T

tail — Command

Print the end of a file
tail [+*n*[bcl]] [*file*]
tail [-*n*[bcl]] [*file*]

tail copies the last part of *file*, or of the standard input if none is named, to the standard output.

The given *number* tells **tail** where to begin to copy the data. Numbers of the form *+number* measure the starting point from the beginning of the file; those of the form *-number* measure from the end of the file.

A specifier of blocks, characters, or lines (**b**, **c**, or **l**, respectively) may follow the number; the default is lines. If no *number* is specified, a default of -10 is assumed.

See Also

commands, egrep

Notes

Because **tail** buffers data measured from the end of the file, large counts may not work.

tan — Mathematics function (libm)

Calculate tangent
#include <math.h>
double tan(*radian*) **double** *radian*;

tan calculates the tangent of its argument *radian*, which must be in radian measure.

Example

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

See Also

mathematics library

Diagnostics

tan returns a very large number where it is singular, and sets **errno** to **ERANGE**.

tanh — Mathematics function (libm)

Calculate hyperbolic cosine
#include <math.h>
double tanh(*radian*) **double** *radian*;

tanh calculates the hyperbolic tangent of *radian*, which is in radian measure.

Example

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

See Also

mathematics library

Diagnostics

tanh sets **errno** to **ERANGE** when an overflow occurs.

tempnam — General function (libc)

Generate a unique name for a temporary file
char *tempnam(*directory, name*)
char **directory*, **name*;

tempnam constructs a unique temporary name that can be used with your program.

directory points to the name of the directory in which you want the temporary file written. If this variable is **NULL**, **tempnam** reads the environmental variable **TMPDIR** and uses it for *directory*. If neither *directory* nor **TMPDIR** is given, **tempnam** uses **\tmp**.

name points to the string of letters that will prefix the temporary name; this string should not be more than three or four characters, to prevent truncation or duplication of temporary file names. If *name* is **NULL**, **tempnam** will set it to **t**.

tempnam uses **malloc** to allocate a buffer for the temporary file name it returns. If all goes well, it returns a pointer to the temporary name it has written; otherwise, it returns **NULL** if the allocation fails or if it cannot build a temporary file name successfully.

See Also

environment, mktemp, tmpnam

tetd_to_tm — Time function (libc)

Convert IKBD time to system calendar format
#include <time.h>
tm_t *tetd_to_tm(*time*) **tetd_t** *time*;

tetd_to_tm converts the time setting for the intelligent keyboard, as returned by the function **Gettime**, into the system's calendar format.

time is of type **tetd_t**, which is defined in the header file **time.h** as being equivalent to an **unsigned long**. It holds the 32-bit map returned by **Gettime**. For information on what the bits of this map signify, see the entry for **Gettime**.

tetd_to_tm returns a pointer to the structure **tm_t**, which is defined in the header file **time.h**. For more information on this structure, see the entry for **time**.

See Also

time (overview), time.h, tm_to_tetd

Tgetdate — gemdos function 42 (osbind.h)

Get the current date

#include <osbind.h>

int Tgetdate()

Tgetdate gets the current date from TOS. It returns an integer whose bits indicate the following:

0-4	day (1-31)
5-8	month (1-12)
9-15	year (0-119, 0=1980)

Example

This examples demonstrates both **Tgetdate** and **Tgettime**. Note that the time returned by this example will be one hour earlier than the time returned by **msh** if the latter is adjusting for daylight savings time.

```
#include <osbind.h>

main() {
        unsigned int date;
        unsigned int time;

        date = Tgetdate();              /* Get system date */
        time = Tgettime();              /* Get system time */
        timeprint("The TOS time is", time);
        dateprint("The TOS date is", date);
}

void fixdig(buf, onumber, size)
char *buf;
int onumber;
int size;
{
        register long limit;
        register long number;
        int o;

        number = onumber;

        limit = 10;
        for (o = 1; o < size ; o++)
                limit *= 10;

        if ((number >= limit)||(number <0)) {
                for (o = 0; o < size; o++)
                        *buf++ = '*';
                *buf = 0;
                return;
        }
        for (o = 0; o < size; o++) {
                limit /= 10;
                *buf++ = '0'+number/limit;
                number = number%limit;
        }
        *buf = '\0';
}

timeprint(string, time)
char *string;
register unsigned int time;
{
        int seconds;
        int minutes;
        int hours;
        char mins[3];
        char secs[3];

        seconds = (time & 0x001F) << 1;         /* Bits  0:4 */
        minutes = (time >> 5)  & 0x3F;          /* Bits  5:10 */
        hours = (time >> 11) & 0x1F;            /* Bits 11:15 */

        fixdig(mins, minutes, 2);
        fixdig(secs, seconds, 2);
        printf("%s %d:%s:%s\n", string, hours, mins, secs);
}

dateprint(string, date)
char *string;
unsigned int date;
{
        int year;
        int month;
        int day;

        day = date & 0x1f;
        month = (date>>5) & 0x0F;
        year = ((date>>9) & 0x7F) + 1980;
        printf("%s %d/%d/%d\n", string, month, day, year);
}
```

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

See Also

gemdos, time, Tsetdate, TOS

Tgettime — gemdos function 44 (osbind.h)

Get the current time

#include <osbind.h>

int Tgettime()

Tgettime obtains the current time from the operating system. It returns the time encoded in the form of an integer whose bits mean the following:

0-4	number of two-second increments (0-29)
5-10	number of minutes (0-59)
11-15	number of hours (0-23)

Example

For example of how to use this function, see the entries for **Tgetdate** and **time**.

See Also

gemdos, time, Settime, TOS

Tickcal — bios function 6 (osbind.h)

Return system timer's calibration.
#include <osbind.h>
#include <bios.h>
long Tickcal()

Tickcal returns the system timer's calibration, rounded to the nearest millisecond.

Example

This example demonstrates **Tickcal**. Also see the example in the entry for **time**.

```
#include <osbind.h>

main()
{
        printf("System clock ticks once every %ld msec.\n", Tickcal());
}
```

See Also

bios, time, TOS

time — Command

Time the execution of a command
time *"command arguments"*

time lets you time the execution of a command under the microshell, **msh**. *command* is the command to execute, and *arguments* its arguments. For example, typing

```
time "cc hello.c"
```

times how long it takes the compiler to compile **hello.c**. Note that *command* and its arguments must be enclosed within quotation marks, for **msh** to parse the command line correctly.

When *command* is finished, **time** prints on the standard output device the amount of time needed to execute the command, in hours, minutes, seconds, and hundreths of a second.

See Also

commands, msh

time — Time function (libc)

Get current time
#include <time.h>
time_t time(*tp***) time_t *tp;**

time reads the current system time. *tp* points to a data element of the type **time_t**, which is defined in the header file **time.h** as being equivalent to a **long**. Note that Mark Williams C defines the current system time as the number of seconds since January 1, 1970, 0h00m00s GMT.

Example

For an example of this function, see the entries for **asctime** and **time**.

See Also

time (overview)

time — Overview

Mark Williams C includes a number of routines that allow the user to set and manipulate time, as recorded on the system's clock, into a variety of formats. These routines should be adequate for nearly any task that involves temporal calculations or the maintenance of data gathered over a long period of time.

All functions, global variables, and manifest constants used in connection with time are defined and described in the header file **time.h**.

The ANSI Draft Time Standard

The draft ANSI standard for the C language describes functions designed to be used with calendar time (i.e., the Gregorian calendar), local time, and daylight savings time.

The basic unit of time is defined as the **CLK_TCK**, which is defined as one tick of the system clock. On the Atari ST, the **CLK_TCK** is equivalent to five milliseconds. Mark Williams C uses three variables to describe time:

clock_t
: This is an implementation-specific type that is capable of encoding clock time. On the Atari ST, this is set to an **unsigned long**.

time_t
: This is an implementation-specific type that can represent time. Mark Williams C defines **time_t** as a 32-bit number that holds the number of seconds since January 1, 1970, 0h00m00s GMT.

struct tm

This structure encodes the elements of calendar time. It is defined as follows:

```
typedef struct tm {
        int tm_sec;   /* second [0-59] */
        int tm_min;   /* minute [0-59] */
        int tm_hour;  /* hour [0-23]: 0 = midnight */
        int tm_mday;  /* day of the month [1-31] */
        int tm_mon;   /* month [0-11]: 0=January */
        int tm_year;  /* year since 1900 A.D. */
        int tm_wday;  /* day of week [0-6]: 0=Sunday */
        int tm_yday;  /* day of the year [0-366] */
        int tm_isdst; /* daylight savings time flag */
} tm_t;
```

The ANSI standard also describes a number of time functions, as follows:

asctime	convert **tm** to ASCII string
clock	return time since system was turned on
ctime	return an ASCII string that gives local time
difftime	compute difference between calendar times
gmtime	return **tm** for Greenwich Mean Time
localtime	return **tm** for local time
stime	set system time (UNIX/COHERENT-compatible)
time	return **time_t**

To print out the local time, a program must perform the following tasks: First, read the system time with **time**. Then, it must pass **time**'s output to **localtime**, which breaks it down into the **tm** structure. Next, it must pass **localtime**'s output to **asctime**, which transforms the **tm** structure into an ASCII string. Finally, it must pass the output of **asctime** to **printf**, which displays it on the standard output device. See the entry for **asctime** for an example of such a program.

Extensions to the ANSI Standard

Mark Williams C includes a number of extensions to the ANSI standard. These are designed to increase the scope and accuracy of the standard, and to ease calculation of some time elements.

To begin, Mark Williams C includes three variables that are used by the function **localtime**; it parses the environmental variable **TIMEZONE** into the following:

timezone	seconds from GMT to give local time
dstadjust	seconds to local standard, if any
tzname	array with names of standard and daylight times

The following functions return information about the calendar:

isleapyear	is this year AD a leap year?
dayspermonth	how many days in this historical month?

Time on Mark Williams C is modelled after time on the COHERENT operating system. As noted above, the variable **time_t** is defined as the number of seconds since January 1, 1970, 0h00m00s GMT; this moment, in turn, is rendered as day 2,440,587.5 on the Julian calendar. This allows accurate calculation of time as far back as January 1, 4713 B.C.

Conversion to the Gregorian calendar is set to October 1582, when it was first adopted in Rome. The issue of when a nation changed from the Julian to the Gregorian calendar is moot in the United States, Canada (except Quebec), Asia, Africa, Australia, and the Middle East; however, users in Quebec, Latin America, Europe, the Soviet Union, and European-influenced areas of Asia (e.g., India) may wish to to write their own functions to convert historical data properly from the Julian to the Gregorian calendar.

The following functions assist in conversion from Julian to Gregorian time:

time_to_jday	convert **time_t** to the Julian date
jday_to_time	convert Julian date to **time_t**
tm_to_jday	convert **tm** structure to Julian date
jday_to_tm	convert Julian date to **tm** structure

Atari ST Time Functions

The Atari ST's ROM BIOS contains a number of functions that manipulate system time. This task is complicated by the fact that the ST has several clocks, which do not reference each other; each can be set independently, and each is used under different circumstances.

The following functions convert between standard time and TOS time:

tm_to_tetd	convert **tm** to TOS time
tetd_to_tm	convert TOS time to **tm**

The intelligent keyboard (IKBD) keeps time to the second, but it not supported by either the **xbios** or the **gemdos** functions. The following two functions convert between time as encoded in **tm** and the IKBD clock:

Kgettime	turn IKBD time to **tm**
Ksettime	turn **tm** to IKBD time

Finally, the Atari **gemdos** and **xbios** routines include a number of functions that directly manipulate system time, as follows:

Fdatime	get/set a file's time and date stamp
Gettime	get the system time (**xbios**)
Settime	set the system time (**xbios**)
Tgettime	get the system time (**gemdos**)
Tgetdate	get the system date (**gemdos**)
Tsettime	get the system time (**gemdos**)
Tsetdate	set the system date (**gemdos**)

Example

For an example of time functions, see the entry for **asctime**. The following example demonstrates the header file **time.h**, and the functions **Gettime**, **Kgettime**, **Ksettime**, **Settime**, **stime**, **tetd_to_tm**, **Tgetdate**, **Tgettime**, **time**, **tm_to_tetd**, **Tsetdate**, and **Tsettime**.

```
#include <time.h>
#include <osbind.h>
tm getdate(p)
char *p;
{
        static tm t;

        sscanf(p,"%4d%2d%2d%2d%2d.%2d", &t.tm_year, &t.tm_mon, &t.tm_mday,
                &t.tm_hour, &t.tm_min, &t.tm_sec);
        t.tm_year -= 1900;
        t.tm_mday -= 1;
        return &t;
}

dodisplay(tp, name)
tm *tp char *name;
{
        printf("%4d%02d%02d%02d%02d.%02d %s\n",
                tp->tm_year+1900, tp->tm_mon+1, tp->tm_mday,
                tp->tm_hour, tp->tm_min, tp->tm_sec, name);
}

#define display(x) dodisplay((tm *)(x), "x");

main(argc, argv)
int argc; char *argv[];
{
        tm *tp;
        tetd_t td;
        time_t t;
        time_t temp;

        if (argc > 1) {
                tp = getdate(argv[1]);
                td = tm_to_tetd(tp);
                stime(&t);
```



```
                Ksettime(tp);
                Settime(td);
                Tsetdate(td.g_date);
                Tsettime(td.g_date);
        }
        temp = time(0L);
        display(localtime(&temp));
        display(gmtime(&temp));
        display(Kgettime());
        display(tetd_to_tm(Gettime()));
        display(tetd_to_tm(((long)Tgetdate()<<16)|(unsigned)Tgettime()));
}
```

See Also

Lexicon

time — Command

Print current time/time execution of a command

time
time *command*
time " *command arguments* "

The command **time** performs two different tasks, depending upon whether it is used with or without arguments.

When **time** is typed without any arguments, it prints the date and time. The date and time are presented in a string of the form:

```
Thu Apr  7 10:35:53 1988 CDT
```

The extension "CDT" stands for "Central Daylight Time". Daylight savings time will be returned only if the macro **TIMEZONE** is set properly in your **profile**. See **TIMEZONE** for more information.

If **time** is used with one or more arguments, it times the execution of a command. For example, typing **time ls** prints the contents of the current directory, then prints a string of the form:

```
00:00:02.340
```

which states how long the command took to execute.

If you wish to time a command that takes arguments, you must enclose the command and its arguments within quotation marks. For example, to time how long it takes to compile the program **window.c** with the **-VGEM** option to the compiler, use the command:

```
time "cc -VGEM window.c"
```

See Also
commands, date, msh, time (overview)

time.h — Header file

Give time-description structure
#include <time.h>

time.h is a header file that contains descriptions and declarations for elements used to manipulate system time under TOS.

See Also
time

time_to_jday — Time function (libc)

Convert system time to Julian date
#include <time.h>
jday_t time_to_jday(*time***) time_t** *time*;

time_to_jday converts system time to Julian days. *time* is the current system time. It is declared to be of type **time_t**, which is defined in the header file **time.h** as being equivalent to a **long**. Mark Williams C defines the current system time as being the number of seconds from January 1, 1970, 0h00m00s GMT. The function **time** returns the current system time in this format.

time_to_jday returns the structure **jday_t**, which is defined in the header file **time.h**. **jday_t** consists of two **unsigned longs**. The first gives the number of the Julian day, which is the number of days since the beginning of the Julian calendar (January 1, 4713 B.C.). The second gives the number of seconds since midnight of the given Julian day.

See Also
jday_to_time, jday_to_tm, time (overview), time.h, tm_to_jday

TIMEZONE — Environmental variable

Time zone information
TIMEZONE=*standard:offset*[*:daylight:date:date:hour:minutes*]

TIMEZONE is an environmental parameter that holds information about the user's time zone. This information is used by Mark Williams C's time routines to construct their description of the current time and day.

To set the **TIMEZONE** parameter, use the **set** command, as follows:

```
set TIMEZONE=[description]
setenv TIMEZONE=[description]
```

where `[description]` is the string that describes your time zone. What this string consists of will be described below. Most users write this command into the file **profile**, so that **TIMEZONE** is set automatically whenever they invoke **msh**.

The description string

A **TIMEZONE** description string consists of seven fields that are separated by colons. Fields 1 and 2 must be filled; fields 3 through 7 are optional.

Field 1 gives the name of your standard time zone. Field 2 gives the time zone's offset from Greenwich Mean Time in minutes. Offsets are positive for time zones west of Greenwich and negative for time zones east of Greenwich. For example, users in Chicago set these fields as follows:

```
TIMEZONE=CST:360
```

CST is an abbreviation for Central Standard Time, that area's time zone; and 360 refers to the fact that Chicago's time zone is 360 minutes (six hours) behind that of Greenwich.

Field 3 gives the name of the local daylight saving time zone. In Chicago, for example, this field would be set as follows:

```
TIMEZONE=CST:360:CDT
```

CDT is an abbreviation for Central Daylight Time. The absence of this field indicates that your area does not use daylight saving time.

Fields 4 and 5 specify the dates on which daylight saving time begins and ends. If field 3 is set but fields 4 and 5 are not, changes between standard time and daylight saving time will be assumed to occur at the times legislated in the United States in 1986: at 2 A.M. standard time on the first Sunday in April, and at 2 A.M. daylight saving time on the last Sunday in October.

Fields 4 and 5 each consist of three numbers separated by periods. The first number specifies which occurrence of the day in the month marks the change, counting positive occurrences from the beginning of the month and negative occurrences from the the end of the month. The second number specifies a day of the week, numbering Sunday as one. The third number specifies a month of the year, numbering January as one. For example, in Chicago fields 4 and 5 are set to the following:

```
TIMEZONE=CST:360:CDT:1.1.4:-1.1.10
```

If the first number in either field is set to zero, then the last two numbers are assumed to indicate an absolute date. This is done because some countries switch to daylight saving time on the same day each year, instead of a given day of the week.

Finally, fields 6 and 7 specify the hour of the day at which daylight saving time begins and ends, and the number of minutes of adjustment. In Chicago, these are set as follows:

```
TIMEZONE=CST:360:CDT:1.1.4:-1.1.10:2:60
```

The '2' of field 6 indicates that the switch to daylight savings time occurs at 2 A.M. The "60" of field 7 indicates that daylight savings time changes the local time by 60 minutes. Although 60 minutes is the standard change, some regions of the world

shift by 30, 45, 90, or 120 minutes; the last shift is also called "double daylight saving time".

Under the microshell **msh**, it usually is not necessary to set the offset field, unless you wish to keep your system set to Greenwich Mean Time.

For an example of this variable's use in a program, see the entry for **asctime**.

See Also

environment, setenv, time

Notes

This environmental variable should be set only if you have set your computer system's time to conform with Greenwich Mean Time. Otherwise, it will cause such functions as **localtime** to incorrectly offset the time they return.

The time zone **time** returns depends on how the time zone was originally set. If **TIMEZONE** has the correct offset from Greenwich, then the system time is GMT; however, if the time was set on the GEM desktop, or if **TIMEZONE** has set the offset from Greenwich incorrectly, then the system time is not GMT.

The default **profile** included with your copy of Mark Williams C has a **TIMEZONE** setting for Central Standard Time (CST/CDT). If you live outside that time zone, you may wish to edit **TIMEZONE** to reflect your local time zone.

For those requiring more information on this subject, much research has been performed by astrologers. See *Time Changes in the World*, compiled by Doris Chase Doane (three volumes, Hollywood, CA, Professional Astrologers, Inc., 1970).

tm_to_jday — Time function (libc)

Convert calendar format to Julian time

#include <time.h>

jday_t tm_to_jday(*time*) tm_t *time;

tm_to_jday converts the system time, as described in the system calendar format, to Julian time. *time* points to a copy of the structure **tm_t**, which is defined in the header file **time.h**. The functions **gmtime** and **localtime** return the current time in this format. For more information on **tm_t**, see the entry for **time**.

tm_to_jday returns the structure **jday_t**, which is defined in the header file **time.h**. jday_t to consist of two **unsigned longs**. The first gives the number of the Julian day, which is the number of days since the beginning of the Julian calendar (January 1, 4713 B.C.). The second gives the number of seconds since midnight of the given Julian day.

See Also

jday_to_time, jday_to_tm, time (overview), time.h, time_to_jday

tm_to_tetd — Time function (libc)

Convert system calendar format to IKBD time

#include <time.h>

tetd_t tm_to_tetd(*time*) tm_t *time;

tm_to_tetd converts the system calendar structure, as returned by the functions **gmtime** and **localtime**, into a form that can be used by the Atari function **Settime** to set the intelligent keyboard's clock.

time points to a copy of the structure **tm_t**, which is defined in the header file **time.h**. For more information on this structure, see the entry for **time**.

tm_to_tetd returns a data element of the type **tetd_t**, which is defined in the header file **time.h** as being equivalent to an **unsigned long**. It holds the 32-bit map used by **Settime** to set the intelligent keyboard's clock. For information on what the bits of this map signify, see the entry for **Settime**.

See Also

tetd_to_tm, time (overview), time.h

TMPDIR — Environmental variable

TMPDIR names the directory into which Mark Williams C writes its temporary files. If this variable is not set, the default is the directory in which the source files are kept. Note that this variable need be set only if space is a problem on the storage device that holds your current directory. For example, the command

```
set TMPDIR=a:\tmp
```

typed at the system prompt tells **cc** to write temporary files in the directory **tmp** on drive A:.

It is a good idea to set **TMPDIR** so that your temporary files are written onto a RAM disk. This will speed compilation noticeably.

See Also

cc, environment, environmental variable

tmpnam — General function (libc)

Generate a unique name for a temporary file

#include <stdio.h>

char *tmpnam(*name*) char *name;

tmpnam constructs a unique temporary name that can be used with your program. *name* is the name of a buffer into which **tmpnam** writes the temporary name. If *name* is **NULL**, **tmpnam** writes the name into an internal buffer that is overwritten each time it is called.

Unlike its cousin **tempnam**, **tmpnam** assumes that the temporary file will be

written into directory \tmp and builds the name accordingly. It returns the address of the internal buffer.

See Also

mktemp, tempnam

toascii – ctype macro (ctype.h)

Convert characters to ASCII

#include <ctype.h>
int toascii(c) int c;

toascii takes any integer value *c*, keeps the low seven bits unchanged, and changes the others to zero; this, in effect, transforms the integer value to an ASCII character. **toascii** then returns the transformed integer. If *c* is already a valid ASCII character, it is returned unchanged.

Example

This example prompts for a file name. It then opens the file and prints its contents, while converting all non-alphanumeric characters to alphanumeric.

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

main()
{
        FILE *fp;
        int ch;
        int filename[20];

        printf("Enter file name: ");
        fflush(stdout);
        gets(filename);
        if ((fp = fopen(filename, "r")) != NULL) {
                while ((ch = fgetc(fp)) != EOF)
                        putchar(isascii(ch) ? ch : toascii(ch));
        }
        else printf("Cannot open %s\n", filename);
}
```

See Also

ctype

tolower – General function (libc)

Convert characters to lower case

int tolower(c) int c;

tolower converts the letter *c* to lower case. **tolower** returns **c** converted to lower case. If *c* is not a letter or is already lower case, then **tolower** returns it unchanged.

Example

The following example demonstrates **tolower** and **toupper**. It reverses the case of every character in a text file.

For an example of its use in a TOS application, see the entry for **Fgetdta**.

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

main()
{
        FILE *fp;
        int ch;
        int filename[20];

        printf("Enter name of file to use: ");
        fflush(stdout);
        gets(filename);

        if ((fp = fopen(filename,"r")) != NULL)
        {
                while ((ch = fgetc(fp)) != EOF)
                        putchar(isupper(ch) ? tolower(ch) : toupper(ch));

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

See Also

ctype, _tolower, toupper

_tolower – ctype macro (ctype.h)

Convert letter to lower case

#include <ctype.h>
int _tolower(c) int c;

_tolower is a macro that converts *c* to lower case and returns it. If *c* is not a letter, the result is undefined.

_tolower differs from its cousin **tolower** in that **_tolower** is a macro that does not check whether its argument is in fact an alphanumeric character, whereas **tolower** is a function that does check its argument.

Example

This example opens a file of text and reverses the cases of all characters. It demonstrates **_tolower** and **_toupper**.

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

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

        if (--argc != 1)
                fatal("Usage: example filename");

        if ((fp = fopen(argv[1], "r")) == NULL)
                fatal("Cannot open file for reading");

        while ((ch = fgetc(fp)) != EOF)
        {
                if ((isascii(ch) != 0) && ch != '\r')
                        fatal("Not a text file");

                if (isalpha(ch) != 0)
                        fputc((isupper(ch) ? _tolower(ch) : _toupper(ch)),
                                stdout);
                else
                        fputc(ch, stdout);
        }
}

fatal(message)
char *message;
{
        fprintf(stderr, "%s\n", message);
        exit(1);
}
```

See Also

ctype, tolower, _toupper

tos – Command

Execute GEM-DOS program

tos *program options*

tos allows you to run under **msh** a program that uses unredirected GEM-DOS file handles. It resets file handle 2 to the **aux:** device; unlike its cousin, the **gem** command, **tos** does not enable the mouse cursor. *program* is the name of the program you wish to execute; note that you should give the full path name of the program and its full name, including suffix. *options* are a list of options that are passed directly to the program to be executed.

See Also

commands, gem

TOS – Overview

TOS is the operating system for the ATARI ST. It includes a number of components, including Digital Research's Graphics Environment Manager (GEM) and the GEM-DOS disk operating system.

The following entries in the Lexicon describe features of TOS:

AES This describes the GEM Application Environment System (AES), which allows the programmer to use predefined windows, icons, pull-down menus, and other GEM elements. It also lists and briefly describes all of the AES routines; each AES routine has its own entry within the Lexicon.

bios This entry describes the TOS function **bios**, and introduces the functions that use it to manipulate the Atari ST's BIOS.

desk accessory
This entry describes how to compile a GEM desk accessory.

error codes
This lists and defines the error codes that can be returned by TOS.

gemdos This entry describes the TOS function **gemdos**, and introduces the functions that use it to manipulate GEM-DOS.

keyboard
This describes the layout of the Atari ST keyboard, with the codes generates by each key.

Line A This describes briefly the Atari "Line A" interface routines, which allow the creation and manipulation of graphics displays.

screen control
This entry lists the escape sequences used to control text on the Atari ST's screen.

system variables
This entry lists all of the "magic locations" within memory where TOS stores its key elements.

VDI This describes the GEM Virtual Device Interface (VDI), which gives the user access to basic graphics routines. It also lists and describes briefly all of the VDI routines; each VDI routine also has its own entry within the Lexicon.

xbios This entry describes the TOS function **xbios**, and introduces the function that use it to manipulate the Atari ST's extended BIOS.

A number of header files are also used with TOS. These include the following:

aesbind.h	bindings for GEM AES routines
basepage.h	TOS basepage structure
bios.h	declarations for **bios** functions
errno.h	**gemdos/bios/xbios** error number enumeration
gemdefs.h	miscellaneous declarations
gemout.h	TOS executable and archive file formats
linea.h	ST linea interface header
obdefs.h	miscellaneous object and variable definitions
osbind.h	bindings for bios/gemdos/xbios functions
signal.h	ST processor exception, extended trap vectors
stat.h	TOS DMABUFFER structure and file attributes
time.h	time and date services
vdibind.h	bindings for GEM VDI routines
xbios.h	declarations for **xbios** functions

Compiling TOS programs

You can include the AES/VDI libraries in your compilations in any of three ways.

First, you can include the libraries with the *library* option to the **cc** command line. To compile the program **sample.c**, use the following form of the **cc** command line:

```
cc sample.c -laes -lvdi
```

The -l option is described in the Lexicon entry for **cc**.

The other two methods involve using a switch on the **cc** command line. **-VGEM** is used to create an ordinary GEM program. It automatically links in the AES and VDI libraries, and calls the special run-time start-up routine **crtsg.o**. For example, to use the **-VGEM** option to compile **sample.c**, use the following command line:

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

crtsg.o has the advantage of being smaller, faster, and simpler than the default run-time start-up routine, **crts0.o**. Note, however, that it differs from the default runtime startup **crts0.o** in the following ways:

1. **argv**, **argc**, and **envp** are all set to zero.
2. **getenv** is not enabled; this means programs that use **crtsg.o** cannot read environmental parameters.
3. **stderr** will send error messages to the auxiliary ports rather to the console.

-VGEMACC is used to create a GEM desktop accessory. It works in much the same way as **-VGEM**, except that it uses the run-time start-up routine **crtsd.o** instead of **crtsg.o**.

The source files for **crtsd.o** and **crtsg.o** are included with your copy of Mark Williams C, should you wish to enhance it.

Finally, **libaes.a** uses the routine **crystal.o** to call traps. This routine is *never* called by the programmer, but it is automatically linked with **libaes.a**.

See Also

AES, bios, crtsg.o, gem, gemdos, keyboard, Lexicon, Line A, screen control, VDI, xbios

touch — Command

Update modification time of a file

touch [**-c**] *file* ...

TOS keeps track of when each file was last modified. **touch** changes the modification time of each *file* to the current time, but does not modify its contents. By default, **touch** creates *file* if it does not already exist; the **-c** flag suppresses this.

See Also

commands, make, msh

toupper — General function (libc)

Convert characters to upper case

#include <ctype.h>
int toupper(c) int c;

toupper is a macro that converts the letter *c* to upper case and returns the converted character. If *c* is not a letter or is already upper case, then **toupper** returns it unchanged.

Example

For examples of this routine, see the entries for **ctype** and **tolower**.

See Also

ctype, tolower, _toupper

_toupper — ctype macro (ctype.h)

Convert letter to upper case

#include <ctype.h>
_toupper(c) int c;

_toupper is a macro that returns *c* converted to upper case. If *c* is not a letter, the result is undefined.

_toupper differs from its cousin **toupper** in that **_toupper** is a macro that does not check whether its argument is in fact an alphanumeric character, whereas **toupper** is a function that does check its argument.

Example

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

See Also

ctype, _tolower, toupper

Tsetdate — gemdos function 43 (osbind.h)

Set a new date

#include <osbind.h>
long Tsetdate(*i*) int*i*;

Tsetdate sets a new date. The 16 bits of the integer *i* encode the date, as follows:

0-4	day (1-31)
5-8	month (1-12)
9-15	year (0-119, 0=1980)

Example

This example demonstrates the macros **Tsetdate** and **Tsettime**, and also uses the macros **Tgetdate** and **Tgettime**. For another example of this function, see the entry for **time**.

```
#include <osbind.h>

main() {
        unsigned int date;
        unsigned int time;
        int seconds;
        int minutes;

        int hours;
        int day;
        int month;
        int year;

        printf("Enter the date and time (MM/DD/YYYY HH:MM): ");
        scanf("%d/%d/%d %d:%d", &month, &day, &year, &hours, &minutes);
        seconds = 0;

        if (year < 100)
                year += 1900;
        date = ((unsigned)(year-1980)<<9)
                |((unsigned)month<<5)
                |(unsigned)day;

        time = ((unsigned)hours<<11)
                |((unsigned)minutes<<5)
                |((unsigned)seconds>>1);

        timeprint("About to set the TOS time to", time);
        dateprint("About to set the TOS date to", date);
        Tsetdate(date);
        Tsettime(time);
```



```
        date = Tgetdate();  /* Get the system date */
        time = Tgettime();  /* Get the system time */
        timeprint("Now the TOS time is", time);
        dateprint("Now the TOS date is", date);
}

void fixdig(buf, onumber, size)
char *buf;
int onumber;
int size;
{
        register long limit;
        register long number;
        int o;

        number = onumber;

        limit = 10;
        for (o = 1; o < size ; o++)
                limit *= 10;

        if ((number >= limit)||(number <0)) {
                for (o = 0; o < size; o++)
                        *buf++ = '*';
                *buf = 0;
                return;
        }

        for (o = 0; o < size; o++) {
                limit /= 10;
                *buf++ = '0'+number/limit;
                number = number%limit;
        }
        *buf = '\0';
}

timeprint(string, time)
char *string;
register unsigned int time;
{
        int seconds;
        int minutes;
        int hours;
        char mins[3];
        char secs[3];

        seconds = (time & 0x001F) << 1;   /* Bits  0:4 */
        minutes = (time >> 5)  & 0x3F;    /* Bits  5:10 */
        hours = (time >> 11) & 0x1F;      /* Bits 11:15 */

        fixdig(mins, minutes, 2);
        fixdig(secs, seconds, 2);
        printf("%s %d:%s:%s\n", string, hours, mins, secs);
}
```

```
dateprint(string, date)
char *string;
unsigned int date;
{
        int year;
        int month;
        int day;

        day = date & 0x1F;
        month = (date>>5) & 0x0F;
        year = ((date>>9) & 0x7F) + 1980;
        printf("%s %d/%d/%d\n", string, month, day, year);
}
```

See Also

gemdos, Tgetdate, time, TOS

Tsettime — gemdos function 45 (osbind.h)

Set a new time
#include <osbind.h>
long Tsettime(*time***) int** *time***;**

Tsettime sets a new system time. The argument **time** is an integer whose bits encode the time, in the following manner:

0-4	two-second increments (0-29)
5-10	minutes (0-59)
11-15	hours (0-23)

Example

For examples of this function, see the entries for **time** and **Tsetdate**.

See Also

gemdos, Tgettime, time, TOS

type checking — Technical information

Every expression has a *type*, such as **int**, **char**, or **double**. C is not strongly typed, which means that it allows different types to be mixed relatively freely, and be changed (or **cast**) from one type to another.

Mark Williams C checks types more strictly than the C standard implies. Mark Williams C's type checking can be enabled or disabled in degrees, using **-VSTRICT** and other "variant" options with the **cc** command.

See Also

cast, cc, type promotion

typedef — C keyword

Define a new data type

typedef is a C facility that lets you define new data types. Such definitions are always made in terms of existing data types; for example,

```
typedef long time_t;
```

establishes the data type **time_t**, and defines it to be equivalent to a **long**. Note that, by convention, programmer-defined data types are written in capital letters.

Judicious use of the **typedef** facility can make programs easier to maintain, and improve their portability.

See Also

C keyword, C language, declarations, manifest constants, portability, storage class
The C Programming Language, page 140

type promotion — Technical information

In arithmetic expressions, Mark Williams C promotes one signed type to another signed type by sign extension, and promotes one unsigned type to another unsigned type by zero padding. For example, **char** promotes to **int** by sign extension, while **unsigned char** promotes to **unsigned int** by zero padding.

See Also

data formats, declarations

U

#undef — Preprocessor instruction

Undefine a manifest constant
#undef *variable*

#undef tells the C preprocessor **cpp** to suspend the current value of *variable*, which had been defined with the **#define** command. For example, in a long program *variable* may be undefined for one function, where its current value conflicts with other elements of the function. The variable can be redefined or restored to its original value with another **#define** statement.

See Also

cc (-D option), cpp, #define

ungetc — STDIO function (libc)

Return character to input stream
#include <stdio.h>
int ungetc (*c*, *fp*) **int** *c*; **FILE** **fp*;

ungetc returns the character *c* to the stream *fp*. *c* can then be read by a subsequent call to **getc, gets, getw, scanf,** or **fread**. No more than one character can be pushed back into any stream at once. A call to **fseek** will nullify the effects of a previous **ungetc**.

Example

The following example opens a file and returns how many lines and sentences it contains. A sentence is defined as being any passage of text that ends in a period.

```
#include <stdio.h>
main()
{
        FILE *fp;
        int ch, nlines, nsents;
        int filename[20];
        nlines = nsents = 0;
        printf("Enter name of file to check: ");
        gets(filename);

        if ((fp = fopen(filename,"r")) != NULL)
        {
                while ((ch = fgetc(fp)) != EOF)
                {
                        if (ch == '\n') ++nlines;
                        else if (ch == '.' || ch == '!'
                                || ch == '?')
                        {
                                if ((ch = fgetc(fp)) != '.')
                                {
                                        ++nsents;
                                        ungetc(ch, fp);
                                }
                                else for(ch='.'; (ch=fgetc(fp))=='.';)
                                        ;
                        }
                }
                printf("%d line(s), %d sentence(s).\n",
                        nlines, nsents);
        }
        else printf("Cannot open %s.\n", filename);
}
```

See Also

fgetc, getc, STDIO
The C Programming Language, page 156

Diagnostics

ungetc normally returns *c*; it returns **EOF** if the character cannot be pushed back.

union — C keyword

Multiply declare a variable

A **union** defines an area of storage that can accept any one of several types of data. In effect, it is a multiple declaration of a variable. For example, a **union** may be declared to consist of an **int**, a **double**, and a **char** *; any one of these three elements can be held by the **union** at a time, and will be handled appropriately by it. For example, the declaration

```
union {
        int number;
        double bignumber;
        char *stringptr;
} example;
```

allows **example** to hold either an **int**, a **double**, or a pointer to a **char**, whichever is needed at the time. The elements of a **union** are accessed like those of a **struct**: for example, to access **number** from the above example, type **example.number**.

unions are helpful in dealing with heterogeneous data, especially within structures; however, the programmer is responsible for keeping track of what data type the **union** is holding at any given time. Passing a **double** to a **union** and then reading the **union** as though it held an **int** will yield results that are unpredictable, and probably unwelcome.

Example

For an example of how to use a **union** in a program, see the entry for **byte ordering**.

See Also

C keywords, C language, struct, structure
The C Programming Language, page 138

uniq — Command

Remove/count repeated lines in a sorted file
uniq [-cdu] [-*n*] [+*n*] [*infile*[*outfile*]]

uniq normally reads input line by line from *infile* and writes all non-duplicated lines to *outfile*. The input file must be sorted. **uniq** uses the standard input or output if either *infile* or *outfile* is omitted. The following describes the available options:

-c Print each line once, discarding duplicate lines; before each line, print the number of times it appears within the file.

-d Print only lines that are duplicated within the file; print each line only once; do not print any counts.

-u Print only lines that are *not* duplicated within the file.

uniq by default behaves as if both **-u** and **-d** were specified, so it prints each unique line once.

Optional specifiers allow **uniq** to skip leading portions of the input lines when comparing for uniqueness.

-*n* Skip *n* fields of each input line, where a field is any number of non-white space characters surrounded by any number of white space characters (blank or tab).

+*n* Skip *n* characters in each input line, after skipping fields as above.

See Also

commands

UNIX routines — Overview

Mark Williams C includes a number of routines that were originally written for version 7 of the UNIX system and related operating systems. These allow Mark Williams C to compile programs that were originally written for these systems.

The routines are as follows:

chdir	change working directory
chmod	change a file's 'mode'
chown	change a file's owner

close	close a file
creat	create/truncate a file
dup	duplicate a file descriptor
dup2	duplicate a file descriptor
errno	integer returned by error routine
execve	execute command from within program
_exit	exit directly from a program
lseek	set read/write position
open	open a file
read	read from a file
unlink	remove a file
write	write to a file

See Also

Lexicon, STDIO

unlink — UNIX system call (libc)

Remove a file
int unlink(*name*) char *name;

unlink removes the directory entry for the given file *name*, which in effect erases *name* from the disk. Note that *name* cannot be open when **unlink**ed. The name is a historical artifact.

Example

This example removes the files named on the command line.

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

        for (i = 1; i < argc; i++)
        {
                if (unlink(argv[i]) == -1)
                {
                        printf("cannot unlink \"%s\"\n", argv[i]);
                        exit(1);
                }
        }
        exit(0);
}
```

See Also

UNIX routines, STDIO

Diagnostics

unlink returns -1 if it cannot remove a file, and zero if it can.

unset – Command

Discard a shell variable
unset *VARIABLE*

unset discards a variable that had been set with the **set** command. For example, if you wished to discard the the variable **b**, simply type

```
unset b
```

and it will be erased.

unset uses the same sort of "in directory" syntax as the **set** command. For example, the command

```
unset in .bin
```

erases **.bin** and everything in it; whereas

```
unset in .bin ls lc cd echo
```

will remove **ls**, **lc**, **cd**, and **echo** from **.bin**.

See Also
commands, msh, set

unsetenv – Command

Discard an environmental variable
unsetenv *VARIABLE*

unset discards an environmental variable. For example, if you wish for some reason to discard the **TMPDIR** variable, type

```
unsetenv TMPDIR
```

See Also
commands, msh, setenv

unsigned – C keyword

Data type

The **unsigned** modifier tells the compiler to treat the variable as an unsigned value. In effect, this doubles the largest positive value storage in that type, and changes the lowest storage value to zero. Note that the 68000 uses "two's complement" storage, not sign magnitude.

See Also
C keywords, C language, data type
The C Programming Language, page 34

V

v_arc – VDI function (libvdi)

Draw a circular arc
#include <aesbind.h>
#include <vdibind.h>
void v_arc(*handle, x, y, radius, beginangle, endangle*)
int *handle, x, y, radius, beginangle, endangle*;

v_arc is a VDI routine that draws a circular arc. *handle* is the virtual device's VDI handle. *x* and *y* give, respectively, the X and Y coordinates of the imaginary center of the circle of which **v_arc** is drawing a section. *radius* is the radius of the imaginary circle. These measurements will differ, depending on whether the device has been set as using normalized device coordinates (NDC) or raster coordinates (RC).

Finally, *beginangle* and *endangle* give, respectively, the beginning and end angles of the arc, measured in tenths of a degree. Counting on an imaginary clock, zero degrees is at 3 o'clock, 90 degrees (900) at noon, 180 degrees (1800) at 9 o'clock, and 270 degrees (2700) at 6 o'clock.

See Also
TOS, v_circle, VDI

v_bar – VDI function (libvdi)

Draw a rectangle
#include <aesbind.h>
#include <vdibind.h>
void v_bar(*handle, xyarray*) **int** *handle, xyarray*[4];

v_bar is a VDI routine that draws a rectangle. Unlike its cousin **vr_recfl**, **v_bar** can draw a perimeter as well as the preset fill pattern.

handle is the virtual device's VDI handle. *xyarray* sets the X and Y coordinates from which to construct the rectangle; the even-numbered entries indicate the X coordinates, and the odd-numbered entries the Y coordinates. Which corner of the rectangle each pair of coordinates indicates will differ depending on whether the virtual device has been set to normalized device coordinates (NDC) or to raster coordinates (RC). On an NDC device, the first pair points to the lower left-hand corner and the second pair to the upper right-hand corner; whereas on an RC device, the first pair points to the upper left-hand corner and the second pair to the lower right-hand corner.

Note that to use this routine, the fill type must be set with **vsf_interior**, the fill style by **vsf_style**, the perimeter flag by **vsf_perimeter**, and the fill color by **vsf_color**. To draw a complex polygon (i.e., a shape other than a rectangle), use the routine **v_fillarea**.

Example

The following program draws a filled rectangle onto the screen. By clicking the mouse's left button and dragging the mouse, you can draw a rectangle on the screen. Pressing the 'T' key changes the rectangle's *type* of fill; and pressing the 'S' key changes its *style*. Pressing **<return>** exits.

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

#define CLICKS 1                        /* no. of clicks expected */
#define BUTTON 1                        /* which button; 1=leftmost */
#define BUTTONSTATE 1                   /* button state; 1=down */

/* global line A variables used by vdi; MUST be included */
int contrl[12],intin[128],ptsin[128],intout[128],ptsout[128];

/* array used by v_bar() */
int xyarray[] = { 1, 1, 1, 1 };

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 1, 1 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* Place for unwanted data */
int nowhere = 0;

main()
{
/* declarations used by evnt_multi() */
      int selection;                    /* code for event that occurred */
      unsigned int which = (MU_KEYBD | MU_BUTTON);
      int buffer[11];                   /* place to write AES messages */
      int mousex;                       /* mouse X coordinate */
      int mousey;                       /* mouse Y coordinate */
      unsigned key;                     /* key typed by user */

/* misc declarations */
      int vdihandle;                    /* virtual device's handle */
      int type = 0;                     /* type of fill */
      int style = 1;                    /* style of fill */
      int width;                        /* width of rubberbox */
      int depth;                        /* depth of rubberbox */

/* OK, here we go ... */
      appl_init();
      graf_mouse(ARROW, &nowhere);
      v_opnvwk(work_in, &vdihandle, work_out);

      /* set clipping array to size of screen */
      cliparray[2] = work_out[0];
      cliparray[3] = work_out[1];
```



```
/* set clipping rectangle; draw rectangle */
vs_clip(vdihandle, 1, cliparray);
vsf_perimeter(vdihandle, 1);

for(;;) {
      selection = evnt_multi(which, CLICKS, BUTTON,
            BUTTONSTATE, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            buffer, 0, 0, &mousex, &mousey,
            &nowhere, &nowhere, &key, &nowhere);

      switch(selection) {
      case MU_KEYBD:
            switch((char)key) {
            case '\r':
                  v_clsvwk(vdihandle);
                  appl_exit();
                  exit(0);

            case 't':
            case 'T':
                  type = (++type%5);
                  vsf_interior(vdihandle, type);
                  v_hide_c(vdihandle);
                  v_bar(vdihandle, xyarray);
                  v_show_c(vdihandle, 0);
                  break;

            case 's':
            case 'S':
                  style = ((style%24)+1);
                  vsf_style(vdihandle, style);
                  v_hide_c(vdihandle);
                  v_bar(vdihandle, xyarray);
                  v_show_c(vdihandle, 0);
                  break;
            }
            break;

      case MU_BUTTON:
            graf_rubbox(mousex,mousey,3,3,&width,&depth);
            xyarray[0] = mousex;
            xyarray[1] = mousey;
            xyarray[2] = (mousex+width);
            xyarray[3] = (mousey+depth);
            v_hide_c(vdihandle);
            v_bar(vdihandle, xyarray);
            v_show_c(vdihandle, 0);
            break;

      default:
            break;
      }
   }
}
```

See Also
TOS, vr_recfl, VDI

v_bit_image — VDI function (libvdi)

Print a bit image file
#include <aesbind.h>
#include <vdibind.h>
void v_bit_image(*handle, filename, aspect, scaling, points, xyarray***)**
int *handle, aspect, scaling, points, xyarray*[4]**; char** **filename*;

v_bit_image is a VDI routine that prints a bit image file. *handle* is the virtual device's VDI handle. *filename* points to the name of the file that holds the bit image; note that this name must be terminated with a NUL character.

aspect gives the code for the aspect ratio used to transfer the bit image onto paper, as follows:

0	ignore aspect ratio
1	honor pixel ratio
2	honor page aspect ratio

Pixel aspect ratio ensures that the figures within the bit image remain constant, e.g., that a circle will remain circular. This may involve some cropping or shrinking of the image when printing it. *Page aspect ratio* ensures that one full page in the bit image file is always printed as one full page of paper. This may result in some distortion of the figures within the bit image, however.

scaling describes how the bit image should be scaled onto to the page being printed. Zero indicates that the X and Y coordinates should be scaled together, whereas one indicates that they should be scaled separately. Note that this argument is meaningful only if the variables in *xyarray* are set. If the X and Y coordinates are scaled together, the printed image may not fully occupy the rectangle defined by *xyarray* on the output device. If they are scaled separately, the bit image will entirely fill the area defined by *xyarray*, but the setting of *aspect* will be ignored.

Finally, *xyarray* defines the upper left-hand and lower right-hand corners of the area on the page into which the bit image will be printed. The even-numbered entries set the X coordinates, and the odd-numbered entries the Y coordinates.

See Also
TOS, VDI

Notes
This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

v_cellarray — VDI function (libvdi)

Draw a table of colored cells
#include <aesbind.h>
#include <vdibind.h>
void v_cellarray(*handle, xyarray, rowlength, cells, rows, mode, cellarray***)**
int *handle, xyarray, rowlength*[4]*, cells, rows, mode, cellarray*[*n*]**;**

v_cellarray is a VDI routine that draws a table of colored cells. *handle* is the virtual device's VDI handle. *xyarray* gives the X and Y coordinates for the rectangle in which the table will be drawn. Note that these values will vary, depending on whether the device is set to normalized device coordinates (NDC) or to raster coordinates (RC). On NDC devices, *xyarray[0]* and *xyarray[1]* give, respectively, the X and Y coordinates of the lower left-hand corner of the rectangle, and *xyarray[2]* and *xyarray[3]* give the coordinates for the upper right-handle corner. On RC devices, *xyarray[0]* and *xyarray[1]* give, respectively, the X and Y coordinates of the upper left-hand corner, whereas *xyarray[2]* and *xyarray[3]* give the X and Y coordinates of the lower right-hand corner.

rowlength gives the horizontal length of the table to be shown, in NDCs or RCs. *cells* is the number of cells to be drawn in each row, and *rows* is the number of rows of cells to draw. *mode* is the writing mode in which the cells will be drawn: one indicates replace mode; two, transparent mode; three, XOR (exclusive or); and four, reverse transparent mode.

Finally, *cellarray* gives the array of colors to be shown in the cells. *n* must equal *cells* times *rows*.

See Also
TOS, VDI, vq_cellarray

Notes
This routine is not present in the resident VDI. It may not be present in the GDOS.

v_circle — VDI function (libvdi)

Draw a circle
#include <aesbind.h>
#include <vdibind.h>
void v_circle(*handle, x, y, radius***) int** *handle, x, y, radius***;**

v_circle is a VDI routine that draws a circle. *handle* is the virtual device's VDI handle. *x* and *y* give, respectively, the X and Y coordinates of the circle's center. *radius* gives the circle's radius. These measurements will vary, depending on whether the device has been defined as using normalized device coordinates (NDC) or raster coordinates (RC).

Example

The following program, called **circle.c**, draws a circle on screen. The first mouse click sets the circle's center; the second mouse click sets its radius. The 'W' key cycles through the available write modes, for truly psychedelic effects. Pressing **<return>** exits. Compile it with the command line

```
cc -V -VGEM circle.c -lm
```

to include the necessary mathematics routine.

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

#define ASTERISK 3                      /* draw an asterisk marker */
#define BUTTON 1                        /* which button; 1=leftmost */
#define CLICKS 1                        /* no. of clicks expected */
#define DOWN 1                          /* mouse button is down */
#define UP 0                            /* mouse button is up */
#define XOR 3                           /* XOR mode */

/* global line A variables used by vdi; MUST be included */
int contrl[12],intin[128],ptsin[128],intout[128],ptsout[128];

/* array used to calculate radius */
int xyarray[4];

/* array used by v_pmarker() */
int xymarker[2];

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 1, 1 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* throw-away declaration */
int nowhere = 0;

main()
{
/* declarations used by evnt_multi() */
      int selection;                    /* code for event */
      unsigned int which = (MU_KEYBD | MU_BUTTON);
      int buffer[8];                    /* place for AES messages */
      int mousex;                       /* mouse X coordinate */
      int mousey;                       /* mouse Y coordinate */
      unsigned key;                     /* key typed by user */

/* misc declarations */
      int vdihandle;                    /* virtual device's handle */
      int writectr = 0;                 /* write modes */
      int fillctr = 1;                  /* circle fill styles */
      int n = 0;                        /* keep track of xyarray */
```



```
appl_init();
graf_mouse(ARROW, &nowhere);
v_opnvwk(work_in, &vdihandle, work_out);

/* set clipping rectangle to size of screen */
cliparray[2] = work_out[0];
cliparray[3] = work_out[1];
vs_clip(vdihandle, 1, cliparray);

vsf_interior(vdihandle, 2);
vsf_perimeter(vdihandle, 1);
vsm_height(vdihandle, 3);
vsm_type(vdihandle, ASTERISK);

for(;;) {
      selection = evnt_multi(which, CLICKS, BUTTON, DOWN,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, buffer, 0, 0,
            &mousex, &mousey, &nowhere, &nowhere, &key,
            &nowhere);

      switch(selection) {
      /* if keyboard is pressed ... */
      case MU_KEYBD:
            if ((char)key == '\r') {
                  v_clsvwk(vdihandle);
                  appl_exit();
                  exit(0);
            }
            if (((char)key == 'w') || ((char)key == 'W'))
                  writectr++;
            break;

      /* if user presses a mouse button ... */
      case MU_BUTTON:
            evnt_button(CLICKS, BUTTON, UP,
                  &nowhere, &nowhere, &nowhere, &nowhere);
            if (n == 0) {
                  /* draw center marker in XOR mode */
                  xymarker[0] = mousex;
                  xymarker[1] = mousey;
                  graf_mouse(M_OFF, &nowhere);
                  vswr_mode(vdihandle, XOR);
                  v_pmarker(vdihandle, 1, xymarker);
                  graf_mouse(M_ON, &nowhere);
            }

            xyarray[n++] = mousex;
            xyarray[n++] = mousey;
            if (n > 3) {
                  n = 0;
                  fillctr++;
                  /* XOR-away the center marker ... */
                  v_pmarker(vdihandle, 1, xymarker);
                  /* ... and set new drawing mode */
                  vswr_mode(vdihandle, (writectr%4)+1);
```

```
                        vsf_style(vdihandle, (fillctr%24)+1);
                        drawcircle(vdihandle);
                    }
                    break;

                default:
                    break;
                }
        }
}

drawcircle(handle)
int handle;
{
        int leg1;                       /* first leg of triangle */
        int leg2;                       /* second leg */
        int radius;                     /* radius of circle=hypotenuse */
        extern double hypot();          /* declare hypot() */

        leg1 = abs(xyarray[2] - xyarray[0]);
        leg2 = abs(xyarray[3] - xyarray[1]);
        /* note casts of variables */
        radius = (int) hypot( (double) leg1, (double) leg2);

/* now, draw the circle */
        graf_mouse(M_OFF, &nowhere);
        v_circle(handle, xyarray[0], xyarray[1], radius);
        graf_mouse(M_ON, &nowhere);
        return;
}
```

See Also

TOS, v_ellipse, VDI

v_clear_disp_list — VDI function (libvdi)

Clear a printer's display list

#include <aesbind.h>
#include <vdibind.h>
void v_clear_disp_list(*handle*) int *handle*;

v_clear_disp_list is a VDI routine that clear's a printer's display list. Unlike the related function **v_clrwk**, it does not set a new page.

See Also

TOS, v_form_adv, v_clrwk, VDI

Notes

This routine is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

v_clrwk — VDI function (libvdi)

Clear the virtual workstation

#include <aesbind.h>

#include <vdibind.h>
void v_clrwk(*handle*) int *handle*;

v_clrwk is a VDI routine that clears the virtual workstation. It is executed automatically after a device is opened. It clears the screen device by setting it to the background color, and clears a hard-copy device (e.g., printer, plotter) by sending a new-page signal. *handle* is the device's VDI handle.

Example

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

See Also

TOS, v_clear_disp_list, v_form_adv, VDI

v_clsvwk — VDI function (libvdi)

Close the screen virtual device

#include <aesbind.h>
#include <vdibind.h>
void v_clsvwk(*handle*) int *handle*;

v_clsvwk is a VDI routine that closes the screen virtual device. It also flushes all appropriate buffers, frees the space assigned to the screen's device driver, and otherwise performs all tasks needed to close the device gracefully. *handle* is the screen's VDI handle.

Example

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

See Also

TOS, VDI, v_clswk, v_opnvwk, v_opnwk

v_clswk — VDI function (libvdi)

Close a virtual workstation

#include <aesbind.h>
#include <vdibind.h>
void v_clswk(*handle*) int *handle*;

v_clswk is a VDI routine that closes a virtual workstation. It also flushes the any associated buffers and frees the memory allocated to the workstation's driver, to conclude matters gracefully. *handle* is the device's VDI handle.

See Also

TOS, VDI, v_opnvwk, v_opnwk

Notes

This routine is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI. To close the screen device, use the related function **v_clsvwk**.

v_contourfill — VDI function (libvdi)

Fill an outlined area

#include <aesbind.h>
#include <vdibind.h>
void v_contourfill(*handle, x, y, color***)**
int *handle, x, y, color*;

v_contourfill is a VDI routine that fills an outlined area with a fill pattern. Note that the fill type must be set by the function **vsf_interior**, and the fill style by the function **vsf_style**.

handle is the virtual device's VDI handle. *x* and *y* give, respectively, the X and Y coordinates of the point at which filling is to begin. Finally, *color* is the code for the color at which filling stops. For a table of color settings, see the entry for **v_opnwk**.

Example

The following example lets the user draw a number of "rubber lines" on the screen. The 'W' key floods an enclosed area with the fill pattern. Pressing **<return>** exits.

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

#define BLACK 1                         /* code for color black */
#define BUTTON 1                        /* which button; 1 = leftmost */
#define CLICKS 1                        /* no. of clicks expected */
#define DOWN 1                          /* mouse button is down */
#define REPLACE 1                       /* make writing mode REPLACE */
#define UP 0                            /* mouse button is up */
#define XOR 3                           /* make writing mode XOR */

/* global line A variables used by vdi; MUST be included */
int contrl[12],intin[128],ptsin[128],intout[128],ptsout[128];

/*
 * array used by vs_clip(); MUST be set, or images that
 * extend beyond the screen perimeters will write over
 * low-level memory (e.g., RAM disks, spoolers, etc.)
 */
int cliparray[] = { 1, 1, 1, 1 };

/* array used by evnt_multi(), for writing AES messages */
int buffer[8];

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* array used by v_pline() */
int xyarray[4];
```



```
/* throw-away declarations */
int nowhere = 0;

main()
{
/* declarations used by evnt_multi() */
        int selection;                  /* code for event */
        unsigned key;                   /* scan code of key */
        int mousex;                     /* mouse X coordinate */
        int mousey;                     /* mouse Y coordinate */
        int vdihandle;                  /* virtual device's handle */
        int flag = 0;                   /* has line been drawn yet? */
        int i = 0;                      /* counter */

/* OK, here we go ... */
        appl_init();
        graf_mouse(ARROW, &nowhere);
        v_opnvwk(work_in, &vdihandle, work_out);

        cliparray[2] = work_out[0];
        cliparray[3] = work_out[1];
        vs_clip(vdihandle, 1, cliparray);
        vsf_interior(vdihandle, 2);
        vsf_style(vdihandle, 23);
        vswr_mode(vdihandle, XOR);

        for(;;) {
                selection = evnt_multi(MU_KEYBD | MU_BUTTON,
                        CLICKS, BUTTON, DOWN, 0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0, buffer, 0, 0, &mousex,
                        &mousey, &nowhere, &nowhere, &key,
                        &nowhere);

                switch(selection) {
                case MU_KEYBD:
                        if ((char)key == '\r') {
                                v_clsvwk(vdihandle);
                                appl_exit();
                                exit(0);
                        }

                        if(((char)key=='w')||((char)key=='W')) {
                                graf_mouse(M_OFF, &nowhere);
                                v_contourfill(vdihandle, mousex,
                                        mousey, BLACK);
                                graf_mouse(M_ON, &nowhere);
                        }
                        break;

                case MU_BUTTON:
                        /* "rubberline" routine */
                        if (flag > 0) {
                                /* if line has moved ... */
                                if((xyarray[2]!=mousex)
                                        || (xyarray[3] != mousey)) {
```

```
                                /* ... undraw old line ... */
                                graf_mouse(M_OFF, &nowhere);
                                v_pline(vdihandle, 2, xyarray);
                                graf_mouse(M_ON, &nowhere);


                                    xyarray[2] = mousex;
                                xyarray[3] = mousey;

                                /* ... and draw new line */
                                graf_mouse(M_OFF, &nowhere);
                                v_pline(vdihandle, 2, xyarray);
                                graf_mouse(M_ON, &nowhere);
                        }
                }

                if (flag == 0) {
                        /* redraw line in REPLACE mode */
                        vswr_mode(vdihandle, REPLACE);
                        graf_mouse(M_OFF, &nowhere);
                        v_pline(vdihandle, 2, xyarray);
                        graf_mouse(M_ON, &nowhere);
                        vswr_mode(vdihandle, XOR);

                        /* reset endpoints */
                        xyarray[0] = mousex;
                        xyarray[1] = mousey;
                        xyarray[2] = mousex;
                        xyarray[3] = mousey;
                        flag = 1;
                }
                flag = check();
                break;

        default:
                break;
        }
    }
}

check()
{
    int buttonstate = 1;                /* button state */

    evnt_multi(MU_TIMER, CLICKS, BUTTON,
        UP, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, buffer,
        0, 0, &nowhere, &nowhere, &buttonstate,
        &nowhere, &nowhere, &nowhere);

    return(buttonstate);
}
```

See Also

TOS, VDI

Notes

Due to the way the AES routine reads the mouse buttons, the example will not always notice that the mouse button has returned to the up position.

v_curdown — VDI function (libvdi)

Move text cursor down one row

#include <aesbind.h>
#include <vdibind.h>
void v_curdown(*handle*) int *handle*;

v_curdown is a VDI routine that moves the text cursor down one row. It does not affect the cursor's horizontal position. Note that the virtual device must first be put into text mode with the function **v_enter_cur** before this function can be used. *handle* is the virtual device's VDI handle.

Example

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

See Also

TOS, v_curhome, v_curleft, v_curright, v_curup, VDI

v_curhome — VDI function (libvdi)

Move text cursor to the home position

#include <aesbind.h>
#include <vdibind.h>
void v_curhome(*handle*) int *handle*;

v_curhome is a VDI routine that moves the text cursor to the home position, i.e., to the upper left-hand corner. Note that the virtual device must first be put into text mode with the function **v_enter_cur** before this function can be used. *handle* is the virtual device's VDI handle.

Example

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

See Also

TOS, v_curdown, v_curleft, v_curright, v_curhome, VDI

v_curleft — VDI function (libvdi)

Move text cursor left one column

#include <aesbind.h>
#include <vdibind.h>
void v_curleft(*handle*) int *handle*;

v_curleft is a VDI routine that moves the text cursor one column to the left. It does not affect the cursor's vertical position. Note that the virtual device must first be put into text mode with the function **v_enter_cur** before this function can be used. *handle* is the virtual device's VDI handle.

Example

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

See Also

TOS, v_curdown, v_curhome, v_curright, v_curup, VDI

v_curright — VDI function (libvdi)

Move text cursor right one column

#include <aesbind.h>
#include <vdibind.h>
void v_curright(*handle*) int *handle*;

v_curright is a VDI routine that moves the text cursor one column to the right. It does not affect the cursor's vertical position. Note that the virtual device must first be put into text mode with the function **v_enter_cur** before this function can be used. *handle* is the virtual device's VDI handle.

Example

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

See Also

TOS, v_curdown, v_curhome, v_curleft, v_curup, VDI

v_curtext — VDI function (libvdi)

Write alphabetic text

#include <aesbind.h>
#include <vdibind.h>
void v_curtext(*handle, string*) int *handle*; char **string*;

v_curtext is a VDI routine that writes alphabetic text on the virtual device. Note that to use this routine, the virtual device must first be placed in text mode, using the routine **v_enter_cur**. *handle* is the virtual device's VDI handle. *string* points to the NUL-terminated string of alphabetic characters to be written.

Example

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

See Also

TOS, VDI

v_curup — VDI function (libvdi)

Move text cursor up one row

#include <aesbind.h>
#include <vdibind.h>
void v_curup(*handle*) int *handle*;

v_curup is a VDI routine that moves the text cursor up one row. It does not affect the cursor's horizontal position. Note that the virtual device must first be put into text mode with the function **v_enter_cur** before this function can be used. *handle* is the virtual device's VDI handle.

Example

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

See Also

TOS, v_curdown, v_curhome, v_curleft, v_curright, VDI

v_dspcur — VDI function (libvdi)

Move mouse pointer to point on screen

#include <aesbind.h>
#include <vdibind.h>
void v_dspcur(*handle, x, y*) int *handle, x, y*;

v_dspcur is a VDI routine that moves the mouse pointer to a specified point on the screen. *handle* is the virtual device's VDI handle. *x* and *y* are, respectively, the X and Y coordinates to which the mouse cursor will be moved.

See Also

TOS, VDI

v_eeol — VDI function (libvdi)

Erase text from cursor to end of screen

#include <aesbind.h>
#include <vdibind.h>
void v_eeol(*handle*) int *handle*;

v_eeol is a VDI routine that erases alphabetic text from the position of the text cursor to the end of the line. Note that the virtual device must first be put into text mode with the function **v_enter_cur** before this function can be used. *handle* is the virtual device's VDI handle.

Example

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

See Also

TOS, v_eeos, VDI

v_eeos — VDI function (libvdi)

Erase from text cursor to end of screen

#include <aesbind.h>
#include <vdibind.h>
void v_eeos(*handle*) int *handle*;

v_eeos is a VDI routine that erases a virtual device from the position of the text cursor to the end. Note that the virtual device must first be put into text mode, with the function **v_enter_cur** before this function can be used. *handle* is the virtual device's VDI handle.

Example

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

See Also

TOS, v_eeol, VDI

v_ellarc — VDI function (libvdi)

Draw an elliptical arc

#include <aesbind.h>
#include <vdibind.h>
void v_ellarc(*handle, x, y, xradius, yradius, beginangle, endangle*)
int *handle, x, y, xradius, yradius, beginangle, endangle*;

v_ellarc is a VDI routine that draws an elliptical arc. *handle* is the virtual device's VDI handle. *x* and *y* give, respectively, the X and Y coordinates of the center of the imaginary ellipse, of which the curve being drawn is part. *xradius* is the horizontal radius of the ellipse, and *yradius* is the vertical radius. Note that all of these values will vary, depending on whether the virtual device uses normalized device coordinates (NDC) or raster coordinates (RC). Finally, *beginangle* and *endangle* represent the beginning and end angles of the ellipse, in tenths of a degree. On an imaginary clock, zero degrees is at 3 o'clock, 90 degrees at noon, 180 degrees at 9 o'clock, and 270 degrees at 6 o'clock.

Example

The following program uses STDIO routines to create a "rough-and-ready" dialogue; the user sets the X radius, the Y radius, the beginning angle, and the end angle, which are then used to draw an elliptical arc on the screen.

```
#include <aesbind.h>
#include <gemdefs.h>
#include <stdio.h>
#include <vdibind.h>

#define ESCAPE 0x1B                     /* ASCII escape code */
#define REPLACE 1                       /* REPLACE writing mode */
#define RCUNDED 2                       /* put rounded ends on lines */
#define XOR 3                           /* XOR writing mode */

/* global line A variables used by vdi; MUST be included */
int contrl[12],intin[128],ptsin[128],intout[128],ptsout[128];
```



```
/*
 * array used by vs_clip(); MUST be set, or images that extend
 * beyond the screen perimeters will write over low-level memory
 * (e.g., RAM disks, spoolers, etc.)
 */
int cliparray[] = { 1, 1, 1, 1 };

/* arrays used by drawline() */
xyvert[] = { 1, 1, 1, 1 };
xyhoriz[] = { 1, 1, 1, 1 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* keep system from scribbling over itself */
int nowhere = 0;

main()
{
        unsigned key;                   /* key pressed by user */
        int vdihandle;                  /* virtual device's handle */
        int xradius;                    /* length of X radius */
        int yradius;                    /* length of Y radius */
        int beginangle;                 /* beginning angle */
        int endangle;                   /* end angle */

 /* OK, here we go ... */
        appl_init();
        graf_mouse(M_OFF, &nowhere);
        v_opnvwk(work_in, &vdihandle, work_out);

        /* set clipping array to match screen dimensions */
        cliparray[2] = work_out[0];
        cliparray[3] = work_out[1];

        /* set line arrays to suit screen dimensions */
        xyvert[0] = work_out[0]/2;
        xyvert[1] = 1;
        xyvert[2] = work_out[0]/2;
        xyvert[3] = work_out[1];

        xyhoriz[0] = 1;
        xyhoriz[1] = work_out[1]/2;
        xyhoriz[2] = work_out[0];
        xyhoriz[3] = work_out[1]/2;

        /* set clipping rectangle & line style */
        vs_clip(vdihandle, 1, cliparray);
        vsl_ends(vdihandle, ROUNDED, ROUNDED);

        /* and execute the program */
        for(;;) {
                printf("Type <return> to continue, <esc> to exit.\n");
                key = evnt_keybd();
                switch((char)key) {
```

```
            /* user wants to exit */
            case ESCAPE:
                  v_clsvwk(vdihandle);
                  appl_exit();
                  exit(0);

            /* user wants to continue */
            case '\r':
                  drawlines(vdihandle);
                  /* Enter X radius */
                  xradius=getdata("Enter X radius (screen, 0-320)");
                  xyhoriz[0] = (work_out[0]/2) - xradius;
                  xyhoriz[2] = (work_out[0]/2) + xradius;
                  drawlines(vdihandle);

                  /* Enter Y radius */
                  yradius=getdata("Enter Y radius (screen, 0-200)");
                  xyvert[1] = (work_out[1]/2) - yradius;
                  xyvert[3] = (work_out[1]/2) + yradius;
                  drawlines(vdihandle);

                  /* Enter beginning angle */
                  beginangle=getdata("Beginning angle (0-360)")*10;
                  drawlines(vdihandle);

                  /* Enter end angle */
                  endangle=getdata("Enter end angle (0-360)")*10;
                  drawlines(vdihandle);

                  /* And now, draw the elliptical arc */
                  vsl_width(vdihandle, 5);
                  v_ellarc(vdihandle, work_out[0]/2, work_out[1]/2,
                        xradius, yradius, beginangle, endangle);
                  break;

            default:
                  break;
            }
      }
}

/* draw cross-hairs on screen */
drawlines(handle)
int handle;
{
      printf("\033E\n");
      vsl_width(handle, 1);
      v_pline(handle, 2, xyvert);
      v_pline(handle, 2, xyhoriz);
      return;
}
```



```
/* get dimensions of arc from user */
getdata(message)
char *message;
{
      for(;;) {
            char string[20];     /* string used with user input */
            int value;           /* value user intended */

            printf("%s: ", message);
            fflush(stdout);
            if((value = atoi(gets(string))) >= 0)
                  return(value);

      }
}
```

See Also

TOS, VDI, v_ellipse

v_ellipse — VDI function (libvdi.a/v_ellipse)

Draw an ellipse

#include <aesbind.h>
#include <vdibind.h>
void v_ellipse(*handle, x, y, xradius, yradius***)**
int *handle, x, y, xradius, yradius***;**

v_ellipse is a VDI routine that draws an ellipse. *handle* is the virtual device's VDI handle. *x* and *y* give, respectively, the X coordinates and Y coordinates of the ellipse's center. Note that these measurements will change, depending on whether the virtual device is set to normalized device coordinates (NDC) or raster coordinates (RC). Finally, *xradius* gives the ellipse's horizontal radius and *yradius* gives its vertical radius.

Example

The following example draws ellipses on the screen. Clicking the mouse draws a rubber box; releasing the mouse fixes the box, whose dimensions are used to calculate the ellipse. Pressing the 'W' key cycles through the available write modes. Pressing **<return>** exits.

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

#define DOWN 1                        /* mouse button is down */
#define CLICKS 1                      /* no. of clicks expected */
#define BUTTON 1                      /* which button; 1=leftmost */

/* global line A variables used by vdi; MUST be included */
int contrl[12],intin[128],ptsin[128],intout[128],ptsout[128];

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 1, 1 };
```

```
/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* throw-away declaration */
int nowhere = 0;

main()
{
/* declarations used by evnt_multi() */
      int selection;                    /* code for event that occurred */
      unsigned int which = (MU_KEYBD | MU_BUTTON);
      int buffer[8];                    /* place to write messages */
      int mousex;                       /* mouse X coordinate */
      int mousey;                       /* mouse Y coordinate */
      unsigned key;                     /* key typed by user */

/* misc declarations */
      int vdihandle;                    /* virtual device's handle */
      int writectr = 0;                 /* cycle through write modes */
      int fillctr = 1;                  /* cycle through fill styles */
      int width;                        /* box width */
      int height;                       /* box height */
      int xcoord;                       /* X coordinate of center */
      int ycoord;                       /* Y coordinate of center */
      int xradius;                      /* X radius of ellipse */
      int yradius;                      /* Y radius of ellipse */

/* OK, here we go ... */
      appl_init();
      graf_mouse(ARROW, &nowhere);
      v_opnvwk(work_in, &vdihandle, work_out);

      /* set clipping array to match screen dimensions */
      cliparray[2] = work_out[0];
      cliparray[3] = work_out[1];
      vs_clip(vdihandle, 1, cliparray);

      vsf_interior(vdihandle, 2);
      vsf_perimeter(vdihandle, 1);
      vsm_height(vdihandle, 3);

      for(;;) {
            selection = evnt_multi(which, CLICKS, BUTTON,
                  DOWN, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                  buffer, 0, 0, &mousex, &mousey,
                  &nowhere, &nowhere, &key, &nowhere);

            switch(selection) {
            case MU_KEYBD:
                  if ((char)key == '\r') {
                        v_clsvwk(vdihandle);
                        appl_exit();
                        exit(0);
                  }
```



```
                  if (((char)key=='w')||((char)key=='W')) {
                        writectr++;
                        vswr_mode(vdihandle, (writectr%4)+1);
                  }
                  break;

            case MU_BUTTON:
                  fillctr++;
                  vsf_style(vdihandle, (fillctr%24)+1);
                  graf_rubbox(mousex, mousey, 0, 0, &width,
                        &height);
                  xcoord = mousex+(width/2);
                  ycoord = mousey+(height/2);
                  xradius = width/2;
                  yradius = height/2;
                  v_ellipse(vdihandle, xcoord, ycoord,
                        xradius, yradius);
                  break;

            default:
                  break;
            }
      }
}
```

See Also

TOS, VDI, v_ellarc, v_ellpie

Notes

v_ellipse can only create ellipses that are oriented horizontally or vertically. It cannot create ellipses that are oriented diagnonally.

v_ellpie — VDI function (libvdi)

Draw an elliptical pie slice

#include <aesbind.h>
#include <vdibind.h>
void v_ellpie(*handle, x, y, xradius, yradius, beginangle, endangle*)
int *handle, x, y, xradius, yradius, beginangle, endangle*;

v_ellpie is a VDI routine that draws an elliptical pie slice. *handle* is the virtual device's VDI handle. *x* and *y* give, respectively, the X and Y coordinates of the imaginary ellipse of which **v_ellpie** draws a part. *xradius* gives the imaginary ellipse's horizontal radius, and *yradius* gives its vertical radius. Finally, *beginangle* and *endangle* give, respectively, the beginning and end angles of the pie slice, in tenths of a degree. On an imaginary clock, zero degrees is at 3 o'clock, 90 degrees at noon, 180 degrees at 9 o'clock, and 270 degrees at 6 o'clock.

See Also

TOS, VDI, v_ellipse

v_enter_cur — VDI function (libvdi)

Enter text mode

#include <aesbind.h>
#include <vdibind.h>
void v_enter_cur(*handle*) int *handle*;

v_enter_cur is a VDI routine that moves a virtual device into text mode. It hides the mouse pointer and draws the text cursor. *handle* is the virtual device's VDI handle.

Example

The following example creates a simple screen editor using VDI text calls. Note that it does not save anything you type, or have the capacity to write what you type into a file, although these features can be added. The keystrokes resemble those used by the MicroEMACS editor.

```
#include <aesbind.h>
#include <vdibind.h>

/* control characters used in example */
#define CTRLA 0x01  /* move to beginning of line */
#define CTRLB 0x02  /* move back one character */
#define CTRLF 0x06  /* move forward one character */
#define CTRLH 0x08  /* home cursor */
#define CTRLK 0x0B  /* kill line of text */

#define CTRLN 0x0E  /* move to next line */
#define CTRLP 0x10  /* move up to previous line */
#define CTRLR 0x12  /* toggle reverse video */
#define CTRLW 0x17  /* kill text to end of screen */
#define CTRLX 0x18  /* secondary set of keystrokes */
#define CTRLZ 0x1A  /* exit from program */

/* standard VDI arrays */
int contrl[12],intin[128],ptsin[128],intout[128],ptsout[128];

/* arrays used by v_opnvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 1, 1 };

/* place to write junk */
long nowhere;

main()
{
        int row, column, videoflag;
        int vdihandle;
        char key[2];
        char string[50];
```



```
key[1] = '\0';
videoflag = 0;

appl_init();
v_opnvwk(work_in, &vdihandle, work_out);

/* set clipping area */
cliparray[2] = work_out[0];       /* width of screen */
cliparray[3] = work_out[1];       /* height of screen */
vs_clip(vdihandle, 1, cliparray);

/* enter text mode */
v_enter_cur(vdihandle);

/* accept characters from the keyboard and process them */
for(;;) {
        key[0] = (char) evnt_keybd();

        switch (key[0]) {
                case '\r':
                        v_curdown(vdihandle);
                        /* Note: no break */

                case CTRLA:
                        vq_curaddress(vdihandle, &row, &column);
                        vs_curaddress(vdihandle, row, 1);
                        break;

                case CTRLB:
                        v_curleft(vdihandle);
                        break;

                case CTRLF:
                        v_curright(vdihandle);
                        break;

                case CTRLH:
                        v_curhome(vdihandle);
                        break;

                case CTRLK:
                        v_eeol(vdihandle);
                        break;

                case CTRLN:
                        v_curdown(vdihandle);
                        break;

                case CTRLP:
                        v_curup(vdihandle);
                        break;
```

```
                case CTRLR:
                        if ((videoflagX2) == 0)
                                v_rvon(vdihandle);
                        else
                                v_rvoff(vdihandle);
                        videoflag++;
                        break;

                case CTRLW:
                        v_eeos(vdihandle);
                        break;

                case CTRLX:
                        switch((char)evnt_keybd()) {
                        /* print out current position */
                        case '=':
                                vq_curaddress(vdihandle,
                                        &row, &column);
                                vs_curaddress(vdihandle,
                                        24, 1);
                                sprintf(string, "Row: %2d  Column %2d",
                                        row, column);
                                v_curtext(vdihandle, string);
                                vs_curaddress(vdihandle,
                                        row, column);
                                break;
                        }
                        break;

                case CTRLZ:
                        v_exit_cur(vdihandle);
                        v_clsvwk(vdihandle);
                        exit(0);

                default:
                        v_curtext(vdihandle, key);
                        break;
        }
    }
}
```

See Also

TOS, v_exit_cur, VDI

v_exit_cur — VDI function (libvdi)

Exit from text mode

#include <aesbind.h>
#include <vdibind.h>
void v_exit_cur(*handle***) int** *handle*;

v_exit_cur is a VDI routine that forces a virtual device to exit from text mode and return to graphics mode, should these modes be separate on that device. It removes the text cursor from the device and restores the mouse pointer, should the virtual device support them. *handle* is the virtual device's VDI handle.

Example

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

See Also

TOS, v_enter_cur, VDI

v_fillarea — VDI function (libvdi)

Draw a complex polygon

#include <aesbind.h>
#include <vdibind.h>
void v_fillarea(*handle, count, xyarray***) int** *handle, count, xyarray*[*n*];

v_fillarea is a VDI routine that draws and fills a complex polygon. Note that to use the full power of this routine, you must first set the fill type with **vsf_interior**, the fill style with **vsf_style**, and the fill color with **vsf_color**.

handle is the virtual device's VDI handle. *count* is the number of corners on the polygon. *xyarray* gives the X and Y coordinates for each of the corners: all of the even-numbered entries hold X coordinates, and all the odd-numbered entries hold Y coordinates. Note that the value of *n* must be exactly double that of *count*.

Example

The following program draws a filled polygon on screen. Use mouse to set markers on the screen, with a maximum of 40 points. Pressing **<esc>** "connects the dots" to draw and fill the polygon. Pressing the 'T' key cycles through types of fill; pressing the 'S' key cycles through styles of fill. Pressing **<return>** exits.

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

#define ASTERISK 3
#define CLICKS 1                /* no. of clicks expected */
#define BUTTON 1                /* which button; 1=leftmost */
#define BUTTONSTATE 1           /* button state expected; 1=down */
#define ESC 0x1B                /* code for <esc> */

/* global line A variables used by vdi; MUST be included */
int contrl[12],intin[128],ptsin[128],intout[128],ptsout[128];

/* array used by v_pmarker() */
int xymarker[2];

/* array used by v_fillarea() */
int xypoly[80];

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 1, 1 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];
```

```
/* throw-away declaration */
int nowhere = 0;

main()
{
/* declarations used by evnt_multi() */
      int selection;                    /* code for event */
      unsigned int which = (MU_KEYBD | MU_BUTTON);
      int buffer[11];                   /* place to write AES messages */
      int mousex;                       /* mouse X coordinate */
      int mousey;                       /* mouse Y coordinate */
      unsigned key;                     /* key typed by user */

/* misc declarations */
      int vdihandle;                    /* virtual device's handle */
      int type = 0;                     /* type of fill */
      int style = 1;                    /* style of fill */
      int n = 0;                        /* used with xyarray[] */
      int flag = 0;                     /* has polygon been drawn yet? */

/* OK, here we go ... */
      appl_init();
      graf_mouse(ARROW, &nowhere);
      v_opnvwk(work_in, &vdihandle, work_out);

      /* set clipping array to match screen dimensions */
      cliparray[2] = work_out[0];
      cliparray[3] = work_out[1];
      vs_clip(vdihandle, 1, cliparray);

      vsm_height(vdihandle, 3);
      vsm_type(vdihandle, ASTERISK);

      for(;;) {
            selection = evnt_multi(which, CLICKS, BUTTON,
                  BUTTONSTATE, 0, 0, 0, 0, 0, 0, 0, 0,
                  0, 0, buffer, 0, 0, &mousex, &mousey,
                  &nowhere, &nowhere, &key, &nowhere);

            switch(selection) {
            case MU_KEYBD:
                  switch((char)key) {
                  case '\r':
                        v_clsvwk(vdihandle);
                        appl_exit();
                        exit(0);

                  case ESC:
                        graf_mouse(M_OFF, &nowhere);
                        v_fillarea(vdihandle, n/2, xypoly);
                        graf_mouse(M_ON, &nowhere);
                        flag = 1;
                        break;
```



```
                  case 't':
                  case 'T':
                        if (flag == 0) {
                              break;
                        } else {
                              type = (++typeX5);
                              vsf_interior(vdihandle, type);
                              graf_mouse(M_OFF, &nowhere);
                              v_fillarea(vdihandle, n/2, xypoly);
                              graf_mouse(M_ON, &nowhere);
                        }
                        break;

                  case 's':
                  case 'S':
                        if (flag == 0) {
                              break;
                        } else {
                              style = ((styleX24)+1);
                              vsf_style(vdihandle, style);
                              graf_mouse(M_OFF, &nowhere);
                              v_fillarea(vdihandle, n/2, xypoly);
                              graf_mouse(M_ON, &nowhere);
                        }
                        break;
                  }
                  break;

            case MU_BUTTON:
                  if (flag > 0) {
                        n = 0;
                        flag = 0;
                  }
                  xymarker[0] = mousex;
                  xymarker[1] = mousey;
                  if (n <= 79) {
                        xypoly[n] = mousex;
                        n++;
                        xypoly[n] = mousey;
                        n++;
                  }
                  graf_mouse(M_OFF, &nowhere);
                  v_pmarker(vdihandle, ASTERISK, xymarker);
                  graf_mouse(M_ON, &nowhere);
                  break;
            default:
                  break;
            }
      }
}
```

See Also
TOS, v_bar, VDI, vr_recfl

v_form_adv — VDI function (libvdi)

Advance the page on a printer
#include <aesbind.h>
#include <vdibind.h>
void v_form_adv(*handle*) int *handle*;

v_form_adv is a VDI routine that advances the page on a printer. Unlike the related function **v_clrwk**, **v_form_adv** does not erase material that has not yet been written onto the printer.

See Also
TOS, v_clear_disp_list, v_clrwk, VDI

v_get_pixel — VDI function (libvdi)

See if a given pixel is set
#include <aesbind.h>
#include <vdibind.h>
void v_get_pixel(*handle, x, y, flag, color*)
int *handle, x, y, *flag, *color*;

v_get_pixel is a VDI routine that indicates whether a pixel is set. *handle* is the virtual device's VDI handle. *x* and *y* are, respectively, the X and Y coordinates of the pixel in question. *flag* is set by **v_get_pixel**; zero indicates that the pixel is not set, whereas one indicates that it is set. Finally, *color* indicates the color of the pixel, if it is set. For a table of color codes, see the entry for **v_opnwk**.

See Also
TOS, VDI

v_gtext — VDI function (libvdi)

Draw graphics text
#include <aesbind.h>
#include <vdibind.h>
void v_gtext(*handle, x, y, text*) int *text, x, y*; char *text;

v_gtext is a VDI routine that draws graphics text on the screen. *handle* is the virtual device's VDI handle. *x* and *y* are, respectively, the X and Y coordinates of the point on the screen where the drawing of the string will begin. Note that these values will change, depending upon the virtual device has been set to normalized device coordinates (NDC) or raster coordinates (RC). Finally, *text* points to the string to drawn.

The font of the string drawn, its size, its color, the angle at which it is displayed, and the manner of its alignment can all be set with separate VDI calls, as follows:

vst_effects	set special effects
vst_alignment	set text alignment
vst_rotation	set angle of rotation
vst_height	set text height

Example
The following example draws cross-hairs on the screen, and then aligns the string "Mark Williams C" against them. Pressing the 'E' key cycles through the available special effects; 'H', the available horizontal alignments; 'R', the text rotation; 'S', the available font sizes; and 'V', the vertical alignments. Typing **<return>** exits from the program

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

#define ESCAPE 0x1B                     /* ASCII code for <esc> */
#define RESERVED 0

/* global line A variables used by vdi; MUST be included */
int contrl[12],intin[128],ptsin[128],intout[128],ptsout[128];

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 1, 1 };

/* arrays used by drawline() */
xyvert[] = { 1, 1, 1, 1 };
xyhoriz[] = { 1, 1, 1, 1 };

/* string used by drawtext() */
char *text = "Mark Williams C";

/* arrays used by v_opnvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* throw-away declaration */
int nowhere = 0;

main()
{
        unsigned key;                   /* key typed by user */
        int vdihandle;                  /* virtual device's handle */
        int size = 1;                   /* text's size, in rasters */
        int effect = 1;                 /* text's special effect used */
        int halign = 0;                 /* text's horizontal alignment */
        int valign = 0;                 /* text's vertical alignment */
        int angle = 0;                  /* angle at which text is drawn */

/* OK, here we go ... */
        appl_init();
        graf_mouse(M_OFF, &nowhere);
        v_opnvwk(work_in, &vdihandle, work_out);
```

```
/* set clipping area to match screen dimensiosn */
cliparray[2] = work_out[0];
cliparray[3] = work_out[1];
vs_clip(vdihandle, 1, cliparray);
drawtext(vdihandle);

/* set drawing arrays to match screen dimensions */
xyvert[0] = work_out[0]/2;
xyvert[1] = 1;
xyvert[2] = work_out[0]/2;
xyvert[3] = work_out[1];

xyhoriz[0] = 1;
xyhoriz[1] = work_out[1]/2;
xyhoriz[2] = work_out[0];
xyhoriz[3] = work_out[1]/2;

for(;;) {
        key = evnt_keybd();
        switch((char)key) {

        case '\r':
                graf_mouse(M_ON, &nowhere);
                v_clsvwk(vdihandle);
                appl_exit();
                exit(0);

        case 'e':
        case 'E':
                vst_effects(vdihandle, effect);
                if (++effect > 32)
                        effect = 1;
                drawtext(vdihandle);
                break;

        case 'h':
        case 'H':
                halign++;
                vst_alignment(vdihandle, (halign%3), (valign%6),
                        &nowhere, &nowhere);
                        /* legal H value 0-2, V value 0-5 */
                drawtext(vdihandle);
                break;

        case 'r':
        case 'R':
                /* Note: ST draws text only at right angles */
                angle += 900;
                if (angle > 3500)
                        angle = 0;
                vst_rotation(vdihandle, angle);
                drawtext(vdihandle);
                break;
```



```
        case 's':
        case 'S':
                vst_height(vdihandle, size, &nowhere,
                        &nowhere, &nowhere, &nowhere);
                        /* character size in rasters */
                if (++size > 26)
                        size = 1;
                drawtext(vdihandle);
                break;

        case 'v':
        case 'V':
                valign++;
                vst_alignment(vdihandle, (halign%3), (valign%6),
                        &nowhere, &nowhere);
                        /* legal H value 0-2, V value 0-5 */
                drawtext(vdihandle);
                break;

        default:
                break;
        }
    }
}

drawlines(handle)
int handle;
{
        v_pline(handle, 2, xyvert);
        v_pline(handle, 2, xyhoriz);
        return;
}

drawtext(handle)
int handle;
{
        v_clrwk(handle);
        drawlines(handle);
        v_gtext(handle, work_out[0]/2, work_out[1]/2, text);
        return;
}
```

See Also

TOS, v_justified, VDI, vqt_extent, vqt_name, vqt_width, vst_alignment, vst_color, vst_effects, vst_height, vst_load_fonts, vst_point, vst_rotation, vst_unload_fonts

v_hardcopy — VDI function (libvdi)

Write the screen to a hard-copy device

#include <aesbind.h>
#include <vdibind.h>
void v_hardcopy(*handle*)

v_hardcopy is a VDI routine that writes a copy of the virtual device to a printer or other attached hard-copy device. *handle* is the virtual device's VDI handle.

See Also

TOS, VDI

Notes

The printer must be installed with TOS before this routine will work properly.

v_hide_c — VDI function (libvdi)

Hide the mouse pointer

#include <aesbind.h>
#include <vdibind.h>
void v_hide_c(*handle***) int** *handle*;

v_hide_c is a VDI routine that hides the mouse pointer. This routine should be invoked when your program redraws the screen; if the pointer is not hidden, it will leave a grayish patch on the screen when it is moved. To display the mouse pointer again, use **v_show_c**.

Example

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

See Also

TOS, v_show_c, VDI

Notes

Mixing VDI mouse calls with AES mouse calls can produce unpredictable results.

v_justified — VDI function (libvdi)

Justify graphics text

#include <aesbind.h>
#include <vdibind.h>
void v_justified(*handle, x, y, string, length, charsp, wordsp***)**
int *handle, x, y, length, charsp, wordsp*; **char** **string*;

v_justified is a VDI routine that justifies a string on a preset line length. *Justification* means that slivers of space are inserted between words or characters to ensure that each string fills exactly the same space. This paragraph is an example of justified text.

handle is, as always, the virtual device's VDI handle. *x* and *y* give, respectively, the X and Y coordinates of the point where the text is to begin printing. *length* is the length to which you want the text set; this value will vary, depending on whether the virtual device is set to normalized device coordinates (NDC) or to raster coordinates (RC). *string* points to the string you want to set. Finally, *charsp* and *wordsp* are flags that indicate whether you want spacing altered between words or characters when performing justification; zero turns off spacing, and one turns it on. Therefore, setting both *charsp* and *wordsp* to zero effectively turns off justification.

Note that if the string is too long to fit into *space*, the characters will overlap.

See Also

TOS, v_gtext, VDI

v_meta_extents — VDI function (libvdi)

Update extents header of metafile

#include <aesbind.h>
#include <vdibind.h>
void v_meta_extents(*handle, minx, miny, maxx, maxy***)**
int *handle, minx, miny, maxx, maxy*;

v_meta_extents is a VDI routine that updates the extents header of a metafile. The extents header gives the minimum space needed to draw all of the VDI primitives contained in the metafile; it is used by some routines in allocating space. *handle* is the virtual device's VDI handle. *minx* and *miny* give, respectively, the minimum width and height of the area needed to hold the VDI primitives contained within the metafile; whereas *maxx* and *maxy* give, respectively, the maximum width and height.

See Also

TOS, v_write_meta, VDI, vm_filename

Notes

This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

If this routine is not used when an item is added to a metafile, the extents parameters will be set to zero.

v_opnvwk — VDI function (libvdi)

Open the virtual screen device

#include <aesbind.h>
#include <vdibind.h>
void v_opnvwk(*work_in, handle, work_out***)**
int *work_in*[11], **handle, *work_out*[57];

v_opnvwk is a VDI routine that opens the virtual screen device. *work_in* is an array of 11 integers that must be set before invoking **v_opnvwk**. These are described in the entry for **v_opnwk**.

handle is the device handle for the screen. Because the GEM desktop has already opened the screen device, you must use the AES routine **graf_handle** to obtain the VDI handle for the screen, as follows:

```
handle = graf_handle(cw, ch, bw, bh);
```

In this example, *handle* is the VDI handle, which is returned by **graf_handle**. *cw* and *ch* point to integers that hold, respectively, the width and height of a character to be displayed, and *bw* and *bh* point to integers that set, respectively, the width and height of a character cell in the screen device.

work_out is an array of 57 integers that are set by **v_opnvwk**. Your program may need to interrogate this array for information; what each integer encodes is described in the entry for **v_opnwk**.

Example

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

See Also

TOS, v_opnwk, VDI

Notes

As of this writing, device attributes cannot be set through the *work_in* array. With the exception of *work_in[10]*, they are all ignored and should be set to one. To set device attributes, use the appropriate attribute functions, as listed in the entry for **VDI**.

v_opnwk — VDI function (libvdi)

Open a virtual workstation

#include <aesbind.h>
#include <vdibind.h>
void v_opnwk(*work_in, handle, work_out***)**
int *work_in*[11], **handle*, *work_out*[57];

v_opnwk is a VDI routine that opens a physical workstation. This routine should used to open all physical workstations *except* the screen, because the screen is opened by the GEM desktop when it boots. To open the screen, use **v_opnvwk**.

work_in is an array of 11 integers that must be set before **v_opnwk** is invoked. Their values are as follows:

work_in[0] Device number. as follows:

1	screen
11	plotter
21	printer
31	metafile
41	camera
51	tablet

work_in[1] Line type, as follows:

1	solid
2	long dashes
3	dots
4	dashes plus dots
5	short dashes
6	dash, dot, dot
7	user-defined
8-*n*	device-independent

work_in[2] Line color, as follows:

0	WHITE
1	BLACK
2	RED
3	GREEN
4	BLUE
5	CYAN
6	YELLOW
7	MAGENTA
8	WHITE
9	BLACK
10	LRED
11	LGREEN
12	LBLUE
13	LCYAN
14	LYELLOW
15	LMAGENTA
16-*n*	device-independent

Note that the names in capital letters are mnemonics that are defined in the header file **obdefs.h**.

work_in[3] Marker type, as follows:

1	dot
2	plus sign
3	asterisk
4	square
5	diagonal cross
6	diamond
7	device-independent

work_in[4] Marker color; same as above.

work_in[5] Text face. These can vary greatly, depending on the device being opened. For a list of the code and names of the the fonts available on a virtual device, use the function **vqt_font_info**.

work_in[6] Text color; same as above.

work_in[7] Fill type, as follows:

0	hollow
1	solid
2	patterned
3	cross-hatched
4	user-defined

work_in[8] Fill style. There are 24 styles of patterned fill, and 12 styles of cross-hatching. See the entry for **vsf_interior** for more information.

work_in[9] Fill color; same as above.

work_in[10] Coordinate system. Zero indicates normalized device coordinates (NDC). This is a system in which the screen is divided into a grid of 32,768 by 32,768 points, with the beginning point in the lower left-hand corner. Two indicates raster coordinates (RC). This uses the absolute number of rasters on the screen, counting from the upper left-hand corner of the screen. The number of rasters varies with screen resolution: high resolution is 640 wide by 400 high; medium resolution, 640 wide by 200 high; and low resolution, 320 wide by 200 high. GDOS is required to use NDC.

handle is the device's VDI handle, and is set by TOS. You can obtain the VDI handle with the AES function **graf_handle**. See the entry for **v_opnvwk** or **graf_handle** for more information.

work_out is an array of 57 integers that is filled in by **v_opnwk**, as follows:

0	width of device, in rasters (number of X coordinates)
1	height of device, in rasters (number of Y coordinates)
2	uses precision scaling? (0=yes, 1=no)
3	width of one pixel, microns
4	height of one pixel, microns
5	number of possible character heights (0=continuous scaling)
6	number of line types
7	number of possible line widths (0=continuous scaling)
8	number of marker types
9	number of possible marker sizes (0=continuous scaling)
10	number of text fonts available
11	number of fill styles available
12	number of cross-hatching styles available
13	number of colors that can be shown simultaneously
14	number of generalized drawing primitives (GDP's)
15-24	first 10 GDP's supported (-1=end of list): 1=rectangle, 2=curve, 3=circle segment, 4=circle, 5=ellipse, 6=elliptical arc, 7=elliptical segment, 8=rounded rectangle, 9=filled, rounded rectangle, 10=justified text
25-34	attribute of corresponding GDP from **work_out**[15]-[24] (-1=end of list): 0=line, 1=marker, 2=text, 3=area fill, 4=no attribute
35	color capability? (0=no, 1=yes)
36	text rotatable? (0=no, 1=yes)
37	can fill areas? (0=no, 1=yes)
38	supports cell arrays? (0=no, 1=yes)
39	number of colors supported: 0=more than 32,767; 1=monochrome; >2=number of colors
40	Cursor control devices: 1=keyboard only; 2=keyboard and mouse
41	number of mappable devices: 1=keyboard, 2=another device
42	number of choice devices: 1=function keys, 2=another key field
43	number of string devices: 1=keyboard
44	workstation type: 0=output only; 1=input only; 2=input/output; 3=reserved; 4=metafile
45	minimum character width
46	minimum character height
47	maximum character width
48	maximum character height
49	minimum visible line width
50	reserved (always zero)
51	maximum line width in X axis
52	reserved (always zero)
53	minimum marker width
54	minimum marker height
55	maximum marker width
56	maximum marker height

See Also

TOS, VDI, v_opnvwk

Notes

This routine is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI. To open the screen device, use the related function **v_opnvwk**.

As of this writing, a virtual device cannot have its attributes set through the *work_in* array. *work_in[0]* through *work_in[9]* should be set to one, and *work_in[10]* should be set to two. Any other settings will either be ignored or will cause system errors.

v_output_window — VDI function (libvdi)

Dump a portion of a virtual device to a printer

#include <aesbind.h>
#include <vdibind.h>
void v_output_window(*handle, xyarray***) int** *handle, xyarray***[4];**

v_output_window is a VDI routine that dumps a portion of a virtual device to the printer. *handle* is the virtual device's VDI handle. *xyarray* gives the two corners of the area to be dumped. On devices set to normalized device coordinates (NDC), *xyarray[0]* and *xyarray[1]* give, respectively, the X and Y coordinates of the lower left-hand corner, and *xyarray[2]* and *xyarray[3]* give the coordinates of the upper right-hand corner. On devices set to raster coordinates (RC), the first two array elements give the coordinates for the upper left-hand corner, and the latter two elements the coordinates of the lower right-hand corner.

See Also

TOS, VDI

Notes

The printer must be correctly described to TOS before this routine will work.

v_pieslice — VDI function (libvdi)

Draw a circular pie slice

#include <aesbind.h>
#include <vdibind.h>
void v_pieslice(*handle, x, y, radius, beginangle, endangle***)**
int *handle, x, y, radius, beginangle, endangle***;**

v_pieslice is a VDI routine that draws a circular pie slice. *handle* is the virtual device's VDI handle. *x* and *y* give, respectively, the X and Y coordinates for the imaginary circle of which the pie slice is a part. *radius* gives the imaginary circle's radius. Note that these measurements vary, depending on whether the device uses normalized device coordinates (NDC) or raster coordinates (RC). Finally, *beginangle* and *endangle* represent the beginning and end angles of the pie slice, given in tenths of a degree. Counting on an imaginary clock, zero degrees is at 3 o'clock; 90 degrees at noon; 180 degrees at 9 o'clock; and 270 degrees at 6 o'clock.

See Also

TOS, v_circle, VDI

v_pline — VDI function (libvdi)

Draw a line

#include <aesbind.h>
#include <vdibind.h>
void v_pline(*handle, howmany, xyarray***)**
int handle, howmany, xyarray[*n***];**

v_pline is a VDI routine that draws a line. For the VDI, a line is built out of one or more line segments, each end of which has its own pair of X and Y coordinates. Thus, it is possible to use **v_pline** to draw polygons or other figures on the screen.

handle is the virtual device's VDI handle. *howmany* is the number of end points being created. *xyarray* is an array of integers that holds the X and Y coordinates for the ends of the line segments; *n* is exactly double the value of *howmany*. Each even value in the array encodes an X coordinate, and each odd value a Y coordinate.

Example

The following example allows you draw lines on the screen while using the mouse. Click the left button to draw a line; holding down the left button lets you draw a continuous squiggle. Exit by typing any key.

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

/* global line A variables used by vdi; MUST be included */
int contrl[12],intin[128],ptsin[128],intout[128],ptsout[128];

/* array used by v_pline & vs_clip */
int xyarray[] = { 1, 1, 1, 1 };
int cliparray[] = { 1, 1, 1, 1 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* throw-away declaration */
int nowhere = 0;

main()
{
/* declarations used by evnt_multi() */
      int selection;                    /* code for event that occurred */
      unsigned int which = (MU_KEYBD | MU_BUTTON);
      int clicks = 1;                   /* no. of clicks expected */
      int button = 1;                   /* which button; 1=leftmost */
      int buttonstate = 1;              /* button state; 1=down */
      int buffer[11];                   /* place to write AES messages */

      int mousex;  /* mouse X coordinate */
      int mousey;  /* mouse Y coordinate */
      int vdihandle;

/* OK, here we go ... */
      appl_init();
      graf_mouse(ARROW, &nowhere);
      v_opnvwk(work_in, &vdihandle, work_out);
```

```
        cliparray[0] = 0;
        cliparray[1] = 1;
        cliparray[2] = work_out[0];
        cliparray[3] = work_out[1];
        vs_clip(vdihandle, 1, cliparray);

        vsl_width(vdihandle, 5);
        vsl_type(vdihandle, 1);
        vsl_ends(vdihandle, 1, 1);

        for(;;) {
                selection = evnt_multi(which, clicks, button,
                        buttonstate, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                        buffer, 0, 0, &mousex, &mousey,
                        &nowhere, &nowhere, &nowhere, &nowhere);

                switch(selection) {
                case MU_KEYBD:
                        v_clsvwk(vdihandle);
                        appl_exit();
                        exit(0);

                case MU_BUTTON:
                        xyarray[0] = xyarray[2];
                        xyarray[1] = xyarray[3];
                        xyarray[2] = mousex;
                        xyarray[3] = mousey;
                        graf_mouse(M_OFF, &nowhere);
                        v_pline(vdihandle, 2, xyarray);
                        graf_mouse(M_ON, &nowhere);
                        break;

                default:
                        break;
                }
        }
}
```

See Also

TOS, VDI, vql_attributes, vsl_color, vsl_ends, vsl_type, vsl_udsty, vsl_width

v_pmarker — VDI function (libvdi)

Draw a marker

#include <aesbind.h>
#include <vdibind.h>
void v_pmarker(*handle, count, array***) int** *handle, count, array*[*n*];

v_pmarker is a VDI routine that draws one or more markers on a virtual device. *handle* is the virtual device's VDI handle. *count* is the number of markers you want to draw. *array* is an array of X and Y coordinates that locate each marker on the screen; *n*, therefore, must be exactly double the size of *count*. Every even number in this array indicates an X coordinate, and every odd number a Y coordinate. Note that the values for each coordinate will differ, depending on whether the device is set to normalized device coordinates (NDC) or raster coordinates (RC).

Example

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

See Also

TOS, VDI, vqm_attributes, vsm_color, vsm_height, vsm_type

v_rbox — VDI function (libvdi)

Draw a rounded rectangle

#include <aesbind.h>
#include <vdibind.h>
void v_rbox(*handle, xyarray***) int** *handle, xyarray*[4];

v_rbox is a VDI routine that draws a rectangle with rounded corners. *handle* is, as always, the virtual device's VDI handle. *xyarray* gives the X and Y coordinates of the two corners that define the rectangle; the even entries in the array give the X coordinates, and the odd entries the Y coordinates. Note that these values will change, depending on whether the virtual device is defined as using normalized device coordinates (NDC) or raster coordinates (RC). On an NDC device, *xyarray[0]* and *xyarray[1]* encode the lower left-hand corner, where on an RC device they encode the upper left-hand corner; likewise, on an NDC device *xyarray[2]* and *xyarray[3]* represent the upper right-hand corner, whereas on an RC device they represent the lower right-hand corner.

Example

The following example lets you use the mouse to draw rectangles on the screen.

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

/* global line A variables used by vdi; MUST be included */
int contrl[12],intin[128],ptsin[128],intout[128],ptsout[128];

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 1, 1 };

/* array used by v_rbox() */
int xyarray[] = { 1, 1, 1, 1 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* throw-away declarations */
int nowhere = 0;
```

```
main()
{
/* declarations used by evnt_multi() */
      int selection;       /* code for event */
      unsigned int which = (MU_KEYBD | MU_BUTTON);
      int clicks = 1;      /* no. of clicks expected */
      int button = 1;      /* which button; 1=leftmost */
      int buttonstate = 1;/* button state expected; 1=down */
      int buffer[11];      /* place to write AES messages */
      int mousex;   /* mouse X coordinate */
      int mousey;   /* mouse Y coordinate */
      unsigned key; /* key typed by user */

/* misc declarations */
      int vdihandle;       /* virtual device's handle */
      int width;    /* width of rubberbox user draws */
      int depth;    /* depth of rubberbox user draws */

      /* open application */
      appl_init();

      /* turn mouse pointer to arrow */
      graf_mouse(ARROW, &nowhere);

      /* open screen device */
      v_opnvwk(work_in, &vdihandle, work_out);

      /* set clipping rectangle */
      cliparray[2] = work_out[0];
      cliparray[3] = work_out[1];
      vs_clip(vdihandle, 1, cliparray);

      /* set perimeter for drawn rectangle */
      vsf_perimeter(vdihandle, 1);

      for(;;) {
            /* wait for something to happen */
            selection = evnt_multi(which, clicks, button,
                  buttonstate, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                  buffer, 0, 0, &mousex, &mousey, &nowhere,
                  &nowhere, &key, &nowhere);
            switch(selection) {

            /* if keyboard is pressed, exit */
            case MU_KEYBD:
                  v_clsvwk(vdihandle);
                  appl_exit();
                  exit(0);

            /* if button is pressed, draw a rectangle */
            case MU_BUTTON:
                  graf_rubbox(mousex, mousey, 3, 3, &width,
                        &depth);
                  xyarray[0] = mousex;
                  xyarray[1] = mousey;
                  xyarray[2] = (mousex+width);
                  xyarray[3] = (mousey+depth);
                  graf_mouse(M_OFF, &nowhere);
                  v_rfbox(vdihandle, xyarray);
                  graf_mouse(M_ON, &nowhere);
                  break;

            default:
                  break;
            }
      }
}
```

See Also

TOS, VDI, v_rfbox

v_rfbox — VDI function (libvdi)

Draw a filled, rounded rectangle

#include <aesbind.h>
#include <vdibind.h>
void v_rfbox(*handle, xyarray***) int** *handle, xyarray***[4];**

v_rfbox is a VDI routine that draws a rectangle with rounded corners. It uses the functions **vsf_interior** and **vsf_style**, which set, respectively, the type and style of the interior fill.

handle is, as always, the virtual device's VDI handle. *xyarray* gives the X and Y coordinates of the two corners that define the rectangle; the even entries in the array give the X coordinates, and the odd entries the Y coordinates. Note that these values will change, depending on whether the virtual device is defined as using normalized device coordinates (NDC) or raster coordinates (RC). On an NDC device, *xyarray[0]* and *xyarray[1]* encode the lower left-hand corner, where on an RC device they encode the upper left-hand corner; likewise, on an NDC device *xyarray[2]* and *xyarray[3]* represent the upper right-hand corner, whereas on an RC device they represent the lower right-hand corner.

See Also

TOS, VDI, v_rbox

v_rmcur — VDI function (libvdi)

Remove last mouse pointer from the screen

#include <aesbind.h>
#include <vdibind.h>
void v_rmcur(*handle***) int** *handle***;**

v_rmcur is a VDI routine that removes the last mouse pointer from the screen. *handle* is the virtual device's VDI handle.

Note that this routine removes only the *last* mouse pointer to have been invoked. If the mouse pointer has been invoked several times, this routine must be called as many times before the mouse pointer finally disappears.

See Also
TOS, VDI

v_rvoff — VDI function (libvdi)

End reverse video for alphabetic text
#include <aesbind.h>
#include <vdibind.h>
void v_rvoff(*handle*) int *handle*;

v_rvoff is a VDI routine that turns off reverse-video display for all alphabetic text written subsequently. *handle* is the virtual device's VDI handle.

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

See Also
TOS, v_rvon, VDI

v_rvon — VDI function (libvdi)

Display alphabetic text in reverse video
#include <aesbind.h>
#include <vdibind.h>
void v_rvon(*handle*) int *handle*;

v_rvon is a VDI routine that causes all subsequent alphabetic text to appear in reverse video. *handle* is the virtual device's VDI handle.

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

See Also
TOS, v_rvoff, VDI

v_show_c — VDI function (libvdi)

Show the mouse cursor
#include <aesbind.h>
#include <vdibind.h>
void v_show_c(*handle, ignore*) int *handle, ignore*;

v_show_c is a VDI routine that reshows the mouse cursor after it has been hidden. Due to a peculiarity in the VDI, this or a similar routine must be invoked the

same number of times that the mouse pointer has been hidden. For example, if the mouse pointer was hidden three times in a row without being redisplayed, it must be recalled three times with **v_show_c** before it will reappear.

handle is the virtual device's VDI handle. *ignore* is a flag that sets the VDI's hide-mouse counting feature: zero indicates that the number of times the mouse pointer was hidden should be ignored, whereas one means that it should be honored.

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

See Also
TOS, v_hide_c, VDI

Notes
Mixing VDI mouse calls with AES mouse calls can produce unpredictable results.

v_updwk — VDI function (libvdi)

Update a virtual workstation
#include <aesbind.h>
#include <vdibind.h>
void v_updwk(*handle*) int *handle*;

v_updwk is a VDI routine that updates a virtual workstation. *handle* is the virtual device's VDI handle.

This routine is used with virtual devices that have buffered output, e.g., printers and plotters, and so is analogous to the STDIO function **fflush**. Note that this function merely executes the commands in buffer, but does not clear the workstation. To clear the workstation, use the function **v_clrwk**.

See Also
TOS, v_clrwk, VDI

Notes
This routine is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

v_write_meta — VDI function (libvdi)

Write a metafile item
#include <aesbind.h>
#include <vdibind.h>
void v_write_meta(*handle, numintin, intin, numptsin, ptsin*)
int *handle, numintin, intin*[*numintin*], *numptsin, ptsin*[*numptsin*];

v_write_meta is a VDI routine that writes a VDI item into a metafile. The item is assigned an opcode by the VDI.

handle is the virtual device's VDI handle. The *intin* and *ptsin* arrays hold the in-

formation needed to build the metafile item. The first entry must hold an opcode that describes the type of item being built. The next 100 entries are reserved by the system. The sub-opcodes used to build the item are entries 101 and higher.

See Also

metafile, TOS, v_meta_extent, VDI, vm_filename

Notes

This routine is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

VDI — Technical information

VDI stands for *virtual device interface*. The VDI is designed to provide the user with graphics routines that can be transported without alteration to a variety of devices: screen, printer, plotter, video camera, graphics tablet, and "metafile". The VDI can perform the following tasks on these devices:

- Draw lines, polygons, circles, curves, and other graphics primitives.
- Fill or flood areas with a preset pattern or cross-hatching.
- Copy ("bitblit") areas of the virtual device or graphics images.
- Load type fonts, and size, justify, and rotate graphics text.
- Write and read "metafiles", or a file of an image that can be incorporated into various other VDI programs (for example, a company logo).
- Return information about virtual devices and drivers.
- Await user events and interrogate the system about them.

Devices and virtual devices

The VDI has drivers that support a number of physical graphics devices, such as printers, plotters, video cameras, and the screen.

The screen device can, in addition, have one or more "virtual devices" output to it. A virtual device is a logical description of the screen that is handed to the screen driver for display.

Every time this virtual device is changed, the device driver updates the screen to reflect this change. More than one virtual device may be created for the screen. This allows you to create a series of "transparencies" that can be manipulated independent of each other and overlaid.

Each virtual device can be described using either normalized device coordinates (NDC) or raster coordinates (RC). The NDC system divides a virtual device into a grid that has 32,767 points on each side. The size of points on the X scale differ from those on the Y scale, to ensure that objects such as circles are drawn in correct proportion. The RC system uses the number of rasters on the physical screen

to scale its objects. The number of rasters will vary, depending on the resolution to which the screen is set, as follows: high resolution, 640 horizontal and 400 vertical; medium resolution, 640 horizontal and 200 vertical; and low resolution, 320 horizontal and 200 vertical.

The NDC and RC systems also differ in where they place the 0,0 point on their X/Y scales. In the NDC system, the 0,0 point is in the lower left-hand corner, whereas in the RC system, 0,0 is in the upper left-hand corner. All objects drawn on the virtual device will be oriented in the same way; for example, a rectangle drawn on an RC virtual device will be measured from the upper left-hand corner to the lower right-hand corner, whereas one on an NDC device will be measured from the lower left-hand corner to the upper right-hand corner. In general, RC are easier for the programmer to visualize and handle, but NDC are more portable. The NDC system can only be used with the VDI GDOS.

VDI components

The VDI is divided into three parts: a library of fonts, a library of device drivers, and GDOS.

The *fonts* display alphanumeric characters in various sizes, weights, and styles. The *device drivers* turn generalized VDI statements into bits that can be understood by particular physical devices. Finally, the most important element is the graphic device operating system (GDOS). The GDOS, as its name implies, coordinates the loading of fonts and drivers.

The GDOS also controls the writing and use of metafiles. These files are extraordinarily useful; for more information, see the entry for **metafile**.

Programming with the VDI

The VDI is "virtual" because it works not directly with physical devices, but with the logical description of a device, or a virtual device. When this logical description is altered, the VDI can either update the phyical device directly or record the changes in a metafile.

To work with the VDI, a program must first *open* a virtual device with one of the functions **v_opnwk** or **v_opnvwk**. To use these functions, you must hand them an array of 11 integers that set various aspects of the graphics environment: for example, the color and thickness of the lines to be drawn, the color of the text, and the type and style of pattern fill for polygons. These routines assign a *handle* to the virtual device, and return an array of 57 integers that give information about the newly opened virtual device.

The VDI uses five global arrays of integers: **intin[]**, **intout[]**, **ptsin[]**, **ptsout[]**, and **contrl[]**. The last of these should be declared as having 12 members; the others should each be declared as having 128. These arrays are manipulated directly by assembly-language programs; they are not used directly within C programs, but must be declared for the VDI routines to work.

Within the program, you can use routines to draw graphics primitives, receive and

process information from the user, modify the virtual device's default settings, and perform many other types of useful tasks.

When finished, the functions **v_clswk** or **v_clsvwk** should be invoked to free the memory used by the virtual device, and otherwise tidy up after the program.

Note that all programs that use the graphics interface must run under the AES; this means that all programs that use the VDI must begin with **appl_init** and close with **appl_exit**.

VDI library routines

The VDI library routines are declared in the header file **vdibind.h**, and are stored in the library **libvdi**. These routines are, in turn, built out of the Atari Line A routines, which form graphic "primitives". The following lists the VDI routines and gives a brief description of each. For more information about a particular routine, see its entry in the Lexicon.

v_arc	draw a circular arc
v_bar	draw an outlined, filled rectangle
v_bit_image	print a bit-image file
v_cellarray	create an array of colored cells
v_circle	draw a circle
v_clear_disp_list	clear a printer's display list
v_clrwk	clear a virtual device
v_clsvwk	close the screen device
v_clswk	close a virtual device
v_contourfill	draw a filled polygon
v_curdown	move the text cursor down one row
v_curhome	move the text cursor to upper left corner
v_curleft	move the text cursor left one column
v_curright	move the text cursor right one column
v_curtext	write a string of text characters
v_curup	move the text cursor up one row
v_dspcur	move the mouse pointer to specified location
v_eeol	erase from text cursor to end of line
v_eeos	erase from text cursor to end of screen
v_ellarc	draw an elliptical arc
v_ellipse	draw an ellipse
v_ellpie	draw an elliptical pie segment
v_enter_cur	enter text mode
v_exit_cur	exit from text mode
v_fillarea	flood an enclosed area with fill pattern
v_form_adv	advance the page on a hard-copy device
v_get_pixel	find if a particular pixel has been set
v_gtext	output graphics text
v_hardcopy	dump virtual device to hard-copy device
v_hide_c	hide the mouse pointer

v_justified	output justified graphics text
v_meta_extents	update extents header of metafile
v_opnvwk	open the screen virtual device
v_opnwk	open a virtual device
v_output_window	print a portion of a virtual device
v_pieslice	draw a circular pie segment
v_pline	draw a polyline
v_pmarker	draw a polymarker
v_rbox	draw a rectangle with rounded corners
v_rfbox	draw rectangular fill area with rounded corners
v_rmcur	remove last graphics cursor from screen
v_rvoff	turn off reverse video for character text
v_rvon	turn on reverse video for character text
v_show_c	show mouse pointer
v_updwk	update workstation (flush buffers)
v_write_meta	add an item to a metafile
vex_butv	change button interrupt routine
vex_curv	change cursor movement interrupt routine
vex_motv	change mouse pointer interrupt routine
vex_timv	change timer interrupt routine
vm_filename	rename a metafile
vq_cellarray	query cell array information
vq_chcells	query no. of characters printable on device
vq_color	query/set mix for a color
vq_curaddress	query text cursor's current position
vq_extnd	perform extended inquiry
vq_key_s	query keyboard status
vq_mouse	query mouse position and button state
vq_tabstatus	query if graphics tablet is available
vqf_attributes	set fill area attributes
vqin_mode	set inquiry mode
vql_attributes	query polyline attributes
vqm_attributes	query polymarker attributes
vqp_error	query message from Polaroid Palette
vqp_films	films supported on Polaroid Palette
vqp_state	read status of Polaroid Palette driver
vqt_attributes	query graphics text attributes
vqt_extent	query length of a string
vqt_font_info	query information about fonts
vqt_name	query name and description of font
vqt_width	query width of a character's cell
vr_recfl	draw a rectangular fill area
vr_trnfm	transform bit image format
vro_cpyfm	copy (blit) a portion of a device
vrq_choice	query choice devices, request mode
vrq_locator	query locator devices, request mode

vrq_string	query string devices, request mode
vrq_valuator	query valuator devices, request mode
vrt_cpyfm	copy (blit) a monochromatic image
vs_clip	clip an area of the virtual device
vs_color	set mix for a color
vs_curaddress	move text cursor to specified point
vs_palette	set the palette for medium resolution
vsc_form	set new mouse pointer shape
vsf_color	set fill color
vsf_interior	set fill type
vsf_perimeter	set drawing of perimeter
vsf_style	set fill style
vsf_udpat	set user-defined fill pattern
vsin_mode	set mode of logical device inquiry
vsl_color	set polyline color
vsl_ends	set polyline end types
vsl_type	set polyline's pattern
vsl_udsty	set user-defined polyline style
vsl_width	set polyline width
vsm_choice	query choice devices, sample mode
vsm_color	set polymarker color
vsm_height	set polymarker height
vsm_locator	query locator devices, sample mode
vsm_string	query string devices, sample mode
vsm_type	set polymarker type
vsm_valuator	query valuator devices, sample mode
vsp_message	suppress Polaroid Palette messages
vsp_save	save settings of driver for Polaroid Palette
vsp_state	set Polaroid Palette driver
vst_alignment	set graphics text alignmment
vst_color	set graphics text color
vst_effects	set graphics text special effects
vst_font	set graphics text font
vst_height	set graphics text height, in pixels
vst_load_fonts	load non-standard fonts
vst_point	set graphics text height, in points
vst_rotation	set angle of graphics text
vst_unload_fonts	unload non-standard fonts
vswr_mode	set writing mode

A sample VDI program

The following program is a game that demonstrates how to use a number of VDI routines. The program draws a small black rectangle, which chases the mouse pointer. If the mouse pointer is caught, the program exults briefly, then asks you if you want to play again.


```
#include <aesbind.h>
#include <gemdefs.h>
#include <osbind.h>
#include <vdibind.h>

#define BUTTON 1                    /* which button; 1 = leftmost */
#define CENTER 1                    /* indicates centering of text */
#define CENTERX 320                 /* center of screen, X coord */
#define CENTERY 200                 /* center of screen, Y coord */
#define CLICKS 1                    /* no. of clicks expected */
#define DOWN 1                      /* mouse button is down */
#define HITIME 0                    /* high word in timer values */
#define LEFT 0                      /* set text flush left */
#define LOTIME 5                    /* low word of timer; 5 ms. */
#define XOR 6                       /* XOR mode */

/* global line A variables used by vdi; MUST be included */
int contrl[12],intin[128],ptsin[128],intout[128],ptsout[128];

/*
 * array used by vs_clip().  Clipping array MUST be set;
 * otherwise, low memory will be written over by graphics
 * forms that extend beyond the edge of the screen.
 */
int cliparray[] = { 1, 1, 639, 399 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* arrays used by vro_cpyfm() to blit cat around screen */
/* solid block rectangle */
int shape[] = {
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
        0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF
};

/* source form block */
FDB cat = { shape, 16, 16, 1, 0, 1, 0, 0, 0 };

/* target form block */
FDB screen = { 0L, 0, 0, 0, 0, 0, 0, 0, 0 };

/* initial position on screen */
int oldarray[] = {
        0, 0, 16, 16, CENTERX, CENTERY, CENTERX+16, CENTERY+16 };

/* new position on screen */
int newarray[] = {
        0, 0, 16, 16, CENTERX, CENTERY, CENTERX+16, CENTERY+16 };

/* throw-away declaration */
int nowhere = 0;
```

```
main()
{
        int mousex = 1;                 /* mouse X coordinate */
        int mousey = 1;                 /* mouse Y coordinate */
        int vdihandle;                  /* virtual device's handle */

/* OK, here we go ... */
        /* open AES process */
        appl_init();
        graf_mouse(BUSY_BEE, &nowhere);
        /* open virtual device */
        vdihandle = graf_handle(&nowhere, &nowhere,
                &nowhere, &nowhere);
        v_opnvwk(work_in, &vdihandle, work_out);
        /* set clipping rectangle */
        vs_clip(vdihandle, 1, cliparray);

        /* blit cat initially */
        vro_cpyfm(vdihandle, XOR, oldarray, &cat, &screen);

        for(;;) {
                /* get mouse pointer's position */
                vq_mouse(vdihandle, &nowhere, &mousex, &mousey);

                /* check cat's position vs. that of mouse */
                if (oldarray[4] < mousex)
                        newarray[4] += 6;
                else if (oldarray[4] > mousex)
                        newarray[4] -= 6;
                if (oldarray[5] < mousey)
                        newarray[5] += 6;
                else if (oldarray[5] > mousey)
                        newarray[5] -= 6;

                /* set cat's kitty corner */
                newarray[6] = newarray[4] + 16;
                newarray[7] = newarray[5] + 16;

                /* synchronize with screen */
                Vsync();

                /* blit cat */
                vro_cpyfm(vdihandle, XOR, newarray, &cat, &screen);
                vro_cpyfm(vdihandle, XOR, oldarray, &cat, &screen);

                /* shuffle cat's new array into old array */
                oldarray[4] = newarray[4];
                oldarray[5] = newarray[5];
                oldarray[6] = newarray[6];
                oldarray[7] = newarray[7];
```



```
                /* check if cat has caught mouse */
                if ((abs(oldarray[4] - mousex) <= 16) &&
                        (abs(oldarray[5] - mousey) <= 16)) {
                        gotcha(vdihandle);
                        playagain(vdihandle);
                }
        }
}

gotcha(vdihandle)
int vdihandle;
{
        char *text = "GOTCHA!";

        /* set text attributes */
        vst_effects(vdihandle, 32);
        vst_alignment(vdihandle, CENTER, 0, &nowhere,
                &nowhere);
        vst_height(vdihandle, 26, &nowhere, &nowhere,
                &nowhere, &nowhere);

        /* ring the bell and write "GOTCHA!" in big letters */
        write(stdout, "\07", 1);
        v_gtext(vdihandle, 320, 200, text);
        evnt_timer(1500, HITIME);
        return;
}

playagain(vdihandle)
int vdihandle;
{
        char *string = "[2][Play again?][Yes|No]";
        int button;

        /* reset text attributes */
        vst_effects(vdihandle, 1);
        vst_alignment(vdihandle, LEFT, 0, &nowhere,
                &nowhere);
        vst_height(vdihandle, 13, &nowhere, &nowhere,
                &nowhere, &nowhere);

        /* draw alert box */
        button = form_alert(1, string);
```

```
        /* do what user requests */
        if (button == 1) {
        /* i.e., if user wants another game ... */
              /* ... clear screen again ... */
              v_clrwk(vdihandle);
              /* ... move cat to center of screen ... */
              newarray[4] = oldarray[4] = CENTERX;
              newarray[5] = oldarray[5] = CENTERY;
              newarray[6] = oldarray[6] = (CENTERX + 16);
              newarray[7] = oldarray[7] = (CENTERY + 16);
              /* ... and redraw cat */
              vro_cpyfm(vdihandle, XOR, oldarray, &cat,
                    &screen);

              /* return pointer shape to bee */
              graf_mouse(BUSY_BEE, &nowhere);
              /* wait a few moments so user can get away */
              evnt_timer(1000, HITIME);
              return;
        } else {
        /* i.e., user wants to quit */
              /* close virtual station, close application, exit */
              v_clsvwk(vdihandle);
              appl_exit();
              exit(1);
        }
}
```

See Also

AES, libvdi, Line A, metafile, TOS, vdibind.h

Notes

A RAM version of GDOS is now available, and it is shipped with several different products. At present, the standard VDI supports drivers only for the screen device, and acknowledges only raster coordinates.

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

vdibind.h — Header file

Declarations for VDI routines
#include <vdibind.h>

vdibind.h is the header file that holds declarations and definitions for the GEM VDI routines, which are contained in the library **libvdi**.

See Also

aesbind.h, header file, TOS, VDI

version — Command

Print/create a version string
version *file ...*
version *directory executable sourcefile ...*

version finds or creates a version string. When given an executable *file* as an argument, **version** scans it for the version string, which it prints on the standard output device.

version can also generate a version number automatically. When given the name of a *directory* that holds source code, and the names of the *executable* (whether or not it has been created yet) and *sourcefile* (or *sourcefiles*) **version** writes a brief program in C that, when compiled and linked, generates a version number for the program and writes it into the executable file.

Example

To generate a version number for an executable called **color.prg** that is compiled from the source file called **color.c**, which is found in directory **examples** on drive B, type the following command:

```
version b:\examples color.prg color.c >version.c
```

It does not matter what you name the file into which you direct the output of **version**; however, be sure that it has the suffix **.c**, so that the compiler will know that it is a C-source file. Also, be sure to include this file on the **cc** command line when you compile the program.

See Also

commands, msh

vertical tab — Character constant

Mark Williams C recognizes the literal character '\v' for the ASCII vertical tab character VT (octal 013). This character may be used as a character constant or in a string constant. The vertical tab character is white space; in particular, **isspace** returns "true" for '\v'.

See Also

ASCII, character constant

vex_butv — VDI function (libvdi)

Set new button interrupt routine
#include <aesbind.h>
#include <vdibind.h>
void vex_butv(*handle, address, oldaddress***)**
int *handle*; **void (***address***); void (***oldaddress***);**

vex_butv is a VDI routine that lets you set a new button interrupt routine. *handle* is the virtual device's VDI handle. *address* is the address of the new interrupt routine. Your routine is responsible for saving registers and resetting registers. Finally, *oldaddress*, which is set by **vex_butv**, is the address of the old interrupt routine.

See Also

TOS, VDI, vex_curv, vex_motv, vex_timv

Notes

If the button interrupt routine is executed in assembly language, note the following:

Invoke the application-dependent code with a **JSR** instruction. The register d0.w contains the mouse button keys. When complete, use the instruction **RTS**, with the mouse button state stored in d0.w.

vex_curv — VDI function (libvdi)

Set new cursor interrupt routine
#include <aesbind.h>
#include <vdibind.h>
void vex_curv(*handle, address, oldaddress*)
int *handle*; **void** (**address*); **void** (**oldaddress*);

vex_curv is a VDI routine that lets you set a new cursor movement interrupt routine. *handle* is the virtual device's VDI handle. *address* points to the new interrupt routine. Note that your new routine is responsible for saving and restoring registers. Finally, *oldaddress*, which is set by **vex_curv**, points to the old interrupt routine.

See Also

TOS, VDI, vex_butv, vex_motv, vex_timv

Notes

If the cursor routine is executed in assembly language, note the following:

Invoke the application-dependent code with a **JSR** instruction. Upon entry to the routine, the registers d0.w and d1.w registers contain, respectively, the X and Y positions of the cursor. If the application-dependent code does not draw its own cursor, a **JSR** instruction should be performed to the address returned in **contrl[9]** and **contrl[10]** (which are initialized by the functions **v_opnwk** and **v_opnvwk**, with d0.w and d1.w holding, respectively, the X and Y positions at which to draw the cursor. This causes VDI to draw a cursor. When complete, perform an **RTS** instruction.

vex_motv — VDI function (libvdi)

Set new mouse movement interrupt routine
#include <aesbind.h>
#include <vdibind.h>
void vex_motv(*handle, address, oldaddress*)
int *handle*; **void** (**address*); **void** (**oldaddress*);

vex_motv is a VDI routine that sets a new interrupt routine to be invoked when the mouse pointer moves. *handle* is the virtual device's VDI handle. *address* gives the address of the new interrupt routine. Note that your routine is responsible for saving and restoring registers. *oldaddress* is set by **vex_motv**; it holds the address of the old interrupt routine.

See Also

TOS, VDI, vex_butv, vex_curv, vex_timv

Notes

If the mouse-movement routine is executed in assembly language, note the following:

Invoke the application-dependent code with a **JSR** instruction. On entry, the registers d0.w and d1.w contain, respectively, the X and Y positions of the mouse. When complete, execute a **JSR** instruction, with d0.w and d1.w holding, respectively, the X and Y positions of the mouse.

vex_timv — VDI function (libvdi)

Set new timer interrupt routine
#include <aesbind.h>
#include <vdibind.h>
void vex_timv(*handle, address, oldaddress, time*)
int *handle*, **time*; **void** (**address*); **void** (**oldaddress*);

vex_timv is a VDI routine that lets you set a new timer interrupt routine. *handle* is the virtual device's VDI handle. *address* points to the address of the new interrupt routine. *oldaddress* is set by **vex_timv** upon exiting, and contains the old interrupt address. Finally, *time* is set by **vex_timv** upon exiting, and contains the interval of the interrupt call, in milliseconds. Note that your new interrupt routine is responsible for saving registers and returning to the system. The interrupt is reactivated by setting the old interrupt address.

See Also

TOS, VDI, vex_butv, vex_curv, vex_motv

Notes

This routine is called **vex_time** in some bindings.

If the timer interrupt routine is written in assembly language, invoke the application-dependent code via a **JSR** routine. When finished, execute an **RTS** instruction.

vm_filename — VDI function (libvdi)

Rename a metafile
#include <aesbind.h>
#include <vdibind.h>
void vm_filename(*handle, filename*) **int** *handle*; **char** **filename*;

vm_filename is a VDI routine that renames a metafile. *handle* is the virtual device's VDI handle. *filename* points to the new file name; this must be an al-

phabetic string that is terminated with NUL.

See Also

TOS, v_meta_extent, v_write_meta, VDI

Notes

This routine is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

void — C keyword

Data type

In addition to the data types described in *The C Programming Language*, Mark Williams C also recognizes the data type **void**. **void** applies *only* to a function declaration, and indicates that the function does not return a value.

Using **void** declarations makes programs clearer and is useful in error checking. For example, a function that prints an error message and calls **exit** to terminate a program should be declared **void** because it never returns. A function that performs a calculation and stores its result in a global variable (rather than returning the result), or one that returns no value, should also be declared **void** to prevent the accidental use of the function in an expression. For example,

```
void cursor_pos(x,y)
int x,y;
{
        printf("\33Y%c%c", x+' ', y+' ');
}
```

could be used to write the current position of the cursor in a screen handling program.

See Also

C keywords, C language, declarations

volatile — C keyword

Qualify an identifier as frequently changing

The type qualifier **volatile** marks an identifier as being frequently changed, either by other portions of the program, by the hardware, by other programs in the execution environment, or by any combination of these. This alerts the translator to refetch the given identifier whenever it encounters an expression that includes the identifier. In addition, an object marked as **volatile** must be stored at the point where an assignment to this object takes place.

See Also

C keyword, const

Notes

Although Mark Williams C recognizes this keyword, the semantics are not implemented in this release. Thus, storage declared to be **volatile** might have references removed by optimizations that the compiler performs. The compiler will generate a warning if the peephole optimizer is enabled and the keyword **volatile** is detected.

vq_cellarray — VDI function (libvdi)

Return information about cell arrays

#include <aesbind.h>
#include <vdibind.h>
void vq_cellarray(*handle, xyarray, rowlength, rows,*
cellused, rowused, status, cellarray)
int *handle, xyarray, rowlength*[**4**], *rows, *cellused, *rowsused, *status, cellarray*[*n*];

vq_cellarray is a VDI routine that returns information about an established cell array.

handle is the virtual device's VDI handle. *xyarray* gives the X and Y coordinates for the rectangle in which the array is drawn. These values will vary, depending on whether the device is set to normalized device coordinates (NDC) or raster coordinates (RC). On NDC devices, *xyarray[0]* and *xyarray[1]* give, respectively, the X and Y coordinates of the lower left-hand corner of the rectangle, whereas *xyarray[2]* and *xyarray[3]* give the coordinates for the upper right-handle corner. On RC devices, *xyarray[0]* and *xyarray[1]* give, respectively, the X and Y coordinates of the upper left-hand corner, whereas *xyarray[2]* and *xyarray[3]* give the X and Y coordinates of the lower right-hand corner. *rowlength* gives the horizontal length of the table to be shown, in NDCs or RCs, and *rows* is the number of rows of cells in the array.

cellused points to an integer that holds the number of horizontal cells used in each row. *rowused* points to the number of rows in the array that were actually used. *status* points to the array's error status: zero indicates that no errors occurred, whereas a number greater than one indicates that a color value could not be found for a given cell.

Finally, *cellarray* gives the array of colors actually displayed in the used cells. *n* must be equal to the number of cells in the entire array. The color index will be set to -1 if the color requested for a given cell could not be found.

See Also

TOS, v_cellarray, VDI

Notes

This routine is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

vq_chcells — VDI function (libvdi)

Find how many characters virtual device can print
#include <aesbind.h>
#include <vdibind.h>
void vq_chcells(*handle, rows, columns***) int** *handle, *rows, *columns*;

vq_chcells is a VDI routine that examines a virtual device and returns the number of rows and columns of characters that can be printed on it. *handle* is the virtual device's VDI handle. *rows* and *columns* point, respectively, to the number of rows and the number of columns of characters that can be printed on the virtual device.

Example

The following example returns the number of rows and columns available on the screen device.

```
#include <aesbind.h>
#include <vdibind.h>

/* global line A variables used by vdi; MUST be included */
int contrl[12],intin[128],ptsin[128],intout[128],ptsout[128];

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* print an alert box on the screen */
alertf(n, p) int n; char *p;
{
        static char buffer[512];
        sprintf(buffer, "%r", &p);
        return form_alert(n, buffer);
}

main()
{
        int nowhere = 0;
        int vdihandle;
        int rows;
        int columns;

        appl_init();
        v_opnvwk(work_in, &vdihandle, work_out);

        vq_chcells(vdihandle, &rows, &columns);
        alertf(1, "[1][Rows: %d|Columns: %d][OK]", rows, columns);

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


See Also

TOS, VDI

vq_color — VDI function (libvdi)

Check/set color intensity
#include <aesbind.h>
#include <vdibind.h>
void vq_color(*handle, color, flag, rgb***) int** *handle, color, flag, rgb*[3];

vq_color is a VDI routine that checks or sets the intensity of a particular color. *handle* is the virtual device's VDI handle. *color* is the code that indicates which color you wish to check or modify: for a table of color indices, see the entry for **v_opnwk**. *flag* indicates whether you want to set the color, or merely check it: zero indicates set the color, and one indicates check it. Finally, *rgb* is an array of three integers that, respectively, set the red, green, and blue guns on the color monitor. Each should be set to a level between one and 1,000, with one being the lowest setting and 1,000 the highest.

See Also

TOS, VDI, vq_extnd

vq_curaddress — VDI function (libvdi)

Get the text cursor's current position
#include <aesbind.h>
#include <vdibind.h>
void vq_curaddress(*handle, row, column***) int** *handle, *row, *column*;

vq_curaddress is a VDI routine that returns the current position of the text cursor on the virtual device. *handle* is the virtual device's VDI handle. *row* and *column* point to integers into which the function will write, respectively, the row and the column in which the text cursor is positioned.

Example

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

See Also

TOS, VDI, vs_curaddress

vq_extnd — VDI function (libvdi)

Perform extend inquire of VDI virtual device
#include <aesbind.h>
#include <vdibind.h>
void vq_extnd(*handle, type, work_out***) int** *handle, type, work_out*[57];

vq_extnd is a VDI routine that performs an extended inquiry on a virtual device. *handle* is the virtual device's VDI handle. *type* is the set of values you want written into the array *work_out*: zero indicates that you want the same values returned by functions **v_opnwk** or **v_opnvwk**. See the entry for **v_opnwk** for a table of these

values. Setting *type* to a non-zero value writes the extended inquiry values into *work_out*.

The following table gives the index into *work_out*, plus the value written there by the extended inquiry:

0	screen type: 0=not a screen; 1=separate alphabetic and graphics screens; 2=separate alphabetic and graphics controllers with common screen; 3=common alphabetic and graphics controller with separate image memories; and 4=common alphabetic and graphics controller and common image memory
1	no. of background colors available
2	which text effects are available
3	scaling possible? 0=no, 1=yes
4	no. of color planes
5	lookup table supported? 0=no, 1=yes
6	no. of 16X16-pixel raster operations done per second
7	contour fill supported? 0=no, 1=yes
8	can rotate characters? 0=no; 1=90 degrees only; 2=can rotate to arbitrary angles
9	no. of writing modes
10	highest level of input mode: 0=none; 1=request mode; 2=sample mode
11	text alignment supported? 0=no; 1=yes
12	handles multi-colored pens (e.g., plotter)? 0=no; 1=yes
13	handles multi-color ribbons (e.g., dot matrix printer)? 0=no; 1=yes
14	maximum no. of points in a polyline; -1=no maximum
15	maximum size of **intin** array: -1=no maximum
16	no. of buttons on the mouse
17	line types usable on wide lines? 0=no; 1=yes
18	drawing modes available for wide lines
19-57	reserved; contains zeroes

See Also

TOS, v_opnwk, VDI

Notes

This routine is called **vq_extend** in some bindings.

vq_key_s — VDI function (libvdi)

Check control key status

#include <aesbind.h>
#include <vdibind.h>
void vq_key_s(*handle, status***) int** *handle*, ***status***;

vq_key_s is a VDI routine that checks the control key status. *handle* is the virtual device's VDI handle. *status* is a bit map that, upon return, indicates the status of the control keys; zero indicates not set and one indicates set, as follows:

0	right shift key
1	left shift key
2	control key
3	alt key

See Also

TOS, VDI, vq_mouse

vq_mouse — VDI function (libvdi)

Check mouse position and button state

#include <aesbind.h>
#include <vdibind.h>
void vq_mouse(*handle, status, x, y***) int** *handle*, ***status*, *x, y;**

vq_mouse is a VDI routine that checks the mouse pointer's position and the status of the mouse buttons. *handle* is the virtual device's VDI handle. *status* is set by **vq_mouse**, and indicates the status of the mouse button: zero indicates not pressed, one indicates pressed. *x* and *y* are set by **vq_mouse** and give, respectively, the X and Y coordinates of the mouse pointer.

See Also

TOS, VDI

vq_tabstatus — VDI function (libvdi)

Find if graphics tablet is available

#include <aesbind.h>
#include <vdibind.h>
int vq_tabstatus(*handle***) int** *handle*;

vq_tabstatus is a VDI routine that checks to see if the graphics tablet is available. *handle* is the virtual device's VDI handle. **vq_tabstatus** returns the status of of the graphics tablet: zero indicates that the tablet is not available, and one indicates that it is.

See Also

TOS, VDI

Notes

This routine is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

vqf_attributes — VDI function (libvdi)

Read the area fill's current attributes

```
#include <aesbind.h>
#include <vdibind.h>
void vqf_attributes(handle, attrib) int handle, attrib[5];
```

vqf_attributes is a VDI routine that returns the attributes currently set for the area fill. *handle* is the virtual device's VDI handle. The fill area's attributes are written into the array *attrib*, as follows:

attrib[0] Fill type. For a table of fill types, see the entry for **vsf_interior**.

attrib[1] Fill color. For a table of color codes, see the entry for **v_opnwk**.

attrib[2] Fill pattern. For a table of fill patterns, see the entry for **vsf_style**.

attrib[3] Writing mode: one indicates replace mode; two, transparent mode; three, XOR (exclusive or) mode; and four, reverse transparent mode.

attrib[4] Draw border: zero indicates that a border is not drawn around a filled area, and one indicates that it will.

See Also

TOS, v_bar, VDI, vql_attributes, vqm_attributes, vqt_attributes

vqin_mode — VDI function (libvdi)

Determine mode of a logical input device

```
#include <aesbind.h>
#include <vdibind.h>
void vqin_mode(handle, device, mode) int handle, device, *mode;
```

vqin_mode is a VDI routine that returns the current mode of a logical input device. *handle* is, as always, the virtual device's VDI handle. *device* is the logical input device whose mode you wish to check, as follows: one, graphic cursor unit (i.e., devices that move the mouse pointer); two, value-changing input (e.g., shift key, control key, etc.); three, selection input unit (i.e., function keys); and four, string input unit (i.e., alphabetic keys). Finally, *mode* points to an integer that will hold the current mode: zero indicates request mode, and one indicates sample mode. *Request mode* waits for a particular event to occur on the device before the function returns, analogous to the AES event library; whereas *sample mode* simply polls the device and returns, without waiting for an event.

See Also

TOS, VDI, vsin_mode

vql_attributes — VDI function (libvdi)

Read the polyline's current attributes

```
#include <aesbind.h>
#include <vdibind.h>
void vql_attributes(handle, attrib) int handle, attrib[6];
```


vql_attributes is a VDI routine that returns the current attributes for the VDI polyline routine. *handle* is the virtual device's VDI handle. The polyline attributes are written into the array *attrib*, as follows:

attrib[0] Line type; see the entry for **vsl_type** for a table of line-type codes.

attrib[1] Line color; see the entry for **v_opnwk** for a table of color codes.

attrib[2] Writing mode: one indicates replace mode; two, transparent mode; three, XOR (exclusive or) mode; and four, reverse transparent mode.

attrib[3] Starting point style: zero indicates square; one, arrowhead; and two, rounded.

attrib[4] Ending point style.

attrib[5] Line width.

See Also

TOS, v_pline, VDI, vqf_attributes, vqm_attributes, vqt_attributes

vqm_attributes — VDI function (libvdi)

Read the marker's current attributes

```
#include <aesbind.h>
#include <vdibind.h>
void vqm_attributes(handle, attrib) int handle, attrib[5];
```

vqm_attributes is a VDI routine that returns the attributes currently set for the marker. *handle* is the virtual device's VDI handle. The marker's attributes are written into the array *attrib*, as follows:

attrib[0] Marker type, as follows:

1	period
2	plus sign
3	asterisk
4	square
5	diagonal cross
6	diamond
7	device-dependent

attrib[1] Marker color. For a table of color codes, see the entry for **v_opnwk**.

attrib[2] Writing mode: one indicates replace mode; two, transparent mode; three, XOR (exclusive or) mode; and four, reverse transparent mode.

attrib[3] Marker width.

attrib[4] Marker height.

See Also

TOS, v_pmarker, VDI, vqf_attributes, vql_attributes, vqt_attributes

vqp_error — VDI function (libvdi)

Inquire if an error occurred with the Polaroid Palette

```
#include <aesbind.h>
#include <vdibind.h>
int vqp_error(handle) int handle;
```

The VDI supports a driver for the Polaroid Palette, a camera that shoots color transparencies. When the driver is loaded, the VDI routine **vqp_error** returns an error message or user prompt for the camera. *handle* is the virtual device's VDI handle. **vqp_error** returns one of the following error messages:

0	no error
1	open dark slide for print film
2	no port at location specified in driver
3	Polaroid Palette not found at specified port
4	video cable is disconnected
5	operating system does not allow memory allocation
6	not enough memory available to allocate buffer
7	memory not freed
8	driver file not found
9	driver file is not of the correct type
10	user should now process print film

See Also

TOS, VDI, vqp_films, vqp_state, vsp_message, vsp_save, vsp_style

Notes

This routine is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

vqp_films — VDI function (libvdi)

Get films supported by driver for Polaroid Palette

```
#include <aesbind.h>
#include <vdibind.h>
void vqp_films(handle, names) int handle, names[125];
```

The VDI supports a driver for the Polaroid Palette, a camera that shoots color transparencies. When the driver is loaded, the VDI routine **vqp_films** returns the names of the fives types of photographic film supported by this driver. *handle* is the virtual device's VDI handle. *films* is an array that holds the names of the films supported.

See Also

TOS, VDI, vqp_error, vqp_state, vsp_message, vsp_save, vsp_style

Notes

This routine is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

vqp_state — VDI function (libvdi)

Read current settings of the Polaroid Palette driver

```
#include <aesbind.h>
#include <vdibind.h>
void vqp_state(handle, port, film, lightness, interlace, planes, indices);
int handle, *port, *film, *lightness, *interlace, *planes, *indices[8][2];
```

The VDI supports a driver for the Polaroid Palette, a camera that shoots color transparencies. When the driver is loaded, the VDI routine **vqp_state** returns a block of data that gives the driver's settings. *handle* is the virtual device's VDI handle. *port* is the port to which the camera is connected; zero indicates the first communications port. *film* is the number of the film for which the driver is currently set.

lightness is the intensity to which the driver is set, from -3 through three. Each number in this range is equivalent to one third of an f-stop, counting from zero. Therefore, -3 has half the intensity of zero, and three is twice as intense as zero.

interlace indicates whether the image is interlaced or not; zero indicates not interlaced, and one indicates interlaced. Note that an interlaced image requires approximately twice the memory of one that is not interlaced.

planes indicates the number of colors supported. It is set to a code, from one through four; one indicates two colors; two, four colors; three, eight colors; and four, 16 colors.

Finally, *indices* holds two-character codes for the eight color indices stored in ADE format.

See Also

TOS, VDI, vqp_error, vqp_films, vsp_message, vsp_save, vsp_style

Notes

This routine is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

vqt_attributes — VDI function (libvdi)

Read the graphic text's current attributes

```
#include <aesbind.h>
#include <vdibind.h>
void vqt_attributes(handle, attrib) int handle, attrib[10];
```

vqt_attributes is a VDI routine that returns the current attributes for the VDI graphics text routine. *handle* is the virtual device's VDI handle. The graphics text attributes are written into the array *attrib*, as follows:

attrib[0]	Character set.
attrib[1]	Text color. For a table of color codes, see the entry for **v_opnwk**.
attrib[2]	Rotation angle, in tenths of a degree (i.e., 0 through 3600).
attrib[3]	Horizontal alignment. For a table of alignment codes, see the entry for **vst_alignment**.
attrib[4]	Vertical alignment.
attrib[5]	Writing mode: one indicates replace mode; two, transparent mode; three, XOR (exclusive or) mode; and four, reverse transparent mode.
attrib[6]	Character width.
attrib[7]	Character height.
attrib[8]	Cell width.
attrib[9]	Cell height.

See Also

TOS, v_gtext, VDI, vqf_attributes, vql_attributes, vqm_attributes

vqt_extent — VDI function (libvdi)

Calculate a string's length
#include <aesbind.h>
#include <vdibind.h>
void vqt_extent(*handle, text, size***) int** *handle, size*[8]**; char** **text*;

vqt_extent is a VDI routine that calculates the length of a string. This is especially useful when positioning proportionally spaced text on a virtual device.

handle is the virtual device's VDI handle. *text* points to the string whose extent you wish to calculate. *size* is an array of eight integers that give the X and Y coordinates of the box that encloses the text, as follows: **size[0]** and **size[1]** give, respectively, the X and Y coordinates of the lower left-hand corner; **size[2]** and **size[3]**, X and Y coordinates of the lower right-hand corner; **size[4]** and **size[5]**, upper right-hand corner; and **size[6]** and **size[7]**, upper left-hand corner. Note that the box extends from the top of the tallest capital letters (e.g., 'M") to the bottom of the lowest descenders (e.g., 'j' or 'y').

See Also

TOS, v_gtext, VDI

vqt_fontinfo — VDI function (libvdi)

Get information about special effects for graphics text
#include <aesbind.h>
#include <vdibind.h>
void vqt_fontinfo(*handle, firstchar, lastchar, sizes, maxwidth, adjust***)**
int *handle, *firstchar, *lastchar, sizes*[5]**, *maxwidth, adjust*[3]**;**

vqt_fontinfo is a VDI routine that returns information about font sizes, especially about the extra space taken up by slanted and shadowed characters. You may need to obtain this information from **vqt_fontinfo** when constructing text to be passed to a specialized output device.

The arguments to **vqt_fontinfo** are as follows: *handle* is the virtual device's VDI handle. *firstchar* points to the first character in the ASCII table that can be set on this device, using the font and special effects that have been set for it; *lastchar* points to the last character in the ASCII table that can be so set. These values, of course, are set by **vqt_fontinfo**. *maxwidth* points to the maximum width of a character in the current font.

sizes points to an array of five integers that are set by **vqt_fontinfo**. Each represents a dimension of the current font, as follows:

sizes[0]	bottom line to baseline
sizes[1]	descent line to baseline
sizes[2]	half line to baseline
sizes[3]	ascent line to baseline
sizes[4]	top line to baseline

These terms are defined in the entry for **vst_alignment**.

Finally, *adjust* points to an array of three integers that are set by **vqt_fontinfo**; each represents a change to the font size represented by the special effects being used, as follows:

adjust[0]	increase in character width
adjust[1]	left offset
adjust[2]	right offset

The *right offset* is the amount of space a slanted letter extends beyond the edge of its "cell", which is defined as the width of the character measured across the bottom. The *left offset* is the extra space that must be set to the left of a slanted character, so its neighbor to the left does not slant over it. The increase in character is the total of the left and right offsets; this is the value you need to figure into the value returned by **vqt_extent** to gain the true extent of a string that uses special effects.

See Also

TOS, v_gtext, VDI, vqt_extent, vqt_name, vst_alignment

vqt_name — VDI function (libvdi)

Get name and description of graphics text font

#include <aesbind.h>
#include <vdibind.h>
int vqt_name(*handle, font, string***) int** *handle, font*; **char** *string*[**32**];

vqt_name is a VDI routine that returns the name and description of a given font. *handle* is the virtual device's VDI handle. *font* is the number of the font whose name you want. Finally, *string* is where **vqt_name** writes the font name and information. The first 16 **chars** hold the name of the font, and the next 16 hold a brief description of it.

vqt_name returns the font ID that is needed to access this face with the function **vst_font**. Note that the number of fonts available on a given virtual device is returned by the functions **v_opnwk** and **v_opnvwk** in the variable *work_out[10]*.

Example

The following example prints a description of each font currently available to the screen device. Note that this example should be compiled with the option **-VGEM**, but that you do not need to run it with the **gem** command.

```
#include <aesbind.h>
#include <vdibind.h>

/* global line A variables used by vdi; MUST be included */
int contrl[12],intin[128],ptsin[128],intout[128],ptsout[128];

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* show an alert box on the screen */
alertf(n, p) int n; char *p;
{
        static char buffer[512];
        sprintf(buffer, "%r", &p);
        return form_alert(n, buffer);
}

main()
{
        int nowhere = 0;
        int vdihandle;
        int info[32]; /* array used by vqt_name */
        int i;

        /* open application */
        appl_init();
        v_opnvwk(work_in, &vdihandle, work_out);
```



```
        /* return code and description of all screen fonts */
        for (i=1; i <= work_out[10]; i++)
                alertf(1, "[1][Font %d|%s][OK]",
                        vqt_name(vdihandle, i, info), info);

        /* close device, exit */
        v_clsvwk(vdihandle);
        appl_exit();
        exit(0);
}
```

See Also

TOS, VDI, vst_font

vqt_width — VDI function (libvdi)

Get character cell width

#include <aesbind.h>
#include <vdibind.h>
int vqt_width(*handle, character, width, left, right***)**
int *handle*, ******width*, ******left*, ******right*; **char** *character*;

vqt_width is a VDI routine that returns the width of a given character's cell, plus information about the "white space" that surrounds the character; it does not take into account the angle at which text is written, or any special effects used.

handle is the virtual device's VDI handle. *character* is the character whose size is to be checked. *width* is returned by **vqt_width**; it is the width of the character's cell. *left* and *right* are also set by **vqt_width**; they indicate the amount of white space left on, respectively, the left and the right of the character within its cell.

vqt_width returns -1 if the character requested is invalid or otherwise cannot be measured.

See Also

TOS, v_gtext, VDI

vr_recfl — VDI function (libvdi)

Draw a rectangular fill area

#include <aesbind.h>
#include <vdibind.h>
void vr_recfl(*handle, xyarray***) int** *handle, xyarray*[**4**];

vr_recfl is a VDI routine that draws a rectangle. Unlike its cousin **v_bar**, **vr_recfl** will draw only a rectangular chunk of the preset fill pattern; it cannot draw a perimeter. *handle* is the virtual device's VDI pattern.

xyarray sets the X and Y coordinates from which to construct the pattern; the even-numbered entries indicate the X coordinates, and the odd-numbered entries the Y coordinates. Which corner of the rectangle each pair of coordinates indicates will differ depending on whether the virtual device has been set to normalized device

coordinates (NDC) or to raster coordinates (RC). On an NDC device, the first pair points to the lower left-hand corner and the second pair to the upper right-hand corner; whereas on an RC device, the first pair points to the upper left-hand corner and the second pair to the lower right-hand corner.

Note that to use this routine, the fill type must be set with **vsf_interior**, the fill style by **vsf_style**, and the fill color by **vsf_color**.

Example

This example uses the random-number routines to create a random pattern, and fills the screen with it. The random-number generator is seeded with the lower portion of the system time. Typing any key repeats the process; typing **<return>** exits. Note that because the Atari ST bumps the system time in two-second increments, you must wait at least two seconds before a new pattern can be drawn.

```
#include <aesbind.h>
#include <gemdefs.h>
#include <time.h>
#include <vdibind.h>

#define USER 4                  /* user-defined fill pattern */

/* global line A variables used by vdi; MUST be included */
int contrl[12],intin[128],ptsin[128],intout[128],ptsout[128];

/* array used by vs_clip() and vr_recfl() */
int xyarray[] = { 1, 1, 1, 1 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];

/* array used by vsf_udpat() */
int fill[16];

/* throw-away declaration */
int nowhere = 0;

main()
{
        int vdihandle;                  /* virtual device's handle */
        int key;

        /* open application */
        appl_init();

        /* open screen device */
        v_opnvwk(work_in, &vdihandle, work_out);

        /* set clipping array */
        xyarray[2] = work_out[0];
        xyarray[3] = work_out[1];
        vs_clip(vdihandle, 1, xyarray);
```



```
        /* hide the mouse pointer; set interior to user-defined */
        v_hide_c(vdihandle);
        vsf_interior(vdihandle, USER);

        dofill(vdihandle);

        for(;;) {
                key = evnt_keybd();
                if ((char)key == '\r') {
                        v_show_c(vdihandle);
                        v_clsvwk(vdihandle);
                        appl_exit();
                        exit(1);

                } else
                        dofill(vdihandle);
        }
}

dofill(vdihandle)
int vdihandle;
{
        int counter;

        /* seed random-number routine with system time */
        srand((int)time(&nowhere));

        /* fill buffer with random numbers */
        for (counter = 0; counter < 16; counter++)
                fill[counter] = rand();

        vsf_udpat(vdihandle, fill, 1);
        vr_recfl(vdihandle, xyarray);
}
```

See Also

TOS, v_bar, VDI

vr_trnfm — VDI function (libvdi)

Transform a raster image

#include <aesbind.h>
#include <gemdefs.h>
#include <vdibind.h>
void vr_trnfm(*handle, sourcemfdb, destmfdb***)**
int *handle*; **FDB** **sourcemfdb*, **destmfdb*;

vr_trnfm is a VDI routine that transforms a raster image between standard (device-independent) and device-dependent forms. *handle* is the virtual device's VDI handle. *sourcemfdb* and *destmfdb* describe the "memory form definition block for the source and destination areas. Note that these are both set to the type **FDB**, which is defined in the header file **gemdefs.h**, as follows:

```
typedef struct fdbstr
{
        long fd_addr;
        int fd_w;
        int fd_h;
        int fd_wdwidth;
        int fd_stand;
        int fd_nplanes;
        int fd_r1;
        int fd_r2;
        int fd_r3;
} FDB;
```

fd_addr points to the beginning of the raster area in RAM. If this value is set to zero, **vro_cpyfm** assumes that it is dealing with the output of the physical device, and uses *handle* to address that device. It also ignores the rest of the **FDB** structure, which should be set to zeroes.

fd_w and *fd_h* give, respectively, the width and height of the area being copied to or copied from, in pixels. *fd_wdwidth* gives the width of the area being copied to/from, in 16-bit words (i.e., divided by 16), and rounded up. This information is needed internally by the VDI's raster copying routines.

fd_stand indicates whether the material is in device-dependent format or in device-independent (standard) format; zero indicates that it is in device-dependent format, and non-zero indicates standard format. Obviously, this should be set to the same value in both *sourcemfdb* and *destmfdb*. *fd_nplanes* is the number of color planes used in the virtual device. The total number of pixels used in the image, then, is the image's height in pixels, times its width in pixels, times the number of planes.

Finally, *fd_r1* through *fd_r3* are used by the system for its own purposes; they should be set to zero.

See Also
TOS, VDI

vro_cpyfm — VDI function (libvdi)

Copy raster form, opaque
#include <aesbind.h>
#include <gemdefs.h>
#include <vdibind.h>
void vro_cpyfm(*handle, logic, xyarray, sourcemfdb, destmfdb*)
int *handle, logic, xyarray*[8]; **FDB** **sourcemfdb, *destmfdb*;

vro_cpyfm is a VDI routine that copies a portion of a virtual image, pixel by pixel, from one location to another.

handle is the virtual device's VDI handle. *logic* defines the mode in which the area being copied will be drawn. The following table lists the available modes; **S** indicates the source pixel, and **D** the destination pixel:

0	Clear destination
1	S & D
2	S & !D
3	S (replace mode)
4	!S & D (erase mode)
5	D (has no effect)
6	S ^ D (exclusive-or mode)
7	S | D (transparent mode)
8	!(S & D)
9	!(S ^ D)
10	!D
11	S | (!D)
12	!S
13	(!S) | D (reverse transparent mode)
14	!(S & D)
15	1 (black out destination area)

Note that setting *logic* to **6** (i.e., to exclusive-or mode) allows you to use **vro_cpyfm** to mimic a hardware sprite, to move images around the screen with minimal fuss.

xyarray defines the area to be copied from and the area to be copied to. *xyarray*[0] through *xyarray*[3] is the area being copied from; the first two numbers define the X and Y coordinates of one corner of the rectangle, and the second two define the corner opposite it. Note that if the virtual device is defined as using normalized device coordinates (NDC), the first corner is the lower left-hand corner and the second the upper right-hand corner; whereas if the device uses raster coordinates (RC), the first corner is the upper left-hand corner and the second is the lower right-hand corner. *xyarray*[4] through *xyarray*[7] define the destination rectangle, in the same manner as the source rectangle. Note that for predictable results, the source and destination rectangles should be of the same size.

Finally, *sourcemfdb* and *destmfdb* point to the "memory form definition blocks" for the source and destination areas. Note that these are both set to the type **FDB**, which is defined in the header file **gemdefs.h**, as follows:

```
typedef struct fdbstr
{
        long fd_addr;
        int fd_w;
        int fd_h;
        int fd_wdwidth;
        int fd_stand;
        int fd_nplanes;
        int fd_r1;
        int fd_r2;
        int fd_r3;
} FDB;
```

fd_addr points to the beginning of the raster area in RAM. If this value is set to zero, **vro_cpyfm** assumes that a virtual device is being used (e.g., the screen), and uses *handle* to address that device; it also ignores the rest of the **FDB** structure, which should be set to zeroes.

fd_w and *fd_h* give, respectively, the the width and height of the area being copied to or copied from, in pixels. *fd_wdwidth* gives the width of the area being copied to/from, in 16-bit words (i.e., divided by 16), and rounded up. This information is needed internally by the VDI's raster copying routines.

fd_stand indicates whether the material is in device-dependent format or in device-independent (standard) format; zero indicates that it is in device-dependent format, and non-zero indicates standard format. Obviously, this should be set to the same value in both *sourcemfdb* and *destmfdb*. *fd_nplanes* is the number of color planes used in the virtual device. The total number of pixels used in the image, then, is the image's height in pixels, times its width in pixels, times the number of planes.

Finally, *fd_r1* through *fd_r3* are used by the system for its own purposes; they should be set to zero.

Example

The following example lets you copy one portion of the screen to another. When you click the mouse the first time, you draw a rectangle on the screen; clicking the mouse again lets you drag the rectangle to another part of the screen. When the mouse button is lifted the second time, the contents of the rectangle are copied to where the rectangle stopped. Pressing the 'W' key changes the writing mode, and pressing **<return>** exits.

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

#define BUTTON 1                        /* which button; 1=leftmost */
#define CLICKS 1                        /* no. of clicks expected */
#define DOWN 1                          /* mouse button is down */
#define FUJI 4                          /* "fuji" fill pattern */
#define HOLLOW 0                        /* make fill type hollow */
#define REPLACE 1                       /* make writing mode REPLACE */
#define RETURN 0x0D                     /* code for <return> */
#define W_KEY 0x77                      /* code for W key */
#define XOR 3                           /* make writing mode XOR */

/* global line A variables used by vdi; MUST be included */
int contrl[12],intin[128],ptsin[128],intout[128],ptsout[128];

/* array used by vs_clip() */
int cliparray[] = { 1, 1, 1, 1 };

/* arrays used by v_opvwk() */
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];
```



```
/* array used by v_bar() */
int xyarray[] = { 1, 1, 1, 1 };

/* arrays used by vro_cpyfm() */
int copyarray[8];

/* throw-away declaration */
int nowhere = 0;

main()
{
        /* declarations used by evnt_multi() */
        int selection;                          /* code for event that occurred */
        unsigned int which = (MU_KEYBD | MU_BUTTON);
        int buffer[11];                         /* place to write AES messages */
        unsigned key;                           /* scan code of key pressed */
        int mousex;                             /* mouse X coordinate */
        int mousey;                             /* mouse Y coordinate */

        /* misc declarations */
        int vdihandle;                          /* virtual device's handle */
        int logic = 3;                          /* logic type */
        FDB holder  = { 0L, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
                                                /* used by vro_cpyfm; all zeros */


        /* open application */
        appl_init();

        /* set mouse pointer to arrow */
        graf_mouse(ARROW, &nowhere);

        /* open screen device */
        v_opnvwk(work_in, &vdihandle, work_out);

        /* set clipping array */
        cliparray[2] = work_out[0];
        cliparray[3] = work_out[1];
        vs_clip(vdihandle, 1, cliparray);

        /* set interior of rectangle to Atari "fuji" */
        vsf_interior(vdihandle, FUJI);

        /* turn on rectangle perimeter */
        vsf_perimeter(vdihandle, 1);

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

        /* draw rectangle; turn on pointer again */
        xyarray[0] = work_out[0]/3;
        xyarray[1] = work_out[1]/3;
        xyarray[2] = work_out[0]/3;
        xyarray[3] = work_out[1]/3;
        v_bar(vdihandle, xyarray);
        graf_mouse(M_ON, &nowhere);
```

```
        for(;;) {
              /* wait for use to do something */
              selection = evnt_multi(which, CLICKS, BUTTON,
                    DOWN, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                    buffer, 0, 0, &mousex, &mousey,
                    &nowhere, &nowhere, &key, &nowhere);

              switch(selection) {
              /* if keyboard is touched ... */
              case MU_KEYBD:
                    /* if <return>, exit */
                    if ((char)key == RETURN) {
                          v_clsvwk(vdihandle);
                          appl_exit();
                          exit(0);
                    }
                    /* if "W" is pressed, change pattern */
                    if ((char)key == W_KEY)
                          logic++;
                    break;

              /* if mouse button is pressed, draw rubberbox */
              case MU_BUTTON:
                    getarray(vdihandle, mousex, mousey);
                    v_bar(vdihandle, xyarray);
                    vswr_mode(vdihandle, REPLACE);

                    graf_mouse(M_OFF, &nowhere);
                    /* do the blitting */
                    vro_cpyfm(vdihandle, (logic%16),
                          copyarray, &holder, &holder);
                    graf_mouse(M_ON, &nowhere);
                    break;

              default:
                    break;
              }
        }
}

getarray(handle, mousex, mousey)
int handle, mousex, mousey;
{
        int width;                          /* box width */
        int height;                         /* box height */
        int newx;                           /* X coordinate */
        int newy;                           /* Y coordinate */

        /* set source rectangle's coordinates */
        copyarray[0] = xyarray[0] = mousex;
        copyarray[1] = xyarray[1] = mousey;
        graf_rubbox(mousex, mousey, 0, 0, &width, &height);
        copyarray[2] = xyarray[2] = (mousex + width);
        copyarray[3] = xyarray[3] = (mousey + height);
```



```
        /* Now draw a rectangle around source area */
        graf_mouse(M_OFF, &nowhere);
        vswr_mode(handle, XOR);
        vsf_interior(handle, HOLLOW);
        v_bar(handle, xyarray);
        graf_mouse(M_ON, &nowhere);

        /*
         * wait for second button event; then set coordinates
         * for destination rectangle.
         */
        evnt_button(CLICKS, BUTTON, DOWN, &nowhere, &nowhere,
              &nowhere, &nowhere);
        graf_dragbox(width, height, mousex, mousey,
              0, 0, 639, 399, &newx, &newy);

        copyarray[4] = newx;
        copyarray[5] = newy;
        copyarray[6] = (newx + width);
        copyarray[7] = (newy + height);
        return;
}
```

See Also

TOS, VDI, vrt_cpyfm

vrq_choice — VDI function (libvdi)

Return status of function keys when any key is pressed

#include <aesbind.h>
#include <vdibind.h>
void vrq_choice(*handle, in, out***) int** *handle, in, *out*;

vrq_choice is a VDI routine that returns the status of the function keys when any key is pressed. In VDI jaragon, it operates the select device in request mode; these terms are described more fully in the entry for **vsin_mode**.

handle is the virtual device's VDI handle. *in* is the number of the function key you want to check, one through ten. The function terminates when any key is pressed; if the key was a function key, *out* holds its number; if another key was struck, *out* holds its ASCII value.

See Also

TOS, VDI, vsm_choice

Notes

Before this function can be used, the function **vsin_mode(***handle***, 3, 1)** must be entered, which will place the valuator device into request mode.

vrq_locator — VDI function (libvdi)

Find location of mouse cursor when a key is pressed

#include <aesbind.h>

#include <vdibind.h>
void vrq_locator(*handle, x, y, xout, yout, key*)
int *handle, x, y, *xout, *yout, *key*;

vrq_locator is a VDI routine that returns the location of the mouse cursor when a mouse button is pressed. In VDI jargon, it operates the position input device in request mode; these terms are described more fully in the entry for **vsin_mode**.

handle is the virtual device's VDI handle. *x* and *y* are, respectively, the X and Y coordinates of the mouse pointer's initialized position.

xout and *yout* are, respectively, the X and Y coordinates of the mouse pointer when a key is pressed. Finally, the low byte of *key* gives the ASCII code of the key that was pressed to terminate the polling of the screen. The left and right buttons on the mouse can also terminate polling. These return, respectively, 0x20 and 0x21. Note that because any key can end polling of the screen, you must write a loop if you want to terminate on a particular key.

See Also
TOS, VDI, vsm_locator

Notes
Before this function can be used, the function **vsin_mode**(*handle*, **1, 1**) must be entered, which will place the locator device into request mode.

vrq_string — VDI function (libvdi)

Read a string from the keyboard
#include <aesbind.h>
#include <vdibind.h>
void vrq_string(*handle, length, echo, xyarray, string*)
int *handle, length, echo, xyarray*[2]; **char** **string*;

vrq_string is a VDI routine that reads a string from the keyboard. The string is automatically terminated with a NUL character. The systems stops accepting characters either when the user presses the **<return>** key, or when the string exceeds the maximum length set by the user. In VDI jaragon, it operates the string device in request mode; these terms are described more fully in the entry for **vsin_mode**.

handle is, as always, the virtual device's VDI handle. *length* is the maximum length of the string, in characters. *echo* indicates whether or not you want the string echoed to the screen as the user types; zero indicates no echo, whereas one indicates to echo. *xyarray* gives the X and Y coordinates of where on the screen to begin echoing the string. Finally, *string* points to where the string will be written.

See Also
TOS, VDI, vsm_string

Notes
Before this function can be used, the function **vsin_mode**(*handle*, **4, 1**) must be entered, which will place the valuator device into request mode.

vrq_valuator — VDI function (libvdi)

Return status of shift and cursor keys
#include <aesbind.h>
#include <vdibind.h>
void vrq_valuator(*handle, in, out, key*) **int** *handle, in, *out, *key*;

vrq_valuator is a VDI routine that returns the status of the valuator keys. In VDI jargon, it operates the valuator keys in request mode; these terms are described more fully in the entry for **vsin_mode**.

handle is the virtual device's VDI handle. *in* is the code of the valuator key whose status you wish to check. *key* is the code of the key that was pressed to terminate this function. Finally, *out* is the value of *key* plus a specific value that indicates which valuator key was pressed along with it, as follows:

Cursor up	*key* plus ten
Cursor down	*key* minus ten
Shift/cursor up	*key* plus one
Shift/cursor down	*key* minus one

See Also
TOS, VDI, vsm_valuator

Notes
Before this function can be used, the function **vsin_mode**(*handle*, **2, 1**) must be entered, which will place the valuator device into request mode.

vrt_cpyfm — VDI function (libvdi)

Copy raster form, transparent
#include <aesbind.h>
#include <gemdefs.h>
#include <vdibind.h>
void vrt_cpyfm(*handle, mode, xyarray, sourcemfdb, destmfdb, color*)
int *handle, mode, xyarray*[8], *color*[2]; **FDB** **sourcemfdb*, **destmfdb*;

vrt_cpyfm is a VDI routine that copies a monochromatic image onto a polychromatic device, such as the screen. It resembles the blitting function, **vro_cpyfm**, but it is designed particularly for moving images around the screen.

handle is the virtual device's VDI handle. *mode* is the mode in which the image is written, as follows: one, replace mode; two, transparent mode; three, XOR (exclusive or); and four, reverse transparent. Note that these are the same codes used by the VDI routine **vswr_mode**, which is usually used to set the writing mode.

xyarray defines the area to be copied from and the area to be copied to. *xyarray*[0] through *xyarray*[3] is the area being copied from; the first two numbers define the X and Y coordinates of one corner of the rectangle, and the second two define the corner opposite it. Note that if the virtual device is defined as using normalized device coordinates (NDC), the first corner is the lower left-hand corner and the second the upper right-hand corner; whereas if the device uses raster coordinates (RC), the first corner is the upper left-hand corner and the second is the lower right-hand corner. *xyarray*[4] through *xyarray*[7] define the destination rectangle, in the same manner as the source rectangle. Note that for predictable results, the source and destination rectangles should be of the same size.

color is an array of two integers that set the color indices: *color*[0] sets the index for the foreground color, and *color*[1] sets the index for the background color. The color indices are as follows:

0	WHITE
1	BLACK
2	RED
3	GREEN
4	BLUE
5	CYAN
6	YELLOW
7	MAGENTA
8	WHITE
9	BLACK
10	LRED
11	LGREEN
12	LBLUE
13	LCYAN
14	LYELLOW
15	LCYAN
16-*n*	device-independent

sourcemfdb and *destmfdb* point to the "memory form definition blocks for the source and destination areas. Note that these are both set to the type **FDB**, which is defined in the header file **gemdefs.h**, as follows:


```
typedef struct fdbstr
{
        long fd_addr;
        int fd_w;
        int fd_h;
        int fd_wdwidth;
        int fd_stand;
        int fd_nplanes;
        int fd_r1;
        int fd_r2;
        int fd_r3;
} FDB;
```

fd_addr points to the beginning of the raster area in RAM. If this value is set to zero, **vrt_cpyfm** assumes that a virtual device is being used (e.g., the screen), and uses *handle* to address that device; it also ignores the rest of the **FDB** structure, which should be set to zeroes.

fd_w and *fd_h* give, respectively, the the width and height of the area being copied to or copied from, in pixels. *fd_wdwidth* gives the width of the area being copied to/from, in 16-bit words (i.e., divided by 16), and rounded up. This information is needed internally by the VDI's raster copying routines.

fd_stand indicates whether the material is in device-dependent format or in device-independent (standard) format; zero indicates that it is in device-dependent format, and non-zero indicates standard format. Obviously, this should be set to the same value in both *sourcemfdb* and *destmfdb*. *fd_nplanes* is the number of color planes used in the virtual device. The total number of pixels used in the image, then, is the image's height in pixels, times its width in pixels, times the number of planes.

Finally, *fd_r1* through *fd_r3* are used by the system for its own purposes; they should be set to zero.

See Also

TOS, VDI, vro_cpyfm

vs_clip — VDI function (libvdi)

Set the virtual device's clipping rectangle

#include <aesbind.h>
#include <vdibind.h>
void vs_clip(*handle, flag, xyarray*) **int** *handle, flag, xyarray*[4];

vs_clip is a VDI routine that sets the clipping rectangle for a virtual device. The *clipping rectangle* is the portion of an image that is actually displayed on the physical device; if any portion of the image drawn on the virtual device extends beyond the clipping rectangle, it is trimmed off. If an image is not clipped, it could extend beyond the borders of the physical device; this, in turn, causes memory to be drawn over, possibly with catastrophic results.

handle is the virtual device's VDI handle. *flag* indicates whether clipping should be

turned on or off: zero indicates off, one indicates on.

Finally, *xyarray* is an array of four integers that place the clipping rectangle, as follows:

xyarray[0]	X coordinate of first corner
xyarray[1]	Y coordinate of first corner
xyarray[2]	X coordinate of opposite corner
xyarray[3]	Y coordinate of opposite corner

Note that if the device is set to normalized device coordinates (NDC), the first corner is the upper left-hand corner of the image; whereas if the device is set to raster coordinates, the first corner is the lower left-hand corner.

Example

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

See Also

TOS, VDI

vs_color — VDI function (libvdi)

Set color intensity
#include <aesbind.h>
#include <vdibind.h>
void vs_color(*handle, index, rgbarray***) int** *handle, index, rgbarray***[3];**

vs_color is a VDI routine that sets the intensity of a color. Each color is set by adjusting the intensity of three electron guns, one for red pixels, another for green pixels, and a third for blue. **vs_color** allows you to adjust the intensity of each gun for given color.

handle is a virtual device's VDI handle. *index* is the code for the color being adjusted; for a table of these indices, see the entry for **v_opnwk**. Finally, *rgbarray[0]* through *rgbarray[2]* hold, respectively, the new value for the red, blue, and green guns; each value is an integer between one and 1,000.

See Also

TOS, VDI

vs_curaddress — VDI function (libvdi)

Move text cursor to specified row and column
#include <aesbind.h>
#include <vdibind.h>
void vs_curaddress(*handle, row, column***) int** *handle, row, column***;**

vs_curaddress is a VDI routine that moves the text cursor to a specified row and column on the virtual device. Note that to use this routine, the virtual device must first be placed in alphabetic mode, with the routine **v_enter_cur**. *handle* is the virtual device's VDI handle. *row* and *column* give, respectively, the row and column where you wish to position the text cursor.

Example

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

See Also

TOS, VDI, vq_curaddress

vs_palette — VDI function (libvdi)

Select color palette on medium-resolution screen
#include <aesbind.h>
#include <vdibind.h>
void vs_palette(*handle, palette***) int** *handle, palette***;**

vs_palette is a VDI routine that selects a palette for use on the medium-resolution screen. *handle* is the virtual device's VDI handle. *palette* is a pre-set color palette: zero (the default) indicates a palette of red, green, and brown; and one indicates a palette of cyan, magenta, and white.

See Also

TOS, VDI

vsc_form — VDI function (libvdi)

Draw a new shape for the mouse pointer
#include <aesbind.h>
#include <vdibind.h>
void vsc_form(*handle, form***) int** *handle, form***[37];**

vsc_form is a VDI routine that draws a new shape for the mouse pointer. *handle* is the virtual device's VDI handle.

form is an array of 37 integers. *form[0]* and *form[1]* give, respectively, the X and Y coordinates for the "action point", or the point on the pointer that is considered significant; in most instances, this is the upper left-hand corner. These values are set relative to the upper left-hand corner.

form[2] is reserved by the VDI, and must be set to one. *form[3]* is the color index mask, and is normally set to zero. *form[4]* is the color index cursor form, and is normally set to one. *form[5]* through *form[20]* gives the bit form of the mouse pointer's mask, or its monochromatic image. Finally, *form[21]* through *form[36]* gives the cursor form in color; bits set to one in this map are shown in the background color.

Once the new shape is loaded with **vsc_form**, it can be called with **graf_mouse(USER_DEF,** *form***);**. The header file **gemdefs.h** must be included to use this call. **graf_mouse** is another way to set the mouse to a user-defined form; it is implemented using **vsc_form**.

See Also

TOS, VDI

vsf_color — VDI function (libvdi)

Set a polygon's fill color

```
#include <aesbind.h>
#include <vdibind.h>
void vsf_color(handle, color) int handle, color;
```

vsf_color is a VDI routine that sets a polygon's fill color. *handle* is the virtual device's VDI handle. *color* is the color to which the polygon's fill should be set; for a table of color settings, see the entry for **v_opnwk**. Note that this routine can be used only with **vsf_interior** and **vsf_style**.

See Also

TOS, v_bar, v_opnwk, VDI, vsf_interior, vsf_style

vsf_interior — VDI function (libvdi)

Set a polygon's fill type

```
#include <aesbind.h>
#include <vdibind.h>
void vsf_interior(handle, type) int handle, type;
```

vsf_interior is a VDI routine that lets you choose what type of filling will be used for a polygon. *handle* is the virtual device's VDI handle. *type* is the type of fill you choose, as follows:

0	empty (erased, set to color 0)
1	solid
2	patterned
3	cross-hatched
4	user-defined type

Using the "empty" setting and having the "transparent" flag set by the routine **vswr_mode** will result in only the outline of a polygon being drawn, with what is in the background filling its interior.

Example

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

See Also

TOS, v_bar, VDI, vsf_style

vsf_perimeter — VDI function (libvdi)

Set whether to draw a perimeter around a polygon

```
#include <aesbind.h>
#include <vdibind.h>
void vsf_perimeter(handle, flag) int handle, flag;
```


vsf_perimeter is a VDI routine that lets you choose whether or not to draw a perimeter around a polygon you are creating. The perimeter is in the color that you have set with the routine **vsf_color**, and it is always one raster wide. *handle* is the virtual device's VDI handle. *flag* indicates whether or not to draw a perimeter: zero indicates not to draw a perimeter, and one indicates to draw one.

Example

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

See Also

TOS, v_bar, VDI, vsf_color

vsf_style — VDI function (libvdi)

Set a polygon's fill style

```
#include <aesbind.h>
#include <vdibind.h>
void vsf_style(handle, style) int handle, style;
```

A polygon's fill *type* is set with the routine **vsf_interior**, and can be one of the following: hollow, solid, patterned, cross-hatched, or user defined. If one of the last three types is selected, then **vsf_style** can be used to selected the *style* of filling.

handle is the virtual device's VDI handle. *style* is the code number of the fill style selected. For a patterned fill, 24 styles are available, as follows:

1-8	gray tones, from lightest (1) to solid (8)
9	horizontal "brick" pattern
10	diagnonal "brick" pattern
11	inverted 'v's
12	arch
13	cross-hatched line segments
14	heavy random dots
15	light random dots
16	interwoven hollow lines
17	zig-zagged thin lines plus dots
18	horizontal and vertical lines of dots
19	black balls in checkerboard pattern
20	overlapping scale shapes
21	overlapping diagonal rectangles
22	rectangles in checkboard pattern
23	diamond pattern
24	lines in herringbone pattern

For a cross-hatched fill, 12 styles are available, as follows:

1	light, closely spaced diagonal lines
2	heavy, closely spaced diagonal lines
3	heavy, closely spaced, diagonal cross-hatched lines
4	closely spaced vertical lines
5	closely spaced horizontal lines
6	heavy, closely spaced, perpendicular cross-hatched lines
7	light, widely spaced diagonal lines
8	heavy, widely spaced diagonal lines
9	light, closely spaced, diagonal cross-hatched lines
10	widely spaced vertical lines
11	widely spaced horizontal lines
12	widely spaced perpendicular lines

The styles for a user-defined fill are set with the function **vsf_udpat**. The default user-defined fill is the "fuji" (the Atari symbol).

Example

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

See Also

TOS, v_bar, VDI, vsf_interior

vsf_udpat — VDI function (libvdi)

Define a fill pattern

#include <aesbind.h>
#include <vdibind.h>
void vsf_udpat(*handle, pattern, planes***) int** *handle, pattern*[*n*]**,** *planes***;**

vsf_udpat is a VDI routine that allows a user to define a customized fill pattern. *handle* is the virtual device's VDI handle. *planes* is the number of color planes used in the pattern; the fill pattern must have a 16-integer array for each color plane. *pattern* is an array of 16 integers that defines the dot pattern, beginning in the upper left-hand corner and working through the lower right-hand corner. *n* must be set to 16 times *planes*. Note that once a pattern has been set, it must be loaded using **vsf_interior** and **vsf_style**.

Example

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

See Also

TOS, v_bar, VDI, vsf_interior, vsf_style

vsin_mode — VDI function (libvdi)

Set input mode for logical input device

#include <aesbind.h>
#include <vdibind.h>
int vsin_mode(*handle, device, mode***) int** *handle, device, mode***;**

vsin_mode is a VDI routine that sets the input mode for a given logical input device. This mode is used by a set of functions that poll the input devices for information about their current status.

The VDI recognizes four types of input devices: *Position input devices* control the position of the mouse cursor on the screen; these are the mouse itself or the cursor keys. *Value-changing devices* affect only the value returned by another input device; these include the shift key, the control key, and the alt key. *Selection input devices* return a selection number; these refer only to the Atari ST's function keys. Finally, *string input devices* are the alphabetic keys on the Atari ST's keyboard, by which strings are input.

handle is the virtual device's VDI handle. *device* indicates the logical device you wish to set: one indicates the position devices; two, value-changing input devices; three, the selection devices; and four, the string-input devices.

Finally, *mode* is the mode to which you want to set the device. *Request mode* tells the polling function to wait for input from a given device, e.g., for a key to be struck or a mouse button to be pressed. *Sample mode* simply polls the device and returns, without waiting for an event. One indicates request mode, and two indicates sample mode.

vsin_mode returns the mode to which the device was set.

See Also

TOS, VDI, vqin_mode, vrq_choice, vrq_locator, vrq_string, vrq_valuator, vsm_choice, vsm_locator, vsm_string, vsm_valuator

vsl_color — VDI function (libvdi)

Set a line's color

#include <aesbind.h>
#include <vdibind.h>
int vsl_color(*handle, color***) int** *handle, color***;**

vsl_color is a VDI routine that sets the color of a line. *handle* is the virtual device's VDI handle. *color* is the color to which the line is being set. For a list of the available values, see the entry for **v_opnwk**. If the color requested is not available on the target virtual device, the line color will be set to one (black).

vsl_color returns the color to which the line was set.

See Also

TOS, VDI, v_pline

vsl_ends — VDI function (libvdi)

Attach ends to a line

#include <aesbind.h>
#include <vdibind.h>

void vsl_ends(*handle, beginning, end*) int *handle, beginning, end*;

vsl_ends is a VDI routine that attaches ends to a line. *handle* is the virtual device's VDI handle. *beginning* and *end* refer to the type of figure drawn at, respectively, the beginning and the end of the line, as follows:

0	squared end (default)
1	arrowhead
2	rounded end

Example

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

See Also

TOS, VDI, v_pline

vsl_type — VDI function (libvdi)

Set a line's type
#include <aesbind.h>
#include <vdibind.h>
int vsl_type(*handle, type*) int *handle, type*;

vsl_type is a VDI routine that sets a line's type. *handle* is the virtual device's VDI handle.

type is the type to which the line is being set, as follows:

1	solid
2	long dashes
3	dots
4	dash-dot
5	dashes
6	dash-dot-dot
7	user-defined
8-*n*	device-dependent

vsl_type returns the type to which the line was set.

Example

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

See Also

TOS, v_pline, VDI, vsl_udsty

vsl_udsty — VDI function (libvdi)

Set user-defined line type
#include <aesbind.h>
#include <vdibind.h>
void vsl_udsty(*handle, pattern*) int *handle, pattern*;

vsl_udsty is a VDI routine that lets the user design a line type to be drawn by **v_pline**. *handle* is the virtual device's VDI handle. *pattern* is a bit map for the pattern to be drawn. Setting a bit to one means that its 1/16 portion of a line unit will be drawn; setting it to zero means that its portion will be blank.

Note that once the bit pattern is set with **vsl_udsty**, it must be loaded with the function **vsl_type**.

See Also

TOS, v_pline, VDI, vsl_type

vsl_width — VDI function (libvdi)

Set a line's width
#include <aesbind.h>
#include <vdibind.h>
int vsl_width(*handle, width*) int *handle, width*;

vsl_width is a VDI routine that sets a line's width. *handle* is the virtual device's VDI handle.

width is the width of the line to be drawn; this will vary depending on whether the virtual device being drawn on is set in normalized device coordinates (NDC) or raster coordinates (RC). The value *work_out[7]* indicates how many line widths are available for you to use on that device; see the entry for **v_opnwk** for more information. If the line width you request is not available on the virtual device, the line width will be set to the next *smaller* width.

vsl_width returns the width to which the line was actually set.

Example

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

See Also

TOS, VDI, v_opnwk, v_pline

vsm_choice — VDI function (libvdi)

Return last function key pressed
#include <aesbind.h>
#include <vdibind.h>
int vsm_choice(*handle, key*) int *handle*, *key*;

vsm_choice is a VDI routine that returns the last function key pressed, whether or not another key is pressed. To use VDI jargon, it operates the valuator device in sample mode; these terms are explained more fully in the entry for **vsin_mode**. *handle* is the virtual device's VDI handle. *choice*, which is set by **vsm_choice**, is the number of the function key last pressed, from one to ten. If no function key was pressed, the ASCII code of the last key pressed is returned.

vsm_choice returns either zero or one; the former indicates that no key was pressed, whereas the latter indicates that a key was pressed.

See Also

TOS, VDI, vrq_choice

Notes

Before this function can be used, the function **vsin_mode**(*handle*, **3, 2**) must be entered, which will place the locator device into sample mode.

vsm_color — VDI function (libvdi)

Set a polymarker's color
#include <aesbind.h>
#include <vdibind.h>
int vsm_color(*handle, color*) **int** *handle, color*;

vsm_color is a VDI routine that sets a marker's color. *handle* is the virtual device's VDI handle. *color* is the color you select for the marker; for a list of the legal color codes, see the entry for **v_opnwk**. If the color you requested is not available, the marker's color will be set by default to one (black).

vsm_color returns the color to which marker is actually set.

See Also

TOS, VDI, v_pmarker

vsm_height — VDI function (libvdi)

Set a polymarker's height
#include <aesbind.h>
#include <vdibind.h>
int vsm_height(*handle, height*) **int** *handle, height*;

vsm_height is a VDI routine that set the height of a polymarker. *handle* is the virtual device's VDI handle.

height is new size of the image, in Y coordinate units; these are used to avoid problems with scaling. Note that not every device will support every requested size of marker. Interrogating the variable *work_out[9]*, which is a member of the array returned by the routine used to open the virtual device, will indicate the number of marker sizes available; zero indicates continuous scaling, i.e., that every size is supported. See the entry for **v_opnwk** for more information. Note that if a particular size is unavailable, the marker will be rescaled automatically to the next available *smaller* size.

vsm_height returns the height to which the marker is set.

Example

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

See Also

TOS, VDI, v_opnwk, v_pmarker

vsm_locator — VDI function (libvdi)

Return mouse pointer's position
#include <aesbind.h>
#include <vdibind.h>
int vsm_locator(*handle, x, y, xout, yout, key*)
int *handle, x, y, *xout, *yout, *key*;

vsm_locator is a VDI routine that returns the mouse pointer's position whether or not a key was pressed. To use VDI jargon, it operates the position input device in sample mode; these terms are explained more fully in the entry for **vsin_mode**. Because VDI programs work by interrupts, a program does not know where the mouse pointer is at any given point; this function is handed an initial set of coordinates for the mouse pointer, then polls the screen to find where it is now. It returns and indicates whether the pointer has changed from the initializing coordinates, whether a key was pressed, or both; and it sets values for the new X and Y coordinates (if any) and for the key that was pressed (if any).

handle is, as always, the virtual device's VDI handle. *x* and *y* give, respectively, the X and Y coordinates of the mouse pointer's initialized position; these may be set by another function.

xout and *yout* are set by **vsm_locator**; they give, respectively, the mouse pointer's X and Y coordinates, if they are different from the initializing coordinates. *key* is also set by **vsm_locator**; its low byte gives the ASCII value of a key pressed in the interval, if any.

Finally, **vsm_locator** returns a code, from zero to three, which indicates the following: zero, the mouse pointer was not moved and no key was pressed; one, the mouse pointer was moved, but no key was pressed; two, the mouse pointer was not moved, but a key was pressed; and three, the mouse pointer moved and a key was pressed.

See Also

TOS, VDI, vrq_locator

Notes

Before this function can be used, the function **vsin_mode**(*handle*, **1, 2**) must be entered, which will place the locator device into sample mode.

vsm_string — VDI function (libvdi)

Read a string from the keyboard
#include <aesbind.h>

#include <vdibind.h>
int vsm_string(*handle, length, echo, xyarray, string***)**
int *handle, length, echo, xyarray*[2]; **char** **string*;

vsm_string is a VDI routine that reads a string from the keyboard. The string is automatically terminated with a NUL character. Unlike **vrq_string**, it also notes if any non-alphabetic keys were struck. String entry ends either when a non-alphabetic key is struck, or when the string exceeds the maximum length set by the user.

handle is the virtual device's VDI handle. *length* is the maximum length of the string. *echo* indicates whether or not you want the characters echoed to the screen as they are input: zero indicates not to echo, and one indicates to echo. *xyarray* gives the X and Y coordinates of the position on the screen where to begin echoing the string. Finally, *string* points to the area where the string will be written; be sure to set aside at least *length* amount of space for the string, or you may write over vital memory.

vsm_string returns zero if the string was terminated by a non-alphabetic key, and a number greater than one if it was not. If you plan to have the user terminate the string with the **<return>** key, use **vrq_string** instead of the present function.

See Also
TOS, VDI, vrq_string

Notes
Before this function can be used, the function **vsin_mode(***handle***, 4, 2)** must be entered, which will place the locator device into sample mode.

vsm_type — VDI function (libvdi)

Set polymarker's type
#include <aesbind.h>
#include <vdibind.h>
int vsm_type(*handle, type***) int** *handle, type*;

vsm_type is a VDI routine that sets the type of polymarker displayed on the virtual device. *handle* is the virtual device's VDI handle. *type* is the type of marker being shown, as follows:

1	dot
2	plus sign
3	asterisk
4	square
5	diagonal cross
6	diamond
7	device-dependent

If the type of marker requested is not available on the virtual device, the default marker (an asterisk) will be used. **vsm_type** returns the type of marker to be displayed.

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

See Also
TOS, VDI, v_pmarker

vsm_valuator — VDI function (libvdi)

Return shift/cursor key status
#include <aesbind.h>
#include <vdibind.h>
void vsm_valuator(*handle, in, out, key, status***)**
int *handle, in, *out, *key, *status*;

vsm_valuator is a VDI routine that that returns the status of a shift key or cursor key whether or not another key is pressed. To use VDI jargon, it operates the valuator device in sample mode; these terms are explained more fully in the entry for **vsin_mode**.

handle is the virtual device's VDI handle. *in* is the code of the valuator key whose status you wish to examine. *key* is set by **vsm_valuator**; it is the code of the key pressed before this routine exits, if any. Because all functions in sample mode merely examine the status of a device and return without being triggered by a hardware event, a key may not necessarily have been pressed during this function's operation. *out* is the value of *key*, plus a value that indicates the status of one or more valuator keys, as follows:

Cursor up	*key* plus ten
Cursor down	*key* minus ten
Shift/cursor up	*key* plus one
Shift/cursor down	*key* minus one

Finally, *status* gives the status of the valuator devices, as follows:

0	no action occurred
1	value was changed
2	key was pressed

See Also
TOS, VDI, vrq_valuator

Notes
Before this function can be used, the function **vsin_mode(***handle***, 2, 2)** must be entered, which will place the locator device into sample mode.

vsp_message — VDI function (libvdi)

Suppress messages from Polaroid Palette device
#include <aesbind.h>
#include <vdibind.h>
void vsp_message(*handle***) int** *handle*;

vsp_message is a VDI routine that suppresses messages from the Polaroid Palette device. These messages are normally output to the screen. *handle* is the virtual device's VDI handle.

See Also

TOS, VDI, vqp_error

Notes

This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

vsp_save — VDI function (libvdi)

Save to disk current setting of Polaroid Palette driver
#include <aesbind.h>
#include <vdibind.h>
void vsp_save(*handle***) int** *handle*;

The VDI contains a driver for the Polaroid Palette, a camera that can be used to shoot slides directly from the Atari ST. **vsp_save** is a VDI routine that writes the current settings for this driver to disk. *handle* is the virtual device's VDI handle.

See Also

TOS, VDI, vqp_error, vqp_films, vqp_state, vsp_message, vsp_state

Notes

This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

vsp_state — VDI function (libvdi)

Set the Polaroid Palette driver
#include <aesbind.h>
#include <vdibind.h>
void vsp_state(*handle, port, film, lightness, interlace, planes, indices***);**
int *handle, port, film, lightness, interlace, lanes, indices*[8][2];

The VDI contains a driver for the Polaroid Palette, a camera that can be used to shoot slides directly from the Atari ST. **vsp_state** changes the settings for this driver.

handle is the virtual device's VDI handle. *port* is the port to which the camera is connected; zero indicates the first communications port. *film* is the number of the film for which the driver is currently set.

lightness is the intensity to which the driver is set, from -3 through three. Each number in this range is equivalent to one third of an *f*-stop, counting from zero. Therefore, -3 has half the intensity of zero, and three is twice as intense as zero.

interlace indicates whether the image is interlaced or not; zero indicates not interlaced, and one indicates interlaced. Note that an interlaced image requires approximately twice the memory of one that is not interlaced.

planes indicates the number of colors supported. It is set to a code, from one through four; one indicates two colors; two, four colors; three, eight colors; and four, 16 colors.

Finally, *indices* holds two-character codes for the eight color indices stored in ADE format.

See Also

TOS, VDI, vqp_error, vqp_films, vqp_state, vsp_message, vsp_save

Notes

This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

vst_alignment — VDI function (libvdi)

Realign graphics text
#include <aesbind.h>
#include <vdibind.h>
void vst_alignment(*handle, horiz, vertical, sethoriz, setvert***)**
int *handle, horiz, vertical, *sethoriz, *setvert*;

vst_alignment is a VDI routine that realigns graphics text.

Graphics text is aligned both horizontally and vertically. Horizontal alignment can be to the left (left justified), to the right (right justified), or centered. Vertical alignment can be one of the following: *baseline*, that is, aligned along the bottoms of the characters, excluding descenders (the "tails" on letters like 'j' or 'y'); *half line*, or aligned along the tops of the lower-case letters; *ascent line*, or along the tops of the upper-case letters; *bottom line*, or along the bottom of the character cell (i.e., the bottom of the white space found below the descenders); *descent line*, or along the bottom of the descenders, excluding the white space found below them; and *top line*, or along the top of the character cell (e.g., the top of the white space found above the capital letters). By default, characters are aligned to the left horizontally, and along the baseline vertically.

The following describes the arguments to **vst_alignment**: *handle* is the virtual device's VDI handle. *horiz* is the horizontal alignment you want, as follows:

0	left
1	centered
2	right

vertical is the vertical alignment you want, as follows:

0	baseline
1	half line
2	ascent line
3	bottom line
4	descent line
5	top line

sethoriz and *setvert* point, respectively, to the horizontal and vertical alignments that were actually set. You may wish to check these values, because not every alignment is available with every type face on every virtual device.

Example

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

See Also

TOS, v_gtext, VDI

vst_color — VDI function (libvdi)

Set color for graphics text

```
#include <aesbind.h>
#include <vdibind.h>
int vst_color(handle, color) int handle, color;
```

vst_color is a VDI routine that that sets the color for graphics text. *handle* is the virtual device's VDI handle. *color* is the color being set. See the entry for **v_opnwk** for a table of legal color settings.

If the color requested is not available on this virtual device, **vst_color** sets the color to a default of one (black). It returns the color that was actually set.

See Also

TOS, v_gtext, v_opnwk, VDI,

vst_effects — VDI function (libvdi)

Set special effects for graphics text

```
#include <aesbind.h>
#include <vdibind.h>
int vst_effects(handle, effects) int handle, effect;
```

vst_effects is a VDI routine that sets special effects for graphics text. *handle* is the virtual device's VDI handle. *effect* is the set of effects that you wish to use, as follows:

0x01	thickened letters
0x02	lowered intensity
0x04	slanted letters
0x08	underlining
0x10	outlined letters
0x20	shadowed letters

For example, if you want letters that are underlined and shadowed, set *effects* to 0x28 (i.e., 0x08 plus 0x20). Not every effect will be available on every virtual device. **vst_effects** returns the settings for the effects that were actually set.

Example

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

See Also

TOS, v_gtext, VDI

vst_font — VDI function (libvdi)

Select a new font

```
#include <aesbind.h>
#include <vdibind.h>
int vst_font(handle, font) int handle, font;
```

vst_font is a VDI routine that selects a new font for graphics type. *handle* is the virtual device's VDI handle. *font* is the code number of the new font available. The number of fonts available on a virtual device can be determined either by examined the value returned by the font-loading routine **vst_load_fonts**, or by interrogating the *work_out* array returned by **v_opnwk** and **v_opnvwk**: *work_out[10]* contains this information. Use the routine **vqt_name** to obtain the index number and a description of each available font.

If you select a font that is not available on the virtual device you are working with, the font will be set to a default; on the screen, the default is the system font. **vst_font** returns the code of the font actually selected.

See Also

TOS, v_gtext, VDI, vqt_name, vst_load_fonts, vst_unload_fonts

Notes

This function is not available with every device. To see if it is available on a given virtual device, interrogate *work_out[10]* of the array returned by **v_opnwk** or **v_opnvwk**.

vst_height — VDI function (libvdi)

Reset graphics text height, in absolute values

```
#include <aesbind.h>
#include <vdibind.h>
```

void vst_height(*handle, newheight, charwidth, charheight, cellwidth, cellheight*)
int *handle, newheight, *charwidth, *charheight, *cellwidth, *cellheight*;

vst_height is a VDI routine that sets a new height for graphics text. Note that graphics text can be resized up to twice its original height; this limit is set to reduce the amount of jaggedness, or "aliasing", present in the characters. **vst_height** resets the characters into absolute values; these values can be either in normalized device coordinates (NDC) or raster coordinates (RC), depending on which the virtual device uses. On the high-resolution screen, the normal character height is 13 rasters, which can be increased up to 26 rasters.

The related function **vst_point** resets character height, but uses points rather than absolute values. Note that the current sizes of a character and a character cell can be obtained with the AES routine **graf_handle**. The number of text sizes supported by the virtual device is found in the variable *work_out[5]*, which is part of the array returned by the routines **v_opnwk** and **v_opnvwk**.

handle is the virtual device's VDI handle. *height* is the new height to which the characters are being set. Note that not every height is available; if the height requested is not available, the characters will be set to the next *smaller* size. *charheight* and *charwidth* are, respectively, height and width to which the characters were set; *cellheight* and *cellwidth* are, respectively, the height and width to which the character cell is set. Note that the difference in sizes between a character and its cell controls how much "white space" appears around each character.

Example

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

See Also

TOS, graf_handle, v_gtext, VDI, vst_point

vst_load_fonts — VDI function (libvdi)

Load fonts other than the standard font
#include <aesbind.h>
#include <vdibind.h>
int vst_load_fonts(*handle, reserved*) **int** *handle, font*;

vst_load_fonts is a VDI routine that loads a virtual device's non-standard fonts into memory. The new fonts must be specifically loaded for them to be used; this is done in order to save system memory that would otherwise be taken up by unused fonts. *handle* is the virtual device's VDI handle. *reserved* is reserved by GEM for future use; at present, it should be set to zero. **vst_load_fonts** returns the number of additional fonts loaded. The routine **vst_unload_fonts** should be used to free the memory given over the extra fonts once they are no longer needed.

See Also

TOS, v_gtext, VDI, vst_unload_fonts

Notes

This routine is not available in the ROM-resident VDI. It should not be used if the GDOS is not present in your edition of VDI.

vst_point — VDI function (libvdi)

Reset graphics text height, in printer's points
#include <aesbind.h>
#include <vdibind.h>
void vst_point(*handle, newheight, charwidth, charheight, cellwidth, cellheight*)
int *handle, newheight, *charwidth, *charheight, *cellwidth, *cellheight*;

vst_point is a VDI routine that sets a new height for graphics text. It resets the characters into printer's points; one point equals 1/72 of an inch. The related function **vst_height** resets character height, but uses absolute values rather than points.

The current sizes of a character and a character cell can be obtained with the AES routine **graf_handle**. The number of text sizes supported by the virtual device is found in the variable *work_out[5]*, which is part of the array returned by the routines **v_opnwk** and **v_opnvwk**.

handle is the virtual device's VDI handle. *height* is the new height to which the characters are being set. Note that not every height is available; if the height requested is not available, the characters will be set to the next *smaller* size. *charheight* and *charwidth* are, respectively, height and width to which the characters were set; *cellheight* and *cellwidth* are, respectively, the height and width to which the character cell is set. Note that the difference in sizes between a character and its cell controls how much "white space" appears around each character.

See Also

TOS, graf_handle, v_gtext, VDI, vst_height

vst_rotation — VDI function (libvdi)

Set angle at which graphic text is drawn
#include <aesbind.h>
#include <vdibind.h>
int vst_rotation(*handle, angle*) **int** *handle, angle*;

vst_rotation is a VDI routine that sets the angle at which graphics text is drawn. *handle* is the virtual device's VDI handle. *angle* is the angle at which the text is drawn, in tenths of a degree. On an imaginary clock, zero degrees is set at three o'clock, 90 degrees at noon, 180 degrees at nine o'clock, and 270 degrees at six o'clock. Not every angle is available on every device; therefore, **vst_rotation** returns the angle at which the text is actually drawn.

Example

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

See Also

TOS, v_gtext, VDI

Notes

This function is not available on every virtual device. To see if it is or not, interrogate entry 36 of the array *work_out[]*, which is returned by **v_opnwk** or **v_opnvwk**. Zero indicates no, and one indicates yes.

As of this writing, the Atari ST can rotate text only in 90-degree increments.

vst_unload_fonts — VDI function (libvdi)

Unload fonts

#include <aesbind.h>
#include <vdibind.h>
void vst_unload_fonts(*handle, reserved***) int** *handle, reserved***;**

vst_unload_fonts is a VDI routine that unloads extra fonts used in a VDI program. This routine should be used once there is no more need for the extra fonts, to free up memory given over to the extra fonts. *handle* is the virtual device's VDI handle. *reserved* is reserved for a future application, and should be set to zero.

See Also

TOS, v_gtext, VDI, vst_load_fonts

Notes

This routine uses the VDI's GDOS in its operation. It should not be used if the GDOS is not present in your edition of VDI.

vswr_mode — VDI function (libvdi)

Set the writing mode

#include <aesbind.h>
#include <vdibind.h>
int vswr_mode(*handle, mode***) int** *handle, mode***;**

vswr_mode is a VDI routine that sets the writing mode. *handle* is the device's VDI handle. *mode* indicates the writing mode of the device, as follows: one, replace; two, transparent; three, XOR (exclusive or); and four, reverse transparent. *Replace* mode simply replaces whatever is on the virtual device with the image being drawn. *Transparent* mode replaces all the zero (white) pixels on the device it is overlaying with ones (black), but does not affect black pixels that already exist on the screen. The effect is as if the image were drawn on a sheet of plastic that was then overlaid on the physical device. *Reverse transparent* mode is the same as transparent mode, except that it affects black pixels and ignores white ones. Finally, *XOR* mode draws an image that later can be cancelled out by reversing (or exclusive ORing it), moved elsewhere, and redrawn.

vswr_mode returns the mode set.

Example

For examples of this routine, see the entries for **v_circle** and **v_ellipse**.

See Also

TOS, VDI

Vsync — xbios function 37 (osbind.h)

Synchronize with the screen

#include <osbind.h>
#include <xbios.h>
void Vsync()

Vsync waits for the next picture return from the screen. It is used to synchronize the system's operation with that of the screen, for specialized effects.

Example

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

See Also

TOS, xbios

W

wc — Command

Count words, lines, and characters in files
wc [-clw] [*file...*]

wc counts words, lines, and characters in each *file* named. If no *file* is given, **wc** uses the standard input. If more than one *file* is given, **wc** also prints a total for all of the files.

A *word* is a string of characters surrounded by white space (blanks, tabs, or newlines).

Options control the printing of various counts:

-c Print a count of character.

-l Print a count of lines.

-w Print a count of words.

The default action is to print all counts.

See Also
commands

while — C keyword

Introduce a loop
while(*condition*)

while is a C keyword that introduces a conditional loop. *condition* is tested on reiteration of the loop, and the loop ends when *condition* is no longer satisfied. For example,

```
while (foo < 10)
```

introduces a loop that will continue until the variable **foo** is reset to ten or greater. Note that the statement

```
while (1)
```

will loop forever, unless interrupted by a **break**, **goto**, or **return** statement.

See Also
break, C keywords, C language, continue, do, for
The C Programming Language, page 56

while — Command

Execute a conditional loop
while *word1 word2*

while is a command built into the microshell, **msh**. It controls the operation of conditional loops: as long as **word1** executes successfully, **word2** is executed.

while is often used with the test commands **equal** and **not**.

See Also
commands, equal, if, is_set, msh, not

wildcards — Definition

Wildcards are characters that, under special circumstances, can represent a range of ASCII characters. Another name for them is "metacharacters". The wildcards available under **msh** are as follows:

? Match any one character.

***** Match any number of characters, or no characters at all.

[ ] A set of characters enclosed between '[' and ']' will match any one character of the set. Sets of characters may include ranges, such as **[a-z]** for all lower-case letters or **[0-9]** for all numerals.

/ Remove the special meaning of a wildcard.

See Also
egrep, Fsfirst, msh, patterns, pnmatch

wind_calc — AES function (libaes)

Calculate a window's rectangle
#include <aesbind.h>
int wind_calc(*type, kind, x, y, w, h, xptr, yptr, wptr, hptr*)
int *type, x, y, w, h,* ******xptr,* ******yptr,* ******wptr,* ******hptr*; **unsigned int** *kind*;

wind_calc is an AES routine that calculates the rectangle of a window. If *type* is zero, then *x*, *y*, *w*, and *h* specify the working rectangle of the window (the area within with text or icons are written), and **wind_calc** computes the total rectangle of the window (the entire area the window occupies on the screen). *x*, *y*, *w*, and *h* stand, respectively, for the X coordinate of the window's upper left-hand corner, the Y coordinate of the upper left-hand corner, the width, and the height. All are given in pixels.

If *type* is one, then *x*, *y*, *w*, and *h* specify the total rectangle of the window, and **wind_calc** computes the working rectangle of the window.

The computed rectangle is stored into the integers pointed to by *xptr*, *yptr*, *wptr*, and *hptr*.

kind lists the "gadgets" that appear in the window. The gadgets must be coded as follows:

0x001	NAME	title name
0x002	CLOSER	"close" bar
0x004	FULLER	"full" box
0x008	MOVER	"move" bar
0x010	INFO	information line
0x020	SIZER	"size" box
0x040	UPARROW	up arrow
0x080	DNARROW	down arrow
0x100	VSLIDE	vertical "slider"
0x200	LFARROW	left arrow
0x400	RTARROW	right arrow
0x800	HSLIDE	horizontal "slider"

If this list is not complete, then **wind_calc** will not calculate the rectangle correctly.

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

Example

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

See Also

AES, TOS, window

wind_close — AES function (libaes)

Close a window and preserve its handle

#include <aesbind.h>
int wind_close(*handle***) int** *handle*;

wind_close is an AES routine that closes a window. It preserves the window's handle, which was set by the routine **wind_create**, and all of its allocated resources. A closed window is not visible on the screen, but it can be reopened.

handle is the handle of the window to be opened. **wind_close** returns zero if an error occurred, and a number greater than zero if one did not.

Example

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

See Also

AES, TOS, wind_open, window

wind_create — AES function (libaes)

Create a window

#include <aesbind.h>

int wind_create(*kind, x, y, w, h***) unsigned int** *kind*; **int** *x, y, w, h*;

wind_create is an AES routine that creates a new window. *kind* indicates the elements of the window you wish to create, as follows:

0x001	NAME	title name
0x002	CLOSER	"close" bar
0x004	FULLER	"full" box
0x008	MOVER	"move" bar
0x010	INFO	information line
0x020	SIZER	"size" box
0x040	UPARROW	up arrow
0x080	DNARROW	down arrow
0x100	VSLIDE	vertical "slider"
0x200	LFARROW	left arrow
0x400	RTARROW	right arrow
0x800	HSLIDE	horizontal "slider"

For example, if you wanted to create a window that had only a title bar and an information bar, you would set *kind* to 0x11 (i.e., **NAME|INFO**).

x, *y*, *w*, and *h* give, respectively, the window's X coordinate, its Y coordinate, its width, and its height. The X and Y coordinates are always for the window's upper left-hand corner. These values are returned by the function **wind_get** when given the request **WF_FULLED**. These values are also often used to reset the size of the window when the "full" box is clicked, but nothing requires that this be done.

wind_create returns either the handle of the window it creates, or a negative number if it cannot create a window.

Example

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

See Also

AES, TOS, window

Notes

As of this writing, no more than six windows can be displayed at any given time.

wind_delete — AES function (libaes)

Delete a window and free its resources

#include <aesbind.h>
int wind_delete(*handle***) int** *handle*;

wind_delete is an AES routine that deletes a window and frees the resources allocated to it. *handle* is the handle of the window being deleted; this is returned by the routine **wind_create**. **wind_delete** returns zero if an error occurred, and a number greater than zero if one did not.

Example

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

See Also

AES, TOS, window

wind_find — AES function (libaes)

Determine if the mouse pointer is in a window
#include <aesbind.h>
int wind_find(*x, y***) int** *x, y*;

wind_find is an AES routine that determines if the mouse pointer is positioned over a window. *x* and *y* are the mouse pointer's X and Y coordinates; they can be obtained from the AES routine **graf_mkstate**. **wind_find** returns the handle of the window that the mouse pointer is within, or zero if the pointer is not within any window.

See Also

AES, TOS, window

wind_get — AES function (libaes)

Get information about a window
#include <aesbind.h>
int wind_get(*handle, flag, output1, output2, output3, output4***)**
int *handle, flag, *output1, *output2, *output3, *output4*;

wind_get is an AES routine that gets information about a window. *handle* is the handle of the window in question. A window's handle is first set by the routine **wind_create**, and is passed to your program via messages generated by the window manager.

flag tells **wind_get** just what information you want. Unless noted, **wind_get** will set the values for the X coordinate, Y coordinate, width, and height as follows:

4	WF_WORKXYWH	window's working area
5	WF_CURRXYWH	window's total area
6	WF_PREVXYWH	previous value of **WF_CURRXYWH**
7	WF_FULLXYWH	window's greatest possible size (set by **wind_create**)
8	WF_HSLIDE	*output1* set to relative position of horizontal slider (1-1,000; 1=leftmost)
9	WF_VSLIDE	*output1* set to relative position of vertical slider (1-1,000; 1=top)
10	WF_TOP	*output1* set to handle of topmost window
11	WF_FIRSTXYWH	First rectangle in rectangle list
12	WF_NEXTXYWH	Next rectangle in rectangle list
13	Reserved	
15	WF_HSLSIZE	*output1* set to size of horizontal slider relative to scroll bar; -1 is minimal (small box), 1-1,000 is relative size
16	WF_VSLSIZE	*output1* set to size of vertical slider relative to scroll bar; -1 is minimal (small box), 1-1,000 is relative size

When used with the macros named above, **wind_get** returns zero if an error occurred, and a number greater than zero if one did not. However, when used with the macros **WF_NAME**, **WF_INFO**, or **WF_NEWDESK**, all of which may be used with the function **wind_set**, **wind_get** always returns one, which indicates success, but it returns no useful information.

Example

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

See Also

AES, TOS, window

wind_open — AES function (libaes)

Open or reopen a window
#include <aesbind.h>
int wind_open(*handle, x, y, w, h***) int** *handle, x, y, w, h*;

wind_open is an AES routine that opens or reopens a window. *handle* is the window's handle, as set by **wind_create**. *x*, *y*, *w*, and *h* give, respectively, the X coordinate of the window to be opened, its Y coordinate, its width, and its height.

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

Example

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

See Also

AES, TOS, window

wind_set — AES function (libaes)

Set specified fields within the window
#include <aesbind.h>
int wind_set(*handle, flag, input1, input2, input3, input4***)**
int *handle, flag, input1, input2, input3, input4*;

wind_set is an AES routine that sets specific portions of a window. *handle* is the handle of the window to be altered; the handle is set by **wind_create**. The arguments *input1* through *input4* contain information you wish to insert into the window's definition. Note that not all four of these arguments are used with every task; those that are not used should be set to zero. *flag* indicates what aspect of the window you want to change, as follows:

2	WF_NAME	Point to new name for window: *input1* and *input2* give address passed as two-word pointer
3	WF_INFO	Point to new information line for window: *input1* and *input2* give address passed as two-word pointer
5	WF_CURRXYWH	Window's total area: *input1* gives X coordinate *input2* gives Y coordinate *input3* gives width *input4* gives height
8	WF_HSLIDE	*input1* set to relative position of horizontal slider (1-1,000; 1=leftmost)
9	WF_VSLIDE	*input1* set to relative position of vertical slider (1-1,000; 1=top)
10	WF_TOP	*handle* gives handle of window to be topmost
14	WF_NEWDESK	Address of new default GEM desktop: *input1* and *input2* give address passed as two-word pointer *input3* gives starting object in tree
15	WF_HSLSIZE	Set size of horizontal slider *input1* gives size of slider, 1-1,000
16	WF_VSLSIZE	Set size of vertical slider *input1* gives size of slider, 1-1,000

Ordinarily, **wind_set** returns zero if an error occurred, and a number greater than zero if it was successful. Note the following exceptions: When used with the macros **WF_PREVXYWH** or **WF_FULLXYWH**, which can be used with the function **wind_get**, **wind_set** returns success but does nothing. When the values used with **WF_VSLIDE** or **WF_HSLIDE** are out of bounds (i.e., greater than 1,000 or less than one), **wind_set** returns success and sets the slider to the closest legal value.

Example

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

See Also

AES, TOS, window

Notes

The window manager does not make its own copies of the strings specified by **WF_NAME** or **WF_INFO**. Thus, changing either of these strings after they are passed with **wind_set** will change the appearance of the window as well.

wind_update — AES function (libaes)

Lock or unlock a window
#include <aesbind.h>
int wind_update(*flag***) int** *flag*;

wind_update is an AES routine that locks or unlocks a window. This mechanism is provided to prevent a window's information from being updated while the screen is being redrawn and keep extraneous material from being drawn over the window as it is being redrawn. *flag* indicates what you want done, as follows:

0	END_UPDATE	The update is finished: unlock the window
1	BEG_UPDATE	Beginning an update: lock the window
2	END_MCNTRL	End mouse control through user: lock window
3	BEG_MCNTRL	Begin mouse control through user: unlock

From the time that

```
wind_update(BEG_MCTRL);
```

is called, the AES suspends mouse event handling until the matching

```
wind_update(END_MCTRL);
```

is called. **graf_mkstate** may be used to poll the mouse state while **BEG_MCTRL** is active. The menu bar, any displayed window gadgets, any mouse events requested via **envt_mouse**, **evnt_button**, or **evnt_multi** and any dialogues requested through **form_do** will be inactive until **END_MCTRL** is encountered. This is useful if a desk accessory wishes to perform a dialogue over another application's dialogue screen without activating the application dialogue.

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

Example

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

See Also

AES, TOS, window

window — Technical information

A window is an AES entity that is used to display information. It consists of a number of elements, as follows:

0x001	NAME	Title bar (across top of window)
0x002	CLOSE	Close box (upper left corner)
0x004	FULL	Full box (upper right corner)
0x008	MOVE	Move bar (across top of window)
0x010	INFO	Information bar (just below move bar)
0x020	SIZE	Size box (lower right corner)
0x040	UPARROW	Up arrow
0x080	DNARROW	Down arrow
0x100	VSLIDE	Vertical slider (right side)
0x200	LFARROW	Left arrow
0x400	RTARROW	Right arrow
0x800	HSLIDE	Horizontal slider (bottom of window)

The mnemonics used above are defined in the header file **gemdefs.h**. A window can be built with all of these elements, none of them, or any combination of them.

To create a window, use the function **wind_create**. You must tell this function what kind of window is being created (i.e., which elements compose the window), and the maximum size that the window can assume. It returns a integer *handle* for the window, which can be used to identify it to all other functions. The GEM desktop is always defined as window zero. The desktop is defined as being the entire screen minus the menu bar, and this definition is handy when you wish to expand a window to fill the entire screen.

Once a window is created, its attributes must set with the function **wind_set**. For example, if the window being created has a title bar, the text to be written there must be set before the window is displayed. If you fail to set the **WF_NAME** or **WF_INFO** after creating a window, random text may be written into those areas. All the rest of the attributes may be left to default values without hazard.

Once the attributes have been set, you can open the window with the function **vind_open**. You must pass it the handle of the window being created, and the dimensions to which you want it opened.

When a user clicks one of the elements of the window, such as the full box or the close box, the AES generates a *message* which can be picked up with the routines **evnt_mesag** or **evnt_multi**. Each message is eight **int**s (16 bytes) long. Word **0** is the type of message being sent. Word 1 is the handle of the application that

sends the message. Word 2 is the number of bytes in the message beyond the standard 16 bytes. Normally, a program that handles windows does not need to examine word 1 and 2. Words 3 through 7 give information specific to the message.

The following gives the types of messages that are relevant to handling windows:

WM_REDRAW (redraw a window) Word 3 gives the window's handle; words 4 through 7 give, respectively, the X coordinate, the Y coordinate, the width, and the height of the window to be drawn.

WM_TOPPED (make a window the topmost window) Word 3 gives the window's handle.

WM_CLOSED (close-window box clicked) Word 3 gives the window's handle.

WM_FULLED (full-window box clicked) Word 3 gives the window's handle.

WM_ARROWED (arrow or scroll bar clicked) Word 3 gives the window's handle. Word 4 gives the action requested, as follows:

0	Page up
1	Page down
2	Row up
3	Row down
4	Page left
5	Page right
6	Column left
7	Column right

WM_HSLID (horizontal slider moved) Word 3 gives the window's handle. Word 4 gives the slider's position: zero indicates the leftmost position, and 1,000 the rightmost.

WM_VSLID (vertical slider moved) Word 3 gives the window's handle. Word 4 gives the slider's position: zero indicates the lowest position, and 1,000 the highest.

WM_SIZED (window size altered) Word 3 gives the window's handle. Words 4 through 7 give, respectively, the X coordinate, the Y coordinate, the new width, and the new height.

WM_MOVED (window position altered) Word 3 gives the window's handle. Words 4 through 7 give, respectively, the new X coordinate, the new Y coordinate, the width, and the height.

When a message is received, the user is free either to react appropriately, or ignore the message. For example, if the message **WM_FULLED** is received, this indicates that the user has clicked the fulled box. The size of the box can then be changed with the **wind_set** routine, and another routine then invoked to redraw the screen and remove any debris left.

When you are done with a window, it should be closed with the **wind_close** function, and then removed with the function **wind_delete**. The window should be closed before deletion; otherwise, you may not be able to erase the old, left-over window from the screen.

Redrawing a window

The AES organizes the interior of each window into a set of non-overlapping rectangles, which it records in a list. If only one window appears on the screen, then its interior is described as one rectangle. If there is more than window on the screen, however, and one overlaps the other, then the interior of the lower window is described as a set of rectangles that outline the area being encroached. Therefore, redrawing the interior of a window requires that each rectangle be redrawn in turn; cycling through the rectangles and redrawing them is called "walking the rectangles". The dimensions of the first rectangle in the list can be obtained with the following call:

```
wind_get(handle, WF_FIRSTXYWH, &x, &y, &w, &h);
```

and the dimensions of the next rectangle with this call:

```
wind_get(handle, WF_NEXTXYWH, &x, &y, &w, &h);
```

AES returns zero for the width and height when it reaches the end of its rectangle list.

Example

The following example demonstrates a number of window routines. It draws two windows, one on top of the other. Each has a title bar, a full box, an exit box, sliders, and boxes for moving and changing the size of the window. Clicking the exit box closes the window; when all the windows are closed, the program ends. A redrawing function is included; it "walks the rectangles" to fill the interior of each window with a randomly defined mask. Clicking either slider redraws the mask.

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

#define MAXWIND     4
#define GRAIN 64
#define KIND (0XFFF&~INFO)/* Everything but INFO */

typedef struct { int x, y, w, h; } Box;

/* C note: a macro can fill an array as well as call a function */
#define elements(r) r.x, r.y, r.w, r.h
#define pointers(r) &r.x, &r.y, &r.w, &r.h

int aes_handle;
int vdi_handle;
```



```
int work_in[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2 };
int work_out[57];
int contrl[12], intin[128], intout[128], ptsin[128], ptsout[128];

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

main()
{
      register int w;     /* Current window handle */
      register int nw;    /* Current window count */
      int mb[8];    /* Message buffer */
      Box Full;     /* Fulled area of window */
      Box Prev;     /* Previous area of window */
      Box Work;     /* Working area of window */
      Box Temp;     /* Miscellaneous Box */
      int nowhere; /* Unused pointers point here */

      aes_handle = appl_init();
      graf_mouse(ARROW, &nowhere);

      /* alter size of mask for resolution of screen */
      if (Getrez() < 2)   /* i.e., screen is medium/low res */
            mask.ob_height = 200;
      if (Getrez() < 1)   /* i.e., screen is low res */
            mask.ob_width = 320;

      /* summon the VDI; set some ways it works */
      v_opnvwk(work_in, &vdi_handle, work_out);
      vswr_mode(vdi_handle, 1);  /* "replace" mode */
      vsf_perimeter(vdi_handle, 0);     /* no perimeter on shapes */
      vsf_interior(vdi_handle, 4);      /* user-defined fill */

      /* get size of work area for window 0 (desktop) */
      wind_get(0, WF_FULLXYWH, pointers(Full));

      /* mask the screen with grey */
      objc_draw(&mask, ROOT, 0, elements(Full));

      /* create, set, and open windows */
      for (nw = 0; nw < MAXWIND; nw += 1) {
            /* some variables used only in this block */
            static char *ordinal[] = { "st", "nd", "rd", "th" };
            static char t[MAXWIND][32];

            Temp = Full;
            if ((w = wind_create(KIND, elements(Temp))) < 0)
                  break;
            Temp.w /= 2; Temp.x += rand() % Temp.w;
            Temp.h /= 2; Temp.y += rand() % Temp.h;
            sprintf(t[nw], "%d%s window", nw+1, ordinal[nw>3 ? 3 : nw]);

            /* set the "gadgets" on each window */
            wind_set(w, WF_NAME, t[nw], 0, 0);

            /* "sliders" set to a random number */
            wind_set(w, WF_HSLIDE, rand() % 1000, 0, 0, 0);
            wind_set(w, WF_VSLIDE, rand() % 1000, 0, 0, 0);
```

```
        /* set slider sizes relative to scroll bar */
        wind_set(w, WF_HSLSIZE, Temp.w, 0, 0, 0);
        wind_set(w, WF_VSLSIZE, Temp.h, 0, 0, 0);

        /* draw window with "star wars" effect */
        graf_growbox(1,1,1,1, elements(Temp));
        wind_open(w, elements(Temp));
}

/* clicking a "gadget" generates a message */
while (nw > 0) {
        /* mb[3] gives handle of window */
        evnt_mesag(mb); w = mb[3];

        /* mb[0] holds the message */
        switch(mb[0]) {

        /* redraw window */
        case WM_REDRAW:
        redraw:
                /* get settings of window from window manager */
                wind_get(w, WF_HSLIDE, pointers(Temp)); Prev.x=Temp.x;
                wind_get(w, WF_VSLIDE, pointers(Temp)); Prev.y=Temp.x;
                wind_get(w, WF_HSLSIZE, pointers(Temp)); Prev.w=Temp.x;
                wind_get(w, WF_VSLSIZE, pointers(Temp)); Prev.h=Temp.x;
                wind_get(w, WF_WORKXYWH, pointers(Work));

                graf_mouse(M_OFF, &nowhere);      /* Hide mouse */
                wind_update(BEG_UPDATE);   /* Lock window */
                wind_get(w, WF_FIRSTXYWH, pointers(Temp));

                /* "walk the rectangles" to redraw window interior */
                while (Temp.w || Temp.h) {
                        if (Temp.w && Temp.h)
                                display(&Prev, &Work, &Temp);
                        wind_get(w, WF_NEXTXYWH, pointers(Temp));
                }

                wind_update(END_UPDATE);   /* Unlock window */
                graf_mouse(M_ON, &nowhere);       /* Show mouse */
                continue;

        /* make window the "top", or active, window */
        case WM_TOPPED:
                wind_set(w, WF_TOP, 0, 0, 0, 0);
                continue;

        /* have window fill the whole screen */
        case WM_FULLED:
                wind_get(w, WF_PREVXYWH, pointers(Prev));
                wind_get(w, WF_CURRXYWH, pointers(Temp));
                wind_get(w, WF_FULLXYWH, pointers(Full));
```



```
                if (rc_equal(&Prev, &Full))
                        continue;
                wind_set(w, WF_CURRXYWH, elements(Prev));
        } else {
                wind_set(w, WF_CURRXYWH, elements(Full));
        }

resize:
        wind_get(w, WF_WORKXYWH, pointers(Temp));
        wind_set(w, WF_HSLSIZE, Temp.w, 0, 0, 0);
        wind_set(w, WF_VSLSIZE, Temp.h, 0, 0, 0);
        continue;

/* resize window */
case WM_SIZED:
        Temp = *(Box *)(mb+4);

        if (w & 1) {
                wind_calc(1, KIND, elements(Temp),
                        pointers(Temp));
                Temp.w -= Temp.w % GRAIN;
                if (Temp.w < 2*GRAIN) Temp.w = 2*GRAIN;
                Temp.h -= Temp.h % GRAIN;
                if (Temp.h < 2*GRAIN) Temp.h = 2*GRAIN;
                wind_calc(0, KIND, elements(Temp),
                        pointers(Temp));

        }
        wind_get(w, WF_CURRXYWH, pointers(Prev));
        if (rc_equal(&Temp, &Prev))
                continue;
        wind_set(w, WF_CURRXYWH, elements(Temp));
        goto resize;

/* window moved on screen */
case WM_MOVED:
        wind_set(w, WF_CURRXYWH, mb[4], mb[5], mb[6], mb[7]);
        continue;

/* horizontal slider clicked */
case WM_HSLID:
        wind_get(w, WF_HSLIDE, pointers(Temp));
        if (Temp.x == mb[4])
                continue;
        wind_set(w, WF_HSLIDE, mb[4], 0, 0, 0);
        goto redraw;

/* vertical slider clicked */
case WM_VSLID:
        wind_get(w, WF_VSLIDE, pointers(Temp));
        if (Temp.x == mb[4])
                continue;
        wind_set(w, WF_VSLIDE, mb[4], 0, 0, 0);
        goto redraw;
```

```
		/* arrow or scroll bar clicked */
		case WM_ARROWED:
		{	/* Note parens: limits variables to this block */
			static int dc[] = { -5, 5, -1, 1 };
			register int t;

			/*
			 * mb[4] holds action: 0=page up, 1=page down,
			 * 2=row up, 3= row down, 4=page left, 5=page
			 * right, 6=column left, 7=column right
			 */
			t = mb[4];

			if (t <= 3) {
				wind_get(w, WF_VSLIDE, pointers(Temp));
				t = dc[t] + Temp.x;
				if (t > 1000) t = 1000;
				if (t < 0) t = 0;
				if (t == Temp.x)
					continue;
				wind_set(w, WF_VSLIDE, t, 0, 0, 0);

			} else if (t <= 7) {
				wind_get(w, WF_HSLIDE, pointers(Temp));
				t = dc[t-4] + Temp.x;
				if (t > 1000) t = 1000;
				if (t < 0) t = 0;
				if (t == Temp.x)
					continue;
				wind_set(w, WF_HSLIDE, t, 0, 0, 0);
			} else
				continue;
			goto redraw;
		}

		/* close window */
		case WM_CLOSED:
			wind_get(w, WF_CURRXYWH, pointers(Temp));
			wind_close(w);
			graf_shrinkbox(1,1,1,1, elements(Temp));
			wind_delete(w);
			nw -= 1;
			continue;

		/* unanticipated message */
		default:
			alertf(1, "[0][Message %u received][Okay]", mb[0]);
			continue;
		}
	}

	/* close everything, tidy up, exit gracefully */
	v_clsvwk(vdi_handle);
	appl_exit();
	exit(0);
}
```



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

/* redraw the screen */
display(lp, wp, rp)
Box *lp, *wp, *rp;
/* lp == logical page offsets and sizes 0 .. 999 */
/* wp == working area of window */
/* rp == clipped Box to fill */
{
	Box T; /* VDI temporary box */
	Box U; /* Second temporary box */
	int pattern[16];	/* User-defined pattern */
	register int x, y;	/* Working-area scanner */

	T = *rp; T.w += T.x-1; T.h += T.y-1;

	/* set clipping rectangle */
	vs_clip(vdi_handle, 1, &T);

	for (x = 0; x < wp->w; x += GRAIN)
	    for (y = 0; y < wp->h; y += GRAIN) {
		T.x = wp->x + x; T.w = GRAIN;
		T.y = wp->y + y; T.h = GRAIN;
		U = *rp;

		if ( ! rc_intersect(&T, &U))
			continue;

		/* fill window with randomly generated pattern */
		make_pattern(lp->x+(x/GRAIN), lp->y+(y/GRAIN), pattern);
		vsf_udpat(vdi_handle, pattern, 1);
		T.w += T.x-1; T.h += T.y-1;
		v_bar(vdi_handle, &T);
	    }
}

/* create a pattern for filling each window */
make_pattern(x, y, texture)
register unsigned x, y, *texture;
{
	register unsigned d, z, dz;

	/* fill char array with prime numbers */
	static unsigned char dzs[] = {
		1,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,
		1,67,71,73,79,83,89,97,101,103,107,109,113,127,
		31,137,139,149,151,157,163,167,173,179,181,191,
		93,197,199,211,223,227,229,233,239,241,251
	};
```

```
        /* randomly select a prime number */
        srand(y*x);
        dz = dzs[rand() % sizeof(dzs)];

        /* perform magic with primes
         * to tile interior of window
         */
        d = x % 100;
        x /= 100;
        if (x & 1) d = 100 - d;
        d = (d * 256L) / 100;

        for (y = 0; y < 16; y += 1)
                texture[y] = 0;

        for (; d > 0; d -= 1) {
                for (z = rand(); ; z += dz) {
                        x = 1 << ((z >> 4) & 15);
                        y = z & 15;
                        if ((texture[y] & x) == 0)
                                break;
                }
                texture[y] ^= x;
        }
}
```

See Also

AES, gem, gemdefs.h, object, TOS

write — UNIX system call (libc)

Write to a file

int write(*fd***,** *buffer***,** *n***)**
int *fd***; char *****buffer***; int** *n***;**

write writes *n* bytes of data, beginning at address *buffer*, into the file *fd*. Writing begins at the current write position, as set by the last call to either **write** or **lseek**. **write** advances the position of the file pointer by the number of characters written.

Example

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

See Also

STDIO, UNIX routines

Diagnostics

write returns -1 if an error occurred before the **write** operation commenced, such as a bad file descriptor *fd* or invalid *buffer* pointer. Otherwise, it returns the number of bytes actually written. It should be considered an error if this number is not the same as *n*.

Notes

write is a low-level call that passes data directly to TOS. It should not be intermixed with high-level calls, such as **fread**, **fwrite**, or **fopen** without care.

X

xbios — TOS function

Call a routine from the extended TOS BIOS
#include <osbind.h>
extern long xbios(*n, f1, f2 ... fx***);**

xbios allows you to call a routine directly in the Atari extended ROM BIOS, by triggering trap 14. *n* is the number of the routine, and *f1* through *fx* are the parameter numbers to be used with **xbios**. In most circumstances, it is unnecessary to call **xbios**, for the header file **osbind.h** defines a number of functions that use it directly. The constants and structures used by these functions are contained in the header file **xbios.h**.

The following are the **xbios** functions:

24	**Bioskeys**	restore the default keyboard table
64	**Blitmode**	get/set blitter mode
21	**Cursconf**	set the cursor's configuration
32	**Dosound**	pass data to the sound daemon
10	**Flopfmt**	format a floppy disk
8	**Floprd**	read a floppy disk
19	**Flopver**	verify a floppy disk
9	**Flopwr**	write to a floppy disk
4	**Getrez**	read the current screen resolution
23	**Gettime**	read the current system time
28	**Giaccess**	write to the GI sound chip registers
25	**Ikbdws**	send commands to the intelligent keyboard
0	**Initmous**	initialize the mouse
14	**Iorec**	get a pointer to the serial device input record
26	**Jdisint**	disable an interrupt
27	**Jenabint**	enable an interrupt
16	**Keytbl**	create a new keyboard table
34	**Kbdvbase**	get a pointer to a set of keyboard routines
35	**Kbrate**	set the keyboard's repeat rate
3	**Logbase**	get the screen's logical base
13	**Mfpint**	initialize interrupt routine in multi-function port
12	**Midiws**	send string to musical instrument digital interface
29	**Offgibit**	turn off a bit in the sound chip's A port
30	**Ongibit**	turn on a bit in the sound chip's A port
2	**Physbase**	get the physical base of the screen
18	**Protobt**	create a prototype boot routine
36	**Prtblk**	print a dump of the screen
39	**Puntaes**	make AES go away
17	**Random**	generate a pseudo-random number
15	**Rsconf**	configure the RS-232 (serial) port

20	**Scrdmp**	print a dump of the screen
7	**Setcolor**	set a color
5	**Setscreen**	set the screen parameters
6	**Setpallete**	set the color pallette
33	**Setprt**	configure the printer port
22	**Settime**	set the system time
38	**Supexec**	run a function under supervisor mode
37	**Vsync**	synchronize with the screen refresh
31	**Xbtimer**	initialize a timer on the multi-function port

See Also

osbind.h, TOS

Notes

No **xbios** function checks device numbers. Passing an invalid device number to one will crash the system.

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

All **xbios** I/O routines, including file I/O, are unbuffered. Combining them with buffered I/O routines, such as those in the STDIO library, will lead at best to unpredictable results.

xbios.h — Header file

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

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

See Also

bios.h, header file, TOS, xbios

Xbtimer — xbios function 31 (osbind.h)

Initialize the MFP timer
#include <osbind.h>
#include <xbios.h>
void Xbtimer(*timer, control, data, buffer***) int** *timer, control,data*; **char** **buffer*;

Xbtimer permits you to initialize one of the 68901 chip's timers. *timer* is a value from zero through three, which corresponds to timers A through D, respectively. Timer A handles user applications; timer B handles graphics; timer C is the system

timer; and timer D sets the baud rate for the RS-232 port. *control* sets the timer's control register, and *data* is a byte of data to be written into the timer's data register. *buffer* points to an interrupt handler.

Example

This example sets the timer to a value (an ambiguous time) and loops that many times while incrementing a memory location, then does the same with a register variable.

This program calls the routine **setrte**, which is included with Mark Williams C in the file **setrte.s**. To compile, use the command line

```
cc -o Xbtimer.prg Xbtimer.c setrte.s
```

The following gives the text of **Xbtimer.c**:

```
#include <osbind.h>
unsigned long a_lock;

void timetick()                 /* the clock interrupt handler */
{
        setrte();               /* make this an int return */
        a_lock++;               /* increment the lock */
        Xbtimer(0, 0, 0, 0L);   /* disable the timer */
}

main()
{
        unsigned long tick=0;           /* slow counter */
        register unsigned long tock=0;  /* fast counter */

        a_lock = 0;                     /* clear the timer flag */
        Xbtimer(0, 7, 100, timetick);   /* declare the timer */
        while ( a_lock == 0 )           /* do the memory loop */
                tick++;                 /* print statistics */
        printf("Using memory, loop executed %ld times.\n", tick);

        a_lock = 0;                     /* clear flag */
        Xbtimer(0, 7, 100, timetick);   /* declare timer */
        while ( a_lock == 0 )           /* do register loop */
                tock++;                 /* then print stats. */
        printf("Using register, loop executed %ld times.\n", tock);
        Pterm0();                       /* exit */
}
```

See Also

TOS, xbios

XOFF — Manifest constant

XOFF is a flow-control signal used with asynchronous communications. Usually, it consists of a **<ctrl-S>** character (octal 023). It is sent by the receiving device when its asynchronous buffer is nearly full, or has reached the "high-water mark". Note that when XOFF is used to help control data transmission, binary files cannot be transmitted.

See Also

ASCII, XON

XON — Manifest constant

XON is a flow-control signal used with asynchronous communications. Usually, it consists of a **<ctrl-Q>** character (octal 021). It is sent by the receiving device when its asynchronous buffer is nearly empty, or has reached the "low-water mark". Note that when XON is used to help control data transmission, binary files cannot be transmitted.

See Also

ASCII, XOFF

Permuted Listing of Lexicon Entries