **char** when it is cast to a larger data type.

Note that under Mark Williams C, a **char** by default is signed; this conforms with the description of a character on page 183 of *The C Programming Language*.

See Also

byte, C keywords, C language, data formats, declarations, unsigned

character constant — Overview

A **character constant** is a constant of the form *'X'*, where *X* is any printable character enclosed between two apostrophes. The value of the constant is the machine value of the character it represents, whatever it might happen to be on your system. For example, on the IBM PC and compatible machines, the character constant **'A'** is equivalent to the ASCII value of the letter 'A', or 0x41. This gives you a portable way to manipulate the machine values of characters.

Selected non-printable characters can also be represented as character constants by using the following escape sequences:

\0	NUL
*****NNN*	octal number
\a	bell
\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab
\x*NN*	hexadecimal number
\0x*NN*	hexadecimal number

See Also

ASCII, backspace, carriage return, horizontal tab, newline, vertical tab
The C Programming Language, page 35

Notes

The draft ANSI standard describes the form '\x*NNN*' as a one-character constant. Note, too, that use of this form may not be portable to all compilers. Because it departs from the Kernighan and Ritchie standard for C, it will generate a warning message if the compiler option **-VSBOOK** is used.

chdir — UNIX system call (libc)

Change working directory
chdir(*directory*)
char * *directory*;

The function **chdir** changes the working directory to the directory pointed to by *directory*. This change is in effect until the program exits or calls **chdir** again.

By convention, the working directory has the name '.'.

Diagnostics

chdir returns zero if successful. It returns -1 if an error occurred, e.g., that *directory* does not exist, is not a directory, or is not searchable.

See Also

chmod, directory

chmod — UNIX system call (libc)

Change file protection modes
#include <stat.h>
chmod(*file, mode*)
char * *file*; **int** *mode*;

chmod sets the mode of *file* to *mode*. *mode* is constructed from the following values, which are defined in the header file **stat.h**:

S_IJRON	0x01	Read-only
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

chmod returns -1 if an error occurs.

See Also

chdir, directory

Notes

At present, **chmod** is included for compatibility with the UNIX system libraries. It performs no work, and always returns zero.

chmod — Command

Change the modes of a file
chmod +*modes file*
chmod -*modes file*

The command **chmod** changes the modes of a file. *file* is the file whose modes are being changed. *modes* may be one or more of the following:

s	System file (hidden from normal directory searches)
h	Hidden file (" ") cachur
m	Backed-up (shows up as 'm' in **ls -l**)
w	Write allowed (shows up as 'w' in **ls -l**)

Preceding *modes* with '+' adds the modes to a file, whereas preceding it with '-' deletes them. For example, the command

```
chmod +h example
```

adds the "hidden" mode to the file **example**. The file will be hidden from normal system searches.

Typing the command

```
ls -l example
```

will show the letter 'h' among the file's modes. See **ls** for more information about how that command presents a file's modes.

See Also

commands

chown — UNIX system call (libc)

Change ownership of a file
chown(*file, uid, gid*)
char * *file* ;
short *uid, gid* ;

chown changes the owner of *file* to user id *uid* and group id *gid*.

To change only the user id without changing the group id, **stat** should be used to determine the value of *gid* to pass to **chown**.

chown is included for compatibility with the UNIX operating system. It performs no work, and is always zero.

See Also

UNIX routines

clearerr — STDIO macro (stdio.h)

Present stream status

#include <stdio.h>
clearerr(*fp*) **FILE** **fp*;

clearerr resets the error flag of the argument *fp*. If an error condition is detected by the related macro **ferror**, **clearerr** can be called to clear it.

Example

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

See Also

ferror, STDIO

CLK_TCK – Manifest constant

CLK_TCK is a manifest constant that is set in the header file **time.h**. The draft ANSI standard defines it as being equivalent to the rate at which the system clock ticks. On the Atari ST, this is equivalent to 5 milliseconds.

See Also

manifest constants, time, time.h

clock – Time function (libc)

Get number of clock ticks since system boot
#include <time.h>
clock_t clock()

clock returns the number of times the clock has ticked since the system was last turned on. The number of ticks per second is defined by the manifest constant CLK_TCK, which is declared in the header file **time.h**. Note that this value varies from computer to computer. On the Atari ST, the clock ticks every five milliseconds.

clock returns a value of the type **clock_t**; this type is defined in **time.h** as being equivalent to an **unsigned long**. Note that this value will overflow **clock_t** and be reset to zero approximately 148 days after the machine is turned on.

Example

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

See Also

CLK_TCK, time (overview), time.h

close – UNIX system call (libc)

Close a file
int close(*fd*) **int** *fd*;

close closes the file identified by the file descriptor *fd*, which was returned by **creat**, **dup**, or **open**. **close** frees the associated file descriptor.

Because each program can have only a limited number of files open at any given time, programs that process many files should **close** files whenever possible. Mark Williams C closes all open files automatically when a program exits.

Example

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

See Also

creat, open, STDIO, UNIX routines

Diagnostics

close returns -1 if an error occurs, such as its being handed a bad file descriptor; otherwise, it returns zero.

cmp – Command

Compare bytes of two files
cmp [-ls] *file1 file2* **[***skip1 skip2***]**

cmp is a command that is included with Mark Williams C. It compares *file1* and *file2* character by character, for equality. If *file1* is '-', **cmp** reads the standard input.

Normally, **cmp** notes the first difference and prints the line and character position, relative to any skips. If it encounters EOF on one file but not on the other, it prints the message "EOF on file*n*". The following are the options that can be used with **cmp**:

-l Note each differing byte by printing the positions and octal values of the bytes of each file.

-s Print nothing, but return the exit status.

If the skip counts are present, **cmp** reads *skip1* bytes on *file1* and *skip2* bytes on *file2* before it begins to compare the two files.

See Also

commands, diff, msh

Diagnostics

The exit status is zero for identical files, one for non-identical files, and two for errors, e.g., bad command usage or inaccessible file.

Cnecin – gemdos function 8 (osbind.h)

Perform modified raw input from standard input
#include <osbind.h>
long Cnecin()

Cnecin reads a character from the standard input and returns it. The character is not echoed to the standard output.

Example

This example reads characters from the standard input device, changes their case, and writes them out to the standard output device until a **<ctrl-D>** character is typed.

```
#include <osbind.h>
#include <ctype.h>

main() {
        unsigned char c;

        while((c=Cnecin()) != 0x04) {
                if(isupper(c))                  /* Toggle case of char */
                        c = tolower(c);
                else
                        c = toupper(c);
                Crawio(c);
                if(c == 0x0D)                   /* If a <RETURN> */
                        Crawio(0x0A);           /* Append a line feed */
        }
}
```

See Also

gemdos, screen control, TOS

Notes

This routine has been documented elsewhere as recognizing the special meanings of the characters **<ctrl-C>**, **<ctrl-S>**, and **<ctrl-Q>**; this however, appears not to be correct.

commands — Overview

Mark Williams C includes a number of commands. They are listed below, with the command given on the left and a description on the right.

ar	the archiver/librarian
as	the assembler
as68toas	convert Motorola to Mark Williams assembler
cat	concatenate files
cc	the compiler driver
cd	change directory
chmod	change "mode" of a file
cmp	compare two files
cp	copy a file
cpp	the C preprocessor
cursconf	change cursor style and position
date	print/set the system date and time
db	symbolic debugger
df	measure free space on disk
diff	compare two files

drtomw	convert from DRI to Mark Williams
drvprs	check if drive is present
echo	repeat/expand an argument
egrep	find embedded strings
equal	test if two values are equal
exit	leave **msh**
file	determine file type
gem	run a GEM-DOS program
getcol	get a color palette entry
getpal	get color palette
getphys	get base of physical screen memory
getrez	get screen resolution
help	print help files on screen
hidemouse	hide mouse pointer
htom	redraw screen, moving from high to medium resolution
if	execute a command conditionally
inherit	pass variable to child shell
is_set	test if an environmental variable is set
kbrate	get/set the keyboard's repeat rate
kick	force TOS to reread the floppy disk cache
lc	print directory contents in columns
ld	the linker
ls	list directory contents
ltom	redraw screen, moving from low to medium resolution
make	programming discipline
me	MicroEMACS screen editor
mf	measure free space in RAM
mkdir	create a directory
mousehidden	print number of times mouse pointer has been hidden
msh	the Mark Williams micro-shell
mshversion	print current version of **msh**
msleep	suspend processing for *n* milliseconds
mtoh	redraw screen, moving from medium to high resolution
mtol	redraw screen, moving from medium to low resolution
mv	rename a file
mwtomw	convert old Mark Williams object files to 3.0 format
nm	print symbol tables
not	invert the logical value of its argument
od	print an octal dump of a file
pr	format ASCII files for printing
pwd	print the current directory
rdy	create, save, and load a rebootable RAM disk
rescomp	the Mark Williams resource compiler
resdecom	the Mark Williams resource disassembler
resource	the Mark Williams resource editor
rm	remove a file

rmdir	remove a directory
rsconf	set attributes of serial (auxiliary) port
set	set a shell variable
setcol	set a palette color
setenv	set an environmental variable
setpal	set the color palette
setphys	set the physical base of the screen's memory
setprt	set attributes of parallel port
setrez	set screen resolution
show	display saved screen image
showmouse	show the mouse pointer
size	print size of a file
sleep	suspend processing for *n* seconds
snap	take a "snapshot" of the current screen image
sort	sort ASCII files
strip	strip symbol tables from objects
tail	print the end of a file
time	print current time; time execution of a program
tos	run unredirected GEM-DOS program
touch	change a file's date
uniq	list/destroy duplicate lines
unset	discard a shell variable
unsetenv	discard an environmental variable
version	print/assign version number
wc	count words/lines in ASCII files
while	set a conditional loop

Note that many of the commands are built into **msh** itself, whereas the others are executable programs in their own right. For a list of the commands that are built into **msh**, type the command

```
set in .bin
```

Note that commands not built into **msh** must be stored in one of the directories named in the environmental variable **PATH**, so that they can be found automatically by **msh**. Note, too, that commands not built into **msh** can be run independently from the GEM desktop; in most instances, this will require that the suffix be changed from **.prg** to **.ttp**, so the command in question can receive arguments.

For more information on any of these commands, see its entry within the Lexicon.

See Also

Lexicon, msh

compound number — Definition

A **compound number** is a number that consists of two numbers of different types. In the context of C, this applies usually to floating point numbers, which are constructed of a sign bit; an exponent; and a *fraction*, or base upon which the exponent operates.

See Also

data formats, double, float, fraction

con — Operating system device

Logical device for the console

TOS gives names to its logical devices. Mark Williams C uses these names to allow its STDIO library routines to access these devices via TOS. **con:** is the logical device that describes the console.

Example

The following example demonstrates how to open the console device.

```
#include <stdio.h>
main(){
        FILE *fp, *fopen();
        if ((fp = fopen("con:", "w")) != NULL)
                fprintf(fp, "con: enabled.\n");
        else printf("con: cannot open.\n");
}
```

See Also

aux:, prn:, STDIO

Notes

con: may be spelled **con:** or **CON:**.

const — C keyword

Qualify an identifier as not modifiable

The type qualifier **const** marks an object as being unmodifiable. An object declared as being **const** cannot be used on the left side of an assignment (an *lvalue*), or have its value modified in any way. Because of these restrictions, an implementation may place objects declared to be **const** into a read-only region of storage.

See Also

C keywords, volatile

Notes

Mark Williams C does recognize this keyword, but its semantics are not implemented in release 3.0. Thus, storage declared with the **const** qualifier will *not* be treated as unmodifiable by the compiler, and no warnings will be generated.

continue — C keyword

Force next iteration of a loop

#n filename, so that the parser **cc0** will be able to connect its error messages and debugger output with the original line numbers in your source files.

Options

The following summarizes **cpp**'s options:

-D*VARIABLE*
Define *VARIABLE* for the preprocessor at compilation time. For example, the command

```
cc -DLIMIT=20 foo.c
```

tells the preprocessor to define the variable **LIMIT** to be 20. The compiled program acts as though the directive **#define LIMIT 20** were included before its first line.

-E Strip all comments and line numbers from the source code. This option is used to preprocess assembly-language files or other sources, and should not be used with the other compiler phases.

-I *directory*
C allows two types of **#include** directives in a C program, i.e., **#include "file.h"** and **#include <file.h>**. The **-I** option tells **cpp** to search a specific directory for the files you have named in your **#include** directives, in addition to the directories that it searches by default. By default, **cpp** looks for these files in the directory named by the **INCDIR** environmental variable and the directory of the source file. For information on how to set this variable, see the Lexicon's entries for it and for **setenv**. Note that you can have more than one **-I** option on your **cc** command line.

-o *file*
Write output into *file*. If this option is missing, **cpp** writes its output onto **stdout**, which may be redirected.

-U*VARIABLE*
Undefine *VARIABLE*, as if an **#undef** directive were included in the source program. This is used to undefine the variables that **cpp** defines by default, i.e., **GEMDOS** and **M68000**.

Directives

cpp processes the following directives:

#assert	#ifdef
#define	#ifndef
#elif	#include
#else	#line
#endif	#undef
#if	

Each of the directives has its own entry in the Lexicon. Note that no directive can be indented on the line; if it is not set flush with left margin on the screen, **cpp** will ignore it.

See Also

cc
The C Programming Language, page 86

Cprnos – gemdos function 17 (osbind.h)

Check if printer is ready to receive characters
#include <osbind.h>
long Cprnos()

Cprnos attempts to execute a "handshake" routine to see if the printer is ready to receive characters. It returns -1 if the printer is ready, and 0 if it is not.

Example

The following example demonstrates **Cprnos**.

```
#include <osbind.h>

main() {
        if(Cprnos() != 0)
                Cconws("Printer Ready.\n\r");
        else
                Cconws("Printer not ready.\n\r");
}
```

See Also

gemdos, TOS

Cprnout – gemdos function 5 (osbind.h)

Send a character to the printer port
#include <osbind.h>
void Cprnout(*c***) int** *c*;

Cprnout sends the character *c* to the printer port, and returns nothing.

Example

This example writes a line to the printer.

```
#include <osbind.h>
main() {
        unsigned char *c="This is printed on the printer.\r\n";
        while (*c != '\0')
                Cprnout(*c++);
}
```

See Also
gemdos, TOS

Crawcin — gemdos function 7 (osbind.h)

Read a raw character from standard input
#include <osbind.h>
long Crawcin()

Crawcin reads a raw character from the standard input, and returns it to the calling program. The character is not echoed to the standard output, and the special meanings of the characters **<ctrl-C>**, **<ctrl-S>**, and **<ctrl-Q>** are ignored.

Example
This example reads characters from the standard input device, and writes characters out to the standard output device until a **<ctrl-Z>** is typed. **Crawcin** is also demonstrated in the example for **Cauxin**.

```
#include <osbind.h>
main() {
      unsigned char c;

      while((c = Crawcin()) != 0x1A) {
            Crawio(c);
            if(c == 0x00)
                  Crawio(0x0A);
      }
}
```

See Also
gemdos, TOS

Crawio — gemdos function 6 (osbind.h)

Perform raw I/O with the standard input
#include <osbind.h>
long Crawio(c) int c;

Crawio performs raw I/O with the standard input. If the argument *c* equals 0xFF, then a character is read from the standard input and returned. If *c* does not equal 0xFF, then it is written onto the standard output.

Example
This example reads characters from the standard input device, and writes them on the standard output device until a **<ctrl-Z>** is typed.


```
#include <osbind.h>
main() {
      unsigned char c;

      while ((c = Crawio(0xFF)) != 0x1A) {
            Crawio(c);
            if (c == 0x00)
                  Crawio(0x0A);

      }
}
```

See Also
gemdos, TOS

creat — UNIX system call (libc)

Create/truncate a file
int creat(*file***, *mode*) char *****file*****; int** ***mode*****;**

creat creates a new *file* or truncates an existing *file*. It returns a file descriptor that identifies *file* for subsequent system calls. If *file* already exists, its contents are erased. **creat** ignores its *mode* argument. This argument exists for compatibility with implementations of **creat** under UNIX and related operating systems.

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

See Also
fopen, fdopen, STDIO, UNIX routines

Diagnostics
If the call is successful, **creat** returns a file descriptor. It returns -1 if it could not create the file, typically because of insufficient system resources, or nonexistent path.

crts0.o — Runtime startup

Default C runtime startup

crts0.o is the runtime startup routine for C programs compiled into Mark Williams object format.

crts0 provides an efficient, portable environment for C programs. When used with the micro-shell **msh**, it can provide arbitrarily long argument lists, easily configured environmental parameters, and redirection of up to six input/output channels.

The runtime startup module, **crts0.o**, is the first code executed when your program is run. As its first action, it parses the environment string list passed by TOS into a vector of string pointers. This vector is saved in the the variable **extern char **environ**, for the use of the library routine **getenv()**, and passed as the parameter **char *envp[]**, for the information of the function **main()**.

If the environment vector contains a parameter named **ARGV**, then the run time start-up assumes that the program was executed by **msh**, or by another program that handles arguments, and that the remainder of the environment vector is an argument vector that should be passed as the parameter **char *argv[]** to the function **main()**.

If the parameter **ARGV** has a value, such as **ARGV=CCAP??**, then the value should consist of characters from the set **[CAPF?]**. The characters describe the origin of the system file handles as Console, Auxiliary port, Printer port, File, or unknown. The runtime startup stores the value of ARGV, if it exists, into the external variable **char *_iovector** for the use of the routines that emulate the functions of the COHERENT operating system.

If no **ARGV** parameter is found in the environment, then the run time start-up program assumes that the program was executed by a simple **GEMDOS Pexec()**. The buffer **cmdtail** is parsed to form the argument vector for **main()**. **ARGV[0]** is supplied by the external variable **char _cmdname[]**, which should be supplied by your program, or it will be set to **?** by the library. The value of the variable **_iovector** will be set to the default **CCAP????????????????????????????;**.

See Also

argv, runtime startup, system

crtsd.o – Runtime startup

C runtime startup, GEM environment

crtsd.o is the runtime startup routine for a C programs that is designed to be used as a GEM desktop accessory.

crtsd.o can be specified on the **cc** command line in one of two ways. First, the **-VGEMACC** option will include it, well as the libraries **libaes.a** and **libvdi.a**. Second, **crtsd.o** can be used independently of the libraries by using the **name** option **Nrcrtsd.o**.

See Also

argv, cc, crts0.o, crtsg.o, runtime startup

crtsg.o – Runtime startup

C runtime startup, GEM environment

crtsg.o is the runtime startup routine for C programs that use the GEM VDI and AES routines.

crtsg.o is a simple but fast runtime startup routine. Note the following differences from the default runtime startup **crts0.o**:

1. **ARGV, ARGC,** and **ENVP** are all set to zero.

2. **getenv** is not enabled; this means programs that use **crtsg.o** will cannot read environmental parameters.

3. **stderr** will send error messages to the auxiliary port rather than to the console.

crtsg.o can be invoked on the **cc** command line in one of two ways. First, the **-VGEM** option will include it, well as the libraries **libaes.a** and **libvdi.a**. Second, **crtsg.o** can be used independently of the libraries by using the **name** option **Nrcrtsg.o**.

See Also

argv, cc, crts0.o, crtsd.o, runtime startup

ctime – Time function (libc)

Convert system time to an ASCII string

#include <time.h>

char *ctime(*timep***) time_t *timep;**

ctime converts the system's internal time to a form that can be read by humans. It takes a pointer to the internal time type **time_t**, which is defined in the header file **time.h**, and returns a fixed-length string in the form:

```
Thu Mar 14 11:12:14 1987\n
```

Note that **time_t** is defined as being equivalent to a **long**. Mark Williams C defines the internal system time as being equivalent to the number of seconds that have passed since January 1, 1970 00h00m00s GMT.

ctime is implemented as a call to **localtime** followed by a call to **asctime**.

Example

For another example of this function, see the entry for **asctime**.

```
#include <time.h>

main()
{
        time_t t;
        time(&t);
        printf(ctime(&t));
}
```

See Also

time (overview), time_t, time.h

Notes

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

ctype – Overview

#include <ctype.h>

The **ctype** macros and functions test a character's *type*, and can transform some characters into others. They are:

isalnum	test if alphanumeric character
isalpha	test if alphabetic character
isascii	test if ASCII character
iscntrl	test if a control character
isdigit	test if a numeric digit
islower	test if lower-case character
isprint	test if printable character
ispunct	test if punctuation mark
isspace	test if a tab, space, or return
isupper	test if upper-case character
_tolower	change to lower-case character
_toupper	change to upper-case character

These are defined in the header file **ctype.h**, and each is described further in its own Lexicon entry.

Example

The following example demonstrates the macros **isalnum**, **isalpha**, **isascii**, **iscntrl**, **isdigit**, **islower**, **isprint**, **ispunct**, and **isspace**, and the function **toupper**. It prints information about the type of characters it contains, and converts its name to upper-case characters.

```
#include <ctype.h>
#include <stdio.h>

main()
{
        FILE *fp;
        char fname[20];
        int ch,i;
        int alnum = 0;
        int alpha = 0;

        int control = 0;
        int printable = 0;
        int punctuation = 0;
        int space = 0;

        printf("Enter name of text file to examine: ");
        fflush(stdout);
        gets(fname);

        for(i=0; fname[i] != '\0'; i++)
                fname[i]= islower(fname[i]) ? toupper(fname[i])
                : fname[i];
```



```
        if ((fp = fopen(fname, "r")) != NULL)
        {
                while ((ch = fgetc(fp)) != EOF)
                {
                        if(isascii(ch))
                        {
                                if(isalnum(ch)) alnum++;
                                if(isalpha(ch)) alpha++;
                                if(iscntrl(ch)) control++;
                                if(isprint(ch)) printable++;
                                if(ispunct(ch)) punctuation++;
                                if(isspace(ch)) space++;
                        } else {
                                printf("%s is not ASCII.\n", fname);
                                exit(1);
                        }
                }

                printf("%s has the following:\n", fname);
                printf("%d alphanumeric characters\n", alnum);
                printf("%d alphabetic characters\n", alpha);
                printf("%d control characters\n", control);
                printf("%d printable characters\n", printable);
                printf("%d punctuation marks\n", punctuation);
                printf("%d white space characters\n", space);
                exit(0);
        } else
                printf("Cannot open '%s'.\n", fname);
}
```

See Also

ctype.h, Lexicon

ctype.h — Header file

Header file for data tests

#include <ctype.h>

ctype.h is a header file that holds the texts of the macros described in the overview entry **ctype**.

See Also

ctype, header file

cursconf — Command

Set the cursor's configuration

cursconf *task* [*rate*]

cursconf is a command that uses the **xbios** function **Cursconf** to alter the cursor's configuration. It can take one or two arguments. *task* indicates what to do, as follows:

0	hide the cursor
1	show the cursor
2	set the cursor to blink
3	set the cursor not to blink
4	set the cursor to blink at *rate*
5	return the current blink rate

If *task* is set to 4, then you should give **cursconf** the argument *rate*, which sets the rate at which the cursor blinks. *rate* should be set to proportions of the normal rate parameter, which is one half of the normal cycle time (60 Hz for the color monitor, 70 Hz for the monochrome monitor, and 50 Hz for monitors set in PAL mode). For example, setting *rate* to 35 will cause the cursor to blink twice a second on a monochrome monitor.

All arguments to **cursconf** can be C-style constants.

See Also

commands, TOS

Cursconf — xbios function 21 (osbind.h)

Get or set the cursor's configuration
#include <osbind.h>
#include <xbios.h>
int Cursconf(*function, rate***) int** *function, rate*;

Cursconf gets or sets the cursor's configuration. *function* is an integer that tells TOS to do one of the following:

0	hide the cursor
1	show the cursor
2	set the cursor to blink
3	set the cursor not to blink
4	set the cursor to blink at *rate*
5	return the current blink rate

rate, as noted above, sets the rate at which the cursor blinks. It is used to set the rate only if *function* is set to 4; otherwise it is ignored. *rate* should be set to proportions of the normal rate parameter, which is one-half the normal cycle time (60 Hz for the color monitor, 70 Hz for the monochrome monitor, and 50 Hz for monitors set in PAL mode). For example, setting *rate* to 35 will cause the cursor to blink twice a second on a monochrome monitor.

Note that **Cursconf** returns the current cursor blink rate when *function* is set to 5; otherwise, it returns a meaningless value.

Example

This example creates a utility for the micro-shell **msh** that can turn off or turn on the cursor's blink mode. Because this example uses **argv**, do *not* compile it with the **-VGEM** option. For an example of using **Cursconf** in a GEM program, see the entry for **\auto**.

```
#include <osbind.h>
#define JUNK 50                    /* Place-holding value that has no meaning */

main(argc, argv)
int argc;
char *argv[];
{
        if ((argc-1) == 0) {
                Cursconf(3, JUNK);
                exit(0);
        }
        else if (((argc-1) == 1) && (strcmp(argv[1], "blink") == 0)) {
                Cursconf(2, JUNK);
                exit(0);
        }
        else {
                printf("Usage: cursor [blink]\n");
                exit(1);
        }
}
```

See Also

screen control, TOS, xbios

D

daemon — Definition

A **daemon**, in the context of C programming, is a process that is designed to perform a particular task or control a particular device without requiring the intervention of a human operator.

See Also

process

data formats — Technical information

Mark Williams Company has written C compilers for a number of different computers. Each has a unique architecture and defines data formats in its own way.

The following table gives the sizes, in **chars**, of the data types as they are defined by various microprocessors.

Type	*i8086 SMALL*	*i8086 LARGE*	*Z8001*	*Z8002*	*68000*	*PDP11*	*VAX*
char	1	1	1	1	1	1	1
double	8	8	8	8	8	8	8
float	4	4	4	4	4	4	4
int	2	2	2	2	2	2	4
long	4	4	4	4	4	4	4
pointer	2	4	4	2	4	2	4
short	2	2	2	2	2	2	2

Mark Williams C places some alignment restrictions on data, which conform to all restrictions set by the microprocessor. Byte ordering is set by the microprocessor; see the Lexicon entry on **byte ordering** for more information.

See Also

byte ordering, C language, data types, declarations, double, float, memory allocation

data types — Technical information

The following describes the data types recognized by Mark Williams C. The left-hand column below gives compound type specifiers mentioned in *The C Programming Language*; the right-hand column gives additional specifiers recognized by Mark Williams C.

short int	**unsigned short int**
long int	**unsigned short**
unsigned int	**unsigned long int**
long float	**unsigned long**
	unsigned char

Note that the terms **unsigned short int** and **unsigned short** are synonymous, as are the terms **unsigned long int** and **unsigned long**. The type **unsigned char** is an addition to the language. If used in arithmetic expressions, it is automatically cast to **unsigned int**.

See Also

C language, char, data formats, double, float, int, long, pointer, short, unsigned

date — Command

Print/set the date and time
date [-i] [[[*cc*]*yymmdd*]*hhmm*[.*ss*]]

date prints the time of day and the current date, including the time zone. If an argument is given, the system's current time and date is changed, as follows:

cc	century (AD; default, "19")
yy	year (00-99)
mm	month (01-12)
dd	day (01-31)
hh	hour (00-23)
mm	minute (00-59)
ss	seconds (00-59)

Note that the century and seconds fields are optional. For example, typing

```
date 860512141233
```

sets the date to May 12, 1986, and the time to 2:12:33 P.M. Note that at least *hh* and *mm* must be specified—the rest are optional. The command

```
date -i
```

displays the current date and time in the form acceptable to **date** as input. The command

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

resets the keyboard clock and GEM-DOS times with the output of the system clock. Embedding this command in your **msh profile** will ensure that files are always date-stamped correctly.

The library time conversion routines used by **date** look for the environmental variable **TIMEZONE**, which specifies local time zone and daylight saving time information in the format described in **ctime**.

See Also

commands, ctime, msh, time, TIMEZONE

dayspermonth — Time function (libc)

Return number of days in a given month

#include <time.h>
int dayspermonth(*month, year***) int** *month, year*;

dayspermonth returns the number of days in a given month of a given year A.D. *month* is the number of the month in question, from one to 12. *year* is the year A.D. in which *month* appears. Note that there is no year 0.

See Also

isleapyear, time (overview), time.h

db — Command

Assembler-level symbolic debugger

db [-afkort] [*mapfile*] [*datafile*]

db is an assembly language-level debugger. It allows you to run object files and executable programs under trace control, run programs with embedded breakpoints, and dump and patch files in a variety of forms. You can use it to debug assembly-language programs that have been assembled by **as**, the Mark Williams assembler, as well as those that have been compiled with the Mark Williams C compiler.

What is db*?*

db is a symbolic debugger, which means that it works with the symbol tables that the compiler builds into the object files it generates. Because **db** is designed to work on the level of assembly language, the user needs a working knowledge of 68000 assembly language and microprocessor architecture.

Invoking db

To invoke **db**, type its name, plus the options you want (if any) and the name of the files with which you will be working. *mapfile* is an object file that supplies a symbol table. *datafile* is the executable program to be debugged. If possible, **db** accesses *datafile* with write permission.

The following options to the **db** command specify the format of *program*:

-a Accept commands from the **aux** port. This feature allows you to plug a terminal into the Atari's **aux** port and give commands to **db** from it. The program's output is displayed on the Atari's monitor. This allows you to easily debug programs that use AES or VDI calls.

-f Map *program* as a straight array of bytes (file).

-g GEM option: turn on the mouse pointer for programs that use the GEM interface.

-k The *kernel* option. This allows a user to debug all of the Atari ST's memory. The default *symbolfile* in **tos.sym** defines the documented locations in low memory. The *symbolfile* is used to provide symbolically interpreted output. All of the ST's memory, from address 0 in RAM to the end of the ROM, is available for display or patching. Note that this option allows the user to perform a post-mortem on programs that crash: use the command **:r** to display the registers and the command **:f** to display the fault identifier in the process dump area. These commands are described in detail below.

-o *program* is an object file. If *mapfile* is given, it is another object file that provides the symbol table.

-r Read file only, even though you can write into it. This is used to give a file additional protection.

-t Force **stdin, stdout,** and **stderr** to the console (keyboard and screen), regardless of redirection on the command line or in the shell.

Commands and addresses

db executes commands that you give it from the standard input. A command usually consists of an *address*, which tells **db** where in the program to execute the command; and then the command name and its options, if any.

An address is represented by an *expression*, which can be built out of one or more of the following elements:

- The '.', which represents the current address. When an address is entered, the current address is set to that location. The current address can be advanced by typing **<RETURN>**.
- The name of a register. **db** recognizes the register names d0 through d7, a0 through a7, pc, and sp. Typing the name of a register displays its contents.
- The names of global symbols and symbolic addresses can be used in place of the addresses where they occur. This is useful when setting a breakpoint at the beginning of a subroutine.
- An integer constant, which can be used in the same manner as a global symbol. The default is decimal; a leading **0** indicates octal and **0x** indicates hexadecimal.
- The following binary operators can be used:

  + addition
  - subtraction
  * multiplication

/ integer division

All arithmetic is done in **longs**.

- The following unary operators can be used:

  ~ complementation
  - negation
  * indirection

  All operators are supported with their normal level of precedence. Parentheses '()' can be used for binding.

Display commands

The following commands merely display information about *program*. The symbol '.' represents the *address*, which defaults to the current display address if omitted. *count* defaults to one.

address[,count]?[format]
Display the *format count* times, starting at *address*. The *format* string consists of one or more of the following characters:

^	reset display address to '.'
+	increment display address
-	decrement display address
b	byte
c	**char**; control and non-**chars** escaped
C	like 'c' except '\0' not displayed
d	decimal
f	**float**
F	**double**
i	machine instruction, disassembled
l	**long**
n	output '\n'
o	octal
p	symbolic address
s	string terminated by '\0', with escapes
S	string terminated by '\0', no escapes
u	**unsigned**
w	word
x	hexadecimal
Y	time

The format characters **d**, **o**, **u**, and **x**, which specify a numeric base, can be followed by **b**, **l**, or **w**, which specify a datum size, to describe a single datum for display. A format item may also be preceded by a count that specifies how many times the item is to be applied. Note that *format* defaults to the previously set format for the segment (initially **i** for instructions). Except where otherwise noted, **db** increments the display address by the size of the datum displayed after each format item.

Execution commands

In the following commands, *address* defaults to the address where execution stopped, unless otherwise specified; *count* and *expr* default to 1. *commands* is an arbitrary string of **db** commands, terminated by a newline. A newline may be included by preceding it with a backslash '\'.

[address]=
Print *address* in current display base. *address* defaults to '.'. The command = assigns values to locations in the traced process. The size of the assigned value is determined from the last display format used. You can set and display the registers of the traced process, just like any other address in the traced process. Thus,

```
d0?l
d0=0
```

displays the value of register **d0** as a **long**, and then sets **(long) d0** to zero. To display the character in the low byte of d0, use:

```
d0+3?c
```

To set the low byte of d0 to ASCII **<esc>**, use

```
d0+3=033
```

[address[,count]]=value[,value[,value]...]
Patch the contents starting at *address* to the given *value*. *address* defaults to '.'. Up to ten *values* can be listed.

? Print verbose version of last error message.

[address] **:a**
Print *address* symbolically. *address* defaults to '.'.

[address]:b*[commands]*
Set breakpoint at *address*; save *commands* to be executed when breakpoint is encountered. *commands* defaults to **.:a\nl+.?l\n:x**.

:br *[commands]*
Set breakpoint at return from current routine. The defaults are the same as for **:b**, above.

[address] **:c**
Continue execution from *address*.

[address] **:d[r][s]**
Delete breakpoint at *address*. If optional **r** or **s** is specified, delete return or single-step breakpoint. *address* defaults to '.'.

[address]:**e***[commandline]*
Begin traced execution of the object file at *address* (default, entry point). The *commandline* is parsed and passed to the traced process. **argv[0]** must be typed directly after **:e** if supplied. For example, **:e3 foo bar baz** sets **argv[0]** to **3**, **argv[1]** to **foo**, **argv[2]** to **bar**, and **argv[3]** to **baz**. Quotation marks, apostrophes, and redirection are parsed as by **msh**, but special characters '?*[]' and shell punctuation '(){}|;' are not.

:f Print type of fault which stopped the traced process.

[expr]:**l***[filename]*
The log option. If *expr* is non-zero, open *filename* as a log file; if *expr* is zero, close the currently open log file. **db** echoes all its responses into the open log file.

[expr] **:n**
Set default numeric display base to *expr*: **8**, **10**, and **16** indicate, respectively, octal, decimal, and hexadecimal.

:p Display breakpoints.

[expr] **:q**
If *expr* is nonzero, quit the current level of command input (see **:x**). *expr* defaults to 1. End of file is equivalent to **:q**.

:r Display registers.

[address],[count]:**s[c]***[commands]*
Single-step execution starting at *address*, for *count* steps, executing *commands* at each step. *commands* defaults to **.?i**.

After a single-step command, **<RETURN>** is equivalent to **.,1:s[c]**. If the optional **c** is present, **db** turns off single-stepping at a subroutine call and turns it back on upon return.

[depth] **:t**
Print a call traceback to *depth* levels. If *depth* is 0 (default), unwind the whole stack.

[expr] **:x**
If *expr* is nonzero, read and execute commands from the standard input up to end of file or **:q**. *expr* defaults to 1.

Example of the commands

The following example shows how each **db** command can be used to examine an executable file. It uses the following C program, called **count.c**, which counts the number of ASCII characters in a file:


```
#include <ctype.h>
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
        FILE *fp;
        int result, ch;

        if ((fp = fopen(argv[1], "r")) != NULL) {
                while ((ch = fgetc(fp)) != EOF) {
                        if(isascii(ch)) result++;
                        else fatal(argv[1], "Not ASCII");
                }
                printf("%s: %d characters\n", argv[1], result);
        }
        else fatal(argv[1], "Cannot open");
}

fatal(filename, message)
char *filename, *message;
{
        printf("%s: %s\n", filename, message);
}
```

For purposes of this example, **count.prg** will be used to count the characters in a text file called **tester**. Its contents are as follows:

```
Sonnet 30

When to the sessions of sweet silent thought
I summon up remembrance of things past,
I sigh the lack of many a thing I sought,
And with the old woes new wail my dear time's waste:
Then can I drown an eye, unused to flow,
For precious friends hid in death's dateless night,
And weep afresh love's long since canceled woe,
And moan the expense of many a vanished sight:
Then can I grieve at grievances foregone,
And heavily from woe to woe tell o'er
The sad account of fore-bemoaned moan,
Which I new pay as if not paid before.
But if the while I think on thee, dear friend,
All losses are restored, and sorrows end.
```

To begin, compile **count.c** by typing the following command:

```
cc -V count.c
```

When the program has been compiled, invoke **db** with the following command:

```
db count.prg
```

Addressing commands

As noted above, **db** offers several different ways to set the *address*, or the position within the program that you are examining. One way is by entering a variable name. Type **printf. db** replies:

```
printf_         link    a6, $0x0
```

Another way to set the address is by entering an absolute address. Type **0600. db** replies:

```
main_+0x70      jsr     printf_.l
```

The symbol '.' (dot) echoes the current address. Type a dot; **db** will reply:

```
main_+0x70      jsr     printf_.l
```

which is, as expected, identical to the previous reply.

The equal sign '=' displays the absolute address of any variable that precedes it. To see how this works, type **printf=. db** replies:

```
0x1C6
```

which is the address of **printf**.

Instructions can be shown, beginning at a named address. The *format* must be introduced with a question mark '?'. For example, **.,7i** shows the current line in the instruction space, as indicated by the format string "?i". When this command is typed, **db** replies:

```
main_+0x70      jsr     printf_.l
```

Now, show the next five instructions from the current point by typing **.,5?i. db** replies:

```
main_+0x70      jsr     printf_.l
main_+0x76      lea.l   0xA(a7), a7
main_+0x7A      bra     main_+0x92
main_+0x7C      move.l  $0x24FB, -(a7)
main_+0x82      movea.l 0xA(a6), a0
```

Once a format is set, it remains the default until the format is reset with another format string. For example, the command **printf,20** prints 20 instructions, beginning with **printf**; the format **?i** remains in effect. Type this command. **db** replies:


```
printf_         link    a6, $0x0
printf_+0x4     pea.l   0x8(a6)
printf_+0x8     move.l  $_stdout_, -(a7)
printf_+0xE     jsr     sprintf_+0x3C.l
printf_+0x14    addq.w  $0x8, a7
printf_+0x16    unlk    a6
printf_+0x18    rts
fprintf_        link    a6, $0x0
fprintf_+0x4    pea.l   0xC(a6)
fprintf_+0x8    move.l  0x8(a6), -(a7)
fprintf_+0xC    jsr     sprintf_+0x3C.l
fprintf_+0x12   addq.w  $0x8, a7
fprintf_+0x14   unlk    a6
fprintf_+0x16   rts
sprintf_        link    a6, $0xFFE6
sprintf_+0x4    pea.l   0xFFE6(a6)
sprintf_+0x8    move.w  $0x8000, -(a7)
sprintf_+0xC    move.l  0x8(a6), -(a7)
sprintf_+0x10   jsr     _stropen_.l
sprintf_+0x16   lea.l   0xA(a7), a7
```

Typing **,20** prints the next 20 instructions, beginning from where the previous command left off. When you type this, **db** replies:

```
sprintf_+0x1A   pea.l   0xC(a6)
sprintf_+0x1E   pea.l   0xFFE6(a6)
sprintf_+0x22   jsr     sprintf_+0x3C.l
sprintf_+0x28   addq.w  $0x8, a7
sprintf_+0x2A   pea.l   0xFFE6(a6)
sprintf_+0x2E   clr.w   -(a7)
sprintf_+0x30   jsr     fputc_.l
sprintf_+0x36   addq.w  $0x6, a7
sprintf_+0x38   unlk    a6
sprintf_+0x3A   rts
sprintf_+0x3C   link    a6, $0xFF96
sprintf_+0x40   movem.l d7/a4/a5, (a7)
sprintf_+0x44   move.l  0xC(a6), 0xFFFC(a6)
sprintf_+0x4A   movea.l 0xFFFC(a6), a0
sprintf_+0x4E   move.l  (a0), d0
sprintf_+0x50   movea.l d0, a4
sprintf_+0x52   addq.l  $0x4, 0xFFFC(a6)
sprintf_+0x56   move.b  (a4)+, d0
sprintf_+0x58   ext.w   d0
sprintf_+0x5A   move.w  d0, d7
```

Finally, the command **:a** displays an address symbolically. The default is the current address. Type this command; **db** replies:

```
sprintf_+0x5A
```

which is the same address as that of the last instruction in the previous example; in other words, the address advanced as the command was processed.

To reset and display the address at the point where the instruction **fatal** is, type **fatal:a. db** replies:

```
fatal_
```

Execution commands

db allows you to execute portions of your program; this is done by setting *breakpoints*, or points where execution stops. Breakpoints are set with the command :b. Set breakpoints at **main**, **printf**, and **fatal** as follows:

```
main:b
printf:b
fatal:b
```

The command **:p** displays the current breakpoints:

```
00000110 (main_) i+.7i\n:x\n
000001C6 (printf_) i+.7i\n:x\n
000001A6 (fatal_) i+.7i\n:x\n
```

Now, begin execution with the command **:e**. As noted above, **:e** can take arguments; the arguments correspond to the elements in the array **argv**; in this example, use the following command to pass as an argument the name of the text file **tester**, whose text is given above:

```
:e tester
```

db replies:

```
main_           link      a6, $0xFFF8
```

The program has executed up to the first breakpoint, set on **main**. The command *n*:t performs a call traceback on the stack to *n* levels; the default is zero, which means to unwind the whole stack. Type:

```
:t
```

db replies:

```
0x035E10  main_(0x0002, 0x0003, 0x561A, 0x0003, 0x55F6)
```

Note that the address of **main_** has changed because the program is now loaded into memory.

The command **:c** continues execution of the program to the next breakpoint. When you type it, **db** will reply:

```
printf_         link      a6, $0x0
```

Perform another stack traceback by typing **:t**. **db** replies:

```
0x035DF6  printf_(0x0003, 0x52C8, 0x0003, 0x2D61, 0x0272)
0x035E10  main_(0x0002, 0x0003, 0x561A, 0x0003, 0x55F6)
```

Type **:c** to continue execution to the next breakpoint. **db** replies:

```
tester: 626 characters
Child process terminated (0)
```


The first line shows the output of of the program; in this case, a message that the file **tester** has 626 characters. The message about the child process indicates that the program has finished execution and exited; the number in parentheses is the value that **exit** returned to the calling program (in this case, **db**).

Now, type **:p** to print a list of the breakpoints. **db** makes no reply because no breakpoints remain set; all have been erased as the program executed.

Finally, quit the debugging session by typing **:q**.

Example of debugging

This example shows how to use **db** to track down a simple bug. It uses the following program, called **bug.c**:

```
#include <stdio.h>

main() {
        output(NULL, stdout);   /* send number to stdout */
}

output(number, fp)
int number;
FILE *fp;
{
        fprintf(fp, "The number is %d.\n", number);
}
```

This program passes a number to the routine **output**, which writes it into the named file or device. The program illustrates a common error in C programming.

To begin, compile **bug.c** by using the following command:

```
cc -V bug.c
```

You should see no error messages during compilation. When compilation is finished, try running the program. Instead of writing its message on the standard output device, the program should generate a bus error (as indicated by the appearance of two "bombs" on the screen).

Now, invoke **db** with the following command:

```
db bug.prg
```

One way to approach this problem is to set a breakpoint on **main** and step through the program. The following sets the breakpoint:

```
main:b
```

The **:e** commands performs traced execution at the program's entry point. When you type **:e**, **db** replies as follows:

```
main_           link      a6, $0x0
```

The **:s** commands performs single-step execution. The following commands follows the program through five steps:

```
5:s
```

db replies as follows:

```
main_+0x4       move.l    $_stdout_, -(a7)
main_+0xA       clr.l     -(a7)
main_+0xC       jsr       output_.l
output_         link      a6, $0x0
output_+0x4     move.w    0x8(a6), -(a7)
```

The command **:t** allows you to perform a stack traceback. **db** replies as follows:

```
0x0343F6  output_+0x4(0x0000, 0x0000, 0x0003, 0x3AC6)
0x034406  main_+0x12(0x0001, 0x0003, 0x3C14, 0x0003, 0x3BF0)
```

The number in parentheses indicate what is being passed on the stack to the routine. Each four-digit number represents a machine word (two bytes). The first line indicates the source of the trouble: the routine **output** is being passed *four* words, when it is defined as receiving three: an **int** and a pointer. The problem, of course, is that **main** passed **output** two pointers, NULL and **stdout**; on the 68000, unlike on some other processors, NULL and zero are *not* identical. (For more information on this topic, see the Lexicon entries for **pointer**, **NULL**, and **data formats**.)

Another, simpler approach to this problem is to enter **db** and then immediately set a breakpoint with **:b**, perform a traced execution with **:e** followed by a stack traceback with the **:t** command. **db** replies as follows:

```
0x03435C  fputc_+0x32(0x0054, 0x0000, 0x0003)
0x034304  sprintf_+0x74(0x0000, 0x0003, 0x0003, 0x43F0)
0x0343E4  fprintf_+0x12(0x0000, 0x0003, 0x0003, 0x3A4A, 0x0000)
0x0343F6  output_+0x18(0x0000, 0x0000, 0x0003, 0x3AC6)
0x034406  main_+0x12(0x0001, 0x0003, 0x3C14, 0x0003, 0x3BF0)
```

Again, the display shows how **output** was passed an improper argument, which made it pass an improper argument to **fprintf**.

See Also

commands, od

Notes

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

db now supports symbol tables larger than 64 kilobytes.

Dcreate — gemdos function 57 (osbind.h)

Create a directory
#include <osbind.h>
long Dcreate(*path***) char ***path***;**

Dcreate creates a directory; it returns zero if the directory was created successfully, one if it was not. *path* points to the subdirectory's path name, which should be a NUL-terminated string. **Dcreate** returns a negative value when an error occurs.

Example

The following example uses **Dcreate** to create a directory.

```
#include <osbind.h>
extern int errno;

main(argc, argv) int argc; char **argv; {
        int status;

        if (argc < 2) {
                Cconws("Usage: Dcreate pathname\r\n");
                Pterm(1);
        }

        if ((status = Dcreate(argv[1])) != 0) {
                errno = -status;
                perror("Dcreate failure");
                Pterm(1);
        }
        Cconws("Directory ");
        Cconws(argv[1]);
        Cconws(" created.\r\n");
        Pterm0();
}
```

See Also

gemdos, TOS

Ddelete — gemdos function 58 (osbind.h)

Delete a directory
#include <osbind.h>
long Ddelete(*path***) char ***path***;**

Ddelete deletes a directory; it returns zero if the deletion was successful, non-zero if the deletion failed. *path* points to the subdirectory's path name, which must be a NUL-terminated string.

Example

The following example deletes a directory

```
#include <stdio.h>
#include <osbind.h>
#define EACCESS (-36)       /* Access violation error code    */

extern int errno;

main(argc, argv) int argc; char **argv; {
      int status;

      if (argc < 2) {
            Cconws("Usage: Ddelete pathname\r\n");
            Pterm(1);
      }

      if ((status = Ddelete(argv[1])) != 0) {
            if (status == EACCESS) {
                  fprintf(stderr, "\nDirectory %s contains files\n",
                        argv[1]);
            } else {
                  errno = -status;
                  perror("Ddelete failure");
            }
            Pterm(1);
      }
      printf("Directory %s deleted.\n", argv[1]);
      Pterm0();
}
```

See Also

gemdos, TOS

declarations — Overview

Mark Williams C recognizes the following as legal declarations for data types:

char
double
enum
float
int
long
long float
long int
short
short int
struct
union
unsigned char
unsigned int
unsigned long
unsigned long int
unsigned short
unsigned short int
void

The following pairs of terms are synonymous; the more commonly used term is given on the *right*:

long float	**double**
long int	**long**
short int	**short**
unsigned long int	**unsigned long**
unsigned short int	**unsigned short**

See Also

C language, data formats, data types, Lexicon

default — C keyword

Default label in switch statement

default is a prefix used in **switch** statement. If none of the **case** labels match the parameter in the **switch** statement, then the **default** label is used. Note that a **switch** is not required to have a **default** case, but it is good programming practice to use one.

See Also

C keywords, C language, case, switch
The C Programming Language, page 55

#define — Preprocessor instruction

Define a variable as manifest constant
#define *constant value*

#define tells the C preprocessor **cpp** to define *variable* as a manifest constant. For example, the instruction

```
#define MAXARGS 9
```

tells **cpp** to replace every instance of the string **MAXARGS** with the numeral **9** throughout the program.

The judicious use of **#define** instructions allows you to write code that is more easily understood, maintained, and enhanced. With them, you can modify a major parameter throughout a program by changing one line of code. They also allow you to use a variable name that suggests the function of the parameter it represents; for example, the name **MAXARGS** clearly refers to the maximum number of arguments, whereas the numeral **9** could refer to nearly anything.

See Also

cpp, manifest constant

Notes

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

The present release of Mark Williams C implements the ANSI standard for the C preprocessor. Note that according to the ANSI standard, a macro expansion always occupies no more than one line, no matter how many lines the definition or the actual parameters to the macro span. If you have defined macros that span more than one line, you must either redefine them to occupy one line, or somehow embed the newline character within the macro itself; otherwise, the macro will not expand correctly.

desk accessory — Technical information

A **desk accessory** is a program that is loaded by TOS into the GEM desktop when it is booted. The desktop gives each accessory its own icon, keeps it resident in memory, and gives you direct access to it. When you build a menu, the routine **menu_bar** will automatically include the name of the accessory when it builds the list displayed under the **desk** entry.

To compile a desk accessory with Mark Williams C, use the option **-VGEMACC**. This will automatically link in the special run-time start-up routine **crtsd.o**, and otherwise perform all that is needed to create a desk accessory. Note that all desk accessories must have the suffix **.acc**. Therefore, to compile the program **foo.c** into a desk accessory, use the following form of the **cc** command:

```
cc -VGEMACC -o foo.acc foo.c
```

To install a desk accessory, move the compiled program into your system's root directory. If you have a hard disk, it should be in directory **c:**; otherwise, it should be in the root directory of the disk with which you boot TOS. Do *not* place it into the directory **\auto**; this will cause all manner of unpleasant things to happen. The program will be loaded into the desktop automatically when you reboot your system.

Because of their specialized nature, desk accessories restrict the number and variety of programming tools you can use with them. Note the following:

- Do not use any **stdio** routines.
- Do not use the **malloc** routines found in **libc.a**.
- Do not use **exit**, **Pterm**, **Pterm0**, or **Ptermres**.
- Do not return from **main**.

Also, you should keep the following in mind as you write your accessory:

- If you use **rsc_load**, remember to use **rsrc_free** before you give up control, if possible.

- Do not use **evnt_timer** calls: use **evnt_multi** instead.

Example

The following example, called **desk.c**, demonstrates how to write a desk accessory. It is based on a public-domain program written by Jan Gray in 1986. To compile it, use the following command:

```
cc -o desk.acc -VGEMACC desk.c
```

It displays a digital clock or a calendar in a window in the upper-right hand corner of the desktop.

```
#include <gemdefs.h>
#include <osbind.h>
#include <time.h>

typedef struct { int x, y, w, h; } Rectangle;
#define elements(r) r.x, r.y, r.w, r.h
#define pointers(r) &r.x, &r.y, &r.w, &r.h

char clock_s[] = "hh:mm TZT";
char calend_s[] = "ddd mmm dd YYYY";

/* Faked timezone environment for desk accessory */
char *getenv() { return "EST:300:EDT:1.1.4"; }

main()
{
        register int clock_id;      /* Clock menu identifier */
        register int calend_id;     /* Calendar menu identifier */
        register int clock_w;       /* Clock window handle */
        register int calend_w;      /* Calendar window handle */
        register int w;             /* Temporary window handle */

        Rectangle clock_r;          /* Clock window rectangle */
        Rectangle calend_r;         /* Calendar window rectangle */
        Rectangle r;                /* Temporary rectangle */
        int mb[8];                  /* Message buffer */
        int ret;                    /* Dummy return buffer */

        /* Register menu title */
        ret = appl_init();
        clock_id = menu_register(ret,  "  Clock");
        calend_id = menu_register(ret,  "  Calendar");

        /* Size window titles for templates */
        graf_handle(pointers(r));
        clock_r.w = r.w + r.x * sizeof(clock_s);
        clock_r.h = r.h;
        calend_r.w = r.w + r.x * sizeof(calend_s);
        calend_r.h = r.h;

        /* Position window at upper right corner */
        wind_get(0, WF_FULLXYWH, pointers(r));
        clock_r.x = r.w - clock_r.w; clock_r.y = r.y;
        calend_r.x = r.w - calend_r.w; calend_r.y = r.y;
```

```
/* Initialize window handles as closed */
clock_w = calend_w = -1;
for (;;) {

/* Await message or timer event */
if (MU_MESAG & evnt_multi(
      MU_MESAG | MU_TIMER,
      0, 0, 0,
      0, 0, 0, 0, 0,
      0, 0, 0, 0, 0,
      mb, 30000, 0, /* 30 second timer interval */
      &ret, &ret, &ret, &ret, &ret, &ret)) {

      switch (mb[0]) {

      /* Accessory menu line selected */
      case AC_OPEN:
            if (mb[4] == clock_id) {
                  w = clock_w;
                  r = clock_r;

            } else if (mb[4] == calend_id) {
                  w = calend_w;
                  r = calend_r;

            } else
                  break;

            if (w > 0) {
                  wind_set(w, WF_TOP, 0, 0, 0, 0);
                  break;
            }


            w = wind_create(NAME|CLOSER|MOVER, elements(r));
            if (w > 0) {
                  if (mb[4] == clock_id) {
                  clock_w = w;
                  wind_set(w, WF_NAME, clock_s, 0, 0);
                  } else {
                  calend_w = w;
                  wind_set(w, WF_NAME, calend_s, 0, 0);
                  }
                  wind_open(w, elements(r));
            }
            break;

   /* Screen manager restart */
   case AC_CLOSE:
         if (mb[3] == clock_id)
               clock_w = -1;
         else if (mb[3] == calend_id)
               calend_w = -1;
         break;
```



```
      /* Window close box selected */
      case WM_CLOSED:
            w = mb[3];
            if (w == clock_w)
                  clock_w = -1;
            else if (w == calend_w)
                  calend_w = -1;
            else
                  break;
            wind_close(w);
            wind_delete(w);
            break;

      /* Window dragged to new position */
      case WM_MOVED:
            w = mb[3];
            r = *(Rectangle *)(mb+4);
            if (w == clock_w)
                  clock_r = r;
            else if (w == calend_w)
                  calend_r = r;
            else
                  break;
            wind_set(w, WF_CURRXYWH, elements(r));
            break;

      case WM_NEWTOP:
      case WM_TOPPED:     /* Window clicked to top */
            w = mb[3];
            if (w != clock_w && w != calend_w)
                  break;
            wind_set(w, WF_TOP, 0, 0, 0, 0);
            break;
      }
}

/* Update time on each event if window is open */
if (clock_w > 0 || calend_w > 0) {
      register struct tm *tp;
      register char *p;
      time_t tt;

      time(&tt);
      tp = localtime(&tt);
      p = asctime(tp);
```

```
                calend_s[0] = *p++;
                calend_s[1] = *p++;
                calend_s[2] = *p++;
                calend_s[3] = *p++;
                calend_s[4] = *p++;
                calend_s[5] = *p++;
                calend_s[6] = *p++;
                calend_s[7] = *p++;
                calend_s[8] = *p++;
                calend_s[9] = *p++;
                calend_s[10] = *p++;

                clock_s[0] = *p++;
                clock_s[1] = *p++;
                clock_s[2] = *p++;
                clock_s[3] = *p++;
                clock_s[4] = *p++;
                p += 3;
                clock_s[5] = *p++;

                calend_s[11] = *p++;
                calend_s[12] = *p++;
                calend_s[13] = *p++;
                calend_s[14] = *p++;
                p = tp->tm_isdst <= 0 ? tzname[0] : tzname[1];

                clock_s[6] = *p++;
                clock_s[7] = *p++;
                clock_s[8] = *p++;

                if (clock_w >= 0)
                        wind_set(clock_w, WF_NAME, clock_s, 0, 0);
                if (calend_w >= 0)
                        wind_set(calend_w, WF_NAME, calend_s, 0, 0);
        }
        }
}
```

See Also

crtsd.o, TOS

df — Command

Measure free space on disk

df [-a] *device*

df measures the amount of free space left on a floppy disk, on a logical device on a hard disk, or on a RAM disk. *device* is the name of the device you wish to check; for example, to check the amount of space left on the disk in drive A:, type:

```
df a:
```

The default device is the one you are currently using.

The option **-a** prints the amount of space left on all devices.

See Also

commands, mf, msh

Dfree — gemdos function 54 (osbind.h)

Get information on a drive's free space

#include <osbind.h>

void Dfree(*fs, drive***) long** *fs*[4]**; int** *drive*;

Dfree retrieves information about free space on a disk drive.

fs is an array of four **unsigned longs** into which **Dfree** writes, respectively, the number of free allocation units (also called "clusters") on a disk; the total number of allocation units on the disk; the size of a sector, in bytes; and the size of each allocation unit, in sectors. If you prefer, you can pass *fs* as a pointer to a structure of four **longs**.

drive is the number of the disk drive you wish to check, with zero indicating the default drive, one indicating drive A, etc.

Example

This example displays disk statistics for the default drive.

```
#include <osbind.h>

struct disk_info {
        unsigned long di_free;      /* free allocation units */
        unsigned long di_many;      /* how many AUs on disk */
        unsigned long di_ssize;     /* sector size */
        unsigned long di_spau;      /* sectors per AU */
};

main()
{
        long fs;
        long fb;
        int dd;
        long ts;
        long tb;

        struct disk_info disk;
        dd = Dgetdrv();
        Dfree(&disk, dd+1);
        fs = disk.di_free*disk.di_spau;
        ts = disk.di_spau*disk.di_many;
        fb = fs * disk.di_ssize;
        tb = ts * disk.di_ssize;

        printf("Disk %c: has %ld bytes free in %ld sectors\n",
                dd+'A', fb, fs);
        printf("from total of %ld bytes in %ld sectors (cluster size %ld)\n",
                tb, ts, disk.di_spau*disk.di_ssize);
}
```

See Also

gemdos, TOS

Dgetdrv — gemdos function 25 (osbind.h)

Find current default disk drive
#include <osbind.h>
int Dgetdrv()

Dgetdrv returns an integer that indicates the current drive: 0 corresponds to drive A, and so on through 15 corresponding to drive P.

Example

This example prints the default drive.

```
#include <osbind.h>
main() {
      printf("'%c:' is the current default drive.\n",
            (char) Dgetdrv() + 'A');
}
```

See Also

Dsetdrv, gemdos, TOS

Dgetpath — gemdos function 71 (osbind.h)

Get the current directory name
#include <osbind.h>
long Dgetpath(*buffer, drive***) char *****buffer****; int** ***drive****;*

Dgetpath gets the name of the current directory. *buffer* points to the area where the buffer name is to be stored. *drive* holds a number that indicates the disk drive to be examined, as follows: 0, the default drive; 1, drive A; etc.

Example

This example prints the current path name and device string.

```
#include <osbind.h>

main() {
      int drv;
      char pathbuf[66];                     /* Path buffer */
      char *buf;

      buf = pathbuf;
      *buf++ = (drv=Dgetdrv())+'A';         /* Get drive */
      *buf++ = ':';                         /* d: */
      Dgetpath(buf, drv+1);                 /* Rest of path */
      printf("Current path is %s\n", pathbuf );  /* Display it */
}
```


See Also

Dsetpath, gemdos, TOS

diff — Command

Summarize differences between two files
diff [-b] [-c *symbol***]** ***file1 file2***

diff compares *file1* with *file2*, and summarizes the changes needed to turn *file1* into *file2*.

Two options involve input file specification. First, the standard input may be specified in place of a file by entering a hyphen '-' in place of *file1* or *file2*. Second, if *file1* is a directory, **diff** looks within that directory for a file that has the same name as *file2*, then compares *file2* with the file of the same name in directory *file1*.

The default output script has lines in the following format:

```
1,2 c 3,4
```

The numbers *1,2* refer to line ranges in *file1*, and *3,4* to ranges in *file2*. The range is abbreviated to a single number if the first number is the same as the second. The letter 'c' indicates that lines **1,2** of *file1* should be *changed* to lines **3,4** of *file2*. **diff** then prints the text from each of the two files. Text associated with *file1* is preceded by '< ', whereas text associated with *file2* is preceded by '> '.

The following summarizes **diff**'s options.

-b Ignore trailing blanks and treat more than one blank in an input line as a single blank. Spaces and tabs are considered to be blanks for this comparison.

-c *symbol*
Produce output suitable for the C preprocessor **cpp**; the output contains **#ifdef**, **#ifndef**, **#else**, and **#endif** lines. *symbol* is the string used to build the **#ifdef** statements. If you define *symbol* to the C preprocessor **cpp**, it will produce *file2* as its output; otherwise, it will produce *file1*. Note that this option does *not* work for files that already contain **#ifdef**, **#ifndef**, **#else**, and **#endif** statements.

See Also

commands, egrep

Diagnostics

diff's exit status is 0 when the files are identical, 1 when they are different, and 2 if a problem was encountered (e.g., could not open a file).

Notes

diff cannot handle files with more than 32,000 lines. Handing **diff** a file that exceeds that limit will cause it to fail, with unpredictable side effects.

difftime — Time function (libc)

Return difference between two times
#include <time.h>
double difftime(*newtime, oldtime***) time_t** *newtime, oldtime*;

difftime calculates the difference in seconds between *newtime* and *oldtime*.

Both arguments are of type **time_t**, which is the current system time, and which is defined in the header file **time.h**. Note that the function **time** returns the current time in this format.

Mark Williams C defines the current system time as being the number of seconds since January 1, 1970, 0h00m00s GMT.

See Also
time (overview), time.h

directory — Definition

A **directory** is a table that maps names to files; in other words, it associates the names of a file with their locations on the mass storage device. Under some operating systems, directories are also files, and can be handled like a file.

Directories allow files to be organized on a mass storage device in a rational manner, by function or owner. Note that the documentation for TOS uses the term "folder" as a synonym for "directory".

See Also
file, msh

do — C keyword

Introduce a loop

do is a C control statement that introduces a loop. Unlike **for** and **while** loops, the condition in a **do** loop is evaluated *after* the operation is performed. **do** always works in tandem with **while**; for example

```
do {
        puts("Next entry? ");
        fflush(stdout);
} while(getchar() != EOF);
```

prints a prompt on the screen and waits for the user to reply. The **do** loop is convenient in this instance because the prompt must appear at least once on the screen before the user replies.

See Also
break, C keywords, C language, continue, while
The C Programming Language, page 59

Dosound — xbios function 32 (osbind.h)

Start up the sound daemon
#include <osbind.h>
long Dosound(*buffer***) char** **buffer*;

Dosound starts up a daemon to control the sound generator. *buffer* points to buffer that holds the commands and arguments to be passed to the daemon.

Each command consists of an eight-bit hexadecimal number followed by one or more characters; the commands are as follows:

0x00-0x0F
Each of these commands is followed by a one-character argument; each writes its argument into the appropriate register in the GI sound generator, with 0x00 corresponding to register 0, 0x01 to register 1, and so on. For a fuller explanation of what each register governs in the sound register, see the entry for **Giaccess**.

0x80 This takes a one-character argument and writes it into the temporary register.

0x81 This command takes three one-character arguments. It takes the character that had been loaded into a temporary register with the **0x80** command, loads it into a sound generator register, and controls its execution. The first argument is the number of the register into which the previously stored character is to be loaded. The second argument is a two's-complement number that is added to the contents of the temporary register. The third argument is an end-point value. The instruction that was loaded is executed continually, once each update, and the contents of the temporary register are incremented; this process ends when the value stored in the temporary register equals that of the end-point value.

0x82-0xFF
Each of these commands takes a one-byte argument. If the argument is zero, sound processing is halted. If the argument is greater than zero, it is taken to indicate the number of timer ticks (each tick being 20 milliseconds long) that must pass until the next sound process is performed. In effect, these commands set how long a tone is sustained.

When *buffer* points to a list of commands, **Dosound** returns the old pointer if the sound daemon was active, and NULL if it was not. If *buffer* is set to **-1L**, **Dosound** returns zero if the sound daemon is idle, and a number greater than zero if it is not. This will be helpful in constructing musical resources.

Example
This example generates an interesting series of sounds. Type a key *after* the bell sounds.

```
#include <osbind.h>

char noise[]={
        0xFF, 0x50,         /* Delay a while... */
        0x00, 0xF6,         /* Load reg 0 (Channel A freq, fine) */
        0x01, 0x02,         /* Load reg 1 (Channel A freq, coarse) */
        0x02, 0xDE,         /* Load reg 2 (Channel B freq, fine) */
        0x03, 0x01,         /* Load reg 3 (Channel B freq, coarse) */
        0x04, 0x3F,         /* Load reg 4 (Channel C freq, fine) */
        0x05, 0x01,         /* Load reg 5 (Channel C freq, coarse) */
        0x06, 0x00,         /* Load reg 6 (Noise period) */

        0x07, 0xF8,         /* Load reg 7 (Voice enable) */
        0x08, 0x10,         /* Load reg 8 (Channel A volume) */
        0x09, 0x10,         /* Load reg 9 (Channel B volume) */
        0x0A, 0x10,         /* Load reg A (Channel C volume) */
        0x0B, 0x00,         /* Load reg B (Env period fine tune E) */
        0x0C, 0x30,         /* Load reg C (Env period coarse tune E) */
        0x00, 0x09,         /* Load reg D (Env shape/cycle) */
        0xFF, 0x30,         /* Delay */
        0x00, 0x00,         /* Load reg 0 (Channel A freq, fine) */
        0x01, 0x01,         /* Load reg 1 (Channel A freq, coarse) */
        0x07, 0x3E,         /* Load reg 7 (Voice enable) */
        0x08, 0x0B,         /* Load reg 8 (Channel A vol) */

        0x09, 0x00,         /* Load reg 9 (Channel B vol) */
        0x0A, 0x00,         /* Load reg A (Channel C vol) */
        0x80, 0x01,         /* Init temp register */
        0x81, 0x00, 0x01, 0xFF,
                            /* Loop defined... */
        0x01, 0x02,         /* Next step down */
        0x80, 0x01,         /* Init temp register again */
        0x81, 0x00, 0x01, 0xFF,
                            /* Loop again */
        0x07, 0x3F,         /* Disable voices... */
        0xFF, 0x40,         /* Delay 40 ticks... */
        0x00, 0x34,         /* Load reg 0 (Channel A freq, fine) */
        0x01, 0x00,         /* Load reg 1 (Channel A freq, coarse) */

        0x02, 0x00,         /* Load reg 2 (Channel B freq, fine) */
        0x03, 0x00,         /* Load reg 3 (Channel B freq, coarse) */
        0x04, 0x00,         /* Load reg 4 (Channel C freq, fine) */
        0x05, 0x00,         /* Load reg 5 (Channel C freq coarse) */
        0x06, 0x00,         /* Load reg 6 (Noise period) */
        0x07, 0xFE,         /* Load reg 7 (Voice enable) */
        0x08, 0x10,         /* Load reg 8 (Channel A vol) */
        0x09, 0x00,         /* Load reg 9 (Channel B vol) */
        0x0A, 0x00,         /* Load reg A (Channel C vol) */
        0x0B, 0x00,         /* Load reg B (Env period fine tune E) */
        0x0C, 0x10,         /* Load reg C (Env period coarse tune E) */
        0x00, 0x09,         /* Load reg D (Env shape/cycle) */
        0xFF, 0x00          /* Terminate delay timer */
};
```



```
main() {
        Dosound( noise );           /* Make some noise... */
        while ( Cconis() == 0 )     /* Loop until user types a key */
                Cconws("Listen... ");
        Cconin();                   /* Get the key. */
        Dosound( noise );           /* Make some noise again */
}
```

See Also

daemon, Giaccess, TOS, xbios

double — C keyword

Data type

A **double** is the data type that encodes a double-precision floating-point number. On most machines, **sizeof(double)** is defined as four machine words, or eight **chars**. If you wish your code to be portable, do *not* use routines that depend on a **double** being 64 bits long. Different formats are used to encode **doubles** on various machines. These formats include IEEE, DECVAX, and BCD (binary coded decimal), as described in the entry for **float**. Mark Williams C always uses DECVAX format.

See Also

C keywords, C language, data formats, declarations, float, portability
The C Programming Language, page 34

drtomw — Command

Convert from DRI to Mark Williams format
drtomw *file ...*

drtomw converts an object, an executable object, or an archive from DRI to Mark Williams format. It writes the converted file into a temporary file, which it then writes over the original file. This will fail if the disk with the input files is write-protected or if the input file is set as read-only.

drtomw generates messages to indicate to the user the type of file given as input, whether object file or archive. Normally, the format of a file cannot be distinguished easily by its contents; therefore, **drtomw** distinguishes file format by the suffix to the file name: relocatable objects should have the suffix **.o**, whereas executable objects should have any other extension or no extension at all.

When working with a DRI archive, **drtomw** first converts the archive into a Mark Williams object archive, and then converts all of the object files within it to Mark Williams object files. The archive will still need a "ranlib" header, which may be added by using the command:

```
ar rs archname.a ranlib.sym
```

drtomw converts DRI executable files to Mark Williams format. This involves appending a Mark Williams format header to the end of the file. If characters are present beyond the end of the relocation bytes of the executable file, **drtomw** reports this and aborts the conversion.

See Also

as, as68toas, commands, mwtomw

Notes

The edition of **drtomw** that is shipped with version 3.0 transforms DRI objects into the new Mark Williams object format.

Drvmap — bios function 10 (osbind.h)

Get a map of the logical disk drives
#include <osbind.h>
#include <bios.h>
long Drvmap();

Drvmap returns a bit map of the system's logical configuration of disk drives. In this map, bit 0 corresponds to drive A, bit 1 to drive B, etc.

Example

```
#include <osbind.h>
main() {
        long drivemap;
        int drv;
        long drvmsk=1;
        drivemap = Drvmap();
        puts("Drives on system:\n");

        for(drv = 0 ; drv < 16 ; drv++) {
                if(drvmsk & drivemap)
                        printf("\tdrive %c:\n", (drv+'A'));
                drvmsk <<= 1;
        }
}
```

See Also

bios, bit map, TOS

drvprs — Command

Check if a drive is present on the machine
drvprs [-q] *drive*

The command **drvprs** checks to see if *drive* is present on the machine, where *drive* is the name of a floppy drive, a logical drive on a hard disk, or a RAM disk.

The option **-q** suppresses the message that this command normally returns.

drvprs returns a status value so that it can be used in a **msh** script conditional.

See Also

commands, msh

Dsetdrv — gemdos function 14 (osbind.h)

Make a drive the current drive
#include <osbind.h>
long Dsetdrv(*drive***) int** *drive***;**

Dsetdrv makes *drive* the current disk drive. *drive* can be any integer between 0 and 15, with 0 indicating drive A, 1 indicating drive B, and so on through 15 indicating drive P. **Dsetdrv** returns a bit map of the drive configuration, with bits 0 through 15 indicating drives A through P, respectively. Setting a bit indicates that the corresponding disk drive is present on the system.

Example

This example sets the default drive to **B:**. Upon exiting, the default drive is reset to **A:**.

```
#include <osbind.h>
#define DRIVE_A 0
#define DRIVE_B 1
#define DRIVE_C 2
#define E_DRIVE (-46L) /* Invalid Drive Specified */
main() {
        long drivemap;

        if((drivemap=Dsetdrv(DRIVE_B)) < 0) {
                if(drivemap == E_DRIVE)
                        printf("Invalid drive (%c:) specified.\n",
                                (DRIVE_B + 'A'));
                else
                        printf("GEMDOS error %ld\n", drivemap);
        } else {
                int drv;
                long drvmsk=1;

                printf("Current drive is '%c:'. Others are:\n",
                        (DRIVE_B + 'A'));
                for(drv = 0 ; drv < 16 ; drv++) {
                        if(drvmsk & drivemap)
                                printf("\tdrive %c:\n", (drv+'A'));
                        drvmsk <<= 1;
                }
        }
}
```

See Also

Dgetdrv, Drvmap, gemdos, TOS

Notes

When a program exits, GEMDOS always resets the current drive and the current directory to what they were before the program was run.

Dsetpath — gemdos function 59 (osbind.h)

Set the current directory
#include <osbind.h>
long Dsetpath(*path*) char *path;

Dsetpath sets the current directory; it returns 0 if the directory could be set, and non-zero if it could not. *path* points to the directory's path name, which must be a NUL-terminated string.

Example

This example allows the user to set and display the default path, or get the current path string for device specified. If **drv** equals -1, it uses the default drive and returns a pointer to the path buffer.

```
#include <osbind.h>
char *getpath(pathbuf,drv)
char *pathbuf;
int drv; {
        char *buf;

        buf = pathbuf;                  /* Target buffer */
        if (drv < 0)                    /* If drive is default */
                drv=Dgetdrv();          /* get default drive no. */
        *buf++ = drv+'A';               /* Put drive letter in string */
        *buf++ = ':';                   /* d: */
        Dgetpath(buf, drv+1);           /* get the rest of the path */
        return(pathbuf);                /* Return the buffer address */
}

/*
 * Allow default directory to be changed.
 */

main(argc, argv) int argc; char **argv; {
        char path[80];
        char *dst;
        char *src;

        if(argc < 2) {                  /* No new path? display old */
                Cconws("Current path is ");
                Cconws(getpath(path,-1));
                Cconws("\r\n");
                Pterm0();               /* Then exit. */
        }
        Cconws("Old path was ");
        Cconws(getpath(path,-1));
        Cconws("\r\n");
```



```
        dst = src = argv[1];            /* Get new path */

        while ( *src != '\0' ) {        /* Scan for device */
                if ( *src++ == ':' ) {  /* If found, set device */
                        int drv;        /* Move pointer past ":" */

                        drv = src[-2];
                        if(drv > 'Z')
                                drv -= 'a';
                            else
                                drv -= 'A';
                        if(drv >= 0 && drv <= 15)
                                Dsetdrv(drv);
                        dst = src;
                        break;
                    }
        }
        if (*dst != '\0') {
                if ( Dsetpath(dst) != 0 ) {
                        Cconws("Setpath failed, Path is ");
                        Cconws(getpath(path,-1));
                        Cconws("\r\n");
                        Pterm(1);
                }
        }
        Cconws("Path now set to ");
        Cconws(getpath(path,-1));
        Cconws("\r\n");
        Pterm0();
}
```

See Also

Dgetpath, Dsetdrv, Dgetdrv, gemdos, TOS

Notes

The **msh** functions **pwd** and **cd** maintain their own idea of the current path. Programs, like the example, which reset the current drive tender the shell's data invalid. A **cd** to a completely specified path will fix this.

dup — UNIX system call (libc)

Duplicate a file descriptor
int dup(*fd*) int *fd*;

dup duplicates the existing file descriptor *fd*, and returns the new descriptor. The returned value is the smallest file descriptor that is not already in use by the calling process. *fd* must be less than six under TOS.

Example

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

See Also

fopen, fdopen, STDIO, UNIX routines

Diagnostics

dup returns a number less than zero when an error occurs, such as a bad file descriptor or no file descriptor available.

dup2 — UNIX system call (libc)

Duplicate a file descriptor
int dup2(*fd*, *newfd*) **int** *fd*, *newfd*;

dup2 duplicates the file descriptor *fd*. Unlike its cousin **dup**, **dup2** allows you to specify a new file descriptor *newfd*, rather than having the system select one. If *newfd* is already open, the system closes it before assigning it to the new file. **dup2** returns the duplicate descriptor. Under TOS, *fd* must be greater than five, and *newfd* less than six.

Example

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

See Also

STDIO, UNIX routines

Diagnostics

dup2 returns a number less than zero when an error occurs, such as a bad file descriptor or no file descriptor available.

E

echo — Command

Repeat/expand an argument
echo [-n] [*argument* ...]

echo prints each *argument* on the standard output, placing a space between each *argument*. It appends a newline to the end of the output unless the **-n** flag is present.

If *argument* is a **msh** variable, **echo** will expand it before printing it. For example, if you type

```
set esc=<esc>
set cls=$(esc)E ; echo $cls
```

where **<esc>** indicates the escape character, **echo** will send the characters **<esc>**E to your terminal, which will clear the screen and home the cursor.

See Also

commands, msh

ecvt — General function (libc)

Convert floating-point numbers to strings
char *ecvt(*d*, *prec*, *dp*, *signp*) **double** *d*; **int** *prec*, **dp*, **signp*;

ecvt converts *d* into a NUL-terminated ASCII string of numerals with the precision of *prec*. Its operation resembles that of **printf's %e** operator. **ecvt** rounds the last digit and returns a pointer to the result. On return, **ecvt** sets *dp* to point to an integer that indicates the location of the decimal point relative to the beginning of the string, to the right if positive, to the left if negative; and it sets *signp* to point to an integer that indicates the sign of *d*, zero if positive and nonzero if negative.

Example

The following program demonstrates **ecvt**, **fcvt**, and **gcvt**.

```
char *ecvt(), *fcvt(), *gcvt();
main()
{
        char buf[64];
        double d;
        int i, j;
        char *s, *strcpy();

        d = 1234.56789;
        s = ecvt(d, 5, &i, &j);
        printf("ecvt=\"%s\" i=%d j=%d\n", s, i, j);
        /* prints ecvt="12346" i=4 j=0 */
```

```
    strcpy(s, fcvt(d, 5, &i, &j));
    printf("fcvt=\"%s\" i=%d j=%d\n", s, i, j);
    /* prints fcvt="123456789" i=4 j=0 */

    s = gcvt(d, 5, buf);
    printf("gcvt=\"%s\"\n", s);
    /* prints gcvt="1234.56789" */
}
```

See Also

fcvt, frexp, gcvt, ldexp, modf, printf

Notes

ecvt performs conversions within static string buffers that are overwritten by each execution.

edata — Linker-defined symbol

extern int edata[];

edata is the location after the shared and private data segments. It is defined by the linker when it binds the program together for execution. The value of **edata** is merely an address. The location to which this address points contains no known value, and may be an illegal memory location for the program. The value of **edata** does not change while the program is running.

Example

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

See Also

end, etext

egrep — Command

Extended pattern search

egrep [*option ...*] [*pattern*] [*file ...*]

egrep searches each *file* for occurrences of *pattern* (also called a regular expression). If no *file* is specified, it searches the standard input. Normally, it prints each line matching the *pattern*.

Wildcards

The simplest *patterns* accepted by **egrep** are ordinary alphanumeric strings. **egrep** can also process *patterns* that include the following wildcard characters:

^ Match beginning of line, unless it appears immediately after '[' (see below).

$ Match end of line.

* Match zero or more repetitions of preceding character.

. Match any character except newline.

[*chars*] Match any one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

[^*chars*] Match any character *except* one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

\c Disregard special meaning of character *c*.

| Match the preceding pattern *or* the following pattern. For example, the pattern **cat|dog** matches either **cat** or **dog**. A newline within the *pattern* has the same meaning as '|'.

+ Match one or more occurrences of the immediately preceding pattern element; it works like '*', except it matches at least one occurrence instead of zero or more occurrences.

? Match zero or one occurrence of the preceding element of the pattern.

(...) Parentheses may be used to group patterns. For example, **(Ivan)+** matches a sequence of one or more occurrences of the four letters 'I' 'v' 'a' or 'n'.

Because the metacharacters '*', '?', '$', '(', ')', '[', ']', and '|' are also special to the micro-shell **msh**, patterns that contain those literal characters must be quoted by enclosing *pattern* within double quotation marks.

Options

The following lists the available options:

-b With each output line, print the block number in which the line started (used to search file systems).

-c Print how many lines match, rather than the lines themselves.

-e The next argument is *pattern* (useful if the pattern starts with '-').

-f The next argument is a file that contains a list of patterns separated by newlines; there is no *pattern* argument.

-h When more than one *file* is specified, output lines are normally accompanied by the file name; **-h** suppresses this.

-l Print the name of each file that contains the string, rather than the lines themselves. This is useful when you are constructing a batch file.

-n When a line is printed, also print its number within the file.

-s Suppress all output, just return exit status.

-v Print a line only if the pattern is *not* found in the line.

-y Lower-case letters in the pattern match lower-case *and* upper-case letters on the input lines. A letter escaped with '/' in the pattern must be matched in exactly that case.

See Also

commands

Diagnostics

egrep returns an exit status of zero for success, one for no matches, and two for error.

Notes

egrep uses a deterministic finite automaton (DFA) for the search. It builds the DFA dynamically, so it begins doing useful work immediately. This means that **egrep** is considerably faster than earlier pattern-searching commands, on almost any length of file.

#elif — Preprocessor instruction

Include code conditionally
#elif (*expression*)

#elif is an instruction interpreted by the C preprocessor **cpp**. It can be used within a conditional expression begun with the instructions **#if**, **#ifdef**, or **#ifndef**.

This instruction tells **cpp** that if the condition named in the preceding **#if** or **#ifdef** expression is false and if the current condition succeeds, then include the following lines of code up to the next **#else** or **#endif** instruction.

An **#elif** command can also be coupled with **#else** commands to create several levels of conditions. Note that a conditional expression can have any number of **#elif** statements. For example:

```
#if (condition1)
        int foo = 1;
#elif (condition2)
        int foo = 2;
#elif (condition3)
        int foo = 3;
#else
        int foo = 4;
#endif
```

See Also

cc, cpp, #else, #endif, #if, #ifdef

Notes

Note that all preprocessor commands must be set flush with the left margin, or they will not be executed by **cpp**.

else — C keyword

Introduce a conditional statement

else is the flip side of an **if** statement: if the condition described in the **if** statement fails, then the statements introduced by the **else** statement are executed. For example,

```
if (getchar() == EOF)
        exit(0);
else
        dosomething();
```

exits if the user types **EOF**, but does something if the user types anything else.

See Also

C keywords, C language, if
The C Programming Language, pages 51, 53

#else — Preprocessor instruction

Include code conditionally
#else

#else is an instruction interpreted by the C preprocessor **cpp**. It can be used within a conditional expression initiated by the instructions **#if**, **#ifdef**, or **#ifndef**.

This instruction tells **cpp** that if the conditions named in the preceding **#if** and **#elif** instructions are false, then include the following lines of code up to the next **#endif** instruction. Note that a conditional expression can include any number of **#elif** statements, but can have only one **#else** statement. For example:

```
#if (condition1)
        int foo = 1;
#elif (condition2)
        int foo = 2;
#elif (condition3)
        int foo = 3;
#else
        int foo = 4;
#endif
```

See Also

cc, cpp, #elif #endif, #if, #ifdef, #ifndef

Notes

Note that all preprocessor commands must be set flush with the left margin, or they will not be executed by **cpp**.

end — Linker-defined symbol

extern int end[];

end is the location after the uninitialized data segment; it is defined by the linker when it binds the program together for execution. The value of **end** is merely an address. The location to which it points contains no known value, and may be illegal memory locations for the program. The value of **end** does not change while the program is running.

Example

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

See Also

edata, etext

_end — External data

extern char * _end;

_end is an external variable that points to the end of your program's data space. It is set by the C runtime startup, and can be incremented by the function **sbrk**.

See Also

malloc, maxmem, sbrk

#endif — Preprocessor instruction

End conditional inclusion of code

#endif

#endif is an instruction interpreted by the C preprocessor **cpp**. It ends a conditional expression. For example:

```
#if (condition1)
        int foo = 1;
#elif (condition2)
        int foo = 2;
#elif (condition3)
        int foo = 3;
#else
        int foo = 4;
#endif
```

See Also

cc, cpp, #elif, #else, #if, #ifdef, #ifndef

Notes

Note that all preprocessor commands must be set flush with the left margin, or they will not be executed by **cpp**.

entry — C keyword

Undefined keyword

entry is a C key word that is reserved for future use.

See Also

C keywords, C language

Notes

The draft ANSI standard for the C language eliminates **entry** from the table of keywords.

enum — C keyword

Declare a type and identifiers

An **enum** declaration is a data type whose syntax resembles those of the **struct** and **union** declarations. It lets you enumerate the legal value for a given variable. For example,

```
enum opinion (yes, maybe, no) GUESS;
```

declares type **opinion** can have one of three values: **yes**, **no**, and **maybe**. It also declares the variable **GUESS** to be of type **opinion**.

As with a **struct** or **union** declaration, the tag (**opinion** in this example) is optional; if present, it may be used in subsequent declarations. For example, the statement

```
register enum opinion *op;
```

declares a register pointer to an object of type **opinion**.

All enumerated identifiers must be distinct from all other identifiers in the program. The identifiers act as constants and be used wherever constants are appropriate.

Mark Williams C assigns values to the identifiers from left to right, normally beginning with zero and increasing by one. In the above example, the values of **yes**, **no**, and **maybe** are set, respectively, to one, two, and three. The values often are **int**s, although if the range of values is small enough, the **enum** will be an **unsigned char**. If an identifier in the declaration is followed by an equal sign and a constant, the identifier is assigned the given value, and subsequent values increase by one from that value; for example,

```
enum opinion (yes=50, no, maybe) guess;
```

sets the values of the identifiers **yes**, **no**, and **maybe** to 50, 51, and 52, respectively.

To add **enum** to the formal definition of C, amend the list of type-specifiers in Appendix A of *The C Programming Language* to include **enum-specifier**, and add the following syntax:

```
enum-specifier:
    enum { enum-list }
    enum identifier { enum-list }
    enum identifier
enum-list:
    enumerator
    enum-list , enumerator
enumerator:
    identifier
    identifier = constant-expression
```

See Also

C keywords, C language, declarations

environ — Definition

extern char **environ;

environ is a pointer set by the run-time start-up routine. It points to the environment vector, which is equal to the third argument passed to **main**, **char *envp[];** this, in turn, is the handle that the function **getenv** uses to find the environment.

Example

For an example of how this element is used in a C program, see the entry for **memory allocation**.

See Also

environment, envp

environment — Definition

The **environment** is a set of information that you wish to pass to all programs run on your system. It consists of one or more **environmental variables** that you set; for example, when you set the environmental variable **PATH**, you tell TOS that you wish to pass this information to all programs on your system, including TOS itself.

By changing the environment, you can change the way a command works without rewriting any commands that you may have embedded in batch files, scripts, or **makefiles**.

Mark Williams C's compiler controller **cc** uses the environment extensively to find its subordinate programs and files. For example, the environmental variable **INCDIR** tells **cc** where to find its header files. Embedding the command sets **INCDIR** to indicate the directory **include** on drive A. **cc** will pass this information to the C preprocessor **cpp.prg**, which will then look in that directory for the files that are called with an **#include** statement.

See Also

envp — Definition

Argument passed to main
char *envp[];

envp is an abbreviation for **environmental parameter**. It is the traditional name for a pointer to an array of string pointers passed to a C program's **main** function, and is by convention the third argument passed to **main**.

See the Lexicon entry for **argv** for more information how **envp** and **argv** work together to pass information into a program.

Example

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

See Also

argc, argv, main

EOF — Manifest constant

EOF is an acronym for "end of file". It is the manifest constant defined in **stdio.h** that is used to signal that the end of a file has been reached.

To signal EOF to a program reading from the console keyboard under TOS, you should type **<ctrl-Z>** followed by **<return>** on a line by itself. **<ctrl-Z>** as an EOF signal is implemented by the **read** routine. Programs that use TOS calls to read the console must implement an EOF signal themselves.

Example

The following example echoes characters you type at the keyboard until you type EOF.

```
#include <stdio.h>
main()
{
        int c;
        while((c=getchar())!=EOF)
                putchar(c);
}
```

See Also

manifest constant, stdio.h

equal — Command

Compare two arguments
equal *argument1 argument2*

equal is a test command that is built into **msh**. It tests if *argument1* is equal to *argument2*; it returns zero if they are equal, and a value greater than zero if they are not. The arguments may be absolute values or values returned by another command.

Example

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

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

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

See Also

commands, if, is_set, msh, not, while

errno — UNIX data

External integer for return of error status
extern int errno;

errno is an external integer that Mark Williams C sets to the negative value of any error status returned by TOS to the functions that emulate UNIX system calls. The function **perror** or the array **sys_errlist** can be used to translate the **errno** into text.

The number stored in **errno** is not the TOS error, but is often the absolute value of it. Because all TOS errors are negative, and are defined as such in **errno.h**, the value stored in **errno** must be negated before it can be used to determine what error occurred.

Mathematical functions also use **errno** to indicate classifications of errors on return. **errno** is defined within the header file **errno.h**. Because not every function uses **errno**, it should be polled only in connection with those functions that document its use and the meaning of the various status values.

The error codes returned by TOS are listed in the entry for **error codes**, below.

Example

For an example of using **errno** in a mathematics program, see the entry for **acos**.

See Also

errno.h, error codes, mathematics library, perror, UNIX routines

errno.h — Header file

Error numbers used by errno()
#include <errno.h>

errno.h is a header that defines and describes the error numbers returned by **errno**.

See Also

errno, header file, TOS

error codes — Technical information

The following lists the error codes returned by TOS:

BIOS-level errors:

AE_OK	0L	OK, no error
AERROR	-1L	basic, fundamental error
AEDRVNR	-2L	drive not ready
AEUNCMD	-3L	unknown command
AE_CRC	-4L	CRC error
AEBADRQ	-5L	bad request
AE_SEEK	-6L	seek error
AEMEDIA	-7L	unknown media
AESECNF	-8L	sector not found
AEPAPER	-9L	no paper
AEWRITF	-10L	write fault
AEREADF	-11L	read fault
AEGENRL	-12L	general error
AEWRPRO	-13L	write protect
AE_CHNG	-14L	media change
AEUNDEV	-15L	unknown device
AEBADSF	-16L	bad sectors on format
AEOTHER	-17L	insert other disk

GEMDOS-level errors:

AEINVFN	-32L	invalid function number
AEFILNF	-33L	file not found
AEPTHNF	-34L	path not found
AENHNDL	-35L	too many open files no handles left
AEACCDN	-36L	access denied
AEIHNDL	-37L	invalid handle
AENSMEM	-39L	insufficient memory
AEIMBA	-40L	invalid memory block address

AEDRIVE	-46L	invalid drive was specified
AEXDEV	-48L	cross device rename not documented
AENMFIL	-49L	no more files

Miscellaneous error codes:

AERANGE	-64L	range error
AEINTRN	-65L	internal error
AEPLFMT	-66L	invalid program load format
AEGSBF	-67L	setblock failure due to growth restrictions

See Also

errno, errno.h, perror

etext — Linker-defined symbol

extern int etext[];

etext is the location after the shared and private text (code) segments; it is defined by the linker when it binds the program together for execution. The value of **etext** is merely an address. The location to which it points contains no known value, and may be illegal memory locations for the program. The value of **etext** does not change while the program is running.

Example

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

See Also

edata, end, malloc

evnt_button — AES function (libaes)

Await a specific mouse button event
#include <aesbind.h>
int evnt_button(*clicks, button, state, xptr, yptr, bptr, kptr***)**
int *clicks, button, state, *xptr, *yptr, *bptr, *kptr*;

evnt_button is an AES routine that waits for a specified button event. *clicks* is the number of clicks to await. *button* is the number of the button to await, counting from the left, as follows: 0x1, leftmost button; 0x2, second from left; 0x4, third from left; etc.

state is the button state to await: zero indicates up and one indicates down. **evnt_button** returns zero if an error occurred, and a number greater than zero if one did not.

xptr points to an integer that holds the X coordinate of the mouse pointer. *yptr* points to an integer that holds the Y coordinate of the mouse pointer. *bptr* points to an integer that indicates the button state when the event occurred. Finally, *kptr* points to an integer that represents the states of the control, alt, and shift keys OR'd together, as follows:

0x0	all keys up
0x1	right shift key down
0x2	left shift key down
0x4	control key down
0x8	alt key down

evnt_button returns the number of times the button entered the desired state.

Example

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

See Also

AES, TOS

Notes

evnt_button can wait for only one button event. If you attempt to tell it to wait for button 1 *or* button 2, it will react as if you told it to wait for button 1 *and* button 2, i.e., for both buttons to be pressed simultaneously.

evnt_dclick — AES function (libaes)

Get/set double-click interval
#include <aesbind.h>
int evnt_dclick(*speed, getset***) int** *speed, getset*;

evnt_dclick is an AES routine that gets or sets the mouse's double-click speed. *speed* is the double-click speed, from zero through four, with zero being the slowest and four the fastest. It is ignored if *getset* is set to zero. *getset* is a flag: zero tells AES to return the current speed, and one tells it to set the new speed. **evnt_dclick** returns the old click speed (if *getset* is set to zero) or the new click speed (if it is set to one).

See Also

AES, TOS

evnt_keybd — AES function (libaes)

Await a keyboard event
#include <aesbind.h>
int evnt_keybd()

evnt_keybd is an AES routine that awaits a keyboard event. In other words, it waits for the user to press a key on the keyboard. **evnt_keybd** returns an **int**: the high byte contains the scan code of the key pressed, and the low byte contains the ASCII code, if any, modified by the **<ctrl>** or **<shift>** keys.

Example

The following example prints out the scan code for each key pressed. Pressing the **<return>** key exits.

```
#include <aesbind.h>
#include <gemdefs.h>
#define RETURN 0x1C00

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

main()
{
        unsigned key;
        appl_init();

        for(;;) {
                key = evnt_keybd();

                switch(key) {
                case RETURN:
                        appl_exit();
                        exit(0);

                default:
                        alertf(1, "[1][The scan code is: %x][OK]", key);
                        continue;
                }
        }
}
```

See Also

AES, keyboard, TOS

evnt_mesag — AES function (libaes)

Await a message

#include <aesbind.h>
int evnt_mesag(*buffer***) int** *buffer***[8];**

evnt_mesag is the AES routine that awaits a message. *buffer* is an array of eight integers that holds the message.

GEM uses 12 predefined messages to pass information among its applications. Each is eight **int**s (or "words") long, and each has the following structure:

Word 0	type of message
Word 1	handle of application that sent the message
Word 2	no. of bytes in message beyond 16
Words 3-7	contents of message

The following lists the predefined messages by the value of word 0, as defined in the header file **gemdefs.h**:

MN_SELECTED (menu selected) Word 3 gives the number within its object tree of the title of the selected menu, and word 4 gives the number of the selected item.

WM_REDRAW (redraw a window) Word 3 gives the window's handle; words 4 through 7 give, respectively, the X coordinate, the Y coordinate, the width, and the height of the window to be drawn.

WM_TOPPED (make a window the topmost window) Word 3 gives the window's handle.

WM_CLOSED (close-window box clicked) Word 3 gives the window's handle.

WM_FULLED (full-window box clicked) Word 3 gives the window's handle.

WM_ARROWED (arrow or scroll bar clicked) Word 3 gives the window's handle. Word 4 gives the action requested, as follows:

0	Page up
1	Page down
2	Row up
3	Row down
4	Page left
5	Page right
6	Column left
7	Column right

WM_HSLID (horizontal slider moved) Word 3 gives the window's handle. Word 4 gives the slider's position: zero indicates the leftmost position, and 1,000 the rightmost.

WM_VSLID (vertical slider moved) Word 3 gives the window's handle. Word 4 gives the slider's position: zero indicates the lowest position, and 1,000 the highest.

WM_SIZED (window size altered) Word 3 gives the window's handle. Words 4 through 7 give, respectively, the X coordinate, the Y coordinate, the new width, and the new height.

WM_MOVED (window position altered) Word 3 gives the window's handle. Words 4 through 7 give, respectively, the new X coordinate, the new Y coordinate, the width, and the height.

AC_OPEN (desk accessory opened) Word 3 gives the line in the desk menu that was clicked to open the application. Word 4 gives the desk accessory's menu item identifier, as set by the function **menu_register**.

AC_CLOSE (desk accessory closed) Word 3 gives the desk accessory's menu item identifier, as set by the function **menu_register**.

evnt_mesag always returns one.

Example

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

See Also

AES, TOS, window

Notes

The information included with the message **AC_CLOSE** does not appear to conform to the description given in DRI documentation. The description given above is the result of our experience in working with **evnt_mesag**. Users who wish to perform sophisticated tasks, such as passing messages between applications, should be on guard that the correct information is being passed.

evnt_mouse — AES function (libaes)

Wait for mouse to enter specified rectangle

#include <aesbind.h>
int evnt_mouse(*inout, x, y, w, h, xptr, yptr, bptr, kptr***)**
int *inout, x, y, w, h, *xptr, *yptr, *bptr, *kptr*;

evnt_mouse is the AES routine that waits for the mouse pointer to enter or leave a specified rectangular space on the screen. *inout* tells AES whether to wait for the pointer to enter (zero) or leave (one) the rectangle. Note that the screen manager constantly checks the location of the mouse; it is more accurate to say that **evnt_mouse** waits for the mouse pointer to be found inside or outside the rectangle.

The arguments *x*, *y*, *w*, and *h* hold, respectively, the X coordinate of the target rectangle, its Y coordinate, its width, and its height; all are in rasters.

xptr points to an integer that holds the X coordinate of the mouse pointer. *yptr* points to an integer that holds the Y coordinate of the mouse pointer. *bptr* points to an integer that indicates the button state when the event occurred: zero indicates up and one indicates down. Finally, *kptr* points to an integer that represents the states of the control, alt, and shift keys OR'd together, as follows:

0x0	all keys up
0x1	right shift key down
0x2	left shift key down
0x4	control key down
0x8	alt key down

evnt_mouse always returns one.

See Also

AES, TOS

evnt_multi — AES function (libaes)

Await one or more specified events

#include <aesbind.h>
int evnt_multi(*events, clicks, button, state, m1inout, x1, y1, w1, h1, m2inout, x2, y2, w2, h2, buffer, lowtime, hightime, xptr, yptr, bptr, kptr, key, times***)**
int *events, clicks, button, state, m1inout, x1, y1, w1, h1;*
int *m2inout, x2, y2, w2, h2, buffer*[8], *lowtime, hightime;*
int **key, *times, *xptr, *yptr, *bptr, *kptr;*

evnt_multi is an AES routine that awaits any one of a set of AES events. It is one of the most complex AES functions, and the one most commonly used.

events is a flag that indicates the events for which the process is waiting, as follows:

0x01	keyboard event
0x02	mouse button event
0x04	first defined mouse event
0x08	second defined mouse event
0x10	message from another process
0x20	timer event

clicks is the number of mouse button clicks the process is awaiting. *button* is a mask of the number of the mouse button that the processing is awaiting, from one to 16, as counted from the left:

0x01	leftmost button
0x02	second button from left
0x04	third button from left

Note that as of this writing no mouse has more than three buttons.

state is the button state being awaited: zero indicates up, and one indicates down.

evnt_multi can await either of two mouse events. A "mouse event" occurs when the mouse pointer either enters into or exits from a defined rectangular space on the screen. *m1inout* indicates whether the process is waiting for the mouse pointer to enter (zero) or exit (one) the first mouse rectangle. Note that the screen manager constantly polls the screen to check the location of the mouse; it is more accurate to say that **evnt_multi** waits for the mouse pointer to be found inside or outside the rectangle. The arguments *x1*, *y1*, *w1*, and *h1* define, respectively, the X point of the rectangle to be watched, its Y point, its width, and its height.

m2inout plus *x2*, *y2*, *w2*, and *h2* define the second mouse event. They are defined in exactly the same manner as the arguments for the first mouse event.

buffer is the space into which AES writes any message from another process.

lowtime and *hightime* are, respectively, the low word and the high word of the time interval that the process will wait before it "times out", in milliseconds.

xptr points to an integer that holds the X coordinate of the mouse pointer when an awaited event occurs. *yptr* points to an integer that holds the Y coordinate of the mouse pointer. *bptr* points to an integer that indicates the button state when the event occurred. Finally, *kptr* points to an integer that represents the states of the control, alt, and shift keys OR'd together, as follows:

0x0	all keys up
0x1	right shift key down
0x2	left shift key down
0x4	control key down
0x8	alt key down

If a keyboard event occurs (that is, if the user presses a key on the keyboard), *key* points to the code of the key pressed. See the Lexicon entry **keyboard** for a table of the key codes.

Finally, *times* points to where to number of times the mouse button entered the desired state.

evnt_multi returns a number that indicates which event occurred, encoded in the same manner as the variable *events*, above.

Example

This example demonstrates how to use **evnt_multi**. It displays a window; the mouse pointer changes from an arrow to a bumblebee when it moves from inside to outside the window. The program exits when a key is typed.

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

struct { int x, y, w, h; } Rectangle;

/* place for unused pointers to point at */
int nowhere[11];

main()
{
        /* declarations for window */
        int handle;
        char *title = " TITLE ";

        /* declarations for evnt_multi() */
        unsigned int which = (MU_KEYBD | MU_M1 | MU_M2);

        /* no. of times mouse button enters state */
        int times = 0;
        appl_init();
```



```
        /* get dimensions of desktop window */
        wind_get(0, WF_WORKXYWH, &Rectangle.x, &Rectangle.y,
                &Rectangle.w, &Rectangle.h);

        /* alter size of Rectangle */
        Rectangle.x = Rectangle.w/3;
        Rectangle.y = Rectangle.h/3;
        Rectangle.w /= 3;
        Rectangle.h /= 3;

        /* create window */
        handle = wind_create(NAME, Rectangle);

        /* set window, open */
        wind_set(handle, WF_NAME, title, 0, 0);
        graf_growbox(0, 0, 0, 0, Rectangle);
        wind_open(handle, Rectangle);

        for(;;) {
                switch(evnt_multi(which, 1, 1, 1, 0,
                        Rectangle, 1, Rectangle, nowhere,
                        nowhere[0], nowhere[0], nowhere,
                        nowhere, nowhere, nowhere, nowhere,
                        &times)) {

                case MU_KEYBD:
                        wind_close(handle);
                        graf_shrinkbox(0, 0, 0, 0, Rectangle);
                        appl_exit();
                        exit(0);

                case MU_M1:
                        graf_mouse(ARROW, nowhere);
                        which = (MU_KEYBD | MU_M2);
                        continue;

                case MU_M2:
                        graf_mouse(BUSY_BEE, nowhere);
                        which = (MU_KEYBD | MU_M1);
                        continue;

                default:
                        continue;
                }
        }
}
```

See Also

AES, keyboard, TOS

Notes

Note that, with regard to button events, you can tell **evnt_multi** to wait only for one specified event, e.g., for button 1 to be pressed. If you tell it to wait for button 1 *or* button 2 to be pressed, it will act as if you told it to wait for button 1 *and* button 2 to be pressed.

evnt_timer — AES function (libaes)

Wait for a specified length of time
#include <aesbind.h>
int evnt_timer(*lowtime, hightime***) int** *lowtime, hightime*;

evnt_timer is an AES routine that awaits a timer event, i.e., that waits for a given length of time to pass. The time interval to wait before "timing out" is given in milliseconds. *lowtime* is the low word of the time interval, and *hightime* is the high word. **evnt_timer** always returns one.

Example

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

See Also

AES, TOS

Notes

As of this writing, using **evnt_timer** within a desk accessory will cause the system to crash *if* the desk accessory performs any calls to a GDOS routine. For more information on GDOS, see the entries for **VDI** and **metafile**.

executable file — Definition

An **executable file** is one that can be loaded directly by the operating system and executed. Normally, an executable file is one that has both been *compiled*, where it is rendered into machine language, and *linked*, where the compiled program has received all operating system-specific information and library functions.

See Also

file

execve — UNIX system call (libc)

Execute a command from within a program
int execve(*file, argv, env***)**
**char ****file*, ******argv*[], ******env*[]

execve permits you to tell to TOS execute a specific command. This is done through the GEMDOS call **Pexec**. The calling program is suspended while the command is being executed; it returns when the command has finished executing. *file* is the complete path name of the file to be executed. *argv* points to a list of arguments to be passed to the command. *env* points to a list of status environmental parameters. If the **Pexec** status is negative, then **errno** is set to the absolute value of the status.

See Also

environment, Pexec, system, UNIX routines

exit — General function (libc)

Terminate a program
void exit(*status***) int** *status*;

exit terminates a program gracefully. It flushes all buffers, closes each open file, and then returns *status* to the calling program. By convention, an exit status of zero indicates success, whereas an exit status greater than zero indicates failure. Some systems, such as the Series III under ISIS, throw away the status. On TOS, it is returned to the parent program as the result of **Pexec**.

Example

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

See Also

_exit, runtime startup, system, UNIX routines
The C Programming Language, page 154

exit — Command

Exit from a **msh** shell
exit [*status*]

exit terminates the lowest level of the shell **msh**. If you are working in a sub-shell, such as MicroEMACS invokes with its command **<ctrl-X>!**, **exit** returns you to the program that invoked the sub-shell; otherwise, **exit** terminates **msh** and returns you to the GEM desktop. **msh** executes **exit** directly. The optional argument *status* is an integer which is returned as the exit status.

See Also

commands, msh

_exit — UNIX system call (libc)

Terminate a program
int _exit(*status***) int** *status*;

_exit terminates a program directly. It returns *status* to the calling program, and exits.

Unlike the library function **exit**, **_exit** does not perform extra termination cleanup, such as flushing buffered files and closing open files.

_exit should be used only in situations where you do *not* want buffers flushed or files closed; for example, when your program detects an irreparable error condition, and you want to "bail out" to keep your data files from being corrupted.

_exit should also be used with programs that do not use STDIO. Unlike **exit**, **_exit** does not use STDIO. This will help you create programs that are extremely

small when compiled.

See Also

exit, Pterm, runtime startup, system, UNIX routines

Notes

Programs should normally terminate via **exit**, which flushes buffered I/O and closes associated files. Note that on the Atari ST, **_exit** is implemented via the function **Pterm**.

exp — Mathematics function (libm)

Compute exponent
#include <math.h>
double exp(z) double z;

exp returns the exponential of *z*, or *e^z*.

Example

The following program prompts you for a number, then prints the value for it as returned by **exp**, **pow**, **log**, and **log10**.

```
#include <math.h>

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

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

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

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

                display(x);
                display(exp(x));
                display(pow(10.0,x));
                display(log(exp(x)));
                display(log10(pow(10.0,x)));
        }
}
```


See Also

errno, mathematics library

Diagnostics

exp indicates overflow by an **errno** of **ERANGE** and a huge returned value.

extern — C keyword

Declare storage class

extern indicates that a C element belongs to the *external* storage class. Both variables and functions may be declared to be **extern**. Use of this keyword tells the C compiler that the variable or function is defined outside of the present file of source code. All functions and variables defined outside of functions are implicitly **extern** unless declared **static**.

When a source file references data that are defined in another file, it must declare the data to be **extern**, or the linker will return an error message of the form:

```
undefined symbol name
```

For example, the following declares the array **tzname**:

```
extern char tzname[2][32];
```

When a function calls a function that is defined in another source file or in a library, it should declare the function to be **extern**. In the absence of a declaration, **extern** functions are assumed to return **int**s, which may cause serious problems if the function actually returns a 32-bit pointer (such as on the 68000 or i8086 LARGE model), a **long**, or a **double**.

For example, the function **malloc** appears in a library and returns a pointer; therefore, it should be declared as follows:

```
extern char *malloc();
```

If you do not do so, Mark Williams C will assume that **malloc** returns an **int**, and generate the error message

```
integer pointer pun
```

when you attempt to use **malloc** in your program.

See Also

auto, C keywords, C language, pun, register, static, storage class
The C Programming Language, pages 28, 72, 204

F

fabs — Mathematics function (libm)

Compute absolute value
#include <math.h>
double fabs(*z*) double *z*;

fabs implements the absolute value function. It returns *z* if *z* is zero or positive, or -*z* if *z* is negative.

Example

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

See Also

abs, ceil, floor, frexp, mathematics library

Fattrib — gemdos function 67 (osbind.h)

Get and set file attributes
#include <osbind.h>
long Fattrib(*name, readset, setatrib*) char *name;
int *readset, setatrib*;

Fattrib gets and sets file attributes. *name* points to the file's name, which must be a NUL-terminated string. *readset* contains a 0 if you wish to read the file's attributes, or a 1 if you wish to set them. *setatrib* contains an integer that encodes the file's attributes, as follows: 0x01, read only; 0x02, hidden from directory search; 0x04 set to system, hidden from directory search; 0x08, contains volume label in first 11 bytes; 0x10, file is a subdirectory; and 0x20, file has been written to and closed. **Fattrib** returns the file's attributes if they have been read successfully; otherwise, it cannot be relied on to return meaningful information.

Example

```
#include <osbind.h>
extern int errno;

char *atrtable[] = {
        "read only",
        "hidden",
        "system file",
        "volume label",
        "subdirectory",
        "written to and closed"
};
```



```
main(argc, argv) int argc; char **argv; {
        int attribs;
        unsigned point;
        int i;

        if (argc < 2) {
                printf("Usage: Fattrib file\n");
                Pterm(1);
        }
        if ((attribs = Fattrib(argv[1], 0, 0)) < 0) {
                printf("Can't Fattrib file %s --\n", argv[1]);
                errno = -attribs;
                perror("Fattrib failure");
                Pterm(1);
        }

        printf("File %s:", argv[1]);
        if (attribs == 0) {
                printf(" normal file\n");
                Pterm0();
        }
        point = 1;
        for (i=0 ; i<6 ; i++) {
                if (point & attribs)
                        printf(" (%s)", atrtable[i]);
                point <<= 1;
        }
        printf("\n");
}
```

See Also

gemdos, TOS

Fclose — gemdos function 62 (osbind.h)

Close a file
#include <osbind.h>
long Fclose(*handle*) int *handle*;

Fclose closes a file. *handle* is the file handle that was returned by **Fopen()**, **Fcreate()**, **Fdup()**, or inherited by the process. **Fclose** returns 0 if the file could be closed, and non-zero if it could not.

Example

For example of how to use this macro, see the entries for **Fseek** and **Fcreate**.

See Also

gemdos, TOS

fclose — STDIO function (libc)

Close stream
#include <stdio.h>
int fclose(*fp*) FILE *fp;

fclose closes the stream **fp**. It calls **fflush** on the given *fp*, closes the associated file, and releases any allocated buffer. The library function **exit** calls **fclose** for open streams.

Example

For examples of how to use this function, see the entries for **fopen** and **fseek**.

See Also

STDIO
The C Programming Language, page 153

Diagnostics

fclose returns **EOF** if an error occurs.

Fcreate — gemdos function 60 (osbind.h)

Create a file
#include <osbind.h>
long Fcreate(*name, type***) char ***name*; **int** *type*;

Fcreate creates a file. *name* points to the file's path name, which must be a NUL-terminated string. *type* contains a number that encodes the file's attributes, as follows: 0x01, read-only; 0x02, hidden from directory search; 0x04, set to system, hidden from directory search; and 0x08, contains volume label in first 11 bytes. If a file could be created, **Fcreate** returns a handle with which it can be accessed through TOS. If a file could not be created, **Fcreate** returns an error code. Note that all TOS error codes are negative.

Example

The following example, when compiled, takes two arguments, *file1* and *file2*; it then copies *file1* into *file2*. If *file2* does not exist, it is created.

```
#include <osbind.h>
#include <stdio.h>
#include <stat.h>
extern int errno;

main(argc, argv) int argc; char **argv;
{
        int status;
        int inhand;
        int outhand;
        struct DMABUFFER *mydta;
        char *buffer;
        long copysize;

        if (argc < 3) {
                Cconws("Usage: Fcreate source target\r\n");
                Pterm(1);
        }
```



```
if ((inhand = Fopen(argv[1], 0)) < 0) {
        fprintf(stderr,"\nCan't open input file %s",argv[1]);
        errno = -inhand;
        perror("Fopen failure");
        Pterm(1);
}

Fsetdta(mydta=(struct DMABUFFER *)malloc(sizeof(struct DMABUFFER)));

if ((status=Fsfirst(argv[1], 0xF7)) != 0) {
        Fclose(inhand);
        fprintf(stderr,"\nError getting stats on input file %s",
                        argv[1]);
        errno = -status;
        perror("Fsfirst failure");
        Pterm(1);
}

status = mydta->d_fattr & 7;

if((outhand = Fcreate(argv[2], status)) < 0) {
        Fclose(inhand);
        fprintf(stderr,"\nCan't open output file %s",argv[2]);
        errno = -outhand;
        perror("Fcreate failed");
        Pterm(1);
}

buffer = (char *)malloc(4096);
copysize = mydta->d_fsize;
while (copysize>4096) {
        if ((status=Fread(inhand, 4096L, buffer)) < 0) {
                Fclose(inhand);
                Fclose(outhand);
                Fdelete(argv[2]);
                fprintf(stderr,"\nRead error on %s", argv[1]);
                errno = -status;
                perror("Read failure");
                Pterm(1);
        }

        if ((status=Fwrite(outhand, 4096L, buffer)) < 0) {
                Fclose(inhand);
                Fclose(outhand);
                Fdelete(argv[2]);
                fprintf(stderr,"\nWrite error on file %s", argv[2]);
                errno = -status;
                perror("Write failure");
                Pterm(1);
        }
```

```
            copysize -= 4096;
        }
        if (copysize > 0) {
            if ((status=Fread(inhand, copysize, buffer)) < 0) {
                Fclose(inhand);
                Fclose(outhand);
                Fdelete(argv[2]);
                fprintf(stderr,"\nRead error on %s", argv[1]);
                errno = -status;
                perror("Read failure");
                Pterm(1);
            }

            if ((status=Fwrite(outhand, copysize, buffer)) < 0) {
                Fclose(inhand);
                Fclose(outhand);
                Fdelete(argv[2]);
                fprintf(stderr,"\nWrite error on %s", argv[2]);
                errno = -status;
                perror("Write failure");
                Pterm(1);
            }
        }

        Fclose(inhand);
        Fclose(outhand);
        printf("File %s copied to file %s.\n", argv[1], argv[2]);
        free(mydta);
        Fsetdta(NULL);
        Pterm0();
}
```

See Also

gemdos, TOS

fcvt — General function (libc)

Convert floating point numbers to ASCII strings

char *fcvt(*d, w, dp, signp***) double** *d*; **int** *w*, **dp*, **signp*;

fcvt converts floating point numbers to ASCII strings. Its operation resembles that of the %f operator to **printf**. It converts *d* into a NUL-terminated string of decimal digits with a precision (i.e., the number of characters to the right of the decimal point) of *prec*. It rounds the last digit and returns a pointer to the result. On return, **fcvt** sets *dp* to point to an integer that indicates the location of the decimal point relative to the beginning of the string: to the right if positive, and to the left if negative. Finally, it sets *signp* to point to an integer that indicates the sign of *d*: zero if positive, and nonzero if negative. **fcvt** rounds the result to the FORTRAN F-format.

Example

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

See Also

ecvt, frexp, gcvt, ldexp, modf, printf

Notes

fcvt performs conversions within static string buffers that are overwritten by each execution.

Fdatime — gemdos function 87 (osbind.h)

Get or set a file's date/time stamp

#include <osbind.h>

long Fdatime(*info, handle, getset***)**

int *handle, getset, info*[2];

Fdatime retrieves or sets a file's time/date stamp. *handle* is the file's handle that was set when the file was first opened. *getset* indicates whether the stamp is to be reset or retrieved: 0 indicates get, and 1 indicates set. *info* points to a buffer that holds two integers; this buffer will either has the time/date stamp written into it, or hold the new time/date stamp that is to replace the previous stamp, depending on whether the stamp is to be retrieved or reset. In either case, the first integer of *info* encodes the time and the second integer encodes the date, as follows:

info[1]	0-4	no. of two-second increments (0-29)
	5-10	no. of minutes (0-59)
	11-15	no. of the hour (0-23)
info[2]	0-4	day of the month (1-31)
	5-8	no. of the month (1-12)
	9-15	no. of the year (0-119, 1980 = 0).

Fdatime returns an error status, or zero.

Example

The following example demonstrates **Fdatime**.

```
#include <osbind.h>
#include <errno.h>
#include <time.h>

main(argc, argv)
int argc; char *argv[]; {
        int fd;
        rtetd_t rtd;                    /* Backwards time, date */
        utetd_t utd;                    /* Forwards date, time */
        time_t t;                       /* COHERENT time */
        tm_t *tp;                       /* Time fields */
```

```
        if (argc < 2) {
                printf("Usage: Fdatime <filename>\n");
                exit(1);
        }

        if ((fd = Fopen(argv[1], 0)) < 0) {
                errno = -fd;
                perror(argv[1]);
                exit(1);
        }

        Fdatime(&rtd, fd, 0);
        utd.g_date = rtd.g_rdate;
        utd.g_time = rtd.g_rtime;
        tp = tetd_to_tm(utd);
        t = jday_to_time(tm_to_jday(tp));

        printf("%s", asctime(tp));
        printf("%s", ctime(&t));
        return 0;
}
```

See Also

gemdos, TIMEZONE, TOS

Notes

msh updates the time it returns by one hour if the daylight savings time flag is set in the environmental parameter **TIMEZONE**. Therefore, during the summer months, the time returned by this routine may be one hour behind the time returned by the **date** command.

Note, too, that **Fdatime** overwrites the time/date buffer, even when you are setting the time and date on a file.

Fdelete — gemdos function 65 (osbind.h)

Delete a file

#include <osbind.h>
long Fdelete(*name*) char *name;

Fdelete deletes a file. *name* points to the file's name, which must be a NUL-terminated string. **Fdelete** returns 0 if the file could be deleted, and non-zero if it could not.

Example

For examples of how to use this macro, see the entries for **Fseek** and **Fcreate**.

See Also

gemdos, TOS

fdopen — STDIO function (libc)

Open a stream for standard I/O

#include <stdio.h>

FILE *fdopen(*fd, type*) int *fd*; char **type*;

fdopen allocates and returns a **FILE** structure, or *stream*, for the file descriptor *fd*, as obtained from **open**, **creat**, or **dup**. *type* is the manner in which you want *fd* to be opened, as follows:

r	read a file
w	write into a file
a	append onto a file

Example

The following example obtains a file descriptor with **open**, and then uses **fdopen** to build a pointer to the **FILE** structure.

```
#include <ctype.h>
#include <stdio.h>

main(argc, argv)
int argc; char *argv[];
{
        extern FILE *fdopen();
        FILE *fp;
        int fd;
        int holder;

        if (--argc != 1)
                adios("Usage: example filename");

        if ((fd = open(argv[1], 0)) == -1)
                adios("open failed.");
        if ((fp = fdopen(fd, "r")) == NULL)
                adios("fdopen failed.");

        while ((holder = fgetc(fp)) != EOF)
        {
                if ((holder > '\177') && (holder < ' '))
                        switch(holder)
                        {
                        case '\t':
                        case '\n':
                                break;
                        default:
                                fprintf(stderr, "Seeing char %d\n", holder);
                                exit(1);
                        }
                fputc(holder, stdout);
        }
}

adios(message)
char *message;
{
        fprintf(stderr, "%s\n", message);
        exit(1);
}
```

See Also

creat, dup, fopen, open, STDIO

Diagnostics

fdopen returns NULL if it cannot allocate a FILE structure. Currently, only 20 FILE structures can be allocated per program, including **stdin**, **stdout**, and **stderr**.

Fdup — gemdos function 69 (osbind.h)

Generate a substitute file handle
#include <osbind.h>
long Fdup(*handle*) int *handle*;

Fdup generates a substitute file handle for a standard file handle: between zero and five, inclusive. It returns the new, non-standard file handle if successful, or the error code **EINHNDL** (invalid handle) or **ENHNDL** (no handles left, i.e., too many files open) if not.

See Also

gemdos, TOS

Notes

Fdup returns with no error indication if the argument it is passed is a file handle that has been processed by **Fclose**; however, the system will generate an address error when the process terminates.

feof — STDIO macro (stdio.h)

Discover stream status
#include <stdio.h>
int feof(*fp*) FILE **fp*;

feof is a macro that tests the status of the argument stream *fp*. It returns a number other than zero if *fp* has reached the end of file, and zero if it has not. One use of **feof** is to distinguish a value of -1 returned by **getw** from an **EOF**.

Example

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

See Also

STDIO

ferror — STDIO macro (stdio.h)

Discover stream status
#include <stdio.h>
int ferror(*fp*) FILE **fp*;

ferror is a macro that tests the status of the file stream *fp*. It returns a number other than zero if an error has occurred on *fp*. Any error condition that is discovered will persist either until the stream is closed or until **clearerr** is used to clear it. For write routines that employ buffers, **fflush** should be called before **ferror**, in case an error occurs on the last block written.

Example

This example reads a word from one file and writes it into another.

```
#include <stdio.h>

main()
{
        FILE *fpin, *fpout;
        int word;
        char infile[20], outfile[20];

        printf("Name data file you wish to copy:\n");
        gets(infile);
        printf("Name new file:\n");
        gets(outfile);

        if ((fpin = fopen(infile, "rb")) != NULL) {
                if ((fpout = fopen(outfile, "wb")) != NULL) {
                        while ((word = fgetw(fpin)) != EOF)
                                fputw(word, fpout);

                        if (!ferror(fpin)) {
                                clearerr(fpin);
                                printf("Read error\n");
                                exit(0);
                        }
                }
                else
                        printf("Cannot open output file\n");
        }
        else
                printf("Cannot open output file\n");
        fclose(fpin);
        fclose(fpout);
}
```

See Also

STDIO

fflush — STDIO function (libc)

Flush output stream's buffer
#include <stdio.h>
int fflush(*fp*) FILE **fp*;

fflush flushes any buffered output data associated with the file stream *fp*. The file stream stays open after **fflush** is called. **fclose** calls **fflush**, so there is no need for you to call it when normally closing a file or buffer.

Example

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

See Also

buffer, gets, STDIO, write

Diagnostics

fflush returns **EOF** if it cannot flush the contents of the buffers; otherwise it returns a meaningless value.

Note, also, that all STDIO routines are buffered. **fflush** should be used to flush the output buffer if you follow a STDIO routine with an unbuffered routine, such as **Cconin**.

Fforce – gemdos function 70 (osbind.h)

Force a file handle
#include <osbind.h>
long Fforce(*shandle, nshandle***) int** *shandle, nshandle***;**

Fforce forces the standard file handle, i.e., zero through five, to point to the same file as the non-standard file handle, i.e., six and up. **Fforce** returns **E_OK** (no error) if successful, or **EIHNDL** (invalid handle) if not.

See Also

Fdup, gemdos, TOS

fgetc – STDIO function (libc)

Read character from stream
#include <stdio.h>
int fgetc(*fp***) FILE ***fp***;**

fgetc reads characters from the input stream *fp*. It is a function whose body is the macro **getc**. In general, it behaves the same as **getc**; it runs more slowly than **getc**, but yields a smaller object module when compiled.

Example

This example counts the number of lines and "sentences" in a file.

```
#include <stdio.h>
main()
{
        FILE *fp;
        int ch, nlines, nsents;
        int filename[20];
        nlines = nsents = 0;

        printf("Enter file to test: ");
        gets(filename);
        if ((fp = fopen(filename,"r")) != NULL) {
                while ((ch = fgetc(fp)) != EOF) {
                        if (ch == '\n')
                                ++nlines;

                        else if (ch == '.' ||
                                ch == '!' || ch == '?') {
                                if ((ch = fgetc(fp)) != '.') {
                                        ++nsents;
                                        ungetc(ch, fp);
                                }
                                else for(ch='.'; (ch=fgetc(fp))=='.';)
                                                ;
                        }
                }
                printf("%d line(s), %d sentence(s).\n",
                        nlines, nsents);
        } else
                printf("Cannot open %s.\n", filename);
}
```

See Also

getc, STDIO

Diagnostics

fgetc returns EOF at end of file or on error.

Fgetdta – gemdos function 47 (osbind.h)

Get a disk transfer address
#include <osbind.h>
#include <stat.h>
(DMABUFFER *)Fgetdta()

Fgetdta gets and returns the disk transfer address that had been set by **Fsetdta**, and will be used by **Fsfirst** and **Fsnext**.

Example

The following example creates a version of the **find** utility for TOS. It generates a full path name and description for every file in your file system; its output can be piped to if you wish to find where you stored a particular file, as follows:

```
find | egrep filename
```

This example demonstrates the TOS functions **Fgetdta, Fsetdta, Fsfirst**, and **Fsnext**. It also demonstrates the use of **isascii, isupper, free, malloc, strcat, strcpy, strlen**, and **tolower**.

This example also demonstrates how to use the global variable **_stksize** to check for stack overflow.

```
#include <osbind.h>
#include <stat.h>
#include <ctype.h>
extern long _stksize;

/* Translate string to lower case */
char *lowercase(name)
char *name;
{
      register char *p = name; register int c;
      while (c = *p) *p++ = isascii(c) && isupper(c) ? tolower(c) : c;
      return name;
}

/* Concatentate path suffix to path prefix */
char *dircat(pfx, sfx)
register char *pfx, *sfx;
{
      extern char *malloc(), *strcat();
      register char *p; register int nb, npfx;
      nb = (npfx = strlen(pfx)) + 1 + strlen(sfx) + 1;

      if ((p = malloc(nb)) == 0) exit(1);
      strcpy(p, pfx);
      if (npfx != 0 && pfx[npfx-1] != '\\') strcat(p, "\\");
      return strcat(p, sfx);
}

/* Search the directory specified by dname */
find(name)
char *name;
{
      register char *globname, *newname; DMABUFFER dumb, *saved;

      if ((long)&saved <= _stksize+128) {
            printf("Stack near overflow in find()\n\r"); return;
      }

      globname = dircat(name, "*.*");
      saved = (DMABUFFER *)Fgetdta();
      Fsetdta(&dumb);

      if (Fsfirst(globname, 0xFF) == 0) {
            do {
                  if (dumb.d_fname[0] != '.') {
                        newname = dircat(name, dumb.d_fname);
                        printf("%s\n", lowercase(newname));
                        find(newname);
                        free(newname);
                  }
            } while (Fsnext() == 0);
      }
      free(globname);
      Fsetdta(saved);
}
```



```
main()
{
      find(""); return 0;
}
```

See Also

Fsetdta, Fsfirst, Fsnext, gemdos, TOS

fgets — STDIO function (libc)

Read line from stream

#include <stdio.h>

char *fgets(*s***, *n*, *fp*) char *s; int *n*; FILE *fp;**

fgets reads characters from the stream *fp* into string *s* until either *n*-1 characters have been read, or a newline or EOF is encountered. It retains the newline, if any, and appends a NUL character at the end of of the string. **fgets** returns the argument *s* if any characters were read, and **NULL** if none were read.

Example

This example looks for the pattern given by **argv[1]** in standard input or in file **argv[2]**. It demonstrates the functions **pnmatch**, **fgets**, and **freopen**.

```
#include <stdio.h>
#define MAXLINE 128
char buf[MAXLINE];

main(argc, argv)
int argc; char *argv[];
{
      if (argc != 2 && argc != 3)
            fatal("Usage: pnmatch pattern [ file ]");
      if (argc==3 && freopen(argv[2], "r", stdin)==NULL)
            fatal("cannot open input file");
      while (fgets(buf, MAXLINE, stdin) != NULL)
      {
            if (pnmatch(buf, argv[1], 1))
                  printf("%s", buf);
      }
      if (!feof(stdin))
            fatal("read error");
      exit(0);
}

fatal(s) char *s;
{
      fprintf(stderr, "pnmatch: %s\n", s);
      exit(1);
}
```

See Also
STDIO
The C Programming Language, page 155

Diagnostics
fgets returns **NULL** if an error occurs, or if **EOF** is seen before any characters are read.

fgetw — STDIO function (libc)

Read integer from stream
#include <stdio.h>
int fgetw(*fp*) FILE *fp;

fgetw is a function that reads an integer from the stream *fp*.

Example
This example copies one binary file into another. It demonstrates the functions **fgetw** and **fputw**.

```
#include <stdio.h>

main()
{
        FILE *fpin, *fpout;
        int word;
        char infile[20], outfile[20];

        printf("Name data file to copy:\n");
        gets(infile);

        printf("Name new file:\n");
        gets(outfile);

        if ((fpin = fopen(infile, "rb")) != NULL)
        {
                if ((fpout = fopen(outfile, "wb")) != NULL)
                {
                        while ((word = fgetw(fpin)) != EOF)
                                fputw(word,fpout);
                        if (!ferror(fpin))
                                printf("Read error\n");
                } else
                        printf("Cannot open output file\n");
        } else
                printf("Cannot open output file\n");

        fclose(fpin);
        fclose(fpout);
}
```

See Also
fputw, STDIO

Notes
fgetw returns EOF on errors. A call to **feof** or **ferror** may be necessary to distinguish this value from a genuine end-of-file signal.

field — Definition

A **field** is an area that is set apart from whatever surrounds it, and that is defined as containing a particular type of data. In the context of C programming, a field is either an element of a structure, or a set of adjacent bits within an **int**.

See Also
bit map, data formats, structure
The C Programming Language, page 136

file — Definition

A **file** is a mass of bits that has been given a name and is stored on a nonvolatile medium. These bits may form ASCII characters or machine-executable data. Under the UNIX system, the COHERENT system, and related operating systems, external devices can mimic files, in that they can be opened, closed, read, and written to in a manner identical to that of files.

To manipulate the contents of a file, you must first open it. This can be done with the UNIX-compatible routine **open**, or with the function **fopen**. You can then read the file, write material to it, or append material onto it with the low-level UNIX-system calls **read** and write, or with the functions **fread** and **fwrite**. See the entries on **UNIX system calls** and **STDIO** for more information on manipulating material within a file.

See Also
close, executable file, fopen, fclose, FILE, open

file — Command

Name a file's type
file *file* ...

file names the type of each *file* named. It examines files to make an educated guess about their format.

file recognizes the following classes of text files: files of commands to the shell; files containing the source for a C program; files containing assembly language source; files containing unformatted documents that can be passed to **nroff**; and plain text files that fit into none of the above categories.

file recognizes the following classes of non-text or binary data files: the various forms of archives, object files, and link modules for various machines, and miscellaneous binary data files.

See Also

commands, ls, msh, size

Notes

Because **file** only reads a set amount of data to determine the class of a text file, mistakes can happen.

FILE — Definition

Descriptor for a file stream
#include <stdio.h>

FILE describes a *file stream* which can be either a file on disk or a peripheral device through which data flow. It is defined in the header file **stdio.h**. A pointer to FILE is returned by **fopen, freopen, fdopen**, and related functions.

The **FILE** structure is as follows:

```
typedef struct      FILE
{
      unsigned char *_cp,
                    *_dp,
                    *_bp;
      int   _cc;
      int   (*_gt)(),
            (*_pt)();
      int   _ff;
      char  _fd;
      int   _uc;
} FILE;
```

_cp points to the current character in the file. **_dp** points to the start of the data within the buffer. **_bp** points to the file buffer. **_cc** is the size of the file, in characters. **_gt** and **_pt** point, respectively, to the function **getc** and **putc**. **_ff** is a bit map that holds the various file flags, as follows:

_FINUSE	0x01	unused
_FSTBUF	0x02	used by macro **setbuf**
_FUNGOT	0x04	used by **ungetc**
_FEOF	0x08	tested by macro **feof**
_FERR	0x10	tested by macro **ferror**
_FASCII	0x20	file is in ASCII mode
_FWRITE	0x40	file is is opened for writing
_FDONTC	0x80	don't close file

_fd is the file descriptor, which is used by low-level routines like **open**; it is also used by **reopen**. Finally, **_uc** is the character that has been "ungotten" by **ungetc**, should it be used.

See Also

fopen, freopen, stdio.h, stream

file descriptor — Definition

A **file descriptor** is an integer between 1 and 20 that indexes an area in **_psbase**, which, in turn, points to the operating system's internal file descriptors. It is used by routines like **open, close**, and **lseek** to work with files. Note that a file descriptor is *not* the same as a **FILE** stream, which is used by routines like **fopen, fclose**, or **fread**. Note, too, that TOS routines use the term **handle** as a synonym for "file descriptor".

See Also

file, FILE, UNIX routines

fileno — STDIO function (libc)

Get file descriptor
#include <stdio.h>
int fileno(*fp*) FILE *fp;

fileno returns the file descriptor associated with the file stream *fp*. The file descriptor is the integer returned by **open** or **creat**; it is used by routines such as **fopen** used to create a **FILE** stream.

Example

This example reads a file descriptor and prints it on the screen.

```
#include <stdio.h>

main(argc,argv)
int argc; char *argv[];
{
      FILE *fp;
      int fd;

      if (argc !=2)
      {
            printf("Usage: fd_from_fp filename\n");
            exit(0);
      }

      if ((fp = fopen(argv[1], "rw")) == NULL)
      {
            printf("Cannot open input file\n");
            exit(0);
      }

      fd = fileno(fp);
      printf("The file descriptor for %s is %d\n",
            argv[1], fd);
}
```

See Also
FILE, file descriptor, STDIO

flexible arrays — Definition

Flexible arrays are arrays whose length is not declared explicitly. Each has exactly one empty '[ ]' array-bound declaration. If the array is multidimensional, the flexible dimension of the array must be the *first* array bound in the declaration; for example:

```
int example1[][20]; /* RIGHT */
int example2[20][]; /* WRONG */
```

Note that the C language allows you to declare an indefinite number of array elements of a set length, but not a set number of array elements of an indefinite length.

Flexible arrays occur in only a few contexts; for example, as parameters:

```
char *argv[];
char p[][8];
```

as **extern** declarations:

```
extern int end[];
```

as **extern** or **static** initialized definitions:

```
static char digit[]="01234567";
```

or as a member of a structure — usually, though not necessarily, the last:

```
struct nlist {
        struct nlist *next;
        char name[];
};
```

See Also
array, data types

float — C keyword

Data type

Floating point numbers are a subset of the real numbers. Each has a built-in radix point (or "decimal point") that shifts, or "floats", as the value of the number changes. It consists of the following: one sign bit, which indicates whether the number is positive or negative; bits that encode the number's *exponent*; and bits that encode the number's *fraction*, or the number upon which the exponent works. In general, the magnitude of the number encoded depends upon the number of bits in the exponent, whereas its precision depends upon the number of bits in the fraction.

The exponent often uses a *bias*. This is a value that is subtracted from the exponent to yield the power of two by which the fraction will be increased.

Floating point numbers come in two levels of precision: single precision, called **floats**; and double precision, called **doubles**. With most microprocessors, **sizeof(float)** returns four, which indicates that it is four **chars** (bytes) long, and **sizeof(double)** returns eight.

Several formats are used to encode **floats**, including IEEE, DECVAX, and BCD (binary coded decimal). Mark Williams C uses DECVAX format throughout.

The following describes DECVAX, IEEE, and BCD formats, for your information.

DECVAX Format

The 32 bits in a **float** consist of one sign bit, an eight-bit exponent, and a 24-bit fraction, as follows:

```
Sign  Exponent 1     Fraction
|s   eeeeeee|e   fffffff|ffffffff|ffffffff|
  Byte 4       Byte 3      Byte 2   Byte 1
```

The exponent has a bias of 129.

If the sign bit is set to one, the number is negative; if it is set to zero, then the number is positive. If the number is all zeroes, then it equals zero; an exponent and fraction of zero plus a sign of one ("negative zero") is by definition not a number. All other forms are numeric values.

The most significant bit in the fraction is always set to one and is not stored. It is usually called the "hidden bit".

The format for **doubles** simply adds another 32 fraction bits to the end of the **float** representation, as follows:

```
Sign  Exponent          Fraction
|s    eeeeeee|e  fffffff|ffffffff|ffffffff|
  Byte 8       Byte 7      Byte 6   Byte 5

      fffffff|ffffffff|ffffffff|ffffffff|
       Byte 4   Byte 3   Byte 2   Byte 1
```

IEEE Format

The IEEE encoding of a **float** is the same as that in the DECVAX format. Note, however, that the exponent has a bias of 127, rather than 129.

Unlike the DECVAX format, IEEE format assigns special values to several floating point numbers. Note that in the following description, a *tiny* exponent is one that is all zeroes, and a *huge* exponent is one that is all ones:

- A tiny exponent with a fraction of zero equals zero, regardless of the setting of the sign bit.
- A huge exponent with a fraction of zero equals infinity, regardless of the setting of the sign bit.

- A tiny exponent with a fraction greater than zero is a denormalized number, i.e., a number that is less than the least normalized number.
- A huge exponent with a fraction greater than zero is, by definition, not a number. These values can be used to handle special conditions.

An IEEE **double**, unlike DECVAX format, increases the number of exponent bits. It consists of a sign bit, an 11-bit exponent, and a 53-bit fraction, as follows:

```
Sign      Exponent            Fraction
|s    eeeeeee|eeee  ffff|ffffffff|ffffffff|
  Byte 8         Byte 7    Byte 6   Byte 5

      ffffffff|ffffffff|ffffffff|ffffffff|
       Byte 4   Byte 3   Byte 2   Byte 1
```

Note that the exponent has a bias of 1,023. The rules of encoding are the same as for **floats**.

BCD Format

The BCD ("binary coded decimal") format is used in accounting to eliminate rounding errors that alter the worth of an account by a fraction of a cent. For that reason, BCD format consists of a sign, an exponent, and a chain of four-bit numbers, each of which is defined to hold the digits zero through nine.

A BCD **float** has a sign bit, seven bits of exponent, and six four-bit digits, as follows:

```
Sign Exponent     Fraction
 |s    eeeeeee|   dddd  dddd|dddd dddd|dddd dddd|
     Byte 4         Byte 3      Byte 2     Byte 1
```

A BCD **double** has a sign bit, 11 bits of exponent, and 13 four-bit digits, as follows:

```
Sign     Exponent       Fraction
|s     eeeeeee|eeee  dddd|dddd dddd|dddd dddd|
    Byte 8       Byte 7      Byte 6     Byte 5

      dddd dddd|dddd dddd|dddd dddd|dddd dddd|
        Byte 4     Byte 3     Byte 2     Byte 1
```

Passing the hexadecimal numbers A through F in a digit yields unpredictable results.

The following rules apply when handling BCD numbers:

- A tiny exponent with a fraction of zero equals zero.
- A tiny exponent with a fraction of non-zero indicates a denormalized number.
- A huge exponent with a fraction of zero indicates infinity.
- A huge exponent with a fraction of non-zero is, by definition, not a number; these non-numbers are used to indicate errors.

See Also

C keywords, C language, data formats, declarations, double
The Art of Computer Programming, vol. 2, page 180*ff*
The C Programming Language, page 34

floor — Mathematics function (libm)

Set a numeric floor
#include <math.h>
double floor(z) double z;

floor sets a numeric floor. It returns a double-precision floating point number whose value is the largest integer less than or equal to *z*.

Example

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

See Also

abs, ceil, fabs, frexp, mathematics library

Flopfmt — xbios function 10 (osbind.h)

Format tracks on a floppy disk
#include <osbind.h>
#include <xbios.h>
int Flopfmt(*buffer, intbuf, device, sectors, track, side, interleave, magic, new*)
char **buffer*;
int **intbuf, device, sectors, track, side, interleave, new*;
long *magic*;

Flopfmt formats a track on a floppy disk. The Atari SF314 and SF354 floppy disk drives each support 80 tracks per disk, and zero to ten 512-byte sectors per track. The SF314 supports single- and double-sided media, whereas the SF354 supports only single-sided media.

buffer points to a word-aligned area of memory that is large enough to hold the image of an entire track. This is the number of bytes per sector (512), times the number of sectors (ten), plus the overhead information. This comes to approximately eight kilobytes for a nine-sector track, or nine kilobytes for a ten-sector track. All data in this area are overwritten during the format and verify operation. In the case of a format failure, a zero-terminated list of bad sectors is returned to this array as an array of **int**s.

device is the number of the floppy disk drive. Zero indicates drive A and one indicates drive B. Any other value yields unpredictable results.

sectors is the number of sectors to be formatted onto each track, one through ten. The standard format uses nine tracks.

track is the number of the track that you wish to format, from zero through 79. Any attempt to format beyond track 79 may damage the drive.

side is the side of the floppy disk on which you wish to write, i.e., zero or one. Any attempt to format side 1 with an SF354 drive will fail and may hang the system.

interleave gives the sector interleave factor. With Mega STs, this can be -1, which specifies that the pointer *intbuf* contains the sector numbers in the order in which they appear within the track. This will not work on earlier versions of TOS, and will yield unpredictable results if used.

magic is a magic number that TOS uses to verify the operation. It must be set to 0x87654321L.

new is the value with which to fill the newly formatted sectors. It must not be zero, and the high nybble of neither byte can be 0xF. This is a word value, and the recommended value is 0xE5E5, which sets up a good test pattern in the format.

Flopfmt returns zero if the track was formatted correctly, and nonzero if an error occurred. If bad sectors are discovered, their numbers are written into the area pointed to by *buffer*, in consecutive words terminated by zero. Bad sectors do not cause **Flopfmt** to return an error, so the bad sector list should always be checked. If bad sectors are found, you can either reformat the track, use the bad-sector information to mark the bad sectors in the FAT so that GEMDOS will not use them, or reject the floppy disk altogether.

Flopfmt forces the "changed" status (used by the functions **Mediach** and **Rwabs**) to "definitely changed."

Example

This example formats a single-sided floppy disk and initializes the first two tracks. It demonstrates **Flopfmt**, **Flopwr**, and **Protobt**.

```
#include <stdio.h>
#include <osbind.h>

#define BLANK (0xE5E5)          /* Standard sector format value */
#define MAGIC (0x87654321L)     /* Mandatory magic number value */
#define BUFSIZE (9*1024)        /* Buffer size for 9 sectors */

extern int errno;               /* Error number for perror() */

main()
{
        int track;              /* Track counter */
        int side;               /* Side counter */
        int status;             /* Status word... */
        short *bf;              /* Buffer ptr. */
        char reply;             /* Reply... */
        short *middle;          /* Pointer for bad block dump */
```



```
side = 0;       /* Only format side 0 */
printf("Really format disk in drive B? ");
fflush(stdout);
if ((reply = Crawcin()) != 'y' && reply != 'Y') {
        printf("No. Floppy in drive B not formatted.\n");
        Pterm0();
}

printf("Yes\n");
printf("Press any key when ready...");
fflush(stdout);
Crawcin();
printf("\n");
bf = (short *) malloc(BUFSIZE);

for (track=0; track<80; track++) {
        printf("now formatting track %d:", track);
        fflush(stdout);
        status = Flopfmt(bf, NULL, 1, 9, track, side,
                1, MAGIC, BLANK);

        if (status) {
                middle = bf;
                printf("\t%d\n", status);
                while (*middle) {
                        printf("\tBad sector %d\n", *middle++);
                }
        } else {
                printf("\tokay\n");
        }
}

printf("Format of all tracks completed\n");
printf("Any key to continue...");
fflush(stdout);
Crawcin();
printf("Initializing directory structure\n");

/*
 * Now, clear out the first two tracks (all zeros...
 * First, zero out the buffer...
 */
for (track = 0; track < (BUFSIZE>>1); bf[track++] = 0);

/* Now, write it to all sectors of the first two tracks */
for (track=0;track<2;) {
        printf("Zeroing track %d.\n", track);
        if (status = Flopwr(bf, 0L, 1, 1, track++, 0, 9)) {
                errno = -status;
                perror("Flopwr failure");
        }
}

/* Now, we will prototype the boot block... */
Protobt(bf, (long)Random(), 2, 0);
```

```
        /* Finally, write this out to the boot sector... */
        status = Flopwr(bf, 0L, 1, 1, 0, 0, 1);
        if (status) {
                errno = -status;
                perror("Write of boot-block failed.");
        }
        /* Verify the write... */
        status = Flopver(bf, 0L, 1, 1, 0, 0, 1);
        if (status) {
                errno = -status;
                perror("Verify of boot-block failed.");
        }
        printf("Program done. Disk in drive B is formatted.\n");
        free(bf);
        Pterm0();
}
```

See Also

TOS, xbios

Floprd — xbios function 8 (osbind.h)

Read sectors on a floppy disk

#include <osbind.h>
#include <xbios.h>
int Floprd(*buffer, filler, device, sector, track, side, count***)**
char ****buffer*, ****filler***; int** *device, sector, track, side, count*;

Floprd reads one or more sectors on a floppy disk. *filler* is not used, but must be passed properly for this function to work. *buffer* must point to a buffer that is large enough to hold the number of sectors read. *device* is the number of the device, i.e., zero or one. *sector* is the sector at which to begin reading, i.e., one through nine. *track* is the track number to seek to, i.e., zero through 79. *side* is the side of the floppy to read, zero or one. Finally, *count* is the number of sectors to read; this can be no greater than the number of sectors on the track.

Floprd returns zero if the read succeeded, and returns an error code number if it did not.

Example

```
#include <osbind.h>
#include <bios.h>
#define uword(x)     ((unsigned)(x))
#define ulong(x)     ((unsigned long)(x))
#define can2(x,y)    (uword(x)|(uword(y)<<8))
#define can3(x,y,z) (can2(x,y)|(ulong(z)<<16))
```



```
struct bbpb bb;
main() {
        Floprd(&bb, 0L, 1, 1, 0, 0, 1);     /* read the boot block */
        printf("serial number: %lu\n",
                can3(bb.bp_serial[0],bb.bp_serial[1],bb.bp_serial[2]));

        printf("bytes per sector: %u\n",
                can2(bb.bp_bps[0],bb.bp_bps[1]));
        printf("sectors per cluster: %u\n",
                uword(bb.bp_spc));

        printf("reserved sectors: %u\n",
                can2(bb.bp_res[0],bb.bp_res[1]));
        printf("number of fats: %u\n",
                uword(bb.bp_nfats));
        printf("root directory entries: %u\n",
                can2(bb.bp_ndirs[0],bb.bp_ndirs[1]));
        printf("sectors on media: %u\n",
                can2(bb.bp_nsects[0],bb.bp_nsects[1]));

        printf("media descriptor: %u\n",
                uword(bb.bp_media));
        printf("sectors per fat: %u\n",
                can2(bb.bp_spf[0],bb.bp_spf[1]));
        printf("sectors per track: %u\n",
                can2(bb.bp_spt[0],bb.bp_spt[1]));

        printf("heads per device: %u\n",
                can2(bb.bp_nsides[0],bb.bp_nsides[1]));
        printf("hidden sectors: %u\n",
                can2(bb.bp_nhid[0],bb.bp_nhid[1]));
        printf("check sum: %x\n", can2(bb.bp_chk[0],bb.bp_chk[1]));
        return 0;
}
```

See Also

Flopwr, TOS, xbios

Flopver — xbios function 19 (osbind.h)

Verify a floppy disk

#include <osbind.h>
#include <xbios.h>
int Flopver(*buffer, filler, device, sector, track, side, count***)**
char ****buffer*, ****filler***; int** *device, sector, track, side, count*;

Flopver reads a sector from a floppy disk, to verify that it can in fact be read. *buffer* points to a buffer of 1,024 bytes into which a list of bad sectors (if any) will be written. *filler* is not used, and can be initialized to anything. *device* is the number of the floppy disk, and can be set to zero or one. *sector* is the number of the sector to read, one through nine. *track* is the track on which to seek the sector in question, zero through 79. *side* is the side of the disk to read, zero or one. Finally, *count* is the number of sectors to read, and can be no greater than the number of sectors available on a track.

Flopver returns zero if it could read the sector, and returns an error code if it could not. If it found bad sectors, it writes a NUL-terminated string of the numbers of those sectors into *buffer*; otherwise, it writes zero into *buffer*.

Example

For an example of how to use this macro, see the entry for **Flopfmt**.

See Also

Flopfmt, Floprd, Flopwr, TOS, xbios

Flopwr – xbios function 9 (osbind.h)

Write sectors on a floppy disk
#include <osbind.h>
#include <xbios.h>
int Flopwr(*buffer, filler, device, sector, track, side, count*)
char **buffer, *filler*; **int** *device, sector, track, side, count*;

Flopwr writes one or more sectors on a floppy disk. *filler* is not used, but must be passed properly for this function to work. *buffer* points to a buffer that holds the information to written onto the disk. *device* is the number of the device, i.e., zero or one. *sector* is the sector at which to begin writing, i.e., one through nine. *track* is the track number to seek to, i.e., zero through 79. *side* is the side of the floppy on which to write, zero or one. Finally, *count* is the number of sectors to write; this can be no greater than the number of sectors on the track.

Flopwr returns zero if it succeeded in writing the information, and returns an error code number if it did not. Note that writing over the boot sector on the disk (sector 1, side 0, track 0) is not recommended.

Example

For an example of how to use this macro, see the entry for **Flopfmt**.

See Also

Floprd, TOS, xbios

fopen – STDIO function (libc)

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

fopen allocates and initializes a **FILE** structure, or *stream*; opens or creates the file *name*; and returns a pointer to the structure for use by other **STDIO** routines. *name* may refer either to a real file or to one of the devices **aux:**, **con:**, or **prn:**. *type* is a string that consists of one or more of the characters "rwab", to indicate the mode of the string, as follows:

r	read ASCII; error if file not found
rb	read binary data
w	write ASCII; truncate if found, create if not found
wb	write binary data
a	append ASCII; no truncation, create if not found
ab	append binary data
r+	read and write ASCII; no truncation, error if not found
r+b	read and write binary data
w+	write and read ASCII; truncate if found, create if not found
w+b	write and read binary data
a+	append and read ASCII; no truncation, create if not found
a+b	append and read binary data

The modes that contain 'a' set the seek pointer to point at the end of the file so that data may be appended; all other modes set it to point at the beginning of the file.

Note that files opened in ASCII mode, which is the default, will return only printable characters and the newline character '\n'. Text files that use the return character '\r' must be opened in binary mode.

Example

This example copies **argv[1]** to **argv[2]** using STDIO routines. It demonstrates the functions **fopen, fread, fwrite, fclose,** and **feof**.

```
#include <stdio.h>
char buf[BUFSIZ];

main(argc, argv)
int argc; char *argv[];
{
        register FILE *ifp, *ofp;
        register unsigned int n;

        if (argc != 3)
                fatal("Usage: copy source destination");
        if ((ifp = fopen(argv[1], "rb")) == NULL)
                fatal("cannot open input file");
        if ((ofp = fopen(argv[2], "wb")) == NULL)
                fatal("cannot open output file");

        while ((n = fread(buf, 1, BUFSIZ, ifp)) != 0) {
                if (fwrite(buf, 1, n, ofp) != n)
                        fatal("write error");
        }
        if (!feof(ifp))
                fatal("read error");
        if (fclose(ifp) == EOF || fclose(ofp) == EOF)
                fatal("cannot close");
        exit(0);
}
```

```
fatal(s) char *s; {
        fprintf(stderr, "copy: %s\n", s);
        exit(1);
}
```

See Also

FILE, fdopen, freopen, STDIO
The C Programming Language, page 151, 167

Diagnostics

fopen returns **NULL** if it cannot allocate a **FILE** structure, if the *type* string is nonsense, or if the call to **open** or **creat** fails. Currently, only 20 **FILE** structures can be allocated per program, including **stdin**, **stdout**, and **stderr**.

Fopen – gemdos function 61 (osbind.h)

Open a file
#include <osbind.h>
long Fopen(*name, mode*) **char** **name*; **int** *mode*;

Fopen opens a file. *name* points to the file's path name, which must be a NUL-terminated string. *mode* is an integer than encodes the mode in which the file is opened, as follows: zero, read only; one, write only; and two, read or write. If the file can opened, **Fopen** returns a handle by which the file can be accessed through TOS.

If the file cannot be opened, then **Fopen** returns a negative number. Note that this is a **long** negative; some devices, such as **con:** or **prn:**, may return a handle that is negative when examined as a word but positive when examined as a **long**.

Example

For examples of how to use this macro, see the entries for **Fseek** and **Fcreate**.

See Also

gemdos, TOS

for – C keyword

Control a loop
for(*initialization*; *endcondition*; *modification*)

for is a C keyword that introduces a loop. It takes three arguments, which are separated by semicolons ';'. *initialization* is executed before the loop begins. *endcondition* describes the condition which will end the loop. *modification* is the statement that modifies *variable* to control the number of iterations of the loop. For example,

```
for (i=0; i<10; i++)
```

first sets the variable **i** to zero; then it declares that the loop will continue as long as **i** remains less than ten; and finally, increments **i** by one after every iteration of the loop. This ensures that the loop will iterate exactly ten times (from **i==0** through **i==9**). The statement

```
for(;;)
```

will loop until its execution is interrupted by a **break**, **goto**, or **return** statement.

See Also

break, C keywords, C language, continue, while
The C Programming Language, page 56

form_alert – AES function (libaes)

Display an alert box
#include <aesbind.h>
int form_alert(*button, string*) **int** *button*; **char** **string*;

form_alert is an AES routine that displays an alert dialogue box on the screen. An alert dialogue box consists of three elements: an icon, which is selected from a predefined set of three; text, which describes the alert; and one or more "exit buttons", or little boxes that the user clicks to indicate what he wants to do.

button defines which exit button is the default. The default button is drawn with a heavier outline. It is the one selected if the user presses the return key instead of using the mouse. The default is set as follows: zero, no default button; one, first exit button; two, second exit button; and three, third exit button.

string points to the string used with the alert box. The string has the following format:

```
[n] [text] [exit]
```

The square brackets are entered literally. *n* refers to the number of the icon you wish to display, as follows:

0	no icon
1	NOTE icon (exclamation point)
2	WAIT icon (question mark)
3	STOP icon (stop sign)

text is the text displayed within the alert box. An alert box can hold no more than five lines of text, each no longer than 31 characters. A vertical bar '|' indicates a line break. *exit* describes the exit buttons. It can have no more than 20 characters. If you want more than one exit button, separate their texts with a vertical bar. For example,

```
[3] [Cannot find file|Try again?] [Quit|Try again]
```

indicates that you want the STOP icon (icon no. 3), that the box is to have two lines of text ("Cannot find file/Try again?"), and that you want two exit buttons, one marked "Quit" and the other marked "Try again".

form_alert returns the number of the exit button selected.

Example

The following example demonstrates **form_alert**.

```
#include <aesbind.h>

main()
{
        alertf(2, "[1][Alert Box][I dig it]");
}

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

See Also

AES, cc, gem, TOS

form_center — AES function (libaes)

Center an object on the screen

#include <aesbind.h>
#include <obdefs.h>
int form_center(*picture, xptr, yptr, wptr, hptr***)**
OBJECT ****picture*; int ****xptr***, ****yptr***, ****wptr***, ****hptr***;**

form_center is an AES routine that centers an object on the screen.

picture points to the object being manipulated. The type **OBJECT** is defined in the header file **obdefs.h**. See the Lexicon entry on **object** for more information.

The arguments *xptr*, *yptr*, *wptr*, and *hptr* point, respectively, to **int**s that hold the object's X coordinate, its Y coordinate, its width, and its height.

form_center always returns one.

Example

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

See Also

AES, obdefs.h, object, TOS

form_dial — AES function (libaes)

Reserve/free screen space for dialogue

#include <aesbind.h>
int form_dial(*flag, openx, openy, openw, openh, endx, endy, endw, endh***)**
int *flag, openx, openy, openw, openh, endx, endy, endw, endh***;**

form_dial is an AES routine that either reserves space for a dialogue box, or frees space previously reserved. *flag* indicates whether the space is to be reserved or freed, as follows:

0	FMD_START	reserve screen space
1	FMD_GROW	draw "growing box"
2	FMD_SHRINK	draw "shrinking box"
3	FMD_FINISH	free memory, send redraw message

The variables *openx*, *openy*, *openw*, and *openh* give, respectively, the X position, the Y position, the width, and the height of the rectangle from which the dialogue box grows. These values are used only with flags **FM_GROW** or **FM_SHRINK**. *endx*, *endy*, *endh*, and *endw* give, respectively, the X position, the Y position, the width, and the height of the dialogue box itself. All values should be given in rasters.

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

Note that you are responsible for saving and restoring the portion of the screen that is obscured by the **form_dial** dialogue box. **form_dial** will generate a redraw message when invoked with flag **FMD_FINISH**; your program must capture this message and process it properly.

Example

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

See Also

AES, form_do, object, TOS

Notes

The call **form_dial(FMD_FINISH ...);** can be used to force the screen manager to redraw any portion of the screen.

form_do — AES function (libaes)

Handle user input in form dialogue

#include <aesbind.h>
int form_do(*tree, object***)** **OBJECT ****tree***; int** *object***;**

form_do is an AES routine that handles text the user may need to input into an object. *tree* points to the object tree that will accept the text. *object* indicates the object within the tree that has an editable text field; zero indicates that the tree contains no editable text field. **form_do** returns the index of the object that closed the dialogue.

Example

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

See Also

AES, form_dial, object, TOS

form_error — AES function (libaes)

Display a TOS error

#include <aesbind.h>
int form_error(*error*) int *error*;

form_error is an AES routine that displays a preset error alert. *error* is an integer that indicates which error message you wish to display, as follows:

0	Undefined
1	Undefined
2	Cannot find file or folder
3	Same as 2
4	No room to open another document
5	Item with this name already exists
6	Undefined
7	Undefined
8	Not enough RAM to run application
9	Undefined
10	Same as 8
11	Same as 8
12	Undefined
13	Undefined
14	Undefined
15	Specified drive does not exist
16	Cannot delete current folder
17	Undefined
18	Same as 2

The above numbers correspond to error codes under MS-DOS; note, however, that they are *not* the same as GEMDOS or TOS errors. All codes greater than 18 are associated with no specific error message. **form_error** returns the number of the exit button that the user clicked, from one through three. At present, all error alerts have only one exit button.

Example

This example displays the preset error forms.

```
#include <aesbind.h>

main()
{
      int counter;
      appl_init();
      for (counter = 0; counter <= 20; counter++)
            form_error(counter);
      appl_exit();
}
```

See Also

AES, TOS

fprintf — STDIO function (libc)

Print formatted output onto file stream

int fprintf(*fp, format,* [*arg1, ... argN*])
FILE **fp*; **char **format*;**
[data type] *arg1, ... argN*;**

fprintf prints formatted strings onto the file stream *fp*. It uses the *format* string to specify an output format for *arg1* through *argN*.

See **printf** for a description of **fprintf's** formatting codes.

Example

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

See Also

printf, sprintf, STDIO
The C Programming Language, page 152

Notes

Because C does not perform type checking, it is essential that an argument match its specification. For example, if the argument is a **long** and the specification is for an **int**, **fprintf** will peel off the first word of that **long** and present it as an **int**.

At present, **fprintf** does not return a meaningful value.

fputc — STDIO function (libc)

Write character onto file stream

#include <stdio.h>
int fputc(*c, fp*) char *c*; FILE **fp*;

fputc writes the character c onto the file stream *fp*. It returns *c* if *c* was written successfully.

Example

The following example uses **fputc** to write the contents of one file into another.

```
#include <stdio.h>
main()
{
        FILE *fp, *fout;
        int ch;
        int infile[20];
        int outfile[20];

        printf("Enter name to copy: ");
        gets(infile);
        printf("Enter name of new file: ");
        gets(outfile);

        if ((fp = fopen(infile,"r")) != NULL)
        {
                if ((fout = fopen(outfile,"w")) != NULL)
                        while ((ch = fgetc(fp)) != EOF)
                                fputc(ch, fout);
                else
                        printf("Cannot write %s.\n", outfile);
        }

        else
                printf("Cannot read %s.\n", infile);

        fclose(fp);
        fclose(fout);
}
```

See Also
STDIO

Diagnostics
fputc returns EOF when a write error occurs, e.g., when a disk runs out of space.

fputs — STDIO function (libc)

Write string to file stream
#include <stdio.h>
fputs(*string***, *fp*) char *****string*****; FILE *****fp*****;**

fputs writes *string* onto the file stream *fp*. Unlike its cousin **puts**, it does not append a newline character to the end of *string*.

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

See Also
STDIO
The C Programming Language, page 155

fputw — STDIO function (libc)

Write an integer to a stream
#include <stdio.h>

int fputw(*word***, *fp*) int *****word*****; FILE *****fp*****;**

fputw writes *word* onto the file stream *fp*, and returns the value written.

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

See Also
fgetw, STDIO

Diagnostics
fputw returns EOF when an error occurs. A call to **ferror** or **feof** may be needed to distinguish this value from a valid end-of-file signal.

fraction — Definition

A **fraction**, in the context of C programming, is the fractional portion of a floating point number. The term "mantissa" is often used as a synonym for it.

See Also
data formats, double, float, frexp

fread — STDIO function (libc)

Read data from file stream
#include <stdio.h>
int fread(*buffer***, *size*, *n*, *fp*)**
char *****buffer***; unsigned *****size*****, *n*; FILE *****fp*****;**

fread reads *n* items, each being *size* bytes long, from file stream *fp* into *buffer*.

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

See Also
fwrite, STDIO

Diagnostics
fread returns zero upon reading EOF or on error; otherwise, it returns the number of items read.

Fread — gemdos function 63 (osbind.h)

Read a file
#include <osbind.h>
long Fread(*handle***, *n*, *buffer*)**
int *****handle***; long *n*; char *****buffer*****;**

Fread reads *n* bytes from a file opened by **Fopen** or **Fcreate**.

handle is the file handle generated when the file was opened; *buffer* points to the location where the material being read is stored.

Fread returns the number of bytes read if the data were read successfully. If the value returned does not equal *n*, you have reached the end of the file or an error has occurred — usually the former.

Example

For examples of how to use this function, see the entries for **Fseek** and **Fcreate**.

See Also

gemdos, TOS

free — General function (libc)

Return dynamic memory to free memory pool
void free(*ptr*) **char** **ptr*;

free helps you manage the arena. It returns to the free memory pool memory that had previously been allocated by **malloc, calloc,** or **realloc**. **free** marks the block indicated by *ptr* as unused, so the **malloc** search can coalesce it with contiguous free blocks. *ptr* must have been obtained from **malloc, calloc,** or **realloc**.

Example

For an example of how to use this routine, see the entry for **malloc**. For an example of this function in a TOS application, see the entry for **Fgetdta**.

See Also

arena, calloc, malloc, notmem, realloc, setbuf

Diagnostics

free prints a message and calls **abort** if it discovers that the arena has been corrupted. This most often occurs by storing data beyond the bounds of an allocated block.

Frename — gemdos function 86 (osbind.h)

Rename a file
#include <osbind.h>
long Frename(*n, oldpath, newpath*) **int** *n*;
char **oldpath, newpath*;

Frename renames a file. *oldpath* points to the file's old path name, and *newpath* to its new path name; both names must be NUL-terminated strings. *newpath* must not be the name of an existing file. *n* is reserved for TOS, and must be zero. **Frename** can move a file to another subdirectory, but only on the same disk drive. It returns zero if the file could be renamed, non-zero if it could not.

Example

This example renames a file.


```
#include <stdio.h>
#include <osbind.h>

extern int errno;                       /* global for last error... */

main(argc, argv) int argc; char **argv; {
        int status;

        if (argc < 3) {
                printf("Usage: Frename oldname newname\n");
                Pterm(1);
        }
        if ((status=Frename(0, argv[1], argv[2])) != 0) {
                errno = -status;
                perror("Rename failed");
                Pterm(1);
        }
        printf("File %s renamed to %s\n", argv[1], argv[2]);
        Pterm0();
}
```

See Also

gemdos, TOS

freopen — STDIO function (libc)

Open file stream for standard I/O
#include <stdio.h>
FILE *freopen (*name, type, fp*)
char **name*, **type*; **FILE** **fp*;

freopen reinitializes the file stream *fp*. It closes the file currently associated with it, opens or creates the file *name*, and returns a pointer to the structure for use by other STDIO routines. *name* may refer either to a real file or to one of the devices **aux:**, **con:**, or **prn:**.

type is a string that consists of one or more of the characters "**rwab**" (for, respectively, read, write, append, and binary) to indicate the mode of the stream. For further discussion of the *type* variable, see the entry for **fopen**. **freopen** differs from **fopen** only in that *fp* specifies the stream to be used. Any stream previously associated with *fp* is closed by **fclose**. **freopen** is usually used to change the meaning of **stdin, stdout,** or **stderr**.

Example

This example, called **match.c**, looks in **argv[2]** for the pattern given by **argv[1]**. If the pattern is found, the line that contains the pattern is written into the file **argv[3]** or to **stdout**.

```
#include <stdio.h>
#define MAXLINE 128
char buffer[MAXLINE];
```

```
main(argc,argv)
int argc; char *argv[];
{
        FILE *fpin, *fpout;

        if (argc != 3 && argc != 4)
                fatal("Usage: match pattern infile [outfile]");
        if (argc >= 3 && (fpin = fopen(argv[2], "r"))==NULL)
                fatal("Cannot open input file");
        if ((fpout = freopen(argv[3], "w", stdout))==NULL)
                fatal("Cannot open output file");

        while (fgets(buffer, MAXLINE, fpin) != NULL)
        {
                if (pmmatch(buffer, argv[1], 1))
                        fputs(buffer, stdout);
        }

        if (!feof(fpin))
                fatal("read error");
        exit(0);
}

fatal(s)
char *s;
{
        fprintf(stderr, "match: %s\n", s);
        exit(1);
}
```

See Also

fopen, STDIO

Diagnostics

freopen returns **NULL** if the *type* string is nonsense or if the file cannot be opened. Currently, only 20 **FILE** structures can be allocated per program, including **stdin, stdout,** and **stderr.**

frexp — General function (libc)

Separate fraction and exponent
double frexp(*real, ep***) double** *real*; **int** **ep*;

frexp breaks double-precision floating point numbers into fraction and exponent. It returns the fraction *m* of its *real* argument, such that $0.5 <= m < 1$ or $m=0$, and stores the binary exponent *e* in location *ep*. These numbers satisfy the equation $real = m * 2^e$.

Example

This example prompts for a number, then uses **frexp** to break it into its fraction and exponent.


```
#include <stdio.h>

main()
{
        extern char *gets();
        extern double frexp(), atof();
        double real, fraction;
        int *ep;

        char string[64];

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

                real = atof(string);
                fraction = frexp(real, ep);
                printf("%lf is the fraction of %lf\n",
                        fraction, real);
                printf("%d is the binary exponent of %lf\n",
                        *ep, real);
        }
}
```

See Also

atof, ceil, fabs, floor, ldexp, modf

fscanf — STDIO function (libc)

Format input from a file stream
#include <stdio.h>
int fscanf(*fp, format, arg1, ... argN***)**
FILE **fp*; **char** **format*;
[data type] **arg1*, ... **argN*;

fscanf reads the file stream *fp*, and uses the string *format* to format the arguments *arg1* through *argN*, each of which must point to a variable of the appropriate data type.

fscanf returns either the number of arguments matched, or EOF if no arguments matched.

For more information on **fscanf**'s conversion codes, see **scanf.**

Example

The following example uses **fprintf** to write some data into a file, and then reads it back using **fscanf.**

```
#include <stdio.h>

main ()
{
        FILE *fp;
        char let[4];

        /* open file into write/read mode */
        if ((fp = fopen("tmpfile", "wr")) == NULL)
        {
                printf("Cannot open 'tmpfile'\n");
                exit(1);
        }

        /* write a string of chars into file */
        fprintf(fp, "1234");

        /* move file pointer back to beginning of file */
        rewind(fp);

        /* read and print data from file */
        fscanf(fp, "%c %c %c %c",
                &let[0], &let[1], &let[2], &let[3]);
        printf("%c %c %c %c\n",
                let[3], let[2], let[1], let[0]);
}
```

See Also

scanf, sscanf, STDIO
The C Programming Language, page 152

Notes

Because C does not perform type checking, it is essential that an argument match its specification. For that reason, **fscanf** is best used only to process data that you are certain are in the correct data format, such as data previously written out with **fprintf**.

fseek — STDIO function (libc)

Seek on file stream
#include <stdio.h>
int fseek(*fp, where, how***)**
FILE ***fp*; long** *where***; int** *how***;**

fseek changes where the next read or write operation will occur within the file stream *fp*. It handles any effects the seek routine might have had on the internal buffering strategies of the system. The arguments *where* and *how* specify the desired seek position. *where* indicates the new seek position in the file; it is measured from the start of the file if *how* equals zero, from the current seek position if *how* equals one, and from the end of the file if *how* equals two.

fseek differs from its cousin **lseek** in that **lseek** is a UNIX system call and takes a file number, whereas **fseek** is a STDIO function and takes a **FILE** pointer.

Example

This example opens file **argv[1]** and prints its last **argv[2]** characters (default, 100). It demonstrates the functions **fseek, ftell,** and **fclose.**

```
#include <stdio.h>
extern long atol();

main(argc, argv)
int argc; char *argv[];
{
        register FILE *ifp;
        register int c;
        long nchars, size;

        if (argc < 2 || argc > 3)
                fatal("Usage: tail file [ nchars ]");
        nchars = (argc == 3) ? atol(argv[2]) : 100L;
        if ((ifp = fopen(argv[1], "r")) == NULL)
                fatal("cannot open input file");
        if (fseek(ifp, 0L, 2) == -1)            /* Seek to end */
                fatal("seek error");
        size = ftell(ifp);                      /* Find current size */
        size = (size < nchars) ? 0L : size - nchars;

        if (fseek(ifp, size, 0) == -1)          /* Seek to point */
                fatal("seek error");
        while ((c = getc(ifp)) != EOF)
                putchar(c);                     /* Copy rest to stdout */
        if (fclose(ifp) == EOF)
                fatal("cannot close");
        exit(0);
}

fatal(s)
char *s;
{
        fprintf(stderr, "tail: %s\n", s);
        exit(1);
}
```

See Also

ftell, lseek, STDIO

Diagnostics

For any diagnostic error, **fseek** returns -1; otherwise, it returns zero. Note that if **fseek** goes beyond the end of the file, it will not return an error message until the corresponding read or write is performed.

Fseek — gemdos function 66 (osbind.h)

Move a file pointer
#include <osbind.h>
long Fseek(*n, handle, mode***) long** *n***; int** *handle, mode***;**

Fseek moves a file pointer. *handle* is the file's handle, which was generated when the file was opened; *n* is a signed long integer that indicates the number of bytes the pointer is to be moved. *mode* contains an integer that encodes the manner in which the pointer is to be moved, as follows: zero, move *n* bytes from beginning of file; one, move *n* bytes from current location; and two, move *n* bytes from the end of the file. **Fseek** returns the number of bytes that the file pointer is now located from the beginning of the file.

Example

This example demonstrates **Fseek**. It copies one file into another.

```
#include <osbind.h>
#include <stat.h>
#include <errno.h>
char buffer[8192];              /* 8K buffer */

void reverse(buffer,len)
char *buffer; int len;
{
        register char place, *forward, *backward;
        forward = &buffer[0];
        backward = &buffer[len];

        while (forward < backward) {
                place = *--backward;
                *backward = *forward;
                *forward++ = place;
        }
}

fatal(error, msg)
int error; char *msg;
{
        errno = -error;
        perror(msg);
        exit(1);
}

main(argc, argv)
int argc; char *argv[];
{
        int status, infd, outfd, size;
        DMABUFFER dma;

        if (argc < 3) {
                printf("Usage: Fseek source target\n");
                exit(1);
        }

        if ((infd = Fopen(argv[1], 0)) < 0)
                fatal(infd, argv[1]);
        Fsetdta(&dma);
```



```
        if ((status=Fsfirst(argv[1], 0xF7)) != 0)
                fatal(status, argv[1]);
        status = dma.d_fattr & 7;

        if ((outfd = Fcreate(argv[2], status)) < 0)
                fatal(outfd, argv[2]);
        while (dma.d_fsize > 0) {
                if (dma.d_fsize > sizeof(buffer))
                        size = sizeof(buffer);
                else
                        size = dma.d_fsize;

                Fseek(dma.d_fsize-size, infd, 0);
                if ((status=Fread(infd, (long)size, buffer)) < 0)
                        Fdelete(argv[2]), fatal(status, argv[1]);
                reverse(buffer, size);

                if ((status=Fwrite(outfd, (long)size, buffer)) < 0)
                        Fdelete(argv[2]), fatal(status, argv[2]);
                dma.d_fsize -= size;
        }

        Fclose(infd);
        Fclose(outfd);
        printf("File %s copied to file %s.\n", argv[1], argv[2]);
        return 0;
}
```

See Also

Fsnext, gemdos, TOS

Diagnostics

For any diagnostic error, **Fseek** returns -1; otherwise, it returns zero. Note that if **Fseek** goes beyond the end of the file, it will not return an error message until the corresponding read or write is performed.

fselinput — AES function (libaes)

Select a file

#include <aesbind.h>

int fselinput(*directory, file, button***) char *****directory, *****file*****; int *****button**;

fselinput is an AES routine that allows the user to select a file in the current directory, or create a new file. It displays a box on the screen; within the box is a window that shows the contents of *directory*.

The user can use the mouse to scroll through the contents of *directory* and select one; she can also move up or down within the directory tree, or specify a new directory. The box also contains two "exit buttons", one marked "Cancel" and the other marked "OK".

directory, as noted above, points to a buffer that holds the name of the directory being read. Note that *directory* must be large enough to hold the full path name for any file selected, including those selected from subdirectories within the direc-

tory first displayed.

To avoid accidentally creating a C-language escape character, be sure to use two backslashes '\\' to separate elements of the path name. The default directory is named **a:**. The path name must end with a string that indicates which files you wish to examine in the directory; for example, "*.*" displays all the files in a directory, whereas "*.c" displays only the C programs.

If the user clicks a directory, **fsel_input** alters the name in the buffer to which *directory* points in order to reflect this change.

file is the name of the first file in *directory*. It is initialized by AES. If the user selects a file other than the first one in the directory, what *file* points to is also altered to reflect this change.

button points to a integer that indicates which exit button the user selected: zero indicates that she selected the Cancel button, and one indicates that the OK button was selected.

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

Example

The following example demonstrates **fsel_input**. It checks to see if the file you select is present or not.

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

#define assert(x) if (!(x)) alertf(1, "[3][assert| %s  |failed]", #x);

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

main()
{
    register char *cp;
    int result, button;

    static char prefix[128];
    static char separator[] = "\\";
    static char suffix[16] = "*.*";
    static char filename[128];
    static char name[16];
    extern char *strrchr();

    /* open application */
    appl_init();
```



```
    /* build path for current directory */
    cp = prefix;
    *cp++ = Dgetdrv()+'A';
    *cp++ = ':';
    Dgetpath(cp, 0);

    /* ensure mouse pointer is an arrow */
    graf_mouse(ARROW, (int *)0);

    for (;;) {
        /* build string of form "A:foo\\*.*" */
        strcpy(filename, prefix);
        strcat(filename, separator);
        strcat(filename, suffix);

        /* call fsel_input to select file */
        result = fsel_input(filename, name, &button);
        assert(filename[0] != 0);

        if (result == 0 || name[0] == 0 || filename[0] == 0 ||
                button == 0) {
            if (alertf(2, "[2][Cancel|file|selection ][Yes|No]")
                == 1)
                break;
            continue;
        }

        cp = strrchr(filename, '\\');
        assert(cp != 0);
        *cp = 0;

        strcpy(prefix, filename);
        *cp++ = '\\';
        strcpy(suffix, cp);

        /* build query string */
        if (strchr(name, '*') || strchr(name, '?')) {
            strcpy(suffix, name);
            name[0] = 0;
            continue;
        }

        strcpy(cp, name);
        name[0] = 0;

        /* check if file is present */
        if (Fsfirst(filename, 0xFF) >= 0) {
            alertf(1, "[0][File| %s  |found  ][Ok]",
                filename);
        } else {
            alertf(1, "[1][File| %s  |not found  ][Ok]",
                filename);
        }
```

```
        /* see if user wishes to continue */
        if (alertf(1, "[2][Try|another  |file][Yes|No]") == 2)
            break;
    }

    /* tidy up, exit */
    appl_exit();
    return 0;
}
```

See Also

AES, TOS

Fsetdta — gemdos function 26 (osbind.h)

Set disk transfer address

#include <osbind.h>
#include <stat.h>
void Fsetdta(c) DMABUFFER *c;

Fsetdta sets the pointer *c* to the address of a DMA buffer, a 44-byte buffer that can be subsequently used by the macro **Fsfirst**. It returns nothing.

Example

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

See Also

Fgetdta, Fsfirst, gemdos, TOS

Fsfirst — gemdos function 78 (osbind.h)

Search for first occurrence of a file

#include <osbind.h>
#include <stat.h>
int Fsfirst(*name, attrib***) char ****name***; int** ***attrib***;

Fsfirst searches for the first occurrence of a file name. *name* points to the file's name, which must be a NUL-terminated string. *attrib* is an integer that encodes the search's attributes, as follows:

0x00	normal files only; no hidden files, subdirectories, system files, or volume labels will match
0x01	include read-only files
0x02	include files hidden from directory search
0x04	include system files
0x08	include volume-label files
0x10	include subdirectory files
0x20	include files that have been written to and closed

If you specify volume label, no other type of file can be sought. The order in which file matches are found depends on the order in which the files are arranged in the directory, and is not governed by alphabetical order or creation date.

If the search is successful, **Fsfirst** takes the 44-byte DMA buffer that had been created with **Fsetdta**, and fills it as follows: bytes zero through 20, reserved for TOS; byte 21, file attributes; bytes 22-23, the file's time stamp; bytes 24-25, the file's date stamp; bytes 26-29, the file's size; and bytes 30-43, the file's name. The DMA buffer is declared in the header file **stat.h**.

Fsfirst returns **AE_OK** (success) if the search succeeded, and **AEFILNF** (file not found) if it did not.

Example

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

See Also

Fsetdta, Fsnext, gemdos, stat.h, TOS

Fsnext — gemdos function 79 (osbind.h)

Search for next occurrence of file name

#include <osbind.h>
#include <stat.h>
int Fsnext()

Fsnext continues the search for a file, by using the information that had been written into the 44-byte file name buffer by **Fsfirst** or by a previous call to **Fsnext**. If **Fsnext** finds another file with the given name, it updates the DMA buffer to accommodate the name and attributes of the newly found file. The DMA buffer is declared in the header file **stat.h**.

Fsnext returns **E_OK** (success) if the search was successful, and **ENMFIL** (no more files) if it was not.

Example

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

See Also

Fsfirst, gemdos, stat.h, TOS

fstat — General function (libc)

Find file attributes

#include <stat.h>
fstat(*descriptor, statptr***) int** ***descriptor***; struct stat *****statptr**;

fstat returns a structure that contains the attributes of a file. *descriptor* points to the file descriptor, as returned by the library function **fopen**, and *statptr* points to a structure of the type **stat**, which is defined in the header file **stat.h**.

The following summarizes the structure **stat** and defines the permission and file type bits.

```
struct stat {
        dev_t st_dev;
        int_t st_ino;
        unsigned short st_mode;
        short st_nlink;
        short st_uid;
        short st_gid;
        dev_t st_rdev;
        size_t st_size;
        time_t st_atime;
        time_t st_mtime;
        time_t st_ctime;
};

#define S_IJRON 0x01      /* Read-only */
#define S_IJHID 0x02      /* Hidden from search */
#define S_IJSYS 0x04      /* System, hidden from search */
#define S_IJVOL 0x08      /* Volume label in first 11 bytes */
#define S_IJDIR 0x10      /* Directory */
#define S_IJWAC 0x20      /* Written to and closed */
```

The majority of entries in the structure **stat** are there to preserve compatibility with the COHERENT operating system. Most return meaningless values when used on the Atari ST, with the following exceptions: **st_atime**, **st_mtime**, and **st_ctime** all return the time that the file or directory was last modified.

See Also

ls, msh, open, stat, stat.h

Diagnostics

fstat returns -1 if the file is not found or if *statptr* is invalid.

ftell — STDIO function (libc)

Return current position of file pointer
#include <stdio.h>
long ftell(*fp*) FILE *fp;

ftell returns the current position of the seek pointer. Like its cousin **fseek**, **ftell** takes into account any buffering that is associated with the stream *fp*.

Example

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

See Also

fseek, STDIO

function — Definition

A **function** is the C term for a portion of code that is named, can be invoked by name, and that performs a task. Many functions can accept data in the form of arguments, modify the data, and return a value to the statement that invoked it.

See Also

data types, executable file, library, portability

fwrite — STDIO function (libc)

Write onto file stream
#include <stdio.h>
int fwrite(*buffer, size, n, fp*)
char *buffer; unsigned *size, n*; FILE *fp;

fwrite writes *n* items, each of *size* bytes, from *buffer* onto the file stream *fp*.

Example

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

See Also

fread, STDIO

Diagnostics

fwrite normally returns the number of items written. If an error occurs, the returned value will not be the same as *n*.

Fwrite — gemdos function 64 (osbind.h)

Write into a file
#include <osbind.h>
long Fwrite(*handle, n, buffer*) int *handle*; long *n*; char *buffer;

Fwrite writes *n* bytes into a file. *handle* is the file handle that was generated when the file was opened by **Fopen** or **Fcreate**. *buffer* points to the material to be written. **Fwrite** returns *n* if the material was written successfully, and an error code if it was not.

Example

For examples of how to use this macro, see the entries for **Fseek** and **Fcreate**.

See Also

gemdos, TOS

G

galaxy.a — Archive

galaxy.a is an archive that holds the source files for **galaxy**, a program that allows you to simulate on your Atari ST the birth and evolution of spiral galaxies. The source code is self-documenting, and is offered as an extended example of how to manipulate the Atari ST's graphics.

If you wish to compile **galaxy**, you must first extract the source files from the archive. Use the command **cd** to move to the directory where you have stored this archive, then give **msh** the following command:

```
ar xv galaxy.a
```

See Also

ar

gcvt — General function (libc)

Convert floating point number to ASCII string
char *gcvt(*d, prec, buffer***)**
double *d*; **int** *prec*; **char ***buffer*;

gcvt converts a floating point number into an ASCII string. Its operation resembles that of the %g operator to **printf**. **gcvt** converts its argument *d* into a NUL-terminated string of decimal numerals with a precision (i.e., the number of numerals to the right of the decimal point) of *prec*. Unlike its cousins **ecvt** and **fcvt**, **gcvt** uses a buffer that is defined by the caller. *buffer* must point to a buffer large enough to hold the result; 64 characters will always be sufficient.

When generating its output, **gcvt** will mimic **fcvt** if possible; otherwise, it mimics **ecvt**. **gcvt** returns *buffer*.

Example

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

See Also

ecvt; fcvt, frexp, ldexp, modf, printf

gem — Command

Run a GEM program
gem *command args*

gem allows you to run a GEM *command* under the micro-shell **msh**. It resets file handle 2 to the **aux:** device. Unlike its cousin, the **tos** command, **gem** enables the mouse cursor.

gem reads the environment, and will properly use the environmental variables **PATH** and **SUFF**. A GEM program will execute correctly if both the executable and its associated resource file are located in a directory named in **PATH**.

Another way to use **gem** is with a **cd** command. For example,

```
set game='cd c:\games; gem game.prg; cd'
```

allows you to run the GEM application **game.prg** by typing **$game**. When you exit from **game**, you will be returned to your **HOME** directory.

When you are finished, just exit from the GEM program in the normal way, and **gem** will return you to **msh**.

See Also

commands, msh

Notes

Some Atari GEM programs appear to depend on the GEM desktop to perform unspecified clean-up after they run, and thus cannot be run through the **gem** command. These programs include Atari Logo and Atari BASIC. Running these programs under **msh** may damage memory-resident programs, such as RAM disks.

gemdefs.h — Header file

GEM structures and definitions
#include <gemdefs.h>

gemdefs.h is a header file that declares structures and definitions useful for programming in the GEM environment. Many of the mnemonics used through GEM programs are also defined in this file.

See Also

AES, header file, TOS, VDI

gemdos — TOS function

Call a routine from GEM-DOS
#include <osbind.h>
extern long gemdos(*n, arg1...argn***);**

gemdos allows you to call a GEM-DOS routine directly from your program. *n* is the number of the routine, and *arg1* through *argn* are the argument numbers to be used with the routine. In most circumstances, it is unnecessary to use **gemdos** directly, for a library of functions that use it are defined in the header file **osbind.h**.

The following functions use **gemdos**:

0x03	**Cauxin**	Read character from serial port
0x12	**Cauxis**	Return serial port input status
0x13	**Cauxos**	Return serial port output status
0x04	**Cauxout**	Write character to serial port
0x01	**Cconin**	Read character from console

0x0B	**Cconis**	Return console input status
0x02	**Cconout**	Write character to console
0x10	**Cconos**	Return console output status
0x0A	**Cconrs**	Read and edit string from console
0x09	**Cconws**	Write a string to the console
0x08	**Cnecin**	Read character from console, no echo
0x11	**Cprnos**	Check parallel port output status
0x05	**Cprnout**	Write character to parallel port
0x07	**Crawcin**	Read raw character from console
0x06	**Crawio**	Perform raw I/O with console
0x39	**Dcreate**	Create a subdirectory
0x3A	**Ddelete**	Remove a subdirectory
0x36	**Dfree**	Find free space on disk
0x19	**Dgetdrv**	Return current disk drive
0x47	**Dgetpath**	Return current directory
0x0E	**Dsetdrv**	Set the default drive
0x3B	**Dsetpath**	Set the current directory
0x43	**Fattrib**	Get/set file attributes
0x3E	**Fclose**	Close a file
0x3C	**Fcreate**	Create a file
0x57	**Fdatime**	Get/set file's date stamp
0x41	**Fdelete**	Delete a file
0x45	**Fdup**	Duplicate a file's handle
0x46	**Fforce**	Force a file handle
0x2F	**Fgetdta**	Get a disk transfer address
0x3D	**Fopen**	Open a file
0x3F	**Fread**	Read a file
0x56	**Frename**	Rename a file
0x42	**Fseek**	Move a file pointer
0x1A	**Fsetdta**	Set disk transfer address
0x4E	**Fsfirst**	Search for first occurrence of file
0x4F	**Fsnext**	Search for next occurrence of file
0x40	**Fwrite**	Write into a file
0x48	**Malloc**	Allocate dynamic memory
0x49	**Mfree**	Free dynamic memory
0x4A	**Mshrink**	Shrink amount of allocated memory
0x4B	**Pexec**	Load or execute a process
0x4C	**Pterm**	Terminate a process
0x00	**Pterm0**	Terminate a TOS process
0x31	**Ptermres**	Terminate a process but keep in memory
0x20	**Super**	Enter supervisor mode
0x30	**Sversion**	Get current version of TOS
0x2A	**Tgetdate**	Get date
0x2C	**Tgettime**	Get time
0x2B	**Tsetdate**	Set date
0x2D	**Tsettime**	Set time

See Also

osbind.h, TOS

Notes

No **gemdos** function will support a recursive call. Attempting to use a recursive call with a **gemdos** function will crash the system.

Note that all **gemdos** functions are unbuffered. Combining them with buffered I/O routines, such as those in the STDIO library, will lead at best to unpredictable results.

gemout.h — Header file

GEM-DOS file formats and magic numbers

#include <gemout.h>

gemout.h is a header file that declares formats for the GEM-DOS executable files and archives. It also includes a number of "magic numbers" used in handling these formats.

See Also

header file, TOS

Getbpb — bios function 7 (osbind.h)

Get pointer to BIOS parameter block for a disk drive

#include <osbind.h>
#include <bios.h>
(struct bpb *)Getbpb(*device***);**
int *device*;

Getbpb returns a pointer to the BIOS parameter block for a given disk drive. This structure is described in the header file **bios.h**. *device* is an integer that indicates which drive you wish to examine: zero, drive A; one, drive B; etc. If the BIOS parameter block cannot be determined for whatever reason, **Getbpb** returns NULL.

Note that in the DRI bindings, **Getbpb** is declared as returning a **long**. The cast shown in the declaration is necessary to avoid an integer-pointer pun.

Example

The following example dumps the BIOS parameter block for the disk in drive B:.

```
#include <osbind.h>
#include <bios.h>
```

```
main() {
        struct bpb *bp;
        bp = (struct bpb *) Getbpb(1);
        printf("Disk in drive B:\n");
        printf("\tSector Size:\t%5d bytes\n", bp->bp_recsiz);
        printf("\tCluster Size:\t%5d bytes (%d sectors)\n",
              bp->bp_clsizb, bp->bp_clsiz);

        printf("\tDirectory:\t%5d sectors\n", bp->bp_rdlen);
        printf("\tFAT:\t\t%5d sectors\n", bp->bp_fsiz);
        printf("\tData Clusters:\t%5d\n", bp->bp_numcl);
        printf("\tFlags:\t\t %4x\n", bp->bp_flags);
}
```

See Also

bios, TOS

getc — STDIO macro (stdio.h)

Read character from file stream

#include <stdio.h>

int getc(*fp*) FILE *fp;

getc is a macro that reads a character from the file stream *fp*, and returns an **int**.

Example

The following example creates a simple copy utility. It opens the first file named on the command line and copies its contents into the second file named on the command line.

```
#include <stdio.h>

main(argc, argv)
int argc; char *argv[];
{
        int foo;
        FILE *source, *dest;

        if (--argc != 2)
                error("Usage: copy [source] [destination]");

        if ((source = fopen(argv[1], "rb")) == NULL)
                error("Cannot open source file");
        if ((dest = fopen(argv[2], "wb")) == NULL)
                error("Cannot open destination file");

        while ((foo = getc(source)) != EOF)
                putc(foo, dest);
}
```



```
error(string)
char *string;
{
        printf("%s\n", string);
        exit (1);
}
```

See Also

fgetc, getchar, putc, STDIO

The C Programming Language, page 152

Diagnostics

getc returns **EOF** at end of file or on read error.

Notes

Because **getc** is a macro, arguments with side effects probably will not work as expected. Also, because **getc** is a complex macro, its use in expressions of too great a complexity may cause unforeseen difficulties. Use of the function **fgetc** may avoid this.

getchar — STDIO macro (stdio.h)

Read character from standard input

#include <stdio.h>

int getchar()

getchar is a macro that reads a character from the standard input. It is equivalent to **getc(stdin)**.

Example

The following example gets one or more characters from the keyboard, and echoes them on the screen.

```
#include <stdio.h>

main()
{
        int foo;
        while ((foo = getchar()) != EOF)
                putchar(foo);
}
```

See Also

getc, putchar, STDIO

The C Programming Language, page 144, 152

Diagnostics

getchar returns EOF at end of file or on read error.

getcol — Command

Get a color value

getcol *position*

getcol is a command that uses the **xbios** function **Setcolor** to read the color for a position on the current color palette. *position* is the palette position in question, from zero through 15.

See Also

commands, setcolor, Setcolor, TOS

getenv — General function (libc)

Read environmental variable
char *getenv(*VARIABLE***) char ****VARIABLE***;**

A program may read variables from its *environment*. This allows the program to accept information that is specific to it. The environment consists of an array of strings, each having the form *VARIABLE*=*VALUE*. When called with the string *VARIABLE*, **getenv** reads the environment, and returns a pointer to the string *VALUE*.

Example

This example prints the environmental variable **PATH**.

```
#include <stdio.h>

main()
{
        char *env;
        extern char *getenv();

        if ((env = getenv("PATH")) == NULL)
        {
                printf("Sorry, cannot find PATH\n");
                exit(1);
        }
        printf("PATH = %s\n", env);
}
```

See Also

cc, environment, envp, msh

Diagnostics

When *VARIABLE* is not found or has no value, **getenv** returns **NULL**.

Getmpb — bios function 0 (osbind.h)

Copy memory parameter block
#include <osbind.h>
#include <bios.h>
void Getmpb(*pointer***); struct mpb ****pointer***;**

Getmpb tells TOS to copy its memory parameter block into the **mpb** structure pointed to by *pointer*. This structure is described in the header file **bios.h**.

The useful portions of the memory parameter block are described in the example; as of this writing, the memory parameter block does not appear to be utilized by TOS. Note, too, that the lists returned are in system-protected memory; unless the user is in supervisor mode, accessing these lists will generate a bus error.

Example

The following example demonstrates **Getmpb**. It prints out the amount of memory free and memory used.

```
#include <osbind.h>
#include <bios.h>

long chase(cp, mp)
char *cp; register struct mdb *mp;
{
        register long save, total;
        struct mdb mdb;
        printf("%s:\n", cp);
        total = 0;

        while (mp != (struct mdb *)0L) {
                save = Super(0L); mdb = *mp; Super(save);
                total += mdb.md_size;
                printf("\t%06lx: %ld bytes owned by %lx\n",
                        mdb.md_base, mdb.md_size, mdb.md_proc);
                mp = mdb.md_next;
        }

        printf("%ld bytes total.\n", total);
}

main() {
        struct mpb mpb;
        Getmpb(&mpb);
        chase("Free Memory", mpb.mp_free);
        chase("Used Memory", mpb.mp_used);
        return 0;
}
```

See Also

bios, TOS

getpal — Command

Get the color palette settings
getpal

getpal uses the **xbios** function **Setcolor** to read and return the current settings of the color palette.

See Also

commands, Setcolor, setpal, TOS

getphys — Command

Get the base of the physical screen's display
getphys

getphys is a command that uses the **xbios** function **Physbase** to obtain the base of the screen display's physical memory. The address of the base is returned to the standard output.

See Also

commands, Physbase, setphys, TOS

getrez — Command

Get screen's current resolution
getrez

getrez is a command that uses the **xbios** function **Getrez** to read the screen's current resolution. It returns to the standard output a code that indicates the current resolution, as follows: zero indicates low resolution; one, medium resolution; and two, high resolution.

See Also

commands, Getrez, setrez, TOS

Getrez — xbios function 4 (osbind.h)

Read the current screen resolution
#include <osbind.h>
#include <xbios.h>
int Getrez()

Getrez reads the current screen resolution, and returns the following:

0	low resolution
1	medium resolution
2	high resolution

Example

This program prints out the current resolution of the video display. For another example, see the entry for **Prtblk**.

```
#include <osbind.h>
#include <xbios.h>

struct reztab { int r_rez; char *r_name; } reztab[] = {
        GR_LOW, "low",
        GR_MED, "medium",
        GR_HIGH, "high",
        -1, "unknown"
};
```



```
main() {
        register struct reztab *rp;
        register int rez;
        rez = Getrez();

        for (rp = reztab; rp->r_rez != rez && rp->r_rez != -1; rp += 1)
                ;
        printf("Your ST is in %s resolution mode.\n", rp->r_name);
}
```

See Also

TOS, xbios

gets — STDIO function (libc)

Read string from standard input
#include <stdio.h>
char *gets(*buffer***) char *buffer;**

gets reads characters from the standard input into a buffer pointed at by *buffer*. It stops reading as soon as it detects a newline character or EOF. **gets** discards the newline or EOF, appends a NUL character onto the string it has built, and returns another copy of *buffer*.

Example

The following example uses **gets** to get a string from the console; the string is echoed twice to demonstrate what **gets** returns.

```
#include <stdio.h>

main()
{
        char buffer[80];

        printf("Type something: ");
        fflush(stdout);
        printf("%s\n%s\n", gets(buffer), buffer);
}
```

See Also

buffer, fgets, getc, STDIO

Diagnostics

gets returns NULL if an error occurs or if EOF is seen before any characters are read.

Notes

Note that **gets** stops reading the input string as soon as it detects a newline character. If a previous input routine left a newline character in the standard input buffer, **gets** will read it and immediately stop accepting characters; to the user, it will appear as if **gets** is not working at all.

For example, if **getchar** is followed by **gets**, the first character **gets** will receive is

the newline character left behind by **getchar**. A simple statement will remedy this:

```
while (getchar() != '\n')
        ;
```

This throws away the newline character left behind by **getchar**; **gets** will now work correctly.

Getshift — bios function 11 (osbind.h)

Get or set the status flag for shift/alt/control keys
#include <osbind.h>
#include <bios.h>
long Getshift(*flag*) int *flag*;

Getshift gets or sets the status flag for the shift, alt, and control keys. If *flag* is -1, then the status flags of the keys are read and a map returned; if *flag* is any number other than -1, then the flags are set to *flag*, and a map of their previous settings returned. The map is laid out as follows: bit 0, right shift key; bit 1, left shift key; bit 2, control key; bit 3, alt key; and bit 4, caps lock key. If a bit is set to zero, the key is not depressed; if it is set to one, the key is depressed.

Example

This example displays characters, scan codes, and shift states until you type **<ctrl-D>**.

```
#include <osbind.h>
#include <bios.h>
#include <ctype.h>

struct shift { int s_bit; char *s_name; } shift[] = {
        GS_LSH, "left shift",
        GS_RSH, "right shift",
        GS_CTRL, "control",

        GS_ALT,       "alternate",
        GS_CAPS, "caps lock",
        GS_RMB,       "right mouse",
        GS_LMB,       "left mouse",
        0
};

main() {
        register int c, s;
        register long cc;
        register struct shift *sp;

        do {
                cc = Bconin(BC_CON);
                s = Getshift(-1);
                c = cc;                 /* get low word */
                cc >>= 16;    /* get scan code */
                Bconout(BC_RAW, c);
```



```
                if (isascii(c) && ! isprint(c))
                        printf(": ^%c: ", c+'@');
                else
                        printf(":  %c: ", c);
                printf("%02lx:%02x:%02x", cc, c, s);

                for (sp = shift; sp->s_bit > 0; sp += 1)
                        if (s & sp->s_bit)
                                printf("[%s]", sp->s_name);
                printf("\n");
        } while (c != ('D' & (' '-1)));
}
```

See Also

bios, TOS

Gettime — xbios function 23 (osbind.h)

Read the current time
#include <osbind.h>
#include <xbios.h>
long Gettime()

Gettime reads and returns the intelligent keyboard's setting of the current time. It returns a 32-bit value whose bits indicate the following:

0-4	seconds, in two-second increments (0-29)
5-10	minutes (0-59)
11-15	hours (0-23)
16-20	day of the month (1-31)
21-24	month (1-12)
25-31	year (0-119, 0 indicates 1980)

Example

This example gets the keyboard time. If you have not set the keyboard time since you booted your computer, the time returned by this example will not be correct.

```
#include <osbind.h>

main()
{
        register unsigned long time;
        int seconds;
        int minutes;
        int hours;
        int day;
        int month;
        int year;

        time = Gettime();                       /* Get system time */
        seconds = (time & 0x001F) << 1;         /* Bits  0:4 */
        minutes = (time >> 5)  & 0x3F;          /* Bits  5:10 */
        hours = (time >> 11) & 0x1F;            /* Bits 11:15 */
```

```
        day = (time >> 16) & 0x1F; /* Bits 16:20 */
        month = (time >> 21) & 0x0F;      /* Bits 21:24 */
        year = ((time >> 25) & 0x7F)+1980;      /* Bits 25:31 */

        printf("The ATARI ST thinks it is %d sec past %d min\n",
              seconds, minutes);
        printf("past the hour of %d", hours);
        printf(" on %d/%d/%d\n", month, day, year);
}
```

For another example of this function, see the entry for **time**.

See Also

Kgettime, Settime, time, TOS, xbios

Notes

The time data in the bit map returned by **Gettime** is in exactly the reverse order of the data returned by the **gemdos** functions.

getw — STDIO function (libc)

Read word from file stream
#include <stdio.h>
int getw(*fp*) FILE *fp;

getw reads a word (an **int**) from the file stream *fp*.

getw differs from **getc** in that **getw** gets and returns an **int**, whereas **getc** returns either a **char** promoted to an **int**, or EOF. To detect EOF while using **getw**, you must use **feof**.

See Also

getc, STDIO

Notes

getw returns EOF on errors. A call to **feof** or **ferror** may be necessary to distinguish this value from a valid end-of-file signal.

fgetw assumes that the bytes of the word it receives are in the natural byte ordering of the machine; see the entry on **byte ordering** for more information. This means that such files might not be portable between machines.

Giaccess — xbios function 28 (osbind.h)

Access a register on the GI sound chip
#include <osbind.h>
#include <xbios.h>
char Giaccess(*data, register*) char *data*; int *register*;

Giaccess accesses a register on the GI sound chip. *register* is the name of the register being accessed, zero through 15. Bit 7 of this variable indicates whether this register is to be read or written to: zero indicates read, one indicates write.

data is the eight-bit value being passed to the register when this macro is in write mode; if **Giaccess** is in read mode, this value is ignored.

Giaccess returns the value read if in read mode, and a meaningless value if in write mode.

The Atari ST's sound generator is controlled by 16 eight-bit registers. The sound generator itself has three channels, named A, B, and C. Each can be programmed independently. Note that the contents of the address register remain unaltered until reprogrammed, which allows you to use the same data repeatedly without having to resend them. What each register does is listed in the following:

0,1 Set pitch and period length for channel A. The eight bits of register 0 set the pitch, and the first four bits of register 1 control the period length; the lower the number formed by the 12 significant bits of these registers, the higher the pitch of the tone generated.

2,3 Set the pitch and period length for channel B.

4,5 Set the pitch and period length for channel C.

6 The low five bits of this register control the generation of "white noise"; the smaller the value to which these bits are set, the higher the pitch of the noise generated.

7 This register holds an eight-bit map whose bits toggle various aspects of sound generation; for each bit, zero indicates on and one indicates off. The bits control the following functions:

0	Channel A tone
1	Channel B tone
2	Channel C tone
3	Channel A white noise
4	Channel B white noise
5	Channel C white noise
6	Port A; 0=input, 1=output
7	Port B; 0=input, 1=output

8 Bits 0 through 3 set the signal volume for channel A; the settings can be zero through 15, with zero being the softest setting and 15 the loudest. Setting bit 4 indicates that the "envelope" generator, register 13, should be used; in this case, the contents of bits 0 through 3 are ignored.

9 Same as register 8, only for channel B.

10 Same as register 8, only for channel C.

11,12 Control tone generation. A tone is constructed of four aspects: attack, decay, sustain, and release. *Attack* defines how long a tone takes to reach is loudest point; *decay* defines how long that loudest point is held before it softens to the volume that is sustained; *sustain* defines how long the sus-

tained level is held; and *release* defines how long it takes a tone to decay into silence. These registers govern the four aspects of tone generation; register 11 holds the low byte, register 12 the high byte.

13 Bits 0 through 3 set envelope generator's waveform. A tone's "envelope" is the "shape" of the tone generated, which is best studied by experimental listening.

14,15 Control the Atari ST's I/O ports. Register 14 controls port A, and register 15 port B. If set to output by register 7, the contents of these registers can be exported. Note that these ports have nothing to do with sound generation, and are used on the Atari ST to control the floppy disk drives.

Example

This example uses **Giaccess** to set the select lines for the floppy disk drives. It is not recommended that this be done from user programs in general.

```
#include <osbind.h>

prompt(strng)           /* Write prompt; wait for key to be typed */
char *strng;
{
      Cconws(strng);    /* Write the string */
      Crawcin();        /* Wait for a key */
      Cconws("\r\n");   /* CR-LF to console */
}

main() {
      prompt("Let drives stop; then press any key to continue");
      Giaccess((Giaccess(0,14) & 0xF8),14|0x80);
      prompt("Both lights on... Hit any key");
      Giaccess((Giaccess(0,14) & 0xF8)|2,14|0x80);
      prompt("Drive B selected... Hit any key");
      Giaccess((Giaccess(0,14) & 0xF8)|4,14|0x80);
      prompt("Drive A selected... Hit any key");
      Giaccess((Giaccess(0,14) & 0xF8)|6,14|0x80);
      prompt("Neither drive selected... Hit any key");
      Pterm0();
}
```

See Also

Offgibit, Ongibit, TOS, xbios
Programmable Sound Generator Data Manual

GMT — Definition

GMT is an abbreviation of Greenwich Mean Time, the time recorded at the Greenwich Observatory in England, where by international convention the Earth's zero meridian is fixed.

See Also

gmtime, localtime, time, time.h, TIMEZONE

gmtime — Time function (libc)

Convert system time to calendar structure
#include <time.h>
tm *gmtime(*timep***) time_t *timep;**

gmtime converts the internal time from seconds since midnight January 1, 1970 GMT, into fields that give integer years since 1900, the month, day of the month, the hour, the minute, the second, the day of the week, and yearday. It returns a pointer to the structure **tm**, which defines these fields, and which is itself defined in the header file **time.h**. Unlike its cousin, **localtime**, **gmtime** returns Greenwich Mean Time (GMT).

Example

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

See Also

GMT, localtime, time (overview), TIMEZONE

Notes

gmtime is useful only on a system whose time is set to **GMT** rather than to local time. The Mark Williams C time routines read the environmental variable **TIMEZONE** to translate **GMT** automatically into your local time, should you wish. See the entry on **TIMEZONE** for more information on how this works.

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

goto -- C keyword

Unconditionally jump within a function

A **goto** command jumps to the area of the program introduced by a label. Note that a **goto** cannot cross a function boundary.

In the context of C programming, the most common use for **goto** is to exit from a control block or go to the top of a control block. It is used most often to write "ripcord" routines, i.e., routines that are executed when an major error occurs too deeply within a program for the program to disentangle itself correctly.

Example

The following example demonstrates how to use **goto**.

```
#include <stdio.h>

main() {
      char line[80];
```

```
getline:      {
        printf("Enter line: ");
        fflush(stdout);
        gets(line);

/* a series of tests often is best done with goto's */
        if (*line == 'x')
        {
                printf("Bad line\n");
                goto getline;
        }


            else if (*line == 'y')
        {
                printf("Try again\n");
                goto getline;
        }
        else if (*line == 'q')
                goto goodbye;


              else
                goto getline;


    goodbye:
      printf("Goodbye.\n");
      exit(0);
}
```

See Also

C keywords, C language
The C Programming Language, page 62

Notes

The C Programming Language describes **goto** as "infinitely-abusable": *caveat utilitor*.

graf_dragbox — AES function (libaes)

Draw a dragable box
#include <aesbind.h>
int graf_dragbox(*width, height, stx, sty, bx, by, bw, bh, finx, finy*)
int *width, height, stx, sty, bx, by, bw, bh, *finx, *finy*;

graf_dragbox is an AES routine that allows the user to drag a box around the screen. It also sets a boundary rectangle that limits how far the box can be dragged. The boundary can be set to the entire screen, to a window, or to some other delimiter.

width and *height* give, respectively the width and height of the box being dragged, in rasters. Note that the number of raster on the screen varies with the degree of screen resolution; the following gives the dimensions of the screen in rasters, by

resolution:

Resolution	*Width*	*Height*
High	640	400
Medium	640	200
Low	320	200

stx and *sty* give, respectively, the starting X and Y coordinates for the box. *finx* and *finy* point to the coordinates to which the box has been dragged; these values are set by the function.

bx, *by*, *bw*, and *bh* set, respectively, the X coordinate of the boundary rectangle, its Y coordinate, its width, and its height.

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

Example

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

See Also

AES, TOS

Notes

graf_dragbox returns when the mouse button is released. If it is called while the mouse button is up, it returns immediately.

graf_growbox — AES function (libaes)

Draw a growing box
#include <aesbind.h>
int graf_growbox(*stx, sty, stw, sth, finx, finy, finw, finh*)
int *stx, sty, stw, sth, finx, finy, finw, finh*;

graf_growbox is an AES routine that draws a growing box on the screen. The box drawn by **graf_growbox** does not stay on the screen. This routine is designed merely to add a "star wars"-style flourish to GEM programs.

stx, *sty*, *stw*, and *sth* set, respectively, the X coordinate of the origin box (the box from which the growing box starts to grow), its Y coordinate, its width, and its height. *finx*, *finy*, *finw*, and *finh* set in the same way the dimensions of the finish box (the box toward which the growing box grows). The unit of measure for all eight arguments is the number of rasters for the screen. The number rasters on the screen varies with the degree of resolution, as follows:

Resolution	*Width*	*Height*
High	640	400
Medium	640	200
Low	320	200

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

Example

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

See Also

AES, graf_shrinkbox, window AES, gem, graf_shrinkbox, TOS, window

graf_handle – AES function (libaes)

Get a VDI handle
#include <aesbind.h>
int graf_handle(*chwidth, chheight, bwidth, bheight***)**
**int ****chwidth*, ******chheight*, ******bwidth*, ******bheight*;

The AES routine **graf_handle** returns the handle for the physical workstation open for the desktop. It also returns the size of the default system font.

chwidth and *chheight* point, respectively, to the width and height of the default character cell. *bwidth* and *bheight* point, respectively, to the width and height of a square box (corrected for aspect ratio) that contains a character. This is the size of the window boxes. These values are set by GEM.

See Also

AES, TOS

Notes

A desk accessory that does not call **graf_handle** will not have its desk menu item displayed, and the desktop's desk menu item will fail to function, even though it is displayed.

The VDI handle that **graf_handle** returns is the handle that AES uses to implement all of its graphics library routines. You should not use this handle unless you wish to alter the AES graphics. For example, if you reset the fill pattern using the handle returned by **graf_handle**, you may or may not find the window manager using your fill pattern to fill the title bars of windows as it redraws them.

graf_mbox – AES function (libaes)

Move a box
#include <aesbind.h>
int graf_mbox(*width, height, fromx, fromy, tox, toy***)**
int *width, height, fromx, fromy, tox, toy*;

graf_mbox is an AES routine that moves a box without changing its size. *width* and *height* are the dimensions of the box. *fromx* and *fromy* give the original position of the box; *tox* and *toy* the destination position of the box. Note that both of these pairs of coordinates refer to the upper left-hand corner of the box being moved. **graf_mbox** returns zero if an error occurred, and a number greater than zero if one did not.

See Also

AES, TOS

graf_mkstate – AES function (libaes)

Get the current mouse state
#include <aesbind.h>
int graf_mkstate(*xptr, yptr, bptr, kptr***) int ****xptr*, ******yptr*, ******bptr*, ******kptr*;

graf_mkstate is an AES routine that returns the current mouse state. *xptr* point to an integer that holds the X coordinate of the mouse pointer. *yptr* points to a integer that holds the Y coordinate of the mouse pointer. *bptr* points to an intege that indicates the button state when the event occurred: zero indicates up and on indicates down. Finally, *kptr* points to an integer that represents the states of th control, alt, and shift keys OR'd together, as follows:

0x0	all keys up
0x1	right shift key down
0x2	left shift key down
0x4	control key down
0x8	alt key down

These values are set by GEM.

graf_mkstate always returns one.

See Also

AES AES, TOS

graf_mouse – AES function (libaes)

Change the shape of the mouse pointer
#include <aesbind.h>
int graf_mouse(*form, shape***) int** *form*; **int** *shape*[37];

graf_mouse is an AES routine that changes the mouse pointer from the defaul arrow to another shape. *form* is an integer that indicates what new shape yo want, as follows:

0	**ARROW**	arrow (default)
1	**TEXT_CRSR**	vertical line (text cursor)
2	**BUSY_BEE**	bee
3	**POINT_HAND**	hand with pointing finger
4	**FLAT_HAND**	hand with extended fingers
5	**THIN_CROSS**	thin cross hairs
6	**THICK_CROSS**	thick cross hairs
7	**OUTLN_CROSS**	outlined cross hairs
255	**USR_DEF**	user-described shape
256	**M_OFF**	hide mouse pointer

257 M_ON show mouse pointer

shape is a 37-word array that specifies a new shape for the pointer. This argument is ignored if *form* has any value other than 255.

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

Example

The following example cycles through the preset shapes for the mouse pointer.

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

/*
 * array used to build unique mouse-pointer shape.
 * ANSI C standard states that it's OK to end array
 * initialization with a ','.
 */
int mouse[37] = {
        7, 7, 1, 0, 1,
        0x7FFF, 0x7007, 0x780F, 0x5C1D, 0x4E39, 0x4771,
        0x4361, 0x4001, 0x4361, 0x4771, 0x4E39, 0x5C1D,
        0x780F, 0x7007, 0x7FFF, 0x0000, 0x7FFF, 0x7007,
        0x780F, 0x5C1D, 0x4E39, 0x4771, 0x4361, 0x4001,
        0x4361, 0x4771, 0x4E39, 0x5C1D, 0x780F, 0x7007,
        0x7FFF, 0x0000,
};

main()
{
        int counter;
        appl_init();

        /* draw "canned" mouse pointer shapes */
        for (counter = ARROW; counter <= OUTLN_CROSS; counter++) {
                graf_mouse(counter, (int *)0);
                evnt_keybd();
        }

        /* draw user-defined pointer shape */
        graf_mouse(USER_DEF, mouse);
        evnt_keybd();

        appl_exit();
        return(0);
}
```

For further examples, see the entries **evnt_multi, object, window.**

See Also

AES, object, vsc_form, window AES, object, TOS, vsc_form, window

Notes

Mixing AES mouse calls with VDI mouse calls can produce unpredictable results.

graf_mouse and **vsc_form** use the same 37-word mouse form descriptor. The call

```
graf_mouse(USER_DEF, form);
```

is exactly equivalent to:

```
int handle;
handle = graf_handle(&handle, &handle, &handle, &handle);
vsc_form(handle, form);
```

This is an instance of how the AES uses the VDI to implement the higher-level functions that it provides.

graf_rubbox – AES function (libaes)

Draw a rubber box

#include <aesbind.h>
int graf_rubbox(*x, y, w, h, newwidth, newheight*)
int *x, y, w, h, *newwidth, *newheight*;

graf_rubbox is an AES routine that draws a "rubber box" on the screen. A rubber box is one whose dimensions can be altered by the user. *x*, *y*, *w*, and *h* define the initial dimensions of the rubber box: respectively, they define its X coordinate, its Y coordinate, its width, and its height. All dimensions are in rasters.

newwidth and *newheight* point to the values for width and height to be set by the user's manipulation of the box.

This routine can be used to define a block of screen area that can be copied elsewhere. For example, the GEM desktop routine that allows you to select a group of files at once uses **graf_rubbox**.

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

Example

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

See Also

AES, TOS

Notes

This routine is often called **graf_rubberbox** in other bindings.

graf_shrinkbox – AES function (libaes)

Draw a shrinking box

#include <aesbind.h>
int graf_shrinkbox(*beginx, beginy, beginw, beginh, endx, endy, endw, endh*)
int *beginx, beginy, beginw, beginh, endx, endy, endw, endh*;

graf_shrinkbox is an AES routine that draws a shrinking box on the screen. The box drawn by **graf_shrinkbox** does not stay on the screen; this routine is designed merely to add a "star wars"-style flourish to GEM programs. The arguments *beginx*, *beginy*, *beginw*, and *beginh* define the initial dimensions of the shrinking box: respectively, they set its X coordinate, Y coordinate, width, and height. *endx*, *endy*, *endw*, and *endh* set the same dimensions for the rectangle toward which the shrinking box shrinks. The unit of measure is the number of rasters for the screen, as follows:

Resolution	*Width*	*Height*
High	640	400
Medium	640	200
Low	320	200

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

Example

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

See Also

AES, graf_growbox AES, gem, graf_growbox, TOS

graf_slidebox — AES function (libaes)

Track the slider within a box
#include <aesbind.h>
#include <obdefs.h>
int graf_slidebox(*tree, parent, slider, direction***)**
char ****tree*; int** *parent, slider, direction***;**

graf_slidebox is an AES routine that tracks the movement of the "slider". A slider is a box that the user can click to scroll through the contents of the file or directory being displayed.

This function is *not* usable for the window sliders, because the window manager is the only entity that knows where the object that defines those sliders is kept.

All that is needed to define a slider is a box with another box within it. A more complex slider can be made by making the primary boxes invisible and drawing icons within the slider and parent boxes.

tree points to the address of the object tree that contains the slider. *parent* is the index of the parent object within the object tree, and *slider* is the index of the slider object. *direction* is the direction of movement relative to the position of the parent object: zero indicates horizontal movement and one indicates vertical movement.

graf_slidebox returns the position of the center of the slider relative to the parent object. If movement is vertical, then zero indicates the topmost position and 1,000 the bottom-most; and if movement is horizontal, then zero indicates the leftmost position and 1,000 the rightmost.

Example

The following example draws a slider on the screen. By clicking it, you can move the slide bar back and forth; the program informs you of the new position of the slide bar.

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

/* The slider is simply one box inside another */
#define PARENT   0
#define SLIDER   1
OBJECT object[] = {
{ -1, 1, 1, G_BOX, NONE, NORMAL,
    ((-1L&0xFF)<<16)|(BLACK<<12)|(BLACK<<8)|(1<<4)|BLACK, 0, 0, 20, 1 },
{ 0, -1, -1, G_BOX, LASTOB, NORMAL,
    (1L<<16)|(BLACK<<12)|(BLACK<<8)|BLACK, 10, 0, 1, 1 },
};

#define NOBJECT      (sizeof object / sizeof *object)

typedef struct { int x, y, w, h; } Rectangle;
#define elements(r) r.x, r.y, r.w, r.h
#define pointers(r) &r.x, &r.y, &r.w, &r.h

typedef struct { int x, y, b, k; } House_state;
#define melements(r)m.x, m.y, m.b, m.k
#define mpointers(r)&m.x, &m.y, &m.b, &m.k

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

/* Recompute slider position using graf_slidebox return */
slid_repos(op, np, ns, d, s)
register OBJECT *op;
int np, ns, d, s;
{
        if (d == 0)
                op[ns].ob_x = ((long)(op[np].ob_width -
                        op[ns].ob_width)*s)/1000;

        else
                op[ns].ob_y = ((long)(op[np].ob_height -
                        op[ns].ob_height)*s)/1000;
}
```

```
main()
{
        int s;          /* Slide position */
        Rectangle d;  /* Desktop rectangle */
        Rectangle r;
        Mouse_state m;

        appl_init();
        for (s = 0; s < NOBJECT; s += 1)
                rsrc_obfix(object, s);

        /* Get desktop rectangle and center slider */
        wind_get(0, WF_FULLXYWH, pointers(d));
        object[PARENT].ob_x = d.x + d.w / 2 - object[PARENT].ob_width / 2;
        object[PARENT].ob_y += d.y + d.h / 4;

        /* Loop until the alert'ed user quits */
        do {
                /* Redraw the slider */
                objc_draw(object, ROOT, 8, elements(d));

                /* Find the slider rectangle */
                objc_offset(object, SLIDER, &r.x, &r.y);
                r.w = object[SLIDER].ob_width;
                r.h = object[SLIDER].ob_height;

                /* Wait for the slider to be selected */
                do
                        evnt_mouse(0, elements(r), mpointers(m));
                while ((m.b & 1) == 0);

                /* Let the AES track the slider */
                s = graf_slidebox(object, PARENT, SLIDER, 0);

                /* Compute the new slider position */
                slid_repos(object, PARENT, SLIDER, 0, s);
        } while (alertf(1, "[0][slider at %d  ][Ok|Quit]", s) == 1);

        appl_exit();
        return 0;
}
```

See Also

AES, TOS

graf_watchbox — AES function (libaes)

Draw a watched box

#include <aesbind.h>
#include <obdefs.h>
int graf_watchbox(*tree, object, insidepattern, outsidepattern***)**
OBJECT ****tree*; int** *object, insidepattern, outsidepattern*;

graf_watchbox is an AES routine that draws a "watchable box", that is, a box that the screen manager can poll to see if the mouse pointer is inside it or outside it. The user must hold down the leftmost mouse button while moving the pointer;

graf_watchbox returns the position the pointer was at when the button was released.

tree points to object tree that produces the box in question. *object* is the index of this object within the tree. *insidepattern* and *outsidepattern* indicate, respectively, the pattern used to fill the area within the box and outside the box, as follows:

1	normal
2	selected
3	crossed
4	checked
5	outlined
6	shadowed

graf_watchbox returns a value that indicates whether the mouse pointer was inside or outside the box when the button was released: zero indicates outside, and one indicates inside.

See Also

AES, TOS

H

handle — Definition

A **handle** is a generic term for a unique identifier used by TOS and GEM. Three types of handles are commonly used: file handles, workstation handles, and process handles.

A *file handle* identifies a source of bits; it can refer either to a file on disk or to a character device. File handles are returned by **fopen, fcreat**, and **fdup**, and are used by **fwrite, fread**, and **fseek**. See the entry for **FILE** for more information.

A *workstation handle* is used by the GEM VDI to identify a virtual device. It is returned by the routines **graf_handle, v_opnwk, v_opnvwk**. It is always the first argument accepted by a VDI routine.

A *process handle* identifies a process that runs under the AES. At present, these handles have only limited use because the AES currently can run only one process at a time.

A *window handle* identifies each handle as it is created, to distinguish it from all other created windows. It is returned by the routine **wind_create**.

See Also

AES, VDI, UNIX routines

header file — Overview

A *header file* is a file of C code that contains definitions, declarations, and structures commonly used in a given situation. By tradition, a header file always has the suffix ".h". Header files are invoked within a C program by the command **#include**, which is read by **cpp**, the C preprocessor; for this reason, they are also called "include files".

Header files are one of the most useful tools available to a C programmer. They allow you to put into one place all of the information that the different modules of your program share. Proper use of header files will make your programs easier to maintain and to port to other environments.

See Also

#include, portability, stdio.h

help — Command

Print concise description of command
help *command*

help prints a concise description of the options available for each specifed *command*. If the *command* is omitted, **help** prints a simple description of itself. The primary purpose of **help** is to refresh the memory of a user who has forgotten a

command option.

Information used by help is kept in the file named **helpfile**. This file must be kept in a directory that is named in the environmental variable **LIBPATH**, or **help** will not be able to find it. Information about a *command* begins with a line

```
#command
```

and ends with the next line beginning with '#'.

If you wish, you can edit this file and add new descriptions for commands that you want to run under **msh**. Be sure to use the '#', as described above. Once you have edited **helpfile**, you must rebuild its index; otherwise, **help** will no longer work. To rebuild the **helpfile**, use the following command:

```
help -R foo
```

where **foo** is the name of any entry within **helpfile**.

See Also

commands, msh

hidemouse — Command

Hide the mouse pointer
hidemouse

hidemouse is a command that uses the function **lineaa** to hide the mouse pointer. Note that if **hidemouse** is used when the mouse pointer is already hidden, the mouse pointer will need to be called twice before it reappears.

See Also

commands, Line A, mousehidden, showmouse, TOS

HOME — Environmental variable

HOME names where the micro-shell **msh** should look for a file when no other directory is specified. For example, if you type the **cd** command without an argument, **msh** will change the directory to the one you named as the **HOME** directory.

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

See Also

msh, setenv

horizontal tab — Character constant

Mark Williams C recognizes the literal character '\t' as representing the ASCII horizontal tab character HT (octal 011). This character may be used as a character constant or in a string constant.

See Also
ASCII, character constant

htom — Command

Redraw screen from high to medium resolution
htom

htom is a command that redraws the screen, moving from high to medium resolution.

See Also
commands, ltom, mtoh, mtol, TOS

hypot — Mathematics function (libm)

Compute hypotenuse of right triangle
#include <math.h>
double hypot(*x*, *y*) double *x*, *y*;

hypot computes the hypotenuse, or distance from the origin, of its arguments *x* and *y*. The result is the square root of the sum of the squares of *x* and *y*.

Example
For an example of this function, see the entry for **acos**. For an example of its use in a GEM-DOS application, see the entry for **v_circle**.

See Also
cabs, mathematics library

I

if — Command

Execute a command conditionally
if *word1 word2 [word3]*

if is a command built into the microshell **msh**. It governs the conditional execution of commands: If **word1** executes successfully, then **word2** is executed; otherwise, the **word3**, if present, is executed. Each of the words may be a list of commands that is enclosed within parentheses.

Example
The command

```
if (cc -V foo.c >&bar) (cp foo.c b:\src) (me foo.c bar)
```

compiles the program **foo.c**. If the compilation proceeded correctly, then **foo.c** is copied into the directory **src**; however, if something went wrong, then the editor would be invoked to display both **foo.c** and the file into which all error messages had been redirected. This is useful if you keep your source files on a RAM disk.

See Also
commands, equal, is_set, msh, not, while

if — C keyword

Introduce a conditional statement

if is a C keyword that introduces a conditional statement. For example,

```
if (i==10)
        dosomething();
```

will **dosomething** only if **i** equals ten.

if statements can be used with the statements **else if** and **else** to create a chain of conditional statements. Such a chain can include any number of **else if** statements, but only one **else** statement.

See Also
C keywords, C language, else
The C Programming Language, page 51

#if — Preprocessor instruction

Include code conditionally
#if (*expression*)

#if is the initiator for a conditional statement that is processed by the C preprocessor **cpp**. This command tells **cpp** that if the following condition is met, then include the following lines of code in the program until it meets the next **#elif**, **#else**, or **#endif** statement.

Ikbdws writes a string of characters to the intelligent keyboard. *number* is the number of characters to write, minus one, and *buffer* points to the buffer where these characters are kept.

The Atari ST's intelligent keyboard can accept many commands that affect the keyboard itself, the mouse, and the joystick. For more information on how the intelligent keyboard manipulates these devices, see the entry for **Kbdvbase**.

See Also

Gettime, Kbdvbase, Settime, TOS, xbios

INCDIR — Environmental variable

INCDIR names the default directory within which the C preprocessor **cpp** seeks its header files. For example, the command

```
setenv INCDIR=a:\include
```

tells **cpp** to look for header files in directory **include** on drive A. This directory is searched, as is the directory that holds the C source files and the directories named with **-I** options to the **cc** command, if any.

It is recommended that you set **INCDIR** in your **profile** to ensure that it is always set correctly.

See Also

cc, environment, environmental variable

#include — Preprocessor instruction

Copy a header file into a program
#include <*file***.h>**
#include "*file***.h"**

#include is a statement processed by the C preprocessor **cpp**. Its operation is simple: **cpp** replaces the **#include** statement with the contents of *file*.**h**.

The name of the file can be enclosed within angle brackets (**<***file***.h>**) or quotation marks (**"***file***.h"**). Angle brackets tell **cpp** to look for *file*.**h** in the directories named with the **-I** options to the **cc** command line, and then in the directory named by the environmental variable **INCDIR**. Quotation marks tell **cpp** to look for *file*.**h** in the source file's directory, then in directories named with the **-I** options, and then in the directory named by the environmental variable **INCDIR**.

Files that are called with **#include** statements are called *header files* or *include files*.

See Also

cpp, header file, msh
The C Programming Language, page 207

index — String function (libc)

Find a character in a string
char *index(*string***, c) char *string; char c;**

index scans the given *string* for the first occurrence of the character *c*. If *c* is found, **index** returns a pointer to it. If it is not found, **index** returns NULL.

Note that having **index** search for NUL will always produce a pointer to the end of a string. For example,

```
char *string;
assert(index(string, 0)==string+strlen(string));
```

will never fail.

Example

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

See Also

memchr, pnmatch, rindex, string, strchr, strpbrk
The C Programming Language, page 67

Notes

This function is identical to the function **strchr**, which is described in the ANSI standard. Mark Williams C includes **strchr** in its libraries. It is recommended that it be used instead of **index** so that programs more closely approach strict conformity with the ANSI standard.

inherit — Command

Pass variable to child shell
inherit *variable ...*

The command **inherit** allows a sub-shell to inherit a *variable* set in its parent shell. *variable* must be a variable that had been set with the **set** command.

See Also

commands, msh, set, setenv

Initmous — xbios function 0 (osbind.h)

Initialize the mouse
#include <osbind.h>
#include <xbios.h>
void Initmous(*type, parameter, vector***)**
int *type*; **char** **parameter*; **long** *vector*;

Initmous initializes the mouse, and returns nothing.

type indicates the mode into which the mouse is to be set, as follows:

0 turn mouse off

1	enable in relative mode
2	enable in absolute mode
3	unused
4	enable in keycode mode

parameter is the address of the 14-byte parameter block. Bytes 0 through 3 are used under all modes; bytes 4 through 11 are used only if the mouse is initialized into absolute mode. The parameter block's bytes indicate the following:

0	non-zero, set Y axis 0 at bottom; zero, set Y axis 0 at top
1	set the parameter for command to set mouse buttons
2	set parameter for X axis threshhold-scale-delta
3	set parameter for Y axis threshhold-scale-delta
4	most significant byte (MSB) for mouse's absolute maximum position on X axis
5	least significant byte (LSB) for mouse's absolute maximum position on X axis
6	MSB for mouse's absolute maximum position on Y axis
7	LSB for mouse's absolute maximum position on Y axis
8	MSB for mouse's initial position on X axis
9	LSB for mouse's initial position on X axis
A	MSB for mouse's initial position on Y axis
B	LSB for mouse's initial position on Y axis

Finally, *vector* gives the mouse's interrupt vector routine.

See Also

TOS, xbios

int — C keyword

Data type

An **int** is the most commonly used numeric data type, and is normally used to encode integers. On the 68000, as on most microprocessors, **sizeof int** equals 2, that is, two **chars** (15 bits plus a sign bit); therefore, an **int** can contain values from -32768 to +32767. An **int** normally is sign extended when cast to a larger data type; an **unsigned int**, however, will be zero extended.

See Also

C keywords, C language, data formats, data types, declarations, long

interrupt — Definition

An **interrupt** is an interruption of the sequential flow of a program. It can be generated by the hardware, from within the program itself, or from the operating system.

The functions **bios**, **gemdos**, and **xbios** all employ traps, a form of interrupt, to perform their respective tasks.

See Also

bios, gemdos, xbios

Iorec — xbios function 14 (osbind.h)

Set the I/O record

#include <osbind.h>
#include <xbios.h>
Iorec *Iorec(*device***) int** *device*;

Iorec returns a pointer to a serial device's input buffer record. *device* is an integer that encodes the serial device: the legal settings are 0, 1, or 2, for the RS-232 port, the keyboard, or the musical instrument device interface (MIDI) port, respectively.

As noted, **Iorec** returns a pointer to the device's input buffer record. The record is a structure that is laid out as follows:

```
struct Iorec {
        char *io_buff;                  /* Buffer */
        short io_bufsiz;                /* Buffer size in bytes */
        short io_head;                  /* Current write pointer */
        short io_tail;                  /* Current read pointer */
        short io_low;                   /* Low water mark, unstop line */
        short io_high;                  /* High water mark, stop line */
}
```

buffer points to the device's buffer. *size* is the buffer's size; *high* is its "high water mark", or where an XOFF is sent to the transmitting device; and *low* is its "low water mark", or the point where an XON is sent to the transmitting device. Finally, *head* is the head index and *tail* the tail index. Note that for the RS-232 port, the input-buffer record is followed by an output-buffer record that is structured exactly the same.

Example

This example examines all of the input devices and displays their buffers. For an example of using this function from the **\auto** directory, see the entry for **\auto**.

```
#include <osbind.h>
#include <xbios.h>

iodump(ptr)
register struct Iorec *ptr; {
        int ccount;

        if ((ccount = ptr->io_tail - ptr->io_head) < 0)
                ccount += ptr->io_bufsiz;

        printf("Buffer at %lx has %d out of %d characters in it.\n",
                ptr->io_buff, ccount, ptr->io_bufsiz);
        printf("LWM at %d characters, HWM at %d characters\n",
                ptr->io_low, ptr->io_high);
}
```

```
main() {
        struct iorec *bp;

        bp = Iorec(0);              /* get I/O buffer for serial port */
        printf("Serial port input buffer:\n");
        iodump(bp);
        printf("Serial port output buffer:\n");
        bp++;

        iodump(bp);
        bp = Iorec(1);              /* Now for the keyboard */
        printf("Keyboard input buffer:\n");
        iodump(bp);

        bp = Iorec(2);              /* MIDI input buffer */
        printf("MIDI input buffer:\n");
        iodump(bp);
}
```

See Also

TOS, xbios

is_set — Command

Check if an environmental variable is set

is_set *[in dir] name*

is_set is a test command that is built into the microshell, **msh**. It tests to see if the environmental variable *name* is set; **is_set** returns zero if *name* is set, and a value other than zero if it is not.

Example

The following command checks to see if the environmental variable **CMD** is set. If it is, the RAM-disk utility rdy is invoked in command-line mode; otherwise, rdy is invoked under the command **gem**, in graphics mode.

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

See Also

commands, equal, if, msh, not, while

isalnum — ctype macro (ctype.h)

Check if a character is a number or letter

#include <ctype.h>
int isalnum(c) int c;

isalnum tests whether the argument *c* is alphanumeric (0-9, A-Z, or a-z). It returns a number other than zero if *c* is of the desired type, and zero if it is not. **isalnum** assumes that *c* is an ASCII character or EOF.

Example

For an example of how to use this macro, see the entry for **ctype**.

See Also

ASCII, ctype

isalpha — ctype macro (ctype.h)

Check if a character is a letter

#include <ctype.h>
int isalpha(c) int c;

isalpha tests whether the argument *c* is a letter (A-Z or a-z). It returns a number other than zero if *c* is an alphabetic character, and zero if it is not. **isalpha** assumes that *c* is an ASCII character or EOF.

Example

For an example of this macro, see the entry for **ctype**.

See Also

ASCII, ctype

isascii — ctype macro (ctype.h)

Check if a character is an ASCII character

#include <ctype.h>
int isascii(c) int c;

isascii tests whether the argument *c* is an ASCII character ($0 <= c <= 0177$). It returns a number other than zero if *c* is an ASCII character, and zero if it is not. Many other **ctype** macros will fail if passed a non-ASCII value other than EOF.

Example

For an example of how to use this macro, see the entry for **ctype**. For an example of its use in a TOS application, see the entry for **Fgetdta**.

See Also

ASCII, ctype

isatty — General function (libc.a/isatty)

Check if a device is a terminal

isatty(*fd*); int *fd*;

isatty checks to see if a device is a terminal. Given the file descriptor *fd*, **isatty** returns non-zero if *fd* is attached to a terminal, and 0 if it is not.

See Also

FILE, fileno

iscntrl — ctype macro (ctype.h)

Check if a character is a control character
#include <ctype.h>
int iscntrl(c) int c;

iscntrl tests whether the argument *c* is a control character (including a newline character) or a delete character. It returns a number other than zero if *c* is a control character, and zero if it is not. **iscntrl** assumes that *c* is an ASCII character or EOF.

Example

For an example of how to use this macro, see the entry for **ctype**.

See Also

ctype

isdigit — ctype macro (ctype.h)

Check if a character is a numeral
#include <ctype.h>
int isdigit(c) int c;

isdigit tests whether the argument *c* is a numeral (0-9). It returns a number other than zero if *c* is a numeral, and zero if it is not. **isdigit** assumes that *c* is an ASCII character or EOF.

Example

For an example of how to use this macro, see the entry for **ctype**.

See Also

ASCII, ctype

isleapyear — Time function (libc)

Indicate if a year was a leap year
#include <time.h>
int isleapyear(*year*) int *year*;

isleapyear indicates whether a given year A.D. is a leap year or not. *year* is the year A.D. in which you are interested. **isleapyear** returns zero if *year* was not a leap year, and a number greater than zero if it was.

See Also

dayspermonth, time, time.h

islower — ctype macro (ctype.h)

Check if a character is a lower-case letter
#include <ctype.h>
int islower(c) int c;

islower tests whether the argument *c* is a lower-case letter (a-z). It returns a number other than zero if *c* is is a lower-case letter, and zero if it is not. **islower** assumes that *c* is an ASCII character or EOF.

Example

For an example of how to use this macro, see the entry for **ctype**.

See Also

ASCII, ctype

isprint — ctype macro (ctype.h)

Check if a character is printable
#include <ctype.h>
int isprint(c) int c;

isprint is a macro that tests if *c* is printable, i.e, if it is neither a delete nor a control character. It returns a number other than zero if *c* is a printable character, and zero if it is not. **isprint** assumes that *c* is an ASCII character or EOF.

Example

For an example of how to use this macro, see the entry for **ctype**.

See Also

ASCII, ctype

ispunct — ctype macro (ctype.h)

Check if a character is a punctuation mark
#include <ctype.h>
int ispunct(c) int c;

ispunct tests whether the argument *c* is a punctuation mark, i.e., neither an alphanumeric character nor a control character. It returns a number other than zero if the character tested is a punctuation mark, and zero if it is not. **ispunct** assumes that *c* is an ASCII character or EOF.

Example

For an example of how to use this macro, see the entry for **ctype**.

See Also

ASCII, ctype

isspace — ctype macro (ctype.h)

Check if a character prints white space
#include <ctype.h>
int isspace(c) int c;

isspace tests whether the argument *c* is a space, tab, newline, carriage return, or form-feed character. It returns a number other than zero if *c* is a white-space

character, and zero if it is not. **isspace** assumes that *c* is an ASCII character or EOF.

Example

For an example of how to use this macro, see the entry for **ctype**.

See Also

ASCII, ctype

isupper — ctype macro (ctype.h)

Check if a character is an upper-case letter

#include <ctype.h>
int isupper(c) int c;

isupper tests whether the argument *c* is an upper-case letter (A-Z). It returns a number other than zero if *c* is an upper-case letter, and zero if it is not. **isupper** assumes that *c* is an ASCII character or EOF.

Example

For an example of how to use this macro, see the entry for **ctype**. For an example of its use in a TOS application, see the entry for **Fgetdta**.

See Also

ASCII, ctype

J

j0 — Mathematics function (libm)

Compute Bessel function

#include <math.h>
double j0(z) double z;

j0 computes the Bessel function of the first kind for order 0, for its argument *z*.

Example

This example, called **bessel.c**, demonstrates the Bessel functions **j0**, **j1**, and **jn**. Compile it with the following command line

```
cc -f bessel.c -lm
```

to include floating-point functions and the mathematics library.

```
#include <math.h>
dodisplay(value, name)
double value; char *name;
{
        if (errno)
                perror(name);

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

#define display(x) dodisplay((double)(x), #x)
main() {
        extern char *gets();
        double x;
        char string[64];

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

                display(x);
                display(j0(x));
                display(j1(x));
                display(jn(0,x));

                display(jn(1,x));
                display(jn(2,x));
                display(jn(3,x));
        }
}
```

See Also

j1, jn, mathematics library

j1 — Mathematics function (libm)

Compute Bessel function

#include <math.h>
double j1(z) double z;

j1 takes the argument *z* and computes the Bessel function of the first kind for order 1.

Example

For an example of this function, see the entry for **j0**.

See Also

j0, jn, mathematics library

jday_to_time — Time function (libc)

Convert Julian date to system time

#include <time.h>
time_t jday_to_time(*time*) jday_t *time*;

jday_to_time converts Julian time to system time. *time* is the Julian time to be converted. It is of type **jday_t**, which is defined in the header file **time.h**. **jday_t** is a structure that consists of two **unsigned longs**. The first gives the number of the Julian day, which is the number of days since the beginning of the Julian calendar (January 1, 4713 B.C.). The second gives the number of seconds since midnight of the given Julian day.

jday_to_time returns the Julian time as converted to type **time_t**; this type is defined in the header file **time.h** as being equivalent to a **long**. Mark Williams C defines the current system time as being the number of seconds from January 1, 1970, 0h00m00s GMT, which is equivalent to the Julian day 2,440,587.5.

See Also

jday_to_tm, time (overview), time.h, time_to_jday, tm_to_jday

Note

This function is of use mainly to astronomers, geographers, and historians.

jday_to_tm — Time function (libc)

Convert Julian date to system calendar format

#include <time.h>
tm_t *jday_to_tm(*time*) jday_t *time*;

jday_to_tm converts Julian time to the system calendar format. *time* is the Julian time to be converted. It is of type **jday_t**, which is defined in the header file **time.h**. **jday_t** is a structure that consists of two **unsigned longs**. The first gives the number of the Julian day, which is the number of days since the beginning of the Julian calendar (January 1, 4713 B.C.). The second gives the number of seconds since midnight of the given Julian day.

jday_to_tm returns a pointer to a copy of the structure **tm_t**, which is defined in the header file **time.h**. For more information on this structure, see the Lexicon entry for **time**.

See Also

jday_to_time, time (overview), time.h, time_to_jday, tm_to_jday

Note

This function is of use mainly to astronomers, geographers, and historians.

Jdisint — xbios function 26 (osbind.h)

Disable interrupt on muli-function peripheral device

#include <osbind.h>
#include <xbios.h>
void Jdisint(*number*) int *number*;

Jdisint disables an interrupt on the multi-function peripheral device, and returns nothing. *number* is the number of the interrupt to disable. For a table of interrupt codes, see the entry for **Mfpint**.

See Also

Jenabint, Mfpint, TOS, xbios

Jenabint — xbios function 27 (osbind.h)

Enable a multi-function peripheral port interrupt

#include <osbind.h>
#include <xbios.h>
void Jenabint(*number*) int *number*;

Jenabint enables the multi-function peripheral (MFP) interrupt, and returns nothing. *number* is the number of the interrupt to disable. For a table of interrupts, see the entry for **Mfpint**.

See Also

Jdisint, Mfpint, TOS, xbios

jn — Mathematics function (libm)

Compute Bessel function

#include <math.h>
double jn(*n*, *z*) int *n*; double *z*;

jn takes an argument *z* and computes the Bessel function of the first kind for order *n*.

Example

For an example of this function, see the entry for j0.

See Also

j0, j1, mathematics library

K

Kbdvbase — xbios function 34 (osbind.h)

Return a pointer to the keyboard vectors

#include <osbind.h>
#include <xbios.h>
kbdvbase *Kbdvbase()

Kbdvbase returns a pointer to a structure that holds the following elements:

```
struct kbdvbase {
    void (*kb_midivec)();          /* MIDI input data vector */
    void (*kb_vkbderr)();          /* keyboard error vector */
    void (*kb_vmiderr)();          /* MIDI error vector */
    void (*kb_statvec)();          /* keyboard status packet */
    void (*kb_mousevec)();         /* keyboard mouse packet */
    void (*kb_clockvec)();         /* keyboard clock packet */
    void (*kb_joyvec)();           /* keyboard joystick packet */
    void (*kb_midisys)();          /* system midi vector */
    void (*kb_kbdsys)();           /* system keyboard vector */
);
```

kb_midivec points to a routine that moves data from the musical instrument digital interface (MIDI) into the MIDI buffer.

kb_vkbderr and **kb_vmiderr** point to routines that are called whenever an error condition is detected, respectively, on the intelligent keyboard or on the MIDI.

kb_statvec, **kb_mousevec**, **kb_clockvec**, and **kb_joyvec** point to routines that process data received from, respectively, the intelligent keyboard status handler, the mouse, the clock, and the joystick.

Finally, **kb_midisys** and **kb_ikbdsys** point to routines that call handlers when characters become available for, respectively, the MIDI and the intelligent keyboard.

Manipulating peripheral devices

By default, the keyboard reports each make/break contact on the joystick port, each make/break contact on the mouse buttons, and each movement of the mouse that exceeds a preset threshold. Each report consists of a "packet" of three bytes that indicate which device is changing and what change took place. Note that the packet for the joystick has been documented elsewhere as consisting of two bytes; this is incorrect.

The joystick packets consist of three bytes: The first is always 0xFF, which indicates joystick event on port 1; the second is filler, and is always 0x00; and the third records the closed switches on the joystick as set bits in the low nybble. Technically, the high bit of the third byte should encode the state of the joystick fire button. In the default set-up, the fire button is set to the left mouse button. This will change if you instruct the keyboard to adopt some other reporting mode.

The mouse packets consist of three bytes: The first is 0xF8, which indicates relative mouse event and encodes the state of the mouse buttons and joystick fire button in the low bits of the low nybble. The second and third encode, respectively, the relative X- and Y-axis motion as signed characters.

If you do not have a joystick, you can simulate one by plugging your mouse into the joystick port. The mouse quadrature signals show up as the *north south east west* switch closure bits in the joystick packet. In addition, the left mouse button still shows up as a mouse event, but the right button is inoperative.

Example

The following example monitors the keyboard's mouse and joystick vectors.

```
#include <osbind.h>
#include <bios.h>
#include <xbios.h>

union {
        char k_c[4];                    /* translate four-character packet ... */
        long k_s;                       /* ... into a long */
} kst;                                  /* one for joystick and mouse */
long ktm;                               /* packet time stamp */

kbdvec(p) char *p;
{
        kst.k_c[0] = *p++;              /* store four byte packet */
        kst.k_c[1] = *p++;              /* NB: 'p' could be an odd address */
        kst.k_c[2] = *p++;
        kst.k_c[3] = *p++;
        ktm = *((long *)0x4BA);         /* system 200hz clock tick */
}

main()
{
        register struct kbdvbase *kbp;
        register void (*xx_joyvec)(), (*xx_mousevec)();
        register long ks, kt;

        kbp = Kbdvbase();               /* keyboard vector table */
        xx_joyvec = kbp->kb_joyvec;     /* save old joystick vector */
        kbp->kb_joyvec = kbdvec;        /* install new joystick vector */
        xx_mousevec = kbp->kb_mousevec; /* ditto for mouse */
        kbp->kb_mousevec = kbdvec;
        ks = kst.k_s;                   /* initialize state record */

        while (Bconstat(BC_CON) == 0) { /* i.e., until a key is struck */
                if (ks != kst.k_s) {    /* new event? */
                        ks = kst.k_s;   /* then report new state ... */
                        kt = ktm;       /* ... and timestamp */
                        printf("%08lx %lu\n", ks, kt);
                }
        }
```



```
        Bconin(BC_CON);                 /* clear keystroke */
        kbp->kb_joyvec = xx_joyvec;     /* [RESTORE VECTORS] */
        kbp->kb_mousevec = xx_mousevec; /* [OR YOU BOMB ON THE NEXT EVENT] */
        return 0;
}
```

See Also

TOS, xbios

kbrate — Command

Reset the keyboard's repeat rate
kbrate *start, delay*

kbrate uses the **xbios** function **Kbrate** to reset the keyboard's repeat rate. *start* is the amount of time to pass before repeating begins, and *delay* is the time interval between repeats. Both are measured in "system ticks", each tick being 20 milliseconds long. For example, the command

```
kbrate 50 5
```

tells the system that a key must be held down half a second before repeating begins, and then repeating will occur ten times a second thereafter.

See Also

commands, TOS

Kbrate — xbios function 35 (osbind.h)

Get or set the keyboard's repeat rate
#include <osbind.h>
#include <xbios.h>
int Kbrate(*start, delay***) int** *start, delay*;

Kbrate gets or sets the keyboard's repeat rate. Rates are set as multiples of "system ticks"; each tick is 20 milliseconds long. *first* sets the number of ticks to wait before a key begins to repeat; *delay* sets the number of ticks to wait between repeats. If either variable is set to 0xFFFF (-1), that value is not changed. **Kbrate** returns an **int** that holds the previous setting of the keyboard rate: the value of *first* is written as the high byte, and the value of *delay* as the low byte.

Example

This example displays the keyboard repeat rate and delay period; it then sets them to unreasonable values, lets the user try them out, and finally resets the previous values. For an example of using this function from the **\auto** directory, see the entry for **\auto**.

```
#include <osbind.h>
#define DEL 10
#define RT 1

main() {
        int old_rate;
        int old_delay;
        char c;

        old_rate = Kbrate(DEL,RT); /* Set the new rate. */
        old_delay = (old_rate>>8)&0xFF;
        old_rate &= 0xFF;
        printf("The repeat delay is %d/50 seconds\n", old_delay);
        printf("and repeat rate is once every %d/50 seconds\n",
                old_rate);
        printf("Rates are changed to delay=%d, rate=%d\n", DEL, RT);
        printf("Try typing something--end with ^C.\n\n");
        while((c = Crawcin()) != '\03') {
                Crawio(c);
        }
        Kbrate(old_delay,old_rate);
        printf( "\nRates restored.\n" );
}
```

See Also

TOS, xbios

keyboard — Technical information

The Atari keyboard is table-driven. The keyboard tables are vectors of byte values that are indexed by the scan code passed from the intelligent keyboard (IKBD). The table is zero-based, so the first entry is always NULL. The following display shows the layout of the keyboard, with the scan code each key generates being given in hexadecimal: Note that some keys produce different scan codes when used with the **<shift>** or **<alt>** key.

function keys

3B 54	3C 55	3D 56	3E 57	3F 58	40 59	41 5A	42 5B	43 5C	44 5D

<shifted>

keyboard

01	78 02	79 03	7A 04	7B 05	7C 06	7D 07	7E 08	7F 09	80 0A	81 0B	82 0C	83 0D	<alt> 29	0E
0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	53
1D	1E	1F	20	21	22	23	24	25	26	27	28			2B
2A	60	2C	2D	2E	2F	30	31	32	33	34	35	36		
38	39										3A			

keypad

62	61		
52	48	47	
4B	50	4D	

63	64	65	66
67	68	69	4A
6A	6B	6C	4E
6D	6E	6F	72
70		71	

The keyboard sold in North America does not have the key with scan code 60. This key is sometimes called the "ISO Key", and is only on European models.

See Also

ASCII, evnt_keyboard, Keytbl, TOS

Keytbl — xbios function 16 (osbind.h)

Set the keyboard's translation table

#include <osbind.h>
#include <xbios.h>
keytbl *Keytbl(*unshifted, shifted, caplock***) char ****unshifted, *shifted, *caplock***;**

Keytbl sets the keyboard's translation tables.

On the Atari ST, each key generates a unique *scan code*. (See the entry for **keyboard** to find the code for each key.) The scan code is looked up in one of three translation tables. The table used depends upon the states of the shift keys: one table is used when no shift key is pressed, a second is used when the shift key is depressed, and a third is used when the caps-lock key is depressed. The scan code is then translated into the character found in the appropriate table.

The variables *shifted*, *unshifted*, and *capslock* each point to a translation table that you wish to load in place of a default table. Each table must be 128 bytes long. Setting one or more of these arguments to -1L tells **Keytbl** not to load a new key table.

Keytbl returns a pointer to the following structure:

```
struct keytbl {
        char *unshifted;
        char *shifted;
        char *capslock;
}
```

These point to the areas where **Keytbl** has written the current key tables.

Example

This example prints out the default keyboard map in the form of a C source file. This example also demonstrates a good method of obtaining data from the Atari's memory.

```
#include <osbind.h>
#include <xbios.h>

showmap(map, p)
register char *map, *p;
{
        register int i, j;
        printf("char %s[128] = {\06lx\n", map, p);

        for (i = 0; i < 8; i += 1) {
        putchar('\t');

        for (j = 0; j < 16; j += 1)
                if (*p < ' ' || *p >= 0177 || *p == '\'' || *p == '\\')
                        printf("%3d,", *p++ & 0xFF);
                else
                        printf("'%c',", *p++ & 0xFF);

        putchar('\n');
        }
        printf("};\n");
}

main() {
        struct keytbl *kp;
        kp = Keytbl(-1L, -1L, -1L);
        showmap("normal", kp->kt_normal);
        showmap("shifted", kp->kt_shifted);
        showmap("capslock", kp->kt_capslock);
        return 0;
}
```

See Also

Bioskeys, TOS, xbios

Kgettime — Time function (libc)

Read time from intelligent keyboard's clock
#include <time.h>
tm *Kgettime();

Kgettime 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. **Kgettime** returns a pointer to the structure **tm**, which it initializes. **tm** is defined in the header file **time.h**. For more information about it, see the entry for **time**.

See Also

Ksettime, Sgettime, time (overview), time.h

Notes

Unlike the function **Gettime**, which deals in two-second increments, **Kgettime** allows the programmer to work with clock ticks.

This function does not work properly on the Mega ST. To read the clock on the Mega ST, use the function **Sgettime**.

kick — Command

Force TOS to reread the disk cache
kick *drive*

kick forces TOS to read a disk cache. *drive* is the name of the disk drive whose cache is to be read. **kick** should be used when disks are switched in a drive, to ensure that TOS has the correct form of the disk's root directory in memory.

See Also

commands, TOS

Ksettime — Time function (libc)

Set time in intelligent keyboard's clock
#include <time.h>
int Ksettime(*time***) tm *****time**;

Ksettime is a function that sets the time on the intelligent keyboard's clock. This clock is maintained apart from the other clocks on the Atari ST. *time* points to a copy of the structure **tm**, which is filled by the functions **gmtime** or **localtime**. This structure is defined in the header file **time.h**. For more information about it, see the entry for **time**.

See Also

Kgettime, time (overview), time.h

Notes

Unlike the function **Settime**, which deals with two-second increments, **Ksettime** works directly with seconds.