C/C++ Language and Run-Time Reference

C/C++ Topics

- Language
- Run-Time Routines
- Language and Run-Time Examples
- Predefined Identifiers
- Preprocessor Instructions

Other Reference Topics

- iostream Reference
- Run-Time Programming
- Character Codes and Scan Codes

Close

Copyright Notice

Close

Microsoft C/C++ Language Help

MSCXX.HLP

Version 1.00

Close

Language Reference Topics

- Keywords
- Data Types
- Modifiers
- Statements
- Operators
- Operator Precedence
- Escape Sequences
- main Function
- printf Formatting
- Directives
- Preprocessor Operators
- Pragmas
- Predefined Macros
- Header (.H) Files
- Global Variables
- Constants
- Types
- ANSI Character Set
- Key Scan Codes

C/C++ Language Topics

C/C++ Run-Time Routines

C/C++ Run-Time Routines

- Buffer Manipulation
- Character Classification and Conversion
- Data Conversion
- Directory Control
- File Handling
- Graphics (Low level and character font)
- Graphics (Presentation)
- Input and Output
- Internationalization
- Math
- Memory Allocation
- Process and Environment Control
- QuickWin
- Searching and Sorting
- String Manipulation
- System Calls: BIOS Interface
- System Calls: DOS Interface
- Time
- Variable-Length Argument Lists
- Virtual Memory Allocation

C/C++ Preprocessor Instructions

- Directives
- Preprocessor Operators
- Pragmas
- Predefined Macros

Predefined Identifiers

- Header (.H) Files
- Global Variables
- Constants
- Types

Character Codes and Scan Codes

Run-Time Programming Topics

General Information

Programming Topics

Run-Time Programming Topics

General Information

Programming Topics

Example Programs

A _ D

ALARM.C
ANALYZE.C
ANIMATE.C
ARGS.C
ASCII.C
ASSERT.C
ATEXIT.C
ATONUM.C
BARCOL.C
BEEP.C
BESSEL.C
BUFTEST.C
CABS.C
CASE.C
CGAPAL.C
CHMOD1.C
CHMOD2.C
CMPSTR.C
COM.C
COMMIT.C
COPROC.C
COPY2.C
CPYSTR.C
CURSOR.C
DCOMMIT.C
DIRECT.C
DISK.C
DOSMEM.C
DRIVES.C

E _ I

ENVIRON.C
ERROR.C
EXEC.C
EXTDIR.C
EXTERR.C
FCVT.C
FIGURE.C
FILL.C
FINDSTR.C
FONTS.C
FREECT.C
FULL.C
FUNGET.C
FWOPEN.C
GEDIT.C
GETCH.C
HALLOC.C
HANDLER.CPP
HARDERR.C
HEAPBASE.C
HEAPWALK.C
HEXDUMP.C
HMANAGE.C
INTMATH.C
IOTEST.C
IS.C

K _ R

KBHIT.C
LOCK.C
MATH.C
MBLEN.CPP
MBTOWC.CPP
MODES.C
MOVEMEM.C
RECORDS2.C
KEYBRD.C
MATHERR.C
MBSTOWCS.CPP
MKFPSTR.C
MORE.C
MSB.C

MSERIES.C NUMTOA.C
NULLFILE.C PAGER.C
PAGE.C PGPAL.C
PALETTE.C QSORT.C
PRINTF.C RECORDS1.C
REALLOC.C ROTATE.C

S _ V

SCANF.C SCAT.C
SCROLL.C SEEK.C
SETROWS.C SETSTR.C
SETTIME.C SIEVE.C
SIGFP.C SIGNAL.C
SPAWN.C STAR.C
STRTONUM.C SWAB.C
SYSCALL.C SYSINFO.C
TABLE.C TEMPNAME.C
TEXT.C TIMES.C
TOKEN.C TRIG.C
TYPEIT.C UNGET.C
VARARG.C VCNT.C
VLOAD.C VLOCK.C
VRSIZE.C

W

WABOUT.C WMENUCLK.C
WCLOSE.C WOPEN.C
WCSTOMBS.CPP WPRINTF.C
WCTOMB.CPP WRAP.C
WGETFOC.C WSETFOC.C
WGETSIZE.C WSETSIZE.C
WGSCRBUF.C WSSCRBUF.C
WINDOW.C WYIELD.C

Buffer Manipulation Routines

The buffer-manipulation routines are useful for working with areas of memory on a byte-by-byte basis.

Routine	Description
_memccpy, _fmemccpy	Copy characters from one buffer to another until a given character or a given number of characters has been copied
memchr, _fmemchr	Return a pointer to the first occurrence, within a specified number of characters, of a given character in the buffer
memcmp, _fmemcmp	Compare a specified number of characters from two buffers
memcpy, _fmemcpy	Copy a specified number of characters from one buffer to another
_memicmp, _fmemicmp	Compare a specified number of characters from two buffers without regard to the case of the letters (uppercase and lowercase treated as equivalent)
memmove, _fmemmove	Copy a specified number of characters from one buffer to another
memset, _fmemset	Use a given character to initialize a specified number of bytes in the buffer
_swab	Swaps bytes of data and stores them at the specified location

Character Classification and Conversion Routines

The character classification and conversion routines enable you to test individual characters in a variety of ways and to convert between uppercase and lowercase characters.

Routine	Description
isalnum	Tests for alphanumeric character
isalpha	Tests for alphabetic character
__isascii	Tests for ASCII character
iscntrl	Tests for control character
__iscsym	Tests for letter, underscore, or digit
__iscsymf	Tests for letter or underscore
isdigit	Tests for decimal digit

isgraph	Tests for printable character except space
islower	Tests for lowercase character
isprint	Tests for printable character
ispunct	Tests for punctuation character
isspace	Tests for white-space character
isupper	Tests for uppercase character
isxdigit	Tests for hexadecimal digit
__toascii	Converts character to ASCII code
__tolower	Tests character and converts to lowercase if uppercase
_tolower	Converts character to lowercase (unconditional)
toupper	Tests character and converts to uppercase if lowercase
_toupper	Converts character to uppercase (unconditional)

Data Conversion Routines

The data-conversion routines convert numbers to strings of ASCII characters and vice versa.

Routine	Description
abs	Finds absolute value of integer
atof	Converts string to float
atoi	Converts string to int
atol	Converts string to long
_atold	Converts string to long double
_ecvt	Converts double to string
_fcvt	Converts floating-point number to string
_gcvt	Converts floating-point number to string and stores it in a buffer
_itoa	Converts int to string
labs	Finds absolute value of long integer
_ltoa	Converts long to string
strtod	Converts string to double
strtol	Converts string to a long integer
_strtold	Converts string to long double
strtoul	Converts string to an unsigned long integer
_ultoa	Converts unsigned long to string

Directory Control Routines

The directory-control routines let a program access, modify, and obtain information about the directory

structure.

Routine	Description
_chdir	Changes current working directory
_chdrive	Changes current drive
_getcwd	Gets current working directory for the default drive
_getdcwd	Gets current working directory for the specified drive
_getdrive	Gets current working directory
_mkdir	Makes a new directory
_rmdir	Removes a directory
_searchenv	Searches for a given file on specified paths

File Handling Routines

The file-handling routines let you create, manipulate, and delete files. They also set and check file-access permissions.

Routine	Description
_access	Checks file-permission setting
_chmod	Changes file-permission setting
_chsize	Changes file size
_filelength	Gets file length
_fstat	Gets file-status information on handle
_fullpath	Makes an absolute path name from a relative pathname
_isatty	Checks for character device
_locking	Locks areas of file (available with MS-DOS versions 3.0and later)
_makepath	Merges path-name components into a single, full path name
_mktemp	Creates unique filename
remove	Deletes file
rename	Renames file
_setmode	Sets file-translation mode
_splitpath	Splits a path name into component pieces
_stat	Gets file-status information on named file
_umask	Sets default-permission mask
_unlink	Deletes file

Internationalization Routines

Internationalization routines are useful for creating different versions of a program for international markets.

Routine	Description
localeconv	Sets a structure with appropriate values for formatting numeric quantities
mblen, _fmblen	Get the length and determine the validity of a multibyte character
mbstowcs, _fmbstowcs	Convert a sequence of multibyte characters to a corresponding sequence of wide characters
mbtowc, _fmbtowc	Convert a multibyte character to the corresponding wide character
setlocale	Selects the appropriate locale for the program
strcoll	Compares strings using locale-specific information
strftime	Formats a date and time string
strxfrm	Transforms a string based on locale-specific information
wcstombs, _fwcstombs	Convert a sequence of wide characters to a corresponding sequence of multibyte characters
wctomb, fwctomb	Convert a wide character to the corresponding multibyte character

Math Routines

The math routines allow you to perform common mathematical calculations.

Routine	Description
acos, acosl	Calculate the arccosine
asin, asinl	Calculate the arcsine
atan, atanl	Calculate the arctangent
atan2, atan2l	Calculate the arctangent
Bessel	Calculates Bessel functions
_cabs, _cabsl	Find the absolute value of a complex number
ceil, ceill	Find the integer ceiling
_clear87	Gets and clears the floating-point status word
_control87	Gets the old floating-point control word and sets a new control-word value

cos, cosl	Calculate the cosine
cosh, coshl	Calculate the hyperbolic cosine
_dieeetomsbin	Converts IEEE double-precision number to Microsoft (MS) binary format
div	Divides one integer by another, returning the quotient and remainder
_dmsbintoieee	Converts Microsoft binary double-precision number to IEEE format
exp, expl	Calculate the exponential function
fabs, fabsl	Find the absolute value
_fieeetomsbin	Converts IEEE single-precision number to Microsoft binary format
floor, floorl	Find the largest integer less than or equal to the argument
fmod, fmodl	Find the floating-point remainder
_fmsbintoieee	Converts Microsoft binary single-precision number to IEEE format
_fpreset	Reinitializes the floating-point-math package
frexp, frexpl	Calculate an exponential value
_hypot, _hypotl	Calculate the hypotenuse of a right triangle
ldexp, ldexpl	Calculate the product of the argument and 2*exp*
ldiv	Divides one long integer by another, returning the quotient and remainder
log, logl	Calculate the natural logarithm
log10, log10l	Calculate the base-10 logarithm
_lrotl, _lrotr	Shift an unsigned long int item left (_lrotl) or right (_lrotr)
_matherr, _matherrl	Handle math errors
__max, __min	Return the larger or smaller of two values
modf, modfl	Split the argument into integer and fractional parts

pow, powl	Calculate a value raised to a power
rand	Gets a pseudorandom number
_rotl, _rotr	Shift an unsigned int item left (_rotl) or right (_rotr)
sin, sinl	Calculate the sine
sinh, sinhl	Calculate the hyperbolic sine
sqrt, sqrtl	Find the square root
srand	Initializes a pseudo-random series
_status87	Gets the floating-point status word
tan, tanl	Calculate the tangent
tanh, tanhl	Calculate the hyperbolic tangent

Memory Allocation Routines

The memory-allocation routines allow you to allocate, free, and reallocate blocks of memory.

Routine	Description
_alloca	Allocates a block of memory from the program's stack
_bfreeseg	Frees a based heap
_bheapseg	Allocates a based heap
calloc, _bcalloc, _fcalloc, _ncalloc	Allocate storage for an array
_expand, _bexpand, _fexpand, _nexpand	Expand or shrink a block of memory without moving its location
free, _bfree, _ffree, _nfree	Free an allocated block
_freect	Returns approximate number of items of given size that could be allocated in the near heap
_halloc	Allocates storage for huge array
_heapadd, _bheapadd	Add memory to a heap
_heapchk, _bheapchk, _fheapchk, _nheapchk	Check a heap for consistency
_heapmin, _bheapmin,	Release unused memory in a heap

_fheapmin, _nheapmin	
_heapset, _bheapset, _fheapset, _nheapset	Fill free heap entries with a specified value
_heapwalk, _bheapwalk, _fheapwalk, _nheapwalk	Return information about each entry in a heap
_hfree	Frees a block allocated by _halloc
malloc, _bmalloc, _fmalloc, _nmalloc	Allocate a block of memory
_memavl	Returns approximate number of bytes available for allocation in the near heap
_memmax	Returns size of largest contiguous free block in the near heap
_msize, _bmsize, _fmsize, _nmsize	Return size of an allocated block
realloc, _brealloc, _frealloc, _nrealloc	Reallocate a block to a new size
_set_new_handler, _set_bnew_handler, _set_fnew_handler, _set_hnew_handler, _set_nnew_handler	Enable an error-handling mechanism
_stackavail	Returns size of stack space available for allocation with _alloca

Process and Environment Control Routines

The process-control routines allow you to start, stop, and manage processes from within a program.

Routine	Description
abort	Aborts a process without flushing buffers or calling functions registered by atexit and _onexit
assert	Tests for logic error
atexit	Schedules routines for execution at program termination
_cexit	Performs the exit termination procedures (such as flushing buffers) and returns control to the calling program

_c_exit	Performs the _exit termination procedures and returns control to the calling program
_execl	Executes child process with argument list
_execle	Executes child process with argument list and given environment
_execlp	Executes child process using PATH variable and argument list
_execlpe	Executes child process using PATH variable, given environment, and argument list
_execv	Executes child process with argument array
_execve	Executes child process with argument array and given environment
_execvp	Executes child process using PATH variable and argument array
_execvpe	Executes child process using PATH variable, given environment, and argument array
exit	Calls functions registered by atexit and _onexit, then flushes all buffers and closes all open files before terminating the process
_exit	Terminates process without processing atexit or _onexit functions or flushing buffers
_fatexit	Schedules routines for execution at program termination (memory-model independent)
_fonexit	Schedules routines for execution at program termination (memory-model independent)
getenv	Gets the value of an environment variable
_getpid	Gets process ID number
longjmp	Restores a saved stack environment
_onexit	Schedules routines for execution at program termination
perror	Prints error message
_putenv	Adds or changes the value of an

	environment variable
raise	Sends a signal to the calling process
setjmp	Saves a stack environment
signal	Handles an interrupt signal
_spawnl	Executes child process with argument list
_spawnle	Executes child process with argument list and given environment
_spawnlp	Executes child process using PATH variable and argument list
_spawnlpe	Executes child process using PATH variable, given environment, and argument list
_spawnv	Executes child process with argument array
_spawnve	Executes child process with argument array and given environment
_spawnvp	Executes child process using PATH variable and argument array
_spawnvpe	Executes child process using PATH variable, given environment, and argument array
system	Executes an operating-system command

QuickWin Routines

The QuickWin functions make it possible to compile non-Windows MS-DOS programs as simple text-only Windows applications.

Routine	Description
_fwopen	Opens a new window stream
_inchar	Reads a single character from the keyboard
_wabout	Sets the string that appears in the About dialog box
_wclose	Closes a window's file handle
_wgclose	Closes an existing graphics child window
_wgetexit	Gets a QuickWin program's current exit behavior setting
_wgetfocus	Returns a file handle to the window with the input focus
_wgetscreenbuf	Gets a window's current screen-buffer size

_wgetsize	Gets a window's current size and position on the screen
_wggetactive	Returns a file handle to the active graphics child window
_wgopen	Opens a graphics child window
_wgsetactive	Makes a graphics child window active
_wmenuclick	Chooses a menu command
_wopen	Opens a window, returning a file handle to it
_wsetexit	Sets the way a QuickWin program behaves when exit is called
_wsetfocus	Makes a window the active window (sets its focus)
_wsetscreenbuf	Sets a window's screen-buffer size
_wsetsize	Sets a window's size and position on the screen
_wyield	Yields processor time to Windows for queue servicing

Searching and Sorting Routines

Search and sort routines provide binary-search, linear-search, and quick-sort capabilities.

Routine	Description
bsearch	Performs binary search
_lfind	Performs linear search for given value
_lsearch	Performs linear search for given value, which is added to array if not found
qsort	Performs quick sort

String Manipulation Routines

The string manipulation routines allow you to compare strings, copy them, search for strings and characters, and perform various other operations.

Routine	Description
strcat, _fstrcat	Append one string to another
strchr, _fstrchr	Find first occurrence of a given character in a string
strcmp, _fstrcmp	Compare two strings
strcpy, _fstrcpy	Copy one string to another
strcspn,	Find first occurrence of a

_fstrcspn	character from a given character set in a string
_strdup, _fstrdup, _nstrdup	Duplicate a string
strerror	Maps an error number to a message string
_strerror	Maps a user-defined error message to a string
_stricmp, _fstricmp	Compare two strings without regard to case
strlen, _fstrlen	Find length of string
_strlwr, _fstrlwr	Convert string to lowercase
strncat, _fstrncat	Append characters of a string
strncmp, _fstrncmp	Compare characters of two strings
strncpy, _fstrncpy	Copy characters of one string to another
_strnicmp, _fstrnicmp	Compare characters of two strings without regard to case
_strnset, _fstrnset	Set characters of a string to a given character
strpbrk, _fstrpbrk	Find first occurrence of a character from one string in another
strrchr, _fstrrchr	Find last occurrence of a given character in string
_strrev, _fstrrev	Reverse a string
_strset, _fstrset	Set all characters of a string to a given character
strspn, _fstrspn	Find first substring from a given character set in a string
strstr, _fstrstr	Find first occurrence of a given string in another string
strtok, _fstrtok	Find next token in a string
_strupr, _fstrupr	Convert a string to uppercase

System Calls: BIOS

The routines in this category provide direct access to the BIOS interrupt services.

Routine	Description
_bios_disk	Issues service requests for both hard and floppy disks, using INT 0x13

_bios_equiplist	Performs an equipment check, using INT 0x11
_bios_keybrd	Provides access to keyboard services, using INT 0x16
_bios_memsize	Obtains information about available memory, using INT0x12
_bios_printer	Performs printer output services, using INT 0x17
_bios_serialcom	Performs serial communications tasks, using INT 0x14
_bios_timeofday	Provides access to system clock, using INT 0x1A

System Calls: MS-DOS Interface

These routines are implemented as functions and declared in DOS.H.

Routine	Description
_bdos	Invokes MS-DOS system call; uses only DX and AL registers
_chain_intr	Chains one interrupt handler to another
_disable	Disables interrupts
_dos_allocmem	Allocates a block of memory, using MS-DOS system call 0x48
_dos_close	Closes a file, using MS-DOS system call 0x3E
_dos_commit	Flushes a file to disk, using MS-DOS system call 0x68
_dos_creat	Creates a new file and erases any existing file having the same name, using MS-DOS system call 0x3C
_dos_creatnew	Creates a new file and returns an error if a file having the same name exists, using MS-DOS system call 0x5B
_dos_findfirst	Finds first occurrence of a given file, using MS-DOS system call 0x4E
_dos_findnext	Finds subsequent occurrences of a given file, using MS-DOS system call 0x4F
_dos_freemem	Frees a block of memory, using MS-DOS system call

	0x49
_dos_getdate	Gets the system date, using MS-DOS system call 0x2A
_dos_getdiskfree	Gets information on a disk volume, using MS-DOS system call 0x36
_dos_getdrive	Gets the current default drive, using MS-DOS system call 0x19
_dos_getfileattr	Gets current attributes of a file or directory, using MS-DOS system call 0x43
_dos_getftime	Gets the date and time a file was last written, using MS-DOS system call 0x57
_dos_gettime	Gets the current system time, using MS-DOS system call 0x2C
_dos_getvect	Gets the current value of a specified interrupt vector, using MS-DOS system call 0x35
_dos_keep	Installs terminate-and-stay-resident (TSR) programs using MS-DOS system call 0x31
_dos_lock	Synchronizes files and records in a mutitasking environment or network
_dos_open	Opens an existing file, using MS-DOS system call 0x3D
_dos_read	Reads a file, using MS-DOS system call 0x3F
_dos_seek	Moves a file pointer to a specified position in an open file, using MS-DOS system call 0x42
_dos_setblock	Changes the size of a previously allocated block, using MS-DOS system call 0x4A
_dos_setdate	Sets the current system date, using MS-DOS system call 0x2B
_dos_setdrive	Sets the default disk drive, using MS-DOS system call 0x0E
_dos_setfileattr	Sets the current attributes of a file, using MS-DOS system call 0x43
_dos_setftime	Sets the date and time that the specified file was last

	written, using MS-DOS system call 0x57
_dos_settime	Sets the system time, using MS-DOS system call 0x2D
_dos_setvect	Sets a new value for the specified interrupt vector, using MS-DOS system call 0x25
_dos_write	Sends output to a file, using MS-DOS system call 0x40
_dosexterr	Obtains in-depth error information from MS-DOS system call 0x59
_enable	Enables interrupts
_FP_OFF	Returns offset portion of a far pointer
_FP_SEG	Returns segment portion of a far pointer
_harderr	Establishes a hardware error handler
_hardresume	Returns to MS-DOS after a hardware error
_hardretn	Returns to the application after a hardware error
_int86	Invokes MS-DOS interrupts
_int86x	Invokes MS-DOS interrupts with segment register values
_intdos	Invokes MS-DOS system call
_intdosx	Invokes MS-DOS system call with segment register values
_MK_FP	Makes a far pointer from the segment value and the offset value
_segread	Returns current values of segment registers

Time Routines

The time routines allow you to obtain the current time, then convert and store it according to your particular needs. The current time is always taken from the system time.

Routine	Description
asctime	Converts time from type struct tm to a character string
clock	Returns the elapsed CPU time for a process
ctime	Converts time from type time_t to a character string
difftime	Computes the difference

	between two times
_ftime	Puts current system time in variable of type struct_timeb
gmtime	Converts time from type time_t to struct tm
localtime	Converts time from type time_t to struct tm with local correction
mktime	Converts time to a calendar value
_strdate	Returns the current system date as a string
_strtime	Returns the current system time as a string
time	Gets current system time as type time_t
_tzset	Sets external time variables from the environment time variable
_utime	Sets file-modification time

Variable-Length Argument Lists

The va_arg, va_end, and va_start routines are macros that provide a portable way to access the arguments to a routine when the routine takes a variable number of arguments.

Routine	Description
va_arg	Retrieves argument from list
va_end	Resets pointer
va_start	Sets pointer to beginning of argument list

Virtual Memory Allocation Routines

The virtual memory routines allow you to allocate, free, reallocate, lock, and unlock blocks of memory.

Routine	Description
_vfree	Frees an allocated block of virtual memory
_vheapinit	Initializes the virtual memory manager
_vheapterm	Terminates the virtual memory manager
_vload	Loads an allocated block of virtual memory
_vlock	Locks an allocated block of virtual memory
_vlockcnt	Returns the number of locks held on a block of virtual memory
_vmalloc	Allocates a block of virtual memory
_vmsize	Returns the size of an allocated block of virtual memory

_vrealloc	Reallocates a block of virtual memory to a new size
_vunlock	Unlocks a locked block of virtual memory

Low-Level Graphics Routines

The low-level graphics and font routines can be divided into the eight categories listed below, which correspond to the different tasks involved in creating and manipulating graphic objects.

Category	Task
Configuring mode and environment	Selects the proper display mode for the hardware and establishes memory areas for writing and displaying images
Setting coordinates	Specifies the logical origin and the active display area within the screen
Setting low-level graphics palettes	Specifies a palette mapping for low-level graphics routines
Setting attributes	Specifies background and foreground colors, fillmasks, and line styles for low-level graphics routines
Creating graphics output	Draws and fills figures
Creating text output	Writes text on the screen
Transferring images	Stores images in memory and retrieves them
Displaying fonts	Displays text in character fonts compatible with Microsoft Windows

Configuring Mode and Environment Routines

Routines that configure the mode and environment establish the graphics or text mode of operation, determine the current graphics environment, and control the display of the cursor.

Routine	Description
_clearscreen	Erases the screen and fills it with the current background color
_getactivepage	Gets the current active page number
_getbkcolor	Returns the current background color
_getvideoconfig	Obtains status of current graphics environment
_getvisualpage	Gets the current visual page number

_grstatus	Returns the status of the most recent graphics function call
_setactivepage	Sets memory area for the active page for writing images
_setbkcolor	Sets the current background color
_settextrows	Sets the number of text rows
_setvideomode	Selects an operating mode for the display screen
_setvideomoderows	Sets the video mode and the number of rows for text operations
_setvisualpage	Sets memory area for the current visual page

Setting Coordinates Routines

The "set coordinates" routines set the current text or graphics position and convert pixel coordinates between the various graphics coordinate systems.

Routine	Description
_getcurrentposition	Determines current position in view coordinates
_getcurrentposition_w	Determines current position in window coordinates
_getphyscoord	Converts view coordinates to physical coordinates
_getviewcoord	Converts physical coordinates to view coordinates
_getviewcoord_w	Converts window coordinates to view coordinates
_getviewcoord_wxy	Converts window coordinates in _wxycoord structure to view coordinates
_getwindowcoord	Converts view coordinates to window coordinates_setcliprgn. Limits graphic output to a region of the screen
_setvieworg	Positions the view-coordinate origin
_setviewport	Limits graphics output to a region of the screen

	and positions the view-coordinate origin to the upper-left corner of that region
_setwindow	Defines a floating-point window coordinate system

Setting Low-Level Palette Routines

Use the low-level palette routines to select or remap color palettes.

Routine	Description
_remapallpalette	Changes all color indexes in the current palette
_remappalette	Changes a single color index in the current palette
_selectpalette	Selects a predefined palette

Setting Attributes Routines

The low-level output routines that draw lines, arcs, ellipses, and other basic figures do not specify color or line-style information. Instead, the low-level graphics routines rely on a set of attributes set independently by the following routines.

Routine	Description
_getarcinfo	Determines the endpoints in viewport coordinates of the most recently drawn arc or pie
_getcolor	Gets the color
_getfillmask	Gets the fill mask
_getlinestyle	Gets the line-style mask
_getwritemode	Gets the logical write mode
_setcolor	Sets the color
_setfillmask	Sets the fill mask
_setlinestyle	Sets the line-style mask
_setwritemode	Sets the logical write mode for line drawing

Creating Graphics Output Routines

The graphics output routines use a set of specified coordinates and draw various figures.

Routine	Description
_arc, _arc_w, _arc_wxy	Draw an arc
_ellipse, _ellipse_w, _ellipse_wxy	Draw an ellipse or circle
_floodfill, _floodfill_w	Flood-fill an area of the screen with the current color

_getpixel, _getpixel_w	Obtain a pixel's color
_lineto, _lineto_w	Draw a line from the current graphic-output position to a specified point
_moveto, _moveto_w	Move the current graphic-output position to a specified point
_pie, _pie_w, _pie_wxy	Draw a pie-slice-shaped figure
_polygon, _polygon_w, _polygon_wxy	Draw or scan-fill a polygon
_rectangle, _rectangle_w, _rectangle_wxy	Draw or scan-fill a rectangle
_setpixel, _setpixel_w	Set a pixel's color

Creating Text Output Routines

The routines in this category provide text output in graphics and text modes.

Routine	Description
_displaycursor	Sets the cursor on or off upon exit from a graphics routine
_gettextcolor	Gets the text color
_gettextcursor	Returns the cursor attribute (text modes only)
_gettextposition	Obtains the text-output position
_gettextwindow	Gets the text window boundaries
_outmem	Prints text of a specified length from a memory buffer
_outtext	Prints a text string to the screen at the text position
_scrolltextwindow	Scrolls the text window up or down
_settextcolor	Sets the text color
_settextcursor	Sets the cursor attribute (text modes only)
_settextposition	Relocates the text position
_settextwindow	Defines the text-display window
_wrapon	Enables or disables line wrap

Transferring Images Routines

The routines in this category transfer screen images between memory and the display. The routines that end with _w or _wxy use window coordinates; the other routines in this set use view coordinates.

Routine	Description
_getimage, _getimage_w, _getimage_wxy	Store a screen image in memory
_imagesize, _imagesize_w, _imagesize_wxy	Return the size (in bytes) of the buffer needed to store the image
_putimage, _putimage_w	Retrieve an image from memory and display it

Displaying Fonts Routines

The routines listed in this section control the display of font-based characters on the screen.

Routine	Description
_getfontinfo	Obtains the font characteristics
_getgtextextent	Determines the width in pixels of specified text in the font
_getgtextvector	Gets orientation of font text output
_outgtext	Outputs text in the current font to the screen at the specified pixel position
_registerfonts	Initializes font library
_setfont	Finds a single font that matches a specified set of characteristics and makes this font the current font for use by the _outgtext function
_setgtextvector	Sets the orientation for font text output
_unregisterfonts	Frees memory allocated by _registerfonts

Presentation-Graphics Routines

The presentation-graphics routines can be divided into the three categories listed below, corresponding to the different tasks involved in creating and manipulating graphic objects.

Category	Task
Displaying presentation graphics	Initializes video structures for presentation graphics and establishes the default chart type. Displays presentation-graphics chart_

	bar, column, pie, scatter, or line chart.
Analyzing presentation-graphics data	Analyzes data (does not display chart).
Manipulating presentation-graphics structures	Modifies basic chart structures (e.g., palettes, cross-hatching styles).

Displaying Presentation Graphics

The routines listed in this section initialize the presentation-graphics library and display the specified graph type.

Routine	Description
_pg_chart	Displays a single-series bar, column, or line chart
_pg_chartms	Displays a multiseries bar, column, or line chart
_pg_chartpie	Displays a pie chart
_pg_chartscatter	Displays a scatter diagram for a single series of data
_pg_chartscatterms	Displays a scatter diagram for more than one series of data
_pg_defaultchart	Initializes all variables in the chart environment that are necessary for a specified chart type
_pg_initchart	Initializes the presentation-graphics library

Analyzing Presentation Graphic Data Routines

Presentation graphic data routines calculate default values for the specified graph type but do not display the chart.

Routine	Description
_pg_analyzechart	Analyzes a single series of data for a bar, column, or line chart
_pg_analyzechartms	Analyzes a multiseries of data for a bar, column, or line chart
_pg_analyzepie	Analyzes data for a pie chart
_pg_analyzescatter	Analyzes a single series of data for a scatter diagram

_pg_analyzescatterms	Analyzes a multiseries of data for a scatter diagram

Manipulating Presentation Graphics Structures Routines

These routines control low-level aspects of the presentation-graphics package.

Routine	Description
_pg_getchardef	Retrieves the 8-by-8-pixel bit map for a specified character
_pg_getpalette	Retrieves colors, line styles, fill patterns, and plot characters for all presentation-graphics palettes
_pg_getstyleset	Retrieves the contents of the styleset
_pg_hlabelchart	Writes text horizontally on the screen
_pg_resetpalette	Sets current colors, line styles, fill patterns, and plot characters to the default values for the current screen mode
_pg_resetstyleset	Resets the contents of the current styleset to the default value for the current screen mode
_pg_setchardef	Sets the 8-by-8-pixel bit map for a specified character
_pg_setpalette	Sets current colors
_pg_setstyleset	Sets the contents of the styleset
_pg_vlabelchart	Writes text vertically on the screen

Input and Output (I/O) Routines

The input and output (I/O) routines enable you to read and write data to and from files and devices.

Category	Task
Stream	The stream I/O routines treat data as a stream of individual characters.
Low-Level I/O	The low-level I/O routines do not perform buffering and formatting. Instead, they invoke the operating system's input and output capabilities directly.
Console and Port I/O	The console and port I/O routines allow you to read or

write directly to a console (keyboard and screen) or an I/O port (such as a printer port).

Stream Routines

Stream I/O routines handle data as a continuous stream of characters.

Routine	Description
clearerr	Clears the error indicator for a stream
fclose	Closes a stream
_fcloseall	Closes all open streams
_fdopen	Associates a stream with an open file handle
feof	Tests for end-of-file on a stream
ferror	Tests for error on a stream
fflush	Flushes a stream
fgetc	Reads a character from a stream (function version)
_fgetchar	Reads a character from stdin (function version)
fgetpos	Gets the position indicator of a stream
fgets	Reads a string from a stream
_fileno	Gets the file handle associated with a stream
_flushall	Flushes all streams
fopen	Opens a stream
fprintf	Writes formatted data to a stream
fputc	Writes a character to a stream (function version)
_fputchar	Writes a character to stdout (function version)
fputs	Writes a string to a stream
fread	Reads unformatted data from a stream
freopen	Reassigns a FILE pointer to a new file
fscanf	Reads formatted data from a stream
fseek	Moves file position to a given location
fsetpos	Sets the position indicator of a stream
_fsopen	Opens a stream with file sharing
ftell	Gets current file position
fwrite	Writes unformatted data items to

	a stream
getc	Reads a character from a stream
getchar	Reads a character from stdin
gets	Reads a line from stdin
_getw	Reads a binary int item from a stream
printf	Writes formatted data to stdout
putc	Writes a character to a stream
putchar	Writes a character to stdout
puts	Writes a line to a stream
_putw	Writes a binary int item to a stream
rewind	Moves file position to beginning of a stream
_rmtmp	Removes temporary files created by tmpfile
scanf	Reads formatted data from stdin
setbuf	Controls stream buffering
setvbuf	Controls stream buffering and buffer size
_snprintf	Writes formatted data of a specified length to a string
sprintf	Writes formatted data to a string
sscanf	Reads formatted data from a string
_tempnam	Generates a temporary filename in given directory
tmpfile	Creates a temporary file
tmpnam	Generates a temporary filename
ungetc	Places a character in the buffer
vfprintf	Writes formatted data to a stream
vprintf	Writes formatted data to stdout
_vsnprintf	Writes formatted data of a specified length to a string
vsprintf	Writes formatted data to a string

Low-Level Routines

Low-level input and output calls do not buffer or format data.

Routine	Description
_close	Closes a file
_commit	Flushes a file to disk
_creat	Creates a file
_dup	Creates a second handle for a file
_dup2	Reassigns a handle to a file

_eof	Tests for end-of-file
_lseek	Repositions file pointer to a given location
_open	Opens a file
_read	Reads data from a file
_sopen	Opens a file for file sharing
_tell	Gets current file-pointer position
_write	Writes data to a file

Console and Port I/O Routines

These routines perform reading and writing operations on your console or on the specified port.

Routine	Description
_cgets	Reads a string from the console
_cprintf	Writes formatted data to the console
_cputs	Writes a string to the console
_cscanf	Reads formatted data from the console
_getch	Reads a character from the console
_getche	Reads a character from the console and echoes it
_inp	Reads one byte from the specified I/O port
_inpw	Reads a two-byte word from the specified I/O port
_kbhit	Checks for a keystroke at the console
_outp	Writes one byte to the specified I/O port
_outpw	Writes a two-byte word to the specified I/O port
_putch	Writes a character to the console
_ungetch	"Ungets" the last character read from the console so that it becomes the next character read

Header Files

Filename	Major Contents
ASSERT.H	assert debugging macro
BIOS.H	BIOS service functions
CDERR.H	Common dialog error return codes
COLORDLG.H	Common dialog color dialog's control id numbers
COMMDLG.H	Common dialog functions, types, and definitions
CONIO.H	Console and port I/O routines
CPL.H	Control panel extension DLL definitions
CTYPE.H	Character classification
CUSTCNTL.H	Custom Control Library header file
DDE.H	Dynamic Data Exchange structures and definitions
DDEML.H	DDEML API header file
DIRECT.H	Directory control
DLGS.H	Common dialog's dialog element ID numbers
DOS.H	MS-DOS interface functions
DRIVINIT.H	Obsolete: Use print.h instead
ERRNO.H	errno variable definitions
FCNTL.H	Flags used in _open and _sopen functions
FLOAT.H	Constants needed by math functions
FSTREAM.H	Functions used by the filebuf and fstream classes
GRAPH.H	Low-level graphics and font routines
IO.H	File-handling and low-level I/O
IOMANIP.H	definitions/declarations for iostream's parameterized manipulators
IOS.H	Functions used by the ios class
IOSTREAM.H	Functions used by the

	iostream classes
ISTREAM.H	Functions used by the istream class
LIMITS.H	Ranges of integers and character types
LOCALE.H	Localization functions
LZDOS.H	Obsolete: Replaced by #define LIB/#include <lzexpand.h>
LZEXPAND.H	Public interfaces for LZEXPAND.DLL.
MALLOC.H	Memory-allocation functions
MATH.H	Floating-point-math routines
MEMORY.H	Buffer-manipulation routines
MMSYSTEM.H	Include file for Multimedia APIs
NEW.H	declarations and definitions for C++ memory allocation functions
OLE.H	OLE functions, types, and definitions
OSTREAM.H	Functions used by the ostream class
PENWIN.H	Pen Windows functions, types, and definitions
PENWOEM.H	Pen Windows APIs into recognizer layer
PGCHART.H	Presentation graphics
PRINT.H	Printing helper functions, types, and definitions
PROCESS.H	Process-control routines
SCRNSAVE.H	Windows 3.1 screensaver defines and definitions
SEARCH.H	Searching and sorting functions
SETJMP.H	setjmp and longjmp functions
SHARE.H	Flags used in _sopen
SHELLAPI.H	SHELL.DLL functions, types, and definitions
SIGNAL.H	Constants used by signal function
STDARG.H	Macros for variable-length argument-list functions
STDDEF.H	Commonly used data

	types and values
STDIO.H	Standard I/O header file
STDIOSTR.H	Functions used by the stdiostream and stdiobuf classes
STDLIB.H	Commonly used library functions
STREAMB.H	Functions used by the streambuf class
STRESS.H	Stress functions definitions
STRING.H	String-manipulation functions
STRSTREA.H	Functions used by the strstream and strstreambuf classes
TIME.H	General time functions
TOOLHELP.H	TOOLHELP.DLL functions, types, and definitions
VARARGS.H	Variable-length argument-list functions
VER.H	Version management functions, types, and definitions
VMEMORY.H	Virtual memory functions
WFEXT.H	Windows File Manager Extensions definitions
WINDOWS.H	Windows functions, types, and definitions
WINDOWSX.H	Macro APIs, window message crackers, and control APIs
WINMEM32.H	Function protypes and general defines for WINMEM32 DLL
SYS\LOCKING.H	Flags used by _locking function
SYS\STAT.H	File-status structures and functions
SYS\TIMEB.H	time function
SYS\TYPES.H	File-status and time types
SYS\UTIME.H	Run-time function

Global Variables

_amblksiz	_fileinfo	_pgmptr
_cpumode	_fmode	_psp
_daylight	_osmajor	_sys_errlist

Constants Alphabetized

A

B

C

D

E

F

G

H

I

K

L

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MAX
LC_MIN
LC_MONETARY
LC_NUMERIC
LC_TIME
LDBL_DIG
LDBL_EPSILON
LDBL_MANT_DIG
LDBL_MAX
LDBL_MAX_10_EXP
LDBL_MAX_EXP
LDBL_MIN
LDBL_MIN_10_EXP
LDBL_MIN_EXP
_LDBL_RADIX
_LDBL_ROUNDS
_LHUGE_VAL
_LIGHTBLUE
_LIGHTCYAN
_LIGHTGREEN
_LIGHTMAGENTA
_LIGHTRED
_LIGHTYELLOW
_LK_LOCK
_LK_NBLCK
_LK_NBRLCK
_LK_RLCK
_LK_UNLCK
LONG_MAX
LONG_MIN
L_tmpnam

M

_MAGENTA
_MAX_DIR
_MAX_DRIVE
_MAX_EXT
_MAX_FNAME
_MAX_PATH
_MAXCOLORMODE
_MAXRESMODE
_MAXTEXTROWS
MB_CUR_MAX
MB_LEN_MAX
_MCGA
_MDPA
_MODE7HI
_MODE7OFF
_MODE7ON
_MODEFHITOOFF
_MODEFHITOON
_MODEFHI
_MODEFOFF
_MODEFOFFTOHI
_MODEFOFFTOON
_MODEFON
_MODEFONTOHI
_MODEFONTOOFF
_MONO
_MRES16COLOR
_MRES256COLOR
_MRES4COLOR
_MRESNOCOLOR

N

_NKEYBRD_READ
_NKEYBRD_READY
_NKEYBRD_SHIFTSTATUS
NULL
_NULLOFF
_NULLSEG

O

_O_APPEND
_O_BINARY
_OCGA
_O_CREAT
_OEGA
_O_EXCL
_O_NOINHERIT
_O_RAW
_O_RDONLY
_O_RDWR
_ORESCOLOR
_ORES256COLOR
_O_TEXT
_O_TRUNC
_OVERFLOW
_OVGA
_O_WRONLY

P

_P_OVERLAY
_P_WAIT
_PG_BADCHARTSTYLE
_PG_COLUMNCHART
_PG_DECFORMAT
_PG_EXPFORMAT
_PG_PLAINBARS
_PG_POINTANDLINE
_PG_POINTONLY

R

S

T

U

V

W

_WHITE	_WINEXITPERSIST	_WINSIZECHAR
_WINARRANGE	_WINEXITPROMPT	_WINSIZEMAX
_WINBUFDEF	_WINFRAMEHAND	_WINSIZEMIN
_WINBUFINF	_WINMAXREQ	_WINSIZERESTORE
_WINCASCADE	_WIN_MODE	_WINSTATBAR
_WINCURRREQ	_WINNOPERSIST	_WINTILE
_WINEXITNOPERSIST	_WINPERSIST	

XYZ

_XRES16COLOR	_XRES256COLOR	_YELLOW
_ZRES16COLOR	_ZRES256COLOR	

Types

The following types are used by functions in the run-time library.

_axistype	_fontinfo	_timeb
_BYTEREGS	fpos_t	_titletype
_chartenv	_HEAPINFO	tm
clock_t	jmp_buf	_utimbuf
_complex	lconv	va_list
_complexl	ldiv_t	_videoconfig
_diskfree_t	_legendtype	_vmhnd_t
_diskinfo_t	_paletteentry	wchar_t
div_t	ptrdiff_t	_windowtype
_dosdate_t	_rccoord	_wopeninfo
_DOSERROR	_REGS	_WORDREGS
_dostime_t	sig_atomic_t	_wsizeinfo
_exception	size_t	_wxycoord
_exceptionl	_SREGS	_xycoord
FILE	_stat	
_find_t	time_t	

C/C++ Keywords

The following are keywords in Microsoft C and C++. Names with leading underscores are Microsoft extensions.

C Language Keywords

__asm	__fastcall	__self
auto	float	__segment
__based	for	__segname
break	__fortran	short
case	goto	signed
__cdecl	__huge	sizeof
char	if	static
const	__inline	struct
continue	int	switch
default	__interrupt	typedef
do	__loadds	union
double	long	unsigned
else	__near	void
enum	__pascal	volatile
__export	register	while
extern	return	
__far	__saveregs	

C++ Language Keywords

class	operator	try
delete	private	virtual
friend	protected	__multiple_inheritance
inline	public	__single_inheritance
new	this	__virtual_inheritance

The following are not keywords, but they have special meaning in Microsoft C or C++

argc	envp	_setenvp
argv	main	_set_new_handler
__emit	_setargv	

Statements

The following are the statements recognized by the C language:

break	else	return
case	for	switch
continue	goto	typedef

default if while
do

The C++ language adds the following statements:

delete this
new template
operator

Data Types

C/C++ recognizes the types shown in the table below.

Type Name	Bytes	Other Names	Range of Values
int	*	signed, signed int	System dependent
unsigned int	*	unsigned	System dependent
char	1	signed char	-128 to 127
unsigned char	1	none	0 to 255
short	2	short int, signed short int	-32,768 to 32,767
unsigned short	2	unsigned short int	0 to 65,535
long	4	long int, signed long int	-2,147,483,648 to 2,147,483,647
unsigned long	4	unsigned long int	0 to 4,294,967,295
enum	2	none	-32,768 to 32,767
float	4	none	3.4E +/- 38 (7 digits)
double	8	none	1.7E +/- 308 (15 digits)
long double	10	none	1.2E +/- 4932 (19 digits)

Signed and unsigned are modifiers that can be used with any integral type. The char type is signed by default, but you can specify /J to make it unsigned by default.

The int and unsigned int types have the size of the system word. This is two bytes (the same as short and unsigned short) on MS-DOS and 16-bit versions of Windows. However, portable code should not depend on the size of int.

Modifiers

The following are modifiers in Microsoft C and C++. Modifier names with two leading underscores are Microsoft extensions.

C Modifiers

__asm	__fortran	__saveregs
auto	__huge	__segment
__based	__inline	__segname
__cdecl	__interrupt	__self
const	__loadds	short
__export	long	signed
extern	__near	static
__far	__pascal	unsigned
__fastcall	register	volatile

C++ Modifiers

friend	private	public
inline	protected	virtual

Operators

The table below lists C and C++ operators by category.

Arithmetic		Relational	
+	Addition	<	Less than
-	Subtraction	<=	Less than or equal to
*	Multiplication	>	Greater than
/	Division	>=	Greater than or equal to
%	Modulus	==	Equal
		!=	Not equal

Assignment		Increment & Decrement	
=	Assignment	++	Increment
+=	Addition	--	Decrement
-=	Subtraction		
*=	Multiplication		
/=	Division		
%=	Modulus		
<<=	Left shift assignment		
>>=	Right shift assignment		
&=	Bitwise AND		
^=	Bitwise exclusive		

	OR
|=	Bitwise OR

Bitwise		Logical	
&	Bitwise AND	&&	Logical AND
^	Bitwise-exclusive-OR	||	Logical OR
|	Bitwise OR	!	Logical NOT
<<	Left shift		
>>	Right shift		
~	One's complement		

Pointer		Conditional	
&	Address of	? :	Conditional
*	Indirection		Example: `(val >= 0) ? val : -val` If the condition is true, the expression evaluates to val. If not, the expression equals -val.
:>	Base Example: `myseg:>bp` The pointer `bp` acts as an offset into the segment specified by myseg.		

Miscellaneous		C++ Only	
( )	Function call	::	Scope resolution
[ ]	Array element	&	Reference
.	Structure or union member	.*	Pointer to member
->	Pointer to structure member	->*	Pointer to member
(type)	Type cast		
sizeof	Size in bytes		

Directives

The table below lists the directives to the preprocessor.

#define	#error	#include
#elif	#if	#line
#else	#ifdef	#pragma
#endif	#ifndef	#undef

Preprocessor Operators

The following lists the preprocessor operators.

(Stringize)
#@ (Charize)
(Token-paste)
defined

Pragmas

Syntax #pragma *directive*

Summary Instructs the compiler to implement the compiler-specific feature specified by the *directive* argument.

The Microsoft C/C++ compiler recognizes the pragmas listed below.

alloc_text	inline_depth	pagesize
auto_inline	init_seg	pointers_to_members
check_pointer	intrinsic	same_seg
check_stack	linesize	setlocale
code_seg	loop_opt	skip
comment	message	subtitle
data_seg	native_caller	title
function	optimize	vtordisp
hdrstop	pack	warning
inline_recursion	page	

Predefined Macros

The identifiers listed below may be predefined by the compiler for testing and other uses in preprocessor statements.

__cplusplus	_M_I8086	_PCODE
__STDC__	_M_I86	_QC
_DLL	_M_I86mM	_WINDLL
_FAST	_MSC_VER	_WINDOWS
_M_I286	_MSDOS	_CHAR_UNSIGNED
_M_I386		

The following predefined identifiers are supported for compatibility with previous versions of Microsoft C:

M_I86	M_I8086	M_I386
M_I86mM	M_I286	MSDOS

Operator Precedence

The table below lists the C and C++ operators and their precedence and associativity values. The lines separate precedence levels. The highest precedence level is at the top of the table.

Symbol	Name or Meaning	Associativity
	Highest Precedence	
++	Post-increment	Left to right
--	Post-decrement	
( )	Function call	
[ ]	Array element	
->	Pointer to structure	

	member	
.	Structure or union member	
++	Pre-increment	Right to left
--	Pre-decrement	
:>	Base	
!	Logical NOT	
~	Bitwise NOT	
-	Unary minus	
+	Unary plus	
&	Address	
*	Indirection	
sizeof	Size in bytes	
new	Allocate program memory	
delete	Deallocate program memory	
(type)	Type cast [for example, (float) i]	
.*_	Pointer to member (objects)	Left to right
->*	Pointer to member (pointers)	
*	Multiply	Left to right
/	Divide	
%	Remainder	
+	Add	Left to right
-	Subtract	
<<	Left shift	Left to right
>>	Right shift	
<	Less than	Left to right
<=	Less than or equal to	
>	Greater than	
>=	Greater than or equal to	
==	Equal	Left to right
!=	Not equal	
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right

|	Bitwise OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
? :	Conditional	Right to left
=	Assignment	Right to left
*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=	Compound assignment	
,	Comma	Left to right

Lowest Precedence

Escape Sequences

The escape sequences allow you to use a sequence of characters to represent special characters. Escape sequences are listed below.

Sequence	**Name**
\a	Alert (bell)
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\?	Literal quotation mark
\'	Single quotation mark
\"	Double quotation mark
\\	Backslash
ddd	ASCII character in octal notation
\x*dd*	ASCII character in hex notation

Note that for characters notated in hexadecimal, the compiler ignores all leading zeros. It establishes the end of the hex-specified escape character when it encounters either the first non-hex character or more than two hex characters _ not including leading zeros. In the latter case, it reports an error and ignores all characters beyond the second one.

printf Formatting

% [*flags*] [*width*] [**.***precision*] [{F | N | h | l | L}] *type*

See Also

The formats for the printf family of functions are listed below**.**

Flags

-	(left justify)
+	(prefix with sign)
blank	(prefix with blank)
#	(modifies o, x, X, e, E, f, g, G)

Prefix

F	(far pointer)
N	(near pointer)
h	(short int)
l,L	(long int or double)

Type

d,i	(signed decimal)
u	(unsigned decimal integer)
o	(unsigned octal integer)
x,X	(unsigned hex integer)
f	(fixed-point integer)
e, E	(scientific notation)
g, G	(%e or %f; whichever is shorter)
c	(single character)
s	(string)
p	(pointer)
n	(character count)

printf *width* specification
printf *precision* specification
Escape Sequences

Example Programs

A _ D

ALARM.C
ANALYZE.C
ANIMATE.C
ARGS.C
ASCII.C
ASSERT.C
ATEXIT.C
ATONUM.C
BARCOL.C
BEEP.C
BESSEL.C
BUFTEST.C
CABS.C
CASE.C
CGAPAL.C
CHMOD1.C
CHMOD2.C
CMPSTR.C
COM.C
COMMIT.C
COPROC.C
COPY2.C
CPYSTR.C
CURSOR.C
DCOMMIT.C
DIRECT.C
DISK.C
DOSMEM.C
DRIVES.C

E _ I

ENVIRON.C
ERROR.C
EXEC.C
EXTDIR.C
EXTERR.C
FCVT.C
FIGURE.C
FILL.C
FINDSTR.C
FONTS.C
FREECT.C
FULL.C
FUNGET.C
FWOPEN.C
GEDIT.C
GETCH.C
HALLOC.C
HANDLER.CPP
HARDERR.C
HEAPBASE.C
HEAPWALK.C
HEXDUMP.C
HMANAGE.C
INTMATH.C
IOTEST.C
IS.C

K _ R

KBHIT.C
LOCK.C
MATH.C
MBLEN.CPP
MBTOWC.CPP
MODES.C
MOVEMEM.C
MSERIES.C
NULLFILE.C
PAGE.C
PALETTE.C
PRINTF.C
REALLOC.C
RECORDS2.C
KEYBRD.C
MATHERR.C
MBSTOWCS.CPP
MKFPSTR.C
MORE.C
MSB.C
NUMTOA.C
PAGER.C
PGPAL.C
QSORT.C
RECORDS1.C
ROTATE.C

S _ V

SCANF.C
SCROLL.C
SETROWS.C
SETTIME.C
SIGFP.C
SPAWN.C
TOKEN.C
TYPEIT.C
VARARG.C
VLOAD.C
VRSIZE.C
SCAT.C
STAR.C
SWAB.C
SYSINFO.C
TEMPNAME.C
TIMES.C
TRIG.C

W

iostream Class List

Class	Brief
Abstract Stream Base Class	
ios	Stream base class.
Input Stream Classes	
istream	General-purpose input stream class and base class for other input streams.
ifstream	Input file stream class.
istream_withassign	Input stream class for **cin**.
istrstream	Input string stream class.
Output Stream Classes	
ostream	General-purpose output stream class and base class for other output streams.
ofstream	Output file stream class.
ostream_withassign	Output stream class for **cout**, **cerr**, and **clog**.
ostrstream	Output string stream class.
Input/Output Stream Classes	
iostream	General-purpose input/output stream class and base class for other input/output streams.
fstream	Input/output file stream class.
strstream	Input/output string stream class.
stdiostream	Input/output class for standard I/O files.
Stream Buffer Classes	
streambuf	Abstract stream buffer base class.
filebuf	Stream buffer class for disk files.
strstreambuf	Stream buffer class for strings.
stdiobuf	Stream buffer class for standard I/O files.
Predefined Stream Initializer Class	
Iostream_init	Predefined stream initializer class.

iostream Classes

- Abstract Stream Base Class
- Input Stream Classes
- Output Stream Classes
- Input/Output Stream Classes
- Stream Buffer Classes
- Predefined Stream Initializer Class

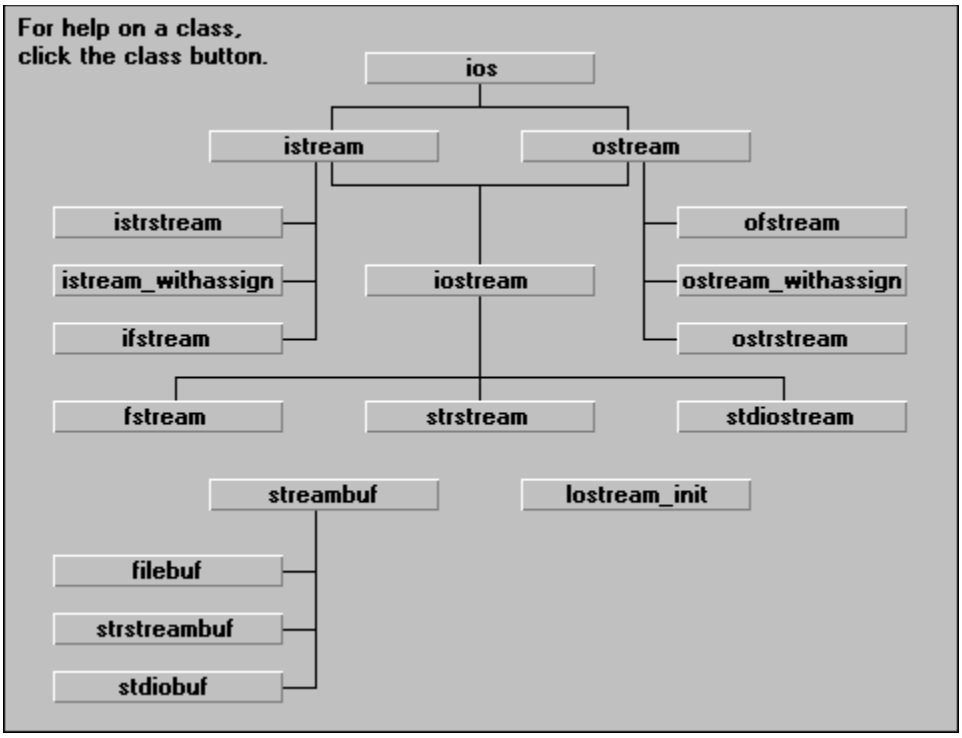
Hypertext Jumps and the Browser Window

Help topics are linked by "hypertext jumps." When you choose text or a bitmap that activates a jump, Windows Help clears the current topic and displays the topic indicated by the jump.

In some cases, instead of having Windows Help jump away from the current topic, you see information displayed in a "popup" windows, such as this one. Popup jumps are dismissed by a mouse click.

Sometimes this help system lists jumps in another kind of window—the browser window. The browser window is opened by choosing the Browser button, as you have done. The browser window lists jumps indicated by a small button with a green center. When you choose a jump in the browser window, Windows Help clears the current topic in the "main" window and displays the topic you chose. This feature allows you to browse long lists of information or jump to a topic you're interested in. Choosing a jump in the main window does not change the state of the browser window. The browser window allows you to keep your place as you browse the help system.

- To close the browser window, either:
  1. Choose the Close button,
  2. Double click the system menu
  3. Open the system menu, then choose Close. Shortcut key: ALT+F4.


For help on a class,
click the class button.
ios
istream
ostream
istrstream
istream_withassign
ifstream
iostream
ofstream
ostream_withassign
ostrstream
fstream
strstream
stdiostream
streambuf
iostream_init
filebuf
strstreambuf
stdiobuf

abort

#include <process.h> Required only for function declarations;

#include <stdlib.h> use either PROCESS.H or STDLIB.H

Syntax void abort(void);

Example

Compatibility

The abort function prints the message

```
abnormal program termination
```

to stderr, then calls raise(SIGABRT). The action taken in response to the SIGABRT signal depends on what action has been defined for that signal in a prior call to the signal function. The default SIGABRT action is for the calling process to terminate with exit code 3, returning control to the parent process or operating system.

In Windows, the abort function does not call raise(SIGABRT). Instead, it terminates the process with an "Abnormal Program Termination" pop-up message. In Windows multithread libraries, the abort function does not call raise(SIGABRT). Instead, it terminates the process with exit code 3.

The abort function does not flush stream buffers or do atexit/_onexit processing.

Return Value

The abort function does not return control to the caller. Rather, it terminates the process and, by default, returns an exit code of 3 to the parent process.

_exec functions
exit
_exit
raise
signal
_spawn functions

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL


```
/* ABORT.C:  This tries to open a file and
 * aborts if the attempt fails.
 */
#include  <stdio.h>
#include  <stdlib.h>
void main( void )
{
   FILE *stream;
   if( (stream = fopen( "NOSUCHF.ILE", "r" )) == NULL )
   {
      perror( "Couldn't open file" );
      abort();
   }
   else
      fclose( stream );
}
```

abs

#include<stdlib.h> Required only for function declarations;

#include <math.h> use STDLIB.H OR MATH.H

Syntax int abs(int *n*);

Example

Compatibility

Parameter	**Description**
n	Integer value

The abs function returns the absolute value of its integer parameter *n*.

Return Value

The abs function returns the absolute value of its parameter. There is no error return.

_cabs
fabs
labs

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL


```
/* ABS.C: This program computes and displays
 * the absolute values of several numbers.
 */
#include  <stdio.h>
#include  <math.h>
#include  <stdlib.h>
void main( void )
{
   int    ix = -4, iy;
   long   lx = -41567L, ly;
   double dx = -3.141593, dy;
   iy = abs( ix );
   printf( "The absolute value of %d is %d\n", ix, iy);
   ly = labs( lx );
   printf( "The absolute value of %ld is %ld\n", lx, ly);
   dy = fabs( dx );
   printf( "The absolute value of %f is %f\n", dx, dy );
}
```

_access

#include <io.h> Required only for function declarations

#include <errno.h> Required for definition of errno constants

Syntax int _access(const char **path*, int *mode*);

Parameter	**Description**
path	File or directory path
mode	Permission setting

With files, the _access function determines whether the specified file exists and can be accessed in *mode*. The possible mode values and their meanings in the _access call are as follows:

Value	**Meaning**
00	Check for existence only
02	Check for write permission
04	Check for read permission
06	Check for read and write permission

With directories, _access determines only whether the specified directory exists; in MS-DOS, all directories have read and write access.

Return Value

The _access function returns the value 0 if the file has the given mode. A return value of -1 indicates that the named file does not exist or is not accessible in the given mode, and errno is set to one of the following values:

Value	**Meaning**
EACCES	Access denied: the file's permission setting does not allow the specified access.
ENOENT	File or path not found.

Use _access for compatibility with ANSI naming conventions of non-ANSI functions. Use access and link with OLDNAMES.LIB for UNIX compatibility.

_chmod
_fstat
_open
_stat

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* ACCESS.C: This example uses _access to check the
 * file named "ACCESS.C" to see if it exists and if
 * writing is allowed.
 */
#include  <io.h>
#include  <stdio.h>
#include  <stdlib.h>
void main( void )
{
   /* Check for existence */
   if( (_access( "ACCESS.C", 0 )) != -1 )
   {
      printf( "File ACCESS.C exists\n" );
      /* Check for write permission */
      if( (_access( "ACCESS.C", 2 )) != -1 )
         printf( "File ACCESS.C has write permission\n" );
   }
}
```

acos Functions

#include <math.h>

#include <errno.h> Required for definition of errno constant

Syntax double acos(double *x*);

long double acosl(long double *x*);

Parameter	Description
x	Value whose arccosine is to be calculated

The acos functions return the arccosine of *x* in the range 0 to π radians. The value of *x* must be between -1 and 1. The acosl function is the 80-bit counterpart, which uses an 80-bit, 10-byte coprocessor form of parameters and return values. See the long double functions for more details on this data type.

Return Value

The acos functions return the arccosine result. If *x* is less than -1 or greater than 1, the function sets errno to EDOM, prints a _DOMAIN error message to stderr, and returns 0. Error handling can be modified with the _matherr (or _matherrl) routine.

acos

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

acosl

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* ASINCOS.C: This program prompts for a value in the range
 * -1 to 1. Input values outside this range will produce
 * _DOMAIN error messages.If a valid value is entered, the
 * program prints the arcsine and the arccosine of that value.
 */
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
void main( void )
{
   double x, y;
   printf( "Enter a real number between -1 and 1: " );
   scanf( "%lf", &x );
   y = asin( x );
   printf( "Arcsine of %f = %f\n", x, y );
   y = acos( x );
   printf( "Arccosine of %f = %f\n", x, y );
}
```

_alloca

#include <malloc.h> Required only for function declarations

Syntax void *_alloca(size_t *size*);

Parameter	Description
size	Bytes to be allocated from stack

The _alloca routine allocates *size* bytes from the program's stack. The allocated space is automatically freed when the calling function is exited.

Observe the following restrictions when using _alloca:

- When you compile with optimization on (either by default or by using one of the /O options), the stack pointer may not be restored properly in functions that have no local variables and that also reference the _alloca function. The following program demonstrates the problem:

```
/* Compile with CL /AM ?Ox ?Fc */
#include <malloc.h>
void main( void )
{
   func( 10 );
}
void func( register int i )
{
   _alloca( i );
}
```

  To ensure that the stack pointer is properly restored, make sure that any function referencing _alloca declares at least one local variable.

- The pointer value returned by _alloca should never be passed as an argument to free.
- The _alloca function should never be used in an expression that is an argument to a function.

Return Value

The _alloca routine returns a void pointer to the allocated space, which is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than char, use a type cast on the return value. The return value is NULL if the space cannot be allocated.

Use _alloca for compatibility with ANSI naming conventions of non-ANSI functions. Use alloca and link with OLDNAMES.LIB for UNIX compatibility.

calloc functions
malloc functions
realloc functions

Standards: UNIX

16-Bit: MS-DOS

```
/* ALLOCA.C: This program checks the stack
 * space available before and after using the
 * _alloca function to allocate space on the stack.
 */
#include <malloc.h>
#include <stdio.h>
void main( void )
{
   char *buffer;
   printf( "Bytes available on stack: %u\n", _stackavail() );
   /* Allocate memory for string. */
   buffer = _alloca( 120 * sizeof( char ) );
   printf( "The _alloca function just allocated" );
   printf( " memory from the program stack.\n" );
   printf( "Enter a string: " );
   gets( buffer );
   printf( "\"%s\" was stored in the program stack.\n", buffer );
   printf( "Bytes available on stack: %u\n", _stackavail() );
}
```

_arc Functions

#include <graph.h>

Syntax short __far _arc(short *x1*, short *y1*, short *x2*, short *y2*, short *x3*, short *y3*, short *x4*, short *y4);*

short __far _arc_w(double *x1*, double *y1*, double *x2*, double *y2*, double *x3*, double *y3*, double *x4*, double *y4*);

short __far _arc_wxy(struct _wxycoord __far **pwxy1*, struct _wxycoord __far **pwxy2*, struct _wxycoord __far **pwxy3*, struct _wxycoord __far **pwxy4*);

Parameter	Description
x1, *y1*	Upper-left corner of bounding rectangle
x2, *y2*	Lower-right corner of bounding rectangle
x3, *y3*	Second point of start vector (center of bounding rectangle is first point)
x4, *y4*	Second point of end vector (center of bounding rectangle is first point)
pwxy1	Upper-left corner of bounding rectangle
pwxy2	Lower-right corner of bounding rectangle
pwxy3	Second point of start vector (center of bounding rectangle is first point)
pwxy4	Second point of end vector (center of bounding rectangle is first point)

The _arc functions draw elliptical arcs. The center of the arc is the center of the bounding rectangle, which is defined by points (*x1*, *y1*) and (*x2*, *y2*) for _arc and _arc_w and by points *pwxy1* and *pwxy2* for _arc_wxy. The arc starts where it intersects an imaginary line extending from the center of the arc through (*x3*, *y3*) for _arc and _arc_w and through *pwxy3* for _arc_wxy. It is drawn counterclockwise about the center of the arc, ending where it intersects an imaginary line extending from the center of the arc through (*x4*, *y4*) for _arc and _arc_w and through *pwxy4* for _arc_wxy.

The _arc routine uses the view coordinate system. The _arc_w and _arc_wxy functions use the real-valued window coordinate system.

In each case, the arc is drawn using the current color. Because an arc does not define a closed area, it is not filled.

Return Value

These functions return a nonzero value if the arc is successfully drawn; otherwise, they return 0.

_ellipse functions
_lineto functions
_pie functions
_rectangle functions
_setcolor

Standards: None
16-Bit: MS-DOS, QWIN

```
/* ARC.C: This program draws a simple arc. */
#include <graph.h>
#include <stdlib.h>
#include <conio.h>
void main( void )
{
   short x, y;
   struct _xycoord xystart, xyend, xyfill;
   /* Find a valid graphics mode */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   /* Draw arcs         */
   x = 100; y = 100;
   _arc( x - 60, y - 60, x, y, x - 30, y - 60, x - 60, y - 30 );
   _arc( x + 60, y + 60, x, y, x, y + 30, x + 30, y );
   /* Get endpoints of second arc and enclose the figure, then fill it. */
   _getarcinfo( &xystart, &xyend, &xyfill );
   _moveto( xystart.xcoord, xystart.ycoord );
   _lineto( xyend.xcoord,   xyend.ycoord );
   _floodfill( xyfill.xcoord, xyfill.ycoord, _getcolor() );
   _getch();
   _setvideomode( _DEFAULTMODE );
}
```

asctime

#include <time.h>

Syntax char *asctime(const struct tm **timeptr*);

Argument	**Description**
timeptr	Time/date structure

The asctime function converts a time stored as a structure to a character string. The *timeptr* value is usually obtained from a call to gmtime or localtime, both of which return a pointer to a tm structure, defined in TIME.H.

The fields of the structure type tm store the following values, each of which is an int:

Field	**Description**
tm_sec	Seconds after the minute (0-59)
tm_min	Minutes after the hour (0-59)
tm_hour	Hours since midnight (0-23)
tm_mday	Day of the month (1-31)
tm_mon	Month (0-11; January = 0)
tm_year	Year (current year minus 1900)
tm_wday	Day of week (0-6; Sunday = 0)
tm_yday	Day of year (0-365; January 1 = 0)
tm_isdst	Nonzero if daylight saving time is in effect, otherwise 0

The string result produced by asctime contains exactly 26 characters and has the form of the following example:

```
Wed Jan 02 02:03:55 1980\n\0
```

A 24-hour clock is used. All fields have a constant width. The newline character (\n) and the null character ('\0') occupy the last two positions of the string. The asctime function uses a single statically allocated buffer to hold the return string. Each call to this routine destroys the result of the previous call.

Return Value

The asctime function returns a pointer to the character string result. There is no error return.

ctime
_ftime
gmtime
localtime
time
_tzset

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* ASCTIME.C: This program places the system time
 * in the long integer aclock, translates it into the
 * structure newtime and then converts it to string
 * form for output, using the asctime function.
 */
#include <time.h>
#include <stdio.h>
struct tm *newtime;
time_t aclock;
void main( void )
{
   time( &aclock );                 /* Get time in seconds */
   newtime = localtime( &aclock );  /* Convert time to struct tm form  */
   /* Print local time as a string */
   printf( "The current date and time are: %s", asctime( newtime ) );
}
```

asin Functions

#include <math.h>

#include <errno.h>

Syntax double asin(double *x*);

long double asinl(long double *x*);

Parameter	Description
x	Value whose arcsine is to be calculated

The asin functions calculate the arcsine of *x* in the range $-\pi/2$ to $\pi/2$ radians. The value of *x* must be between -1 and 1. The asinl function is the 80-bit counterpart, which uses an 80-bit, 10-byte coprocessor form of parameters and return values. See the long double functions for more details on this data type.

Return Value

The asin functions return the arcsine result. If *x* is less than -1 or greater than 1, asin sets errno to EDOM, prints a _DOMAIN error message to stderr, and returns 0.

Error handling can be modified by using the _matherr (or _matherrl) routine.

asin

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

asinl

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* ASINCOS.C: This program prompts for a value in the range
 * -1 to 1. Input values outside this range will produce
 * _DOMAIN error messages.If a valid value is entered, the
 * program prints the arcsine and the arccosine of that value.
 */
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
void main( void )
{
   double x, y;
   printf( "Enter a real number between -1 and 1: " );
   scanf( "%lf", &x );
   y = asin( x );
   printf( "Arcsine of %f = %f\n", x, y );
   y = acos( x );
   printf( "Arccosine of %f = %f\n", x, y );
}
```

assert

#include <assert.h>

#include <stdio.h>

Syntax void assert(int *expression*);

Parameter	Description
expression	C expression specifying assertion being tested

The assert routine prints a diagnostic message and calls the abort routine if *expression* is false (0). The diagnostic message has the form

```
Assertion failed: expression, file filename, line linenumber
```

where *filename* is the name of the source file and *linenumber* is the line number of the assertion that failed in the source file. No action is taken if *expression* is true (nonzero).

In Windows, the diagnostic message appears in an "Assertion Failed" pop-up window.

The assert routine is typically used in program development to identify program logic errors. The given expression should be chosen so that it holds true only if the program is operating as intended. After a program has been debugged, the special "no debug" identifier NDEBUG can be used to remove assert calls from the program. If NDEBUG is defined (by any value) with a /D command-line option or with a #define directive, the C preprocessor removes all assert calls from the program source. If NDEBUG is defined with a #define directive, it must appear before inclusion of ASSERT.H.

The assert routine is implemented as a macro.

Return Value

None.

abort
raise
signal

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* ASSERT.C: In this program, the analyze_string function uses
 * the assert function to test several conditions related to
 * string and length. If any of the conditions fails, the program
 * prints a message indicating what caused the failure.
 */
#include <stdio.h>
#include <assert.h>
#include <string.h>
void analyze_string( char *string );   /* Prototype */
void main( void )
{
   char  test1[] = "abc", *test2 = NULL, test3[] = "";
   printf ( "Analyzing string '%s'\n", test1 );
   analyze_string( test1 );
   printf ( "Analyzing string '%s'\n", test2 );
   analyze_string( test2 );
   printf ( "Analyzing string '%s'\n", test3 );
   analyze_string( test3 );
}
/* Tests a string to see if it is NULL, empty, or longer than 0 characters */
void analyze_string( char * string )
{
   assert( string != NULL );        /* Cannot be NULL */
   assert( *string != '\0' );       /* Cannot be empty */
   assert( strlen( string ) > 2 );  /* Length must be greater than 2 */
}
```

atan Functions

#include <math.h>

Syntax double atan(double *x*);

double atan2(double *y*, double *x*);

long double atanl(long double *x*);

long double atan2l(long double *y*, long double *x*);

Parameter	Description
x, *y*	Any number

The atan family of functions calculates the arctangent of *x*, and the atan2 family of functions calculates the arctangent of *y*/*x*. The atan group returns a value in the range -π/2 to π/2 radians, and the atan2 group returns a value in the range -π to π radians. The atan2 functions use the signs of both parameters to determine the quadrant of the return value. The atan2 functions are well defined for every point other than the origin, even if *x* equals 0 and *y* does not equal 0.

Return Value

The atan family of functions returns the arctangent result. If both parameters of atan2 or atan2l are 0, the function sets errno to EDOM, prints a _DOMAIN error message to stderr, and returns 0.

Error handling can be modified by using the _matherr (or _matherrl) routine.

atan, atan2

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

atanl, atan2l

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* ATAN.C: This program calculates
 * the arctangent of 1 and -1.
 */
#include <math.h>
#include <stdio.h>
#include <errno.h>
void main( void )
{
   double x1, x2, y;
   printf( "Enter a real number: " );
   scanf( "%lf", &x1 );
   y = atan( x1 );
   printf( "Arctangent of %f: %f\n", x1, y );
   printf( "Enter a second real number: " );
   scanf( "%lf", &x2 );
   y = atan2( x1, x2 );
   printf( "Arctangent of %f / %f: %f\n", x1, x2, y );
}
```

atexit, _fatexit

#include <stdlib.h> Required only for function declarations

Syntax int atexit(void (__cdecl **func*)(void));

int __far _fatexit(void (__cdecl __far **func*)(void));

Parameter	**Description**
func	Function to be called

The atexit function is passed the address of a function (*func*) to be called when the program terminates normally. Successive calls to atexit create a register of functions that are executed in LIFO (last-in-first-out) order. No more than 32 functions can be registered with atexit or _onexit. The functions passed to atexit cannot take parameters.

The _fatexit function is a far version of atexit; it can be used with any memory model.

Return Value

Both atexit and _fatexit return 0 if successful, or a nonzero value if an error occurs (e.g., if there are already 32 exit functions defined).

Use the ANSI-standard atexit function (rather than the similar _onexit function) whenever ANSI portability is desired.

abort
exit
_exit
_onexit

atexit

Standards: ANSI
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fatexit

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* ATEXIT.C: This program pushes four functions onto
 * the stack of functions to be executed when atexit
 * is called. When the program exits, these programs
 * are executed on a "last in, first out" basis.
 */
#include <stdlib.h>
#include <stdio.h>
void fn1( void ), fn2( void ), fn3( void ), fn4( void );
void main( void )
{
   atexit( fn1 );
   atexit( fn2 );
   atexit( fn3 );
   atexit( fn4 );
   printf( "This is executed first.\n" );
}
void fn1()
{
   printf( "next.\n" );
}
void fn2()
{
   printf( "executed " );
}
void fn3()
{
   printf( "is " );
}
void fn4()
{
   printf( "This " );
}
```

atof, atoi, atol, _atold

#include<math.h> (for atof, _atold)

#include <stdlib.h> (for atof, _atold, atoi, atol)

Syntax double atof(const char **string*);'

long double _atold(const char **string*);

int atoi(const char **string*);

long atol(const char **string*);

Parameter	**Description**
string	String to be converted

These functions convert a character string to a double-precision floating-point value (atof), an integer value (atoi), a long integer value (atol), or a long double value (_atold). The input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

The string size that can be handled by the atof or _atold function is limited to 100 characters.

The function stops reading the input string at the first character that it cannot recognize as part of a number. This character may be the null character ('\0') terminating the string.

The atof and _atold functions expect *string* to have the following form:

[*whitespace*] [*sign*] [*digits*] [.*digits*] [{d | D | e | E }[*sign*]*digits*]

A *whitespace* consists of space and/or tab characters, which are ignored; *sign* is either plus (+) or minus (-); and *digits* are one or more decimal digits. If no digits appear before the decimal point, at least one must appear after the decimal point. The decimal digits may be followed by an exponent, which consists of an introductory letter (d, D, e, or E) and an optionally signed decimal integer.

The atoi and atol functions do not recognize decimal points or exponents. The *string* argument for these functions has the form

[*whitespace*] [*sign*]*digits*

where *whitespace*, *sign*, and *digits* are exactly as described above for atof.

Return Value

Each function returns the double, long double, int, or long value produced by interpreting the input characters as a number. The return value is 0 (for atoi), 0L (for atol), and 0.0 (for atof and _atold) if the input cannot be converted to a value of that type. The return value is undefined in case of overflow.

_ecvt
_fcvt
_gcvt
strtod

atof, atoi, atol

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_atold

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* ATOF.C: This program shows how numbers stored
 * as strings can be converted to numeric values
 * using the atof, atoi, and atol functions.
 */
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   char *s; double x; int i; long l;
   s = "  -2309.12E-15";    /* Test of atof */
   x = atof( s );
   printf( "atof test:  ASCII string: %s\tfloat:     %e\n", s, x );
   s = "7.8912654773d210";  /* Test of atof */
   x = atof( s );
   printf( "atof test:  ASCII string: %s\tfloat:     %e\n", s, x );
   s = "  -9885 pigs";      /* Test of atoi */
   i = atoi( s );
   printf( "atoi test:  ASCII string: %s\t\tinteger: %d\n", s, i );
   s = "98854 dollars";     /* Test of atol */
   l = atol( s );
   printf( "atol test:  ASCII string: %s\t\tlong:    %ld\n", s, l );
}
```

_bdos

#include <dos.h>

Syntax int _bdos(int *dosfunc*, unsigned int *dosdx*, unsigned int *dosal*);

Parameter	Description
dosfunc	Function number
dosdx	DX register value
dosal	AL register value

The _bdos function invokes the MS-DOS system call specified by *dosfunc* after placing the values specified by *dosdx* and *dosal* in the DX and AL registers, respectively. The _bdos function executes an INT 21H instruction to invoke the system call. When the system call is complete, _bdos returns the contents of the AX register.

The _bdos function is intended to be used to invoke MS-DOS system calls that either take no parameters or take parameters only in the DX (DH, DL) and/or AL registers.

Do not use the _bdos function to call interrupts that modify the DS register. Instead, use the _intdosx or _int86x function.

This call should not be used to invoke system calls that indicate errors by setting the carry flag. Because C programs do not have access to this flag, your program cannot determine whether the return value is an error code. The _intdos function should be used in these cases.

Return Value

The _bdos function returns the value of the AX register after the system call has completed.

_intdos
_intdosx

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* BDOS.C: This example calls DOS function 0x9 (display
 * string) to display a $-terminated string.
 */
#include <dos.h>
/* Function 0x09 assumes that DS will contain segment of the string.
 * This will be true for all memory models if the string is declared near. */
char __near str[] = "Hello world!\r\n$";
void main( void )
{
   /* Offset of string must be in DX, segment in DS. AL is not needed,
    * so 0 is used.
    */
    _bdos( 0x09, (int)str, 0 );
}
```

Bessel Functions

#include <math.h>

Syntax double _j0(double *x);*

double _j1(double *x);*

double _jn(int *n*, double *x*);

double _y0(double *x);*

double _y1(double *x);*

double _yn(int *n*, double *x);*

long double _j0l(long double *x);*

long double _jnl(int *n*, long double *x*);

long double _j1l(long double *x);*

long double _y0l(long double *x);*

long double _y1l(long double *x);*

long double _ynl(int *n*, long double *x);*

Parameter	**Description**
x	Floating-point value
n	Integer order

The _j0, _j1, and _jn routines return Bessel functions of the first kind: orders 0, 1, and *n*, respectively.

The _y0, _y1, and _yn routines return Bessel functions of the second kind: orders 0, 1, and *n*, respectively. The parameter *x* must be positive.

The long double versions of these functions are the 80-bit counterparts and use the 80-bit, 10-byte coprocessor form of parameters and return values. See the long double functions for more details on this data type.

The Bessel functions are explained more fully in most mathematics reference books, such as the *Handbook of Mathematical Functions* (Abramowitz and Stegun; Washington: U.S. Government Printing Office, 1964). These functions are commonly used in the mathematics of electromagnetic wave theory.

Return Value

These functions return the result of a Bessel function of *x*.

For _y0, _y1, or _yn, if *x* is negative, the routine sets errno to EDOM , prints a _DOMAIN error message to stderr, and returns -HUGE_VAL.

Error handling can be modified by using the _matherr (or _matherrl) routine.

Use _j0, _j1, _jn, _y0, _y1, and _yn for compatibility with ANSI naming conventions of non-ANSI functions. Use j0, j1, jn, y0, y1, and yn and link with OLDNAMES.LIB for UNIX compatibility.

_matherr

_j0, _j1, _jn, _y0, _y1, _yn

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_j0l, _j1l, _jnl, _y0l, _y1l, _ynl

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* BESSEL.C: This program illustrates Bessel functions,
 * including:    _j0   _j1   _jn   _y0   _y1   _yn
 */
#include <math.h>
#include <stdio.h>
void main( void )
{
   double x = 2.387;
   int n = 3, c;
   printf( "Bessel functions for x = %f:\n", x );
   printf( "  Kind\t\tOrder\tFunction\tResult\n\n" );
   printf( "  First\t\t0\t_j0( x )\t%f\n", _j0( x ) );
   printf( "  First\t\t1\t_j1( x )\t%f\n", _j1( x ) );
   for( c = 2; c < 5; c++ )
      printf( "  First\t\t%d\t_jn( n, x )\t%f\n", c, _jn( c, x ) );
   printf( "  Second\t0\t_y0( x )\t%f\n", _y0( x ) );
   printf( "  Second\t1\t_y1( x )\t%f\n", _y1( x ) );
   for( c = 2; c < 5; c++ )
      printf( "  Second\t%d\t_yn( n, x )\t%f\n", c, _yn( c, x ) );
}
```

_bfreeseg

#include <malloc.h> Required only for function declarations

Syntax int _bfreeseg(__segment *seg*);

Parameter	**Description**
seg	Segment selected

The _bfreeseg function frees a based heap. The *seg* parameter is a based heap returned by an earlier call to _bheapseg. It specifies the based heap to be freed.

The specified segment is freed completely regardless of whether the blocks it contains are free or allocated. After a _bfreeseg call, the *seg* value is invalid and should not be used.

Return Value

The _bfreeseg function returns 0 if successful and -1 in the case of an error.

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_bheapseg

#include <malloc.h> Required only for function declarations

Syntax __segment _bheapseg(size_t *size*);



Parameter	Description
size	Segment size to allocate

The _bheapseg function allocates a based-heap segment of at least *size* bytes. (The block may be larger than *size* bytes because of space required for alignment and for maintenance information.)

The value returned by _bheapseg is the identifier of the based-heap segment. This value should be saved and used in subsequent calls to other based-heap functions. If the original block of memory is depleted (e.g., by calls to _bmalloc and _brealloc), the run-time code will try to enlarge the heap as necessary.

The _bheapseg function can be called repeatedly. For each call, the run-time library will allocate a new based-heap segment.

Return Value

The _bheapseg function returns the newly allocated segment selector; save this value for use in subsequent based-heap functions. A return value of _NULLSEG indicates failure.

Always check the return from the _bheapseg function (especially when it is used in real mode), even if the amount of memory requested is small.

_bfreeseg
calloc functions
free functions
malloc functions
realloc functions

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* BHEAPSEG.C: This program C illustrates dynamic
 * allocation of based memory using functions
 * _bheapseg, _bfreeseg, _bmalloc, and _bfree.
 */
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>
void main( void )
{
   __segment seg;
   char __based( seg ) *outstr, __based( seg ) *instr;
   char __based( seg ) *pout,   __based( seg ) *pin;
   char tmpstr[80];
   int  len;
   printf( "Enter a string: " );
   gets( tmpstr );
   /* Request a based heap. Use based so that memory won't be
    * taken from near heap.
    */
   if( (seg = _bheapseg( 1000 )) == _NULLSEG )
      exit( 1 );
   /* Allocate based memory for two strings. */
   len = strlen( tmpstr );
   if( ((instr  = _bmalloc( seg, len + 1 )) == _NULLOFF) ||
       ((outstr = _bmalloc( seg, len + 1 )) == _NULLOFF) )
       exit( 1 );
   /* Copy a lowercased string to dynamic memory. The based memory
    * is far when addressed as a whole.
    */
   _fstrlwr( _fstrcpy( (char __far *)instr, (char __far *)tmpstr ) );
    /* Copy input string to output string in reversed order. When reading
     * and writing individual characters from a based heap, the compiler
     * will try to process them as near, thus speeding up the processing.
     */
   for( pin = instr + len - 1, pout = outstr;
        pout < outstr + len; pin--, pout++ )
      *pout = *pin;
   *pout = '\0';
   /* Display strings. Again, strings as a whole are far. */
   printf( "Input:  %Fs\n", (char __far *)instr );
   printf( "Output: %Fs\n", (char __far *)outstr );
   /* Free blocks and release based heap. */
   _bfree( seg, instr );
   _bfree( seg, outstr );
   _bfreeseg( seg );
}
```

_bios_disk

#include <bios.h>

Syntax unsigned _bios_disk(unsigned *service*, struct _diskinfo_t **diskinfo*);

Parameter	Description
service	Disk function desired
diskinfo	Disk parameters

The _bios_disk routine uses INT 0x13 to provide several disk-access functions. The *service* parameter selects the function desired, while the *diskinfo* structure provides the necessary parameters. Note that the low-level disk operations allowed by the _bios_disk routine are very dangerous to use because they perform direct manipulation of the disk.

The *diskinfo* structure provides the following parameters:

Element	Description
unsigned drive	Drive number
unsigned head	Head number
unsigned track	Track number
unsigned sector	Starting sector number
unsigned nsectors	Number of sectors to read, write, or compare
void far *buffer	Memory location to write to, read from, or compare

The *service* parameter can be set to one of the following manifest constants:

_DISK_FORMAT
: Formats the track specified by *diskinfo*. The *head* and *track* fields indicate the track to format. Only one track can be formatted in a single call. The *buffer* field points to a set of sector markers. The format of the markers depends on the type of disk drive; see a technical reference to the PC BIOS to determine the marker format. The high-order byte (AH) of the return value contains the status of the call; 0 equals success. If there is an error, the high-order byte will contain a set of status flags, as defined below under Return Value.

_DISK_READ
: Reads one or more disk sectors into memory. This service uses all fields of the structure pointed to by *diskinfo*, as defined earlier in this section. If no error occurs, the function returns 0 in the high-order byte and the number of sectors read in the low-order byte. If there is an error, the high-order byte (AH) will contain a set of status flags, as defined below under Return Value.

_DISK_RESET
: Forces the disk controller to do a hard reset, preparing for floppy-disk I/O. This is useful after an error occurs in another operation, such as a read. If this service is specified, the *diskinfo* parameter is ignored. Status is returned in the 8 high-order bits (AH) of the return value. If there is an error, the high-order byte will contain a set of status flags, as defined below under Return Value.

_DISK_STATUS
: Obtains the status of the last disk operation. If this service is specified, the *diskinfo* parameter is ignored. Status is returned in the 8 low-order bits (AL) of the return value. If there is an error, the low-order byte (AL) will contain a set of status flags, as defined below under Return Value.

_DISK_VERIFY
: Checks the disk to be sure the specified sectors exist and can be read. It also runs a CRC (cyclic

redundancy check) test. This service uses all fields (except *buffer*) of the structure pointed to by *diskinfo*, as defined earlier in this section. If no error occurs, the function returns 0 in the high-order byte (AH) and the number of sectors compared in the low-order byte (AL). The error status flags are listed below under Return Value.

_DISK_WRITE

Writes data from memory to one or more disk sectors. This service uses all fields of the structure pointed to by *diskinfo*, as defined earlier in this section. If no error occurs, the function returns 0 in the high-order byte (AH) and the number of sectors written in the low-order byte (AL). If there is an error, the high-order byte will contain a set of status flags, as defined below under Return Value.

Return Value

The _bios_disk function returns the value in the AX register after the BIOS interrupt.

Bits	Meaning
0x00	No error
0x01	Invalid request or a bad command
0x02	Address mark not found
0x03	Disk write protected
0x04	Sector not found
0x05	Reset failed
0x06	Floppy disk removed
0x07	Drive parameter activity failed
0x08	Direct Memory Access (DMA) overrun
0x09	DMA crossed 64K boundary
0x0A	Bad sector flag detected
0x0B	Bad track flag detected
0x0C	Media type not found
0x0D	Invalid number of sectors on format
0x0E	Control data access mark detected
0x0F	DMA arbitration level out of range
0x10	Data read (CRC or ECC) error
0x11	Corrected data read (ECC) error
0x20	Controller failure
0x40	Seek error
0x80	Disk timed out or failed to respond
0xAA	Drive not ready
0xBB	Undefined error
0xCC	Write fault on drive
0xE0	Status error
0xFF	Sense operation failed

Standards: None

16-Bit: MS-DOS, WIN, WIN DLL

```
/* BDISK.C: This program first attempts to verify a
 * disk by using an invalid disk head number. After
 * printing the return value error code, the program
 * verifies the disk by using a valid disk head code.
 */
#include <conio.h>
#include <stdio.h>
#include <bios.h>
void main( void )
{
   unsigned status = 0;
   struct _diskinfo_t disk_info;
   disk_info.drive    = 0;
   disk_info.head     = 10;   /* Invalid head number */
   disk_info.track    = 1; disk_info.sector   = 2;
   disk_info.nsectors = 8;
   printf( "Insert disk in drive A: and press any key\n" );
   _getch();
   status = _bios_disk( _DISK_VERIFY, &disk_info );
   printf( "Return value: 0x%.4x\n", status );
   if( status & 0xff00 )      /* Error if high byte is 0 */
      printf( "Seek error\n" );
    else
      printf( "No seek error\n" );
   printf( "Press any key\n" );
   _getch(); disk_info.head = 0;        /* Valid head number */
   status = _bios_disk( _DISK_VERIFY, &disk_info );
   printf( "Return value: 0x%.4x\n", status );
   if( status & 0xff00 )      /* Error if high byte is 0 */
      printf( "Seek error\n" );
   else
      printf( "No seek error\n" );
}
```

_bios_equiplist

#include <bios.h>

Syntax unsigned _bios_equiplist(void);

The _bios_equiplist routine uses INT 0x11 to determine what hardware and peripherals are currently installed on the machine.

Return Value

The function returns the AX value, which is a set of bits indicating what equipment is installed, as defined below:

Bits	Meaning
0	True (1) if disk drive(s) installed
1	True (1) if math coprocessor installed
2 -3	System RAM in 16K blocks (16-64K)
4 -5	Initial video mode:
	00 = Reserved
	01 = 40 x 25 color
	10 = 80 x 25 color
	11 = 80 x 25 monochrome
6 -7	Number of floppy-disk drives installed (00 = 1, 01 = 2, etc.)
8	False (0) if and only if a Direct Memory Access (DMA) chip is installed
9 -11	Number of RS232 serial ports installed
12	True (1) if and only if a game adapter is installed
13	True (1) if and only if an internal modem is installed
14 -15	Number of printers installed

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```c
/* BEQUIPLI.C: This program checks for the presence of diskettes.
 */
#include <bios.h>
#include <stdio.h>
void main( void )
{
   unsigned equipment;
   equipment = _bios_equiplist();
   printf( "Equipment bits: 0x%.4x\n", equipment );
   if( equipment & 0x1000 )      /* Check for game adapter bit */
      printf( "Game adapter installed\n" );
   else
      printf( "No game adapter installed\n" );
}
```

_bios_keybrd

#include <bios.h>

Syntax unsigned _bios_keybrd(unsigned *service*);

Parameter	**Description**
service	Keyboard function desired

The _bios_keybrd routine uses INT 0x16 to access the keyboard services. The *service* parameter can be any of the following manifest constants:

_KEYBRD_READ, _NKEYBRD_READ

Reads the next character from the keyboard. If no character has been typed, the call will wait for one. If the low-order byte of the return value is nonzero, the call contains the ASCII value of the character typed. The high-order byte contains the keyboard scan code for the character. The _NKEYBRD_READ constant is used with enhanced keyboards to obtain the scan codes for function keys F11 and F12 and the cursor control keys.

_KEYBRD_READY, _NKEYBRD_READY

Checks whether a keystroke is waiting to be read and, if so, reads it. The return value is 0 if no keystroke is waiting, or it is the character waiting to be read, in the same format as the _KEYBRD_READ or _NKEYBRD_READ return. This service does not remove the waiting character from the input buffer, as does the _KEYBRD_READ or _NKEYBRD_READ service. The _NKEYBRD_READY constant is used with enhanced keyboards to obtain the scan codes for function keys F11 and F12 and the cursor control keys.

_KEYBRD_SHIFTSTATUS, _NKEYBRD_SHIFTSTATUS

Returns the current SHIFT-key status. _KEYBRD_SHIFTSTATUS returns only low byte. The _NKEYBRD_SHIFTSTATUS constant is used to get a full 16-bit status value. Any combination of the following bits may be set:

Bit	**Meaning if True**
00H	Rightmost SHIFT key pressed
01H	Leftmost SHIFT key pressed
02H	Either CTRL key pressed
03H	Either ALT key pressed
04H	SCROLL LOCK on
05H	NUM LOCK on
06H	CAPS LOCK on
07H	In insert mode (INS)
08H	Left CTRL key pressed
09H	Left ALT key pressed
0AH	Right CTRL key pressed
0BH	Right ALT key pressed
0CH	SCROLL LOCK key pressed
0DH	NUM LOCK key pressed
0EH	CAPS LOCK key pressed
0FH	SYS REQ key pressed

Return Value

With the ...READ and ...SHIFTSTATUS parameters, the _bios_keybrd function returns the contents of the AX register after the BIOS call.

With the ...READY parameter, _bios_keybrd returns 0 if there is no key. If there is a key, _bios_keybrd returns the key waiting to be read (i.e., the same value as _KEYBRD_READ).

With the ...READ and the ...READY parameters, the _bios_keybrd function returns -1 if CTRL+BREAK has been pressed and is the next keystroke to be read.

Standards: None

16-Bit: MS-DOS, WIN, WIN DLL

```
/* BKEYBRD.C: This program prints a message on the
 * screen until the right SHIFT key is pressed.
 */
#include <bios.h>
#include <stdio.h>
void main( void )
{
   while( !(_bios_keybrd( _KEYBRD_SHIFTSTATUS ) & 0001) )
      printf( "Use the right SHIFT key to stop this message\n" );
   printf( "Right SHIFT key pressed\n" );
}
```

_bios_memsize

#include <bios.h>

Syntax unsigned _bios_memsize(void);

The _bios_memsize routine uses INT 0x12 to determine the total amount of conventional main memory installed.

Return Value

The routine returns the total amount of installed memory in 1K blocks. The maximum return value is 640, representing 640K of conventional main memory.

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* BMEMSIZE.C: This program displays the amount of memory installed.
 */
#include <bios.h>
#include <stdio.h>
void main( void )
{
   unsigned memory;
   memory = _bios_memsize();
   printf ( "The amount of memory installed is: %dK\n", memory );
}
```

_bios_printer

#include <bios.h>

Syntax unsigned _bios_printer(unsigned *service*, unsigned *printer*, unsigned *data*);

Parameter	Description
service	Printer function desired
printer	Target printer port
data	Output data

The _bios_printer routine uses INT 0x17 to perform printer output services for parallel printers. The *printer* parameter specifies the affected printer, where 0 is LPT1, 1 is LPT2, and so forth.

Some printers do not support the full set of signals. As a result, the "Out of Paper" condition, for example, may not be returned to your program.

The *service* parameter can be any of the following manifest constants:

_PRINTER_INIT
Initializes the selected printer. The *data* parameter is ignored.

_PRINTER_STATUS
Returns the printer status. The *data* parameter is ignored.

_PRINTER_WRITE
Sends the low-order byte of *data* to the printer specified by *printer*.

Return Value

The _bios_printer function returns the value in the AX register after the BIOS interrupt. The high-order byte (AH) of the return value indicates the printer status after the operation, as defined below:

Bit	Meaning if True
0	Printer timed out
1	Not used
2	Not used
3	I/O error
4	Printer selected
5	Out of paper
6	Acknowledge
7	Printer not busy

Standards: None

16-Bit: MS-DOS, WIN, WIN DLL

```c
/* BPRINTER.C: This program checks the status
 * of the printer attached to LPT1 when it is
 * off line, then initializes the printer.
 */
#include <bios.h>
#include <conio.h>
#include <stdio.h>
#define LPT1 0
void main( void )
{
   unsigned status;
   printf ( "Place printer off line and press any key\n" );
   _getch();
   status = _bios_printer( _PRINTER_STATUS, LPT1, 0 );
   printf( "Status with printer off line: 0x%.4x\n\n", status );
   printf( "Put the printer on line and then\n" );
   printf( "Press any key to initialize printer\n" );
   _getch();
   status = _bios_printer( _PRINTER_INIT, LPT1, 0 );
   printf( "Status after printer initialized: 0x%.4x\n", status );
}
```

_bios_serialcom

#include <bios.h>

Syntax unsigned _bios_serialcom(unsigned *service*, unsigned *serial_port*, unsigned *data*);

Parameter	Description
service	Communications service
serial_port	Serial port to use
data	Port configuration bits

The _bios_serialcom routine uses INT 0x14 to provide serial communications services. The *serial_port* parameter is set to 0 for COM1, to 1 for COM2, and so on.

The _bios_serialcom routine may not be able to establish reliable communications at baud rates in excess of 1,200 baud (_COM_1200) due to the overhead associated with servicing computer interrupts. Faster data communication rates are possible with more direct programming of serial-port controllers. See *C Programmer's Guide to Serial Communications* for more details on serial-communications programming in C.

The *service* parameter can be set to one of the following manifest constants:

_COM_INIT
  Sets the port to the parameters specified in the *data* parameter

_COM_SEND
  Transmits the *data* characters over the selected serial port

_COM_RECEIVE
  Accepts an input character from the selected serial port

_COM_STATUS
  Returns the current status of the selected serial port

The *data* parameter is ignored if *service* is set to _COM_RECEIVE or _COM_STATUS. The *data* parameter for _COM_INIT is created by combining (with the OR operator) one or more of the following constants:

_COM_CHR7	7 data bits
_COM_CHR8	8 data bits
_COM_STOP1	1 stop bit
_COM_STOP2	2 stop bits
_COM_NOPARITY	No parity
_COM_EVENPARITY	Even parity
_COM_ODDPARITY	Odd parity
_COM_110	110 baud
_COM_150	150 baud
_COM_300	300 baud
_COM_600	600 baud
_COM_1200	1200 baud
_COM_2400	2400 baud
_COM_4800	4800 baud
_COM_9600	9600 baud

The default value of *data* is 1 stop bit, no parity, and 110 baud.

Return Value

The function returns a 16-bit integer whose high-order byte contains status bits. The meaning of the low-order byte varies, depending on the *service* value. The high-order bits have the following meanings:

Bit	Meaning if Set
15	Timed out
14	Transmission-shift register empty
13	Transmission-hold register empty
12	Break detected
11	Framing error
10	Parity error
9	Overrun error
8	Data ready

When *service* is _COM_SEND, bit 15 will be set if *data* could not be sent.

When *service* is _COM_RECEIVE, the byte read will be returned in the low-order bits if the call is successful. If an error occurs, any of the bits 9, 10, 11, or 15 will be set.

When *service* is _COM_INIT or _COM_STATUS, the low-order bits are defined as follows:

Bit	Meaning if Set
7	Receive-line signal detected
6	Ring indicator
5	Data set ready
4	Clear to send
3	Change in receive-line signal detected
2	Trailing-edge ring indicator
1	Change in data-set-ready status
0	Change in clear-to-send status

Note that this function works only with IBM personal computers and true compatibles.

Standards: None

16-Bit: MS-DOS, WIN, WIN DLL

```
/* BSERIALC.C: This program checks the
 * status of serial port COM1.
 */
#include <bios.h>
#include <stdio.h>
void main( void )
{
   unsigned com1_status;
   com1_status =  _bios_serialcom( _COM_STATUS, 0, 0 );
   printf ( "COM1 status: 0x%.4x\n", com1_status );
}
```

_bios_timeofday

#include <bios.h>

Syntax unsigned _bios_timeofday(unsigned *service*, long **timeval*);

Parameter	Description
service	Time function desired
timeval	Clock count

The _bios_timeofday routine uses INT 0x1A to get or set the clock count. The *service* parameter can be either of the following manifest constants:

_TIME_GETCLOCK
: Copies the current value of the clock count to the location pointed to by *timeval*. If midnight has not passed since the last time the system clock was read or set, the function returns 0; otherwise, the function returns 1.

_TIME_SETCLOCK
: Sets the current value of the system clock to the value in the location pointed to by *timeval*. There is no return value.

Return Value

The _bios_timeofday function returns the value in the AX register after the BIOS interrupt.

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* BTIMEOFD.C: This program gets the current system
 * clock count before and after a "do-nothing" loop
 * and displays the difference.
 */
#include <bios.h>
#include <stdio.h>
void main( void )
{
   long i, begin_tick, end_tick;
   _bios_timeofday( _TIME_GETCLOCK, &begin_tick );
   printf( "Beginning tick count: %lu\n", begin_tick );
   for( i = 1; i <= 900000; i++ )
      ;
   _bios_timeofday( _TIME_GETCLOCK, &end_tick );
   printf( "Ending tick count:    %lu\n", end_tick );
   printf( "Elapsed ticks:        %lu\n", end_tick - begin_tick );
}
```

bsearch

#include <stdlib.h> Required for ANSI compatibility

#include <search.h> Required only for function declarations

Syntax void *bsearch(const void **key*, const void **base*, size_t *num*, size_t *width*, int (__cdecl **compare*)(const void **elem1*, const void ***elem2*));

Parameter	Description
key	Object to search for
base	Pointer to base of search data
num	Number of elements
width	Width of elements
compare	Function that compares two elements: *elem1* and *elem2*
elem1	Pointer to the key for the search
elem2	Pointer to the array element to be compared with the key

The bsearch function performs a binary search of a sorted array of *num* elements, each of *width* bytes in size. The *base* value is a pointer to the base of the array to be searched, and *key* is the value being sought.

The *compare* parameter is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. The bsearch function calls the *compare* routine one or more times during the search, passing pointers to two array elements on each call. The routine compares the elements, then returns one of the following values:

Value	Meaning
< 0	*elem1* less than *elem2*
= 0	*elem1* identical to *elem2*
> 0	*elem1* greater than *elem2*

If the array you are searching is not in ascending sort order, bsearch does not work properly. If the array contains duplicate records with identical keys, there is no way to predict which of the duplicate records will be located by bsearch.

Return Value

The bsearch function returns a pointer to an occurrence of *key* in the array pointed to by *base*. If *key* is not found, the function returns NULL.

_lfind
_lsearch
qsort

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* BSEARCH.C: This program reads the command-line
 * parameters, sorting them with qsort, and then
 * uses bsearch to find the word "cat." */
#include <search.h>
#include <string.h>
#include <stdio.h>
int compare( char **arg1, char **arg2 ); /* Declare a function for compare */
void main( int argc, char **argv )
{
   char **result;
   char *key = "cat";
   int i;
   /* Sort using Quicksort algorithm: */
   qsort( (void *)argv, (size_t)argc, sizeof( char * ), (int (*)(const void*, const
void*))compare );
   for( i = 0; i < argc; ++i )    /* Output sorted list */
      printf( "%s ", argv[i] );
   /* Find the word "cat" using a binary search algorithm: */
   result = (char **)bsearch( (char *) &key, (char *)argv, argc,
                              sizeof( char * ), (int (*)(const void*, const
void*))compare );
   if( result )
      printf( "\n%s found at %Fp\n", *result, result );
   else
      printf( "\nCat not found!\n" );
}
int compare( char **arg1, char **arg2 )
{
   /* Compare all of both strings: */
   return _strcmpi( *arg1, *arg2 );
}
```

_cabs, _cabsl

#include <math.h>

Syntax double _cabs(struct _complex *z*);

long double _cabsl(struct _complexl *z*);

Parameter	**Description**
z	Complex number

The _cabs and _cabsl functions calculate the absolute value of a complex number, which must be a structure of type _complex (or _complexl). The structure *z* is composed of a real component *x* and an imaginary component *y*. A call to one of the _cabs routines is equivalent to the following:

sqrt(*z.x***z.x* + *z.y***z.y*)

The _cabsl function is the 80-bit counterpart, and it uses the 80-bit, 10-byte coprocessor form of parameters and return values. See the long double functions for more details on this data type.

Return Value

These functions return the absolute value if successful, or HUGE_VAL on overflow (_LHUGE_VAL for _cabsl). The errno variable is set to ERANGE on overflow. Error handling can be changed with the _matherrl function.

Use _cabs for compatibility with ANSI naming conventions of non-ANSI functions. Use cabs and link with OLDNAMES.LIB for UNIX compatibility.

abs
fabs
labs

_cabs

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_cabsl

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* CABS.C: Using _cabs, this program calculates
 * the absolute value of a complex number.
 */
#include <math.h>
#include <stdio.h>
void main( void )
{
   struct _complex number = { 3.0, 4.0 };
   double d;
   d = _cabs( number );
   printf( "The absolute value of %f + %fi is %f\n",
           number.x, number.y, d );
}
```

calloc Functions

#include <stdlib.h> For ANSI compatibility (calloc only)

#include <malloc.h> Required only for function declarations

Syntax void *calloc(size_t *num*, size_t *size*);

void __based(void) *_bcalloc(__segment *seg*, size_t *num*, size_t *size*);

void __far *_fcalloc(size_t *num*, size_t *size*);

void __near *_ncalloc(size_t *num*, size_t *size*);

Parameter	Description
num	Number of elements
size	Length in bytes of each element
seg	Segment selector

The calloc family of functions allocates storage space for an array of *num* elements, each of length *size* bytes. Each element is initialized to 0.

In large data models (compact-, large-, and huge-model programs), calloc maps to _fcalloc. In small data models (tiny-, small-, and medium-model programs), calloc maps to _ncalloc.

The various calloc functions allocate storage space in the data segments shown in the list below:

Function	Data Segment
calloc	Depends on data model of program
_bcalloc	Based heap, specified by *seg* segment selector
_fcalloc	Far heap (outside default data segment)
_ncalloc	Near heap (inside default data segment)

Return Value

The calloc functions return a pointer to the allocated space. The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

The _fcalloc and _ncalloc functions return NULL if there is insufficient memory available. The _bcalloc function returns _NULLOFF in this case.

free functions
_halloc
_hfree
malloc functions
realloc functions

calloc

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_bcalloc, _fcalloc, _ncalloc

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* CALLOC.C: This program uses calloc to allocate space for
 * 40 long integers. It initializes each element to zero.
 */
#include <stdio.h>
#include <malloc.h>
void main( void )
{
   long *buffer;
   buffer = (long *)calloc( 40, sizeof( long ) );
   if( buffer != NULL )
      printf( "Allocated 40 long integers\n" );
   else
      printf( "Can't allocate memory\n" );
   free( buffer );
}
```

ceil, ceill

#include <math.h>

Syntax double ceil(double *x*)*;*

long double ceill(long double *x*)*;*

Parameter	**Description**
x	Floating-point value

The ceil and ceill functions return a double (or long double) value representing the smallest integer that is greater than or equal to *x*.

The ceill function is the 80-bit counterpart, and it uses the 80-bit, 10-byte coprocessor form of parameters and return values. See the long double functions for more details on this data type.

Return Value

These functions return the double or long double result. There is no error return.

floor
fmod

ceil

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

ceill

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FLOOR.C: This example displays the largest integers
 * less than or equal to the floating-point values 2.8
 * and -2.8. It then shows the smallest integers greater
 * than or equal to 2.8 and -2.8.
 */
#include <math.h>
#include <stdio.h>
void main( void )
{
   double y;
   y = floor( 2.8 );
   printf( "The floor of 2.8 is %f\n", y );
   y = floor( -2.8 );
   printf( "The floor of -2.8 is %f\n", y );
   y = ceil( 2.8 );
   printf( "The ceil of 2.8 is %f\n", y );
   y = ceil( -2.8 );
   printf( "The ceil of -2.8 is %f\n", y );
}
```

_cexit, _c_exit

#include <process.h>

Syntax void _cexit(void);

void _c_exit(void);

The _cexit function calls, in LIFO ("last in, first out") order, the functions registered by atexit and _onexit. Then the _cexit function flushes all I/O buffers and closes all open streams before returning.

The _c_exit function is the same as the _exit function but returns to the calling process without processing atexit or _onexit or flushing stream buffers.

The behavior of the exit, _exit, _cexit, and _c_exit functions is described in the following list:

exit
: Performs complete C library termination procedures, terminates the process, and exits with the supplied status code

_exit
: Performs "quick" C library termination procedures, terminates the process, and exits with the supplied status code

_cexit
: Performs complete C library termination procedures and returns to caller, but does not terminate the process

_c_exit
: Performs "quick" C library termination procedures and returns to caller, but does not terminate the process

Return Value

None.

abort
atexit
_exec functions
exit
_onexit
_spawn functions
system

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_cgets

#include <conio.h> Required only for function declarations

Syntax char *_cgets(char **buffer*);

Parameter	**Description**
buffer	Storage location for data

The _cgets function reads a string of characters directly from the console and stores the string and its length in the location pointed to by *buffer*. The *buffer* parameter must be a pointer to a character array. The first element of the array, *buffer*[0], must contain the maximum length (in characters) of the string to be read. The array must contain enough elements to hold the string, a terminating null character ('\0'), and two additional bytes.

The _cgets function continues to read characters until a carriage-return-line-feed (CR-LF) combination is read, or the specified number of characters is read. The string is stored starting at *str*[2]. If a CR-LF combination is read, it is replaced with a null character ('\0') before being stored. The _cgets function then stores the actual length of the string in the second array element, *buffer* [1].

Because all MS-DOS editing keys are active when you call _cgets, pressing F3 repeats the last entry.

Return Value

The _cgets function returns a pointer to the start of the string, at *buffer*[2]. There is no error return.

_getch
_getche

Standards: None

16-Bit: MS-DOS,

```
/* CGETS.C: This program creates a buffer and initializes
 * the first byte to the size of the buffer: 2. Next, the
 * program accepts an input string using _cgets and displays
 * the size and text of that string.
 */
#include <conio.h>
#include <stdio.h>
void main( void )
{
   char buffer[82] = { 80 };  /* Maximum characters in first byte */
   char *result;
   printf( "Input line of text, followed by carriage return:\n");
   result = _cgets( buffer );  /* Input a line of text */
   printf( "\nLine length = %d\nText = %s\n", buffer[1], result );
}
```

_chain_intr

#include <dos.h>

Syntax void _chain_intr(void(__cdecl __interrupt __far **target*)( ));

Parameter	**Description**
target	Target interrupt routine

The _chain_intr routine passes control from one interrupt handler to another. The stack and the registers of the first routine are passed to the second, allowing the second routine to return as if it had been called directly.

The _chain_intr routine is generally used when a user-defined interrupt handler begins processing, then chains to the original interrupt handler to finish processing.

Chaining is one of two techniques, listed below, that can be used to transfer control from a new interrupt routine to an old one:

- Call _chain_intr with the interrupt routine as an parameter. Do this if your routine is finished and you want the second interrupt routine to terminate the interrupt call.

```
void __far __interrupt new_int(unsigned _es, unsigned _ds, unsigned _di, unsigned
_si,... )
{
   ++_di;                  /* Initial processing here */
   _chain_intr( old_int ); /* New DI passed to old_int */
   --_di;                  /* This is never executed */
}
```

- Call the interrupt routine (after casting it to an interrupt function if necessary). Do this if you need to do further processing after the second interrupt routine finishes.

```
void __far __interrupt new_int(unsigned _es, unsigned _ds, unsigned _di, unsigned
_si,... )
{
 ++_di;              /* Initial processing here */
   (*old_int)();     /* New DI passed to old_int */
   __asm mov _di, di /* Put real DI from old_int */
                     /* into _di for return */
}
```

Note that the real registers set by the old interrupt function are not automatically set to the pseudoregisters of the new routine.

Use the _chain_intr function when you do not want to replace the default interrupt handler, but you do need to see its input. An example is a TSR (terminate-and-stay-resident) program that checks all keyboard input for a particular "hot key" sequence.

The _chain_intr function should be used only with C functions that have been declared with __interrupt. The __interrupt declaration ensures that the procedure's entry/exit sequence is appropriate for an interrupt handler.

Return Value

The _chain_intr function does not return to the caller.

_dos_getvect
_dos_keep
_dos_setvect

Standards: None

16-Bit: MS-DOS

_chdir

#include <direct.h> Required only for function declarations

#include <errno.h> Required for errno constants

Syntax int _chdir(const char **dirname*);

Parameter	Description
dirname	Path of new working directory

The _chdir function changes the current working directory to the directory specified by *dirname*. The *dirname* parameter must refer to an existing directory.

This function can change the current working directory on any drive; it cannot be used to change the default drive itself. For example, if A: is the default drive and \BIN is the current working directory, the following call changes the current working directory for drive C:

```
_chdir("c:\\temp");
```

Notice that you must place two backslashes (\\) in a C string in order to represent a single backslash (\); the backslash is the escape character for C strings and therefore requires special handling.

This function call has no apparent immediate effect. However, when the _chdrive function is called to change the default drive to C:, the current working directory becomes C:\TEMP.

With MS-DOS, the new directory set by the program becomes the new current working directory.

Return Value

The _chdir function returns a value of 0 if the working directory is successfully changed. A return value of -1 indicates an error, in which case errno is set to ENOENT, indicating that the specified path could not be found.

Use _chdir for compatibility with ANSI naming conventions of non-ANSI functions. Use chdir and link with OLDNAMES.LIB for UNIX compatibility.

_dos_setdrive
_mkdir
_rmdir
system

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* CHGDIR.C: This program uses the _chdir function to verify
 * that a given directory exists.
 */
#include <direct.h>
#include <stdio.h>
#include <stdlib.h>
void main( int argc, char *argv[] )
{
   if( _chdir( argv[1] )   )
      printf( "Unable to locate the directory: %s\n", argv[1] );
   else
      system( "dir *.c" );
}
```

_chdrive

#include <direct.h> Required only for function declarations

Syntax int _chdrive(int *drive*);

Parameter	Description
drive	Number of new working drive

The _chdrive function changes the current working drive to the drive specified by *drive*. The *drive* parameter uses an integer to specify the new working drive (1=A, 2=B, etc.).

This function changes only the working drive; the _chdir function changes the working directory.

With MS-DOS, the new drive set by the program becomes the new working drive.

Return Value

The _chdrive function returns a value of 0 if the working drive is successfully changed. A return value of -1 indicates an error.

_chdir
_dos_setdrive
_fullpath
_getcwd
_getdrive
_mkdir
_rmdir
system

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* GETDRIVE.C illustrates drive functions including:
 *      _getdrive       _chdrive        _getdcwd
 */
#include <stdio.h>
#include <conio.h>
#include <direct.h>
#include <stdlib.h>
#include <ctype.h>
void main( void )
{
   int ch, drive, curdrive;
   static char path[_MAX_PATH];
   /* Save current drive. */
   curdrive = _getdrive();
   printf( "Available drives are: \n" );
   /* If we can switch to the drive, it exists. */
   for( drive = 1; drive <= 26; drive++ )
      if( !_chdrive( drive ) )
         printf( "%c: ", drive + 'A' - 1 );
   while( 1 )
   {
      printf( "\nType drive letter to check or ESC to quit: " );
      ch = _getch();
      if( ch == 27 )
         break;
      if( isalpha( ch ) )
         _putch( ch );
      if( _getdcwd( toupper( ch ) - 'A' + 1, path, _MAX_PATH ) != NULL )
         printf( "\nCurrent directory on that drive is %s\n", path );
   }
   /* Restore original drive.*/
   _chdrive( curdrive );
   printf( "\n" );
}
```

_chmod

#include <sys\types.h>, <sys\stat.h>, <errno.h>

#include <io.h> Required only for function declarations

Syntax int _chmod(const char **filename,* int *pmode*);

Parameter	**Description**
filename	Name of existing file
pmode	Permission setting for file

The _chmod function changes the permission setting of the file specified by *filename.* The permission setting controls read and write access to the file. The constant expression *pmode* contains one or both of the manifest constants _S_IWRITE and _S_IREAD, defined in SYS\STAT.H. Any other values for *pmode* are ignored. When both constants are given, they are joined with the bitwise-OR operator (|). The meaning of the *pmode* parameter is as follows:

Value	**Meaning**
_S_IWRITE	Writing permitted
_S_IREAD	Reading permitted
_S_IREAD | _S_IWRITE	Reading and writing permitted

If write permission is not given, the file is read-only. Note that all files are always readable; it is not possible to give write-only permission. Thus the modes _S_IWRITE and _S_IREAD | _S_IWRITE are equivalent.

Return Value

The _chmod function returns the value 0 if the permission setting is successfully changed. A return value of -1 indicates an error; in this case, errno is set to ENOENT, indicating that the specified file could not be found.

Use _chmod for compatibility with ANSI naming conventions of non-ANSI functions. Use chmod and link with OLDNAMES.LIB for UNIX compatibility.

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_access
_creat
_fstat
_open
_stat

```
/* CHMOD.C: This program uses _chmod to
 * change the mode of a file to read-only.
 * It then attempts to modify the file.
 */
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
void main( void )
{
   /* Make file read-only: */
   if( _chmod( "CHMOD.C", _S_IREAD ) == -1 )
      perror( "File not found\n" );
   else
      printf( "Mode changed to read-only\n" );
   system( "echo /* End of file */ >> CHMOD.C" );
   /* Change back to read/write: */
   if( _chmod( "CHMOD.C", _S_IWRITE ) == -1 )
      perror( "File not found\n" );
   else
      printf( "Mode changed to read/write\n" );
}
```

_chsize

#include <io.h> Required only for function declarations

#include <errno.h>

Syntax int _chsize(int *handle*, long *size*);

Parameter	**Description**
handle	Handle referring to open file
size	New length of file in bytes

The _chsize function extends or truncates the file associated with *handle* to the length specified by *size*. The file must be open in a mode that permits writing. Null characters ('\0') are appended if the file is extended. If the file is truncated, all data from the end of the shortened file to the original length of the file is lost.

In MS-DOS and Windows, the directory update is done when a file is closed. Consequently, while a program is running, requests to determine the amount of free disk space may receive inaccurate results.

Return Value

The _chsize function returns the value 0 if the file size is successfully changed. A return value of -1 indicates an error, and errno is set to one of the following values:

Value	**Meaning**
EACCES	Specified file is locked against access.
EBADF	Specified file is read-only or an invalid file handle.
ENOSPC	No space is left on device.

Use _chsize for compatibility with ANSI naming conventions of non-ANSI functions. Use chsize and link with OLDNAMES.LIB for UNIX compatibility.

_close
_creat
_open

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* CHSIZE.C: This program uses _filelength to report the size
 * of a file before and after modifying it with _chsize.
 */
#include <io.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <stdio.h>
void main( void )
{
   int fh, result;
   unsigned int nbytes = BUFSIZ;
   /* Open a file */
   if( (fh = _open( "data", _O_RDWR | _O_CREAT, _S_IREAD | _S_IWRITE ))  != -1 )
   {
      printf( "File length before: %ld\n", _filelength( fh ) );
      if( ( result = _chsize( fh, 329678 ) ) == 0 )
         printf( "Size successfully changed\n" );
      else
         printf( "Problem in changing the size\n" );
      printf( "File length after:  %ld\n", _filelength( fh ) );
      _close( fh );
   }
}
```

_clear87

#include <float.h>

Syntax unsigned int _clear87(void);

The _clear87 function gets and clears the floating-point status word. The floating-point status word is a combination of the 8087/80287 status word and other conditions detected by the 8087/80287 exception handler, such as floating-point stack overflow and underflow.

Return Value

The bits in the value returned indicate the floating-point status. See the FLOAT.H include file for a complete definition of the bits returned by _clear87.

Many of the math library functions modify the 8087/80287 status word, with unpredictable results. Return values from _clear87 and _status87 become more reliable as fewer floating-point operations are performed between known states of the floating-point status word.

_control87
_status87

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* This program places the name of the current directory in
 * the buffer array, then displays the name of the current
 * directory on the screen. Specifying a length of _MAX_DIR
 * leaves room for the longest legal directory name.
 */
#include <direct.h>
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   char buffer[_MAX_DIR];
   /* Get the current working directory: */
   if( _getcwd( buffer, _MAX_DIR ) == NULL )
      perror( "_getcwd error" );
   else
      printf( "%s\n", buffer );
}
```

clearerr

#include <stdio.h>

Syntax void clearerr(FILE **stream*);

Parameter	**Description**
stream	Pointer to FILE structure

The clearerr function resets the error indicator and end-of-file indicator for *stream*. Error indicators are not automatically cleared; once the error indicator for a specified stream is set, operations on that stream continue to return an error value until clearerr, fseek, fsetpos, or rewind is called.

Return Value

None.

_eof
feof
ferror
perror

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* CLEARERR.C: This program creates an error
 * on the standard input stream, then clears
 * it so that future reads won't fail.
 */
#include <stdio.h>
void main( void )
{
   int c;
   /* Create an error by writing to standard input. */
   putc( 'c', stdin );
   if( ferror( stdin ) )
   {
      perror( "Write error" );
      clearerr( stdin );
   }
   /* See if read causes an error. */
   printf( "Will input cause an error? " );
   c = getc( stdin );
   if( ferror( stdin ) )
   {
      perror( "Read error" );
      clearerr( stdin );
   }
}
```

_clearscreen

#include <graph.h>

Syntax void __far _clearscreen(short *area*);

Parameter	**Description**
area	Target area

The _clearscreen function erases the target area, filling it with the current background color. The *area* parameter can be one of the following manifest constants (defined in GRAPH.H):

Constant	**Action**
_GCLEARSCREEN	Clears and fills the entire screen
_GVIEWPORT	Clears and fills only within the current view port
_GWINDOW	Clears and fills only within the current text window

Return Value

None.

_getbkcolor
_setbkcolor

Standards: None
16-Bit: MS-DOS, QWIN

```
/* CLRSCRN.C */
#include <conio.h>
#include <graph.h>
#include <stdlib.h>
void main( void )
{
   short xhalf, yhalf, xquar, yquar;
   struct _videoconfig vc;
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   _getvideoconfig( &vc );
   xhalf = vc.numxpixels / (short)2;
   yhalf = vc.numypixels / (short)2;
   xquar = xhalf / (short)2;
   yquar = yhalf / (short)2;
   _setviewport( 0, 0, xhalf - (short)1, yhalf - (short)1 );
   _rectangle( _GBORDER, 0,  0, xhalf - (short)1, yhalf - (short)1 );
   _ellipse( _GFILLINTERIOR, xquar/(short)4, yquar/(short)4,
      xhalf - (xquar/(short)4), yhalf - (yquar/(short)4) );
   _getch();
   _clearscreen( _GVIEWPORT );
   _getch();
   _setvideomode( _DEFAULTMODE );
}
```

clock

#include <time.h>

Syntax clock_t clock(void);

The clock function tells how much processor time has been used by the calling process. The time in seconds is approximated by dividing the clock return value by the value of the CLOCKS_PER_SEC constant.

In other words, the clock function returns the number of processor timer ticks that have elapsed. A timer tick is approximately equal to 1/CLOCKS_PER_SEC seconds.

In versions of Microsoft C prior to version 6.0, the CLOCKS_PER_SEC constant was called CLK_TCK.

Return Value

The clock function returns the product of the time in seconds and the value of the CLOCKS_PER_SEC constant. If the processor time is not available, the function returns the value -1, cast as clock_t.

In MS-DOS, clock returns the time elapsed since the process started. This may not be equal to the actual processor time used by the process.

difftime
time

Standards: ANSI

16-Bit: MS-DOS, QWIN, WIN

```
/* CLOCK.C: This example prompts for how long
 * the program is to run and then continuously
 * displays the elapsed time for that period.
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void sleep( clock_t wait );
void main( void )
{
   long    i = 600000L;
   clock_t start, finish;
   double  duration;
   /* Delay for a specified time. */
   printf( "Delay for three seconds\n" );
   sleep( (clock_t)3 * CLOCKS_PER_SEC );
   printf( "Done!\n" );
   /* Measure the duration of an event. */
   printf( "Time to do %ld empty loops is ", i );
   start = clock();
   while( i-- ) ;
   finish = clock();
   duration = (double)(finish - start) / CLOCKS_PER_SEC;
   printf( "%2.1f seconds\n", duration );
}
/* Pauses for a specified number of microseconds. */
void sleep( clock_t wait )
{
   clock_t goal;
   goal = wait + clock();
   while( goal > clock() )
      ;
}
```

_close

#include <io.h> Required only for function declarations

#include <errno.h>

Syntax int _close(int *handle*);

Parameter	**Description**
handle	Handle referring to open file

The _close function closes the file associated with *handle*.

Return Value

The _close function returns 0 if the file was successfully closed. A return value of -1 indicates an error, and errno is set to EBADF, indicating an invalid file-handle parameter.

Use _close for compatibility with ANSI naming conventions of non-ANSI functions. Use close and link with OLDNAMES.LIB for UNIX compatibility.

_chsize
_creat
_dup
_dup2
_open
_unlink

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* OPEN.C: This program uses _open to open a file
 * named OPEN.C for input and a file named OPEN.OUT
 * for output. The files are then closed.
 */
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>
void main( void )
{
   int fh1, fh2;
   fh1 = _open( "OPEN.C", _O_RDONLY );
   if( fh1 == -1 )
      perror( "open failed on input file" );
   else
   {
      printf( "open succeeded on input file\n" );
      _close( fh1 );
   }
   fh2 = _open( "OPEN.OUT", _O_WRONLY | _O_CREAT, _S_IREAD | _S_IWRITE );
   if( fh2 == -1 )
      perror( "open failed on output file" );
   else
   {
      printf( "open succeeded on output file\n" );
      _close( fh2 );
   }
}
```

_commit

#include <io.h> Required only for function declarations

#include <errno.h>

Syntax int _commit(int *handle*);

Parameter	**Description**
handle	Handle referring to open file

The _commit function forces the operating system to write the file associated with *handle* to disk. This call ensures that the specified file is flushed immediately, not at the operating system's discretion.

Return Value

The _commit function returns 0 if the file was successfully flushed to disk. A return value of -1 indicates an error, and errno is set to EBADF, indicating an invalid file-handle parameter.

_creat
_open
_read
_write

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* COMMIT.C illustrates low-level file I/O functions including:
 *      _close    _commit    memset    _open    _write
 * This is example code; to keep the code simple and readable,
 * return values are not checked.
 */
#include <io.h>
#include <stdio.h>
#include <fcntl.h>
#include <memory.h>

#define MAXBUF 32
int log_receivable( int );
void main( void )
{
   int fhandle;
   fhandle = _open( "TRANSACT.LOG", _O_APPEND | _O_CREAT | _O_BINARY | _O_RDWR );
   log_receivable( fhandle );
   _close( fhandle );
}

 /* The log_receivable function prompts for a name and a monetary
 * amount and places both values into a buffer (buf). The _write
 * function writes the values to the operating system, and the
 * _commit function ensures that they are written to a disk file.
 */
int log_receivable( int fhandle )
{
   int i; char  buf[MAXBUF];
   memset( buf, '\0', MAXBUF );
   /* Begin Transaction. */
   printf( "Enter name: " );
   gets( buf );
   for( i = 1; buf[i] != '\0'; i++ );
   /* Write the value as a '\0' terminated string. */
   _write( fhandle, buf, i+1 );
   printf( "\n" );
   memset( buf, '\0', MAXBUF );
   printf( "Enter amount: $" );
   gets( buf ); for( i = 1; buf[i] != '\0'; i++ );
   /* Write the value as a '\0' terminated string. */
   _write( fhandle, buf, i+1 );
   printf( "\n" );
   /* The _commit function ensures that two important pieces of data are
    * safely written to disk. The return value of the _commit function
    * is returned to the calling function.
    */
   return _commit( fhandle );
}
```

_control87

#include <float.h>

Syntax unsigned int _control87(unsigned int *new*, unsigned int *mask*);

Parameter	Description
new	New control-word bit values
mask	Mask for new control-word bits to set

The _control87 function gets and sets the floating-point control word. The floating-point control word allows the program to change the precision, rounding, and infinity modes in the floating-point-math package. Floating-point exceptions can also be masked or unmasked using the _control87 function.

If the value for *mask* is equal to 0, then _control87 gets the floating-point control word. If *mask* is nonzero, then a new value for the control word is set in the following manner: for any bit that is on (equal to 1) in *mask*, the corresponding bit in *new* is used to update the control word. To put it another way,

```
fpcntrl = ((fpcntrl & ~mask) | (new & mask))
```

where `fpcntrl` is the floating-point control word.

The possible values for the mask constant (*mask*) and new control values (*new*) are shown in the following table.

Mask	Hex Value	Constant	Hex Value
MCW_EM (Interrupt exception)	0x003F		
		_EM_INVALID	0x0001
		_EM_DENORMAL	0x0002
		_EM_ZERODIVIDE	0x0004
		_EM_OVERFLOW	0x0008
		_EM_UNDERFLOW	0x0010
		_EM_INEXACT	0x0020
MCW_IC (Infinity control)	0x1000		
		_IC_AFFINE	0x1000
		_IC_PROJECTIVE	0x0000
MCW_RC (Rounding control)	0x0C00		
		RC_CHOP	0x0C00
		RC_UP	0x0800
		_RC_DOWN	0x0400
		_RC_NEAR	0x0000

MCW_PC (Precision control)	0x0300		
		_PC_24 (24 bits)	0x0000
		_PC_53 (53 bits)	0x0200
		_PC_64 (64 bits)	0x0300

Return Value

The bits in the value returned indicate the floating-point control state. See the FLOAT.H include file for a complete definition of the bits returned by _control87.

_clear87
_status87

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* This program places the name of the current directory in
 * the buffer array, then displays the name of the current
 * directory on the screen. Specifying a length of _MAX_DIR
 * leaves room for the longest legal directory name.
 */
#include <direct.h>
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   char buffer[_MAX_DIR];
   /* Get the current working directory: */
   if( _getcwd( buffer, _MAX_DIR ) == NULL )
      perror( "_getcwd error" );
   else
      printf( "%s\n", buffer );
}
```

cos Functions

#include <math.h>

Syntax double cos(double *x*);

double cosh(double *x*);

long double cosl(long double *x*);

long double coshl(long double *x*);

Parameter	Description
x	Angle in radians

The cos and cosh functions return the cosine and hyperbolic cosine, respectively, of *x*.

The cosl and coshl functions are the 80-bit counterparts and use the 80-bit, 10-byte coprocessor form of parameters and return values. See the reference page on the long double functions for more details on this data type.

Return Value

If *x* is greater than or equal to 2 raised to the power of 27, a partial loss of significance in the result may occur in a call to cos, in which case the function generates a _PLOSS error. If *x* is greater than or equal to 2 raised to the power of 31, significance is completely lost, and cos prints a _TLOSS message to stderr and returns 0. In both cases, errno is set to ERANGE.

If the result is too large in a cosh call, the functions return HUGE_VAL (_LHUGE_VAL for coshl) and set errno to ERANGE. This behavior can be changed with _matherr.

cos, cosh

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

cosl, coshl

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* SINCOS.C: This program displays the sine, hyperbolic
 * sine, cosine, and hyperbolic cosine of pi / 2.
 */
#include <math.h>
#include <stdio.h>
void main( void )
{
   double pi = 3.1415926535;
   double x, y;
   x = pi / 2;
   y = sin( x );
   printf( "sin( %f ) = %f\n", x, y );
   y = sinh( x );
   printf( "sinh( %f ) = %f\n",x, y );
   y = cos( x );
   printf( "cos( %f ) = %f\n", x, y );
   y = cosh( x );
   printf( "cosh( %f ) = %f\n",x, y );
}
```

_cprintf

#include <conio.h> Required only for function declarations

Syntax int _cprintf(const char *_format_ [, _argument_] **...**);

Parameter	**Description**
format	Format control string
argument	Optional parameters

The _cprintf function formats and prints a series of characters and values directly to the console, using the _putch function to output characters. Each *argument* (if any) is converted and output according to the corresponding format specification in *format*. The format has the same form and function as the *format* parameter for the printf function; see printf Format Specification Fields for a description of the format and parameters.

Note that unlike the fprintf, printf, and sprintf functions, _cprintf does not translate line-feed characters into carriage-return-line-feed (CR-LF) combinations on output.

Return Value

The _cprintf function returns the number of characters printed.

_cscanf
fprintf
printf
sprintf
vfprintf

Standards: None

16-Bit: MS-DOS

```
/* CPRINTF.C: This program displays
 * some variables to the console.
 */
#include <conio.h>
void main( void )
{
   int      i = -16, h = 29;
   unsigned u = 62511;
   char     c = 'A';
   char     s[] = "Test";
   /* Note that console output does not translate \n as
    * standard output does. Use \r\n instead.
    */
   _cprintf( "%d  %.4x  %u  %c %s\r\n", i, h, u, c, s );
}
```

_cputs

#include <conio.h> Required only for function declarations

Syntax int _cputs(const char **string*);

Parameter	**Description**
string	Output string

The _cputs function writes the null-terminated string pointed to by *string* directly to the console. Note that a carriage-return-line-feed (CR-LF) combination is not automatically appended to the string.

Return Value

If successful, _cputs returns a 0. If the function fails, it returns a nonzero value.

putch

Standards: None

16-Bit: MS-DOS

```
/* CPUTS.C: This program first displays
 * a string to the console.
 */
#include <conio.h>
void main( void )
{
   /* String to print at console. Note the \r (return) character. */
   char *buffer = "Hello world (courtesy of _cputs)!\r\n";
   _cputs( buffer );
}
```

_creat

#include <sys\types.h>, <sys\stat.h>, <errno.h>

#include <io.h> Required only for function declarations

Syntax int _creat(const char **filename*, int *pmode*);

Parameter	**Description**
filename	Name of new file
pmode	Permission setting

The _creat function either creates a new file or opens and truncates an existing file. If the file specified by *filename* does not exist, a new file is created with the given permission setting and is opened for writing. If the file already exists and its permission setting allows writing, _creat truncates the file to length 0, destroying the previous contents, and opens it for writing.

The permission setting, *pmode*, applies to newly created files only. The new file receives the specified permission setting after it is closed for the first time. The integer expression *pmode* contains one or both of the manifest constants _S_IWRITE and _S_IREAD, defined in SYS\STAT.H. When both of the constants are given, they are joined with the bitwise-OR operator (|). The *pmode* parameter is set to one of the following values:

Value	**Meaning**
_S_IWRITE	Writing permitted
_S_IREAD	Reading permitted
_S_IREAD | _S_IWRITE	Reading and writing permitted

If write permission is not given, the file is read-only. Note that all files are always readable; it is not possible to give write-only permission. Thus, the modes _S_IWRITE and _S_IREAD | _S_IWRITE are equivalent. With MS-DOS versions 3.0 and later, files opened using _creat are always opened in compatibility mode (see _sopen).

The _creat function applies the current file-permission mask to *pmode* before setting the permissions (see _umask).

Note that the _creat routine is provided primarily for compatibility with previous libraries. A call to _open with _O_CREAT and _O_TRUNC in the *oflag* parameter is equivalent to _creat and is preferable for new code.

Return Value

If successful, _creat returns a handle for the created file. Otherwise, it returns -1 and sets errno to one of the following constants:

Value	**Meaning**
EACCES	The filename specifies an existing read-only file or specifies a directory instead of a file
EMFILE	No more handles available (too many open files)
ENOENT	File not found

Use _creat for compatibility with ANSI naming conventions of non-ANSI functions. Use creat and link with OLDNAMES.LIB for UNIX compatibility.

_chmod
_chsize
_close
_dup
_dup2
_open
_sopen
_umask

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* CREAT.C: This program uses _creat to create
 * the file (or truncate the existing file)
 * named data and open it for writing.
 */
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
void main( void )
{
   int fh;
   fh = _creat( "data", _S_IREAD | _S_IWRITE );
   if( fh == -1 )
      perror( "Couldn't create data file" );
   else
   {
      printf( "Created data file.\n" );
      _close( fh );
   }
}
```

_cscanf

#include <conio.h> Required only for function declarations

Syntax int _cscanf(const char *_format_ [, _argument_] **...**);

Parameter	**Description**
format	Format-control string
argument	Optional parameters

The _cscanf function reads data directly from the console into the locations given by *argument*. The _getche function is used to read characters. Each optional parameter must be a pointer to a variable with a type that corresponds to a type specifier in *format*. The format controls the interpretation of the input fields and has the same form and function as the *format* parameter for the scanf function; see scanf for a description of *format*.

While _cscanf normally echoes the input character, it will not do so if the last call was to _ungetch.

Return Value

The _cscanf function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF for an attempt to read at end-of-file. This may occur when keyboard input is redirected at the operating system command-line level. A return value of 0 means that no fields were assigned.

_cprintf
fscanf
scanf
sscanf

Standards: None

16-Bit: MS-DOS

```
/* CSCANF.C: This program prompts for a string
 * and uses _cscanf to read in the response.
 * Then _cscanf returns the number of items
 * matched, and the program displays that number.
 */
#include <stdio.h>
#include <conio.h>
void main( void )
{
   int   result, i[3];
   _cprintf( "Enter three integers: ");
   result = _cscanf( "%i %i %i", &i[0], &i[1], &i[2] );
   _cprintf( "\r\nYou entered " );
   while( result-- )
      _cprintf( "%i ", i[result] );
   _cprintf( "\r\n" );
}
```

ctime

#include <time.h> Required only for function declarations

Syntax char *ctime(const time_t **timer*);

Parameter	**Description**
timer	Pointer to stored time

The ctime function converts a time stored as a time_t value to a character string. The *timer* value is usually obtained from a call to time, which returns the number of seconds elapsed since midnight (00:00:00), January 1, 1970, Universal Coordinated Time.

The string result produced by ctime contains exactly 26 characters and has the form of the following example:

```
Wed Jan 02 02:03:55 1980\n\0
```

A 24-hour clock is used. All fields have a constant width. The newline character ('\n') and the null character ('\0') occupy the last two positions of the string.

Calls to the ctime function modify the single statically allocated buffer used by the gmtime and the localtime functions. Each call to one of these routines destroys the result of the previous call. The ctime function also shares a static buffer with the asctime function. Thus, a call to ctime destroys the results of any previous call to asctime, localtime, or gmtime.

Return Value

The ctime function returns a pointer to the character string result. If *time* represents a date before midnight, January 1, 1970, Universal Coordinated Time, ctime returns NULL.

asctime
_ftime
gmtime
localtime
time

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* CTIME.C: This program gets the current
 * time in time_t form, then uses ctime to
 * display the time in string form.
 */
#include <time.h>
#include <stdio.h>
void main( void )
{
   time_t ltime;
   time( &ltime );
   printf( "The time is %s\n", ctime( &ltime ) );
}
```

_dieeetomsbin, _dmsbintoieee

#include <math.h>

Syntax int _dieeetomsbin(double **src8*, double **dst8*);

int _dmsbintoieee(double **src8*, double ***dst8*);

Parameter	Description
src8	Buffer containing value to convert
dst8	Buffer to store converted value

The _dieeetomsbin routine converts a double-precision number in IEEE (Institute of Electrical and Electronic Engineers) format to Microsoft (MS) binary format. The routine _dmsbintoieee converts a double-precision number in MS binary format to IEEE format.

These routines allow C programs (which store floating-point numbers in the IEEE format) to use numeric data in random-access data files created with those versions of Microsoft Basic that store floating-point numbers in MS binary format, and vice versa.

The parameter *src8* is a pointer to the double value to be converted. The result is stored at the location given by *dst8*.

These routines do not handle IEEE NANs ("not a number") and infinities. IEEE denormals are treated as 0 in the conversions.

Return Value

These functions return 0 if the conversion is successful and 1 if the conversion causes an overflow.

_fieeetomsbin
_fmsbintoieee

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

difftime

#include <time.h> Required only for function declarations

Syntax double difftime(time_t *timer1*, time_t *timer0*);

Parameter	**Description**
timer0	Beginning time
timer1	Ending time

The difftime function computes the difference between the supplied time values, *timer0* and *timer1*.

Return Value

The difftime function returns, in seconds, the elapsed time from *timer0* to *timer1*. The value returned is a double-precision number.

time

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* DIFFTIME.C: This program calculates the amount of time
 * needed to do a floating-point multiply 50000 times.
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void main( void )
{
   time_t   start, finish;
   unsigned loop;
   double   result, elapsed_time;
   printf( "This program will do a floating point multiply 50000 times\n" );
   printf( "Working...\n" );
   time( &start );
   for( loop = 0; loop < 50000L; loop++ )
      result = 3.63 * 5.27; time( &finish );
   elapsed_time = difftime( finish, start );
   printf( "\nProgram takes %6.0f seconds.\n", elapsed_time );
}
```

_disable

#include <dos.h>

Syntax void _disable(void);

The _disable routine disables interrupts by executing an 8086 CLI machine instruction. Use _disable before modifying an interrupt vector.

Return Value

None.

_enable

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_displaycursor

#include <graph.h>

Syntax short __far _displaycursor(short *flag*);

Parameter	**Description**
flag	Cursor state

Upon entry into each graphic routine, the screen cursor is turned off. The _displaycursor function determines whether the cursor will be turned back on when programs exit graphic routines. If *flag* is set to _GCURSORON, the cursor will be restored on exit. If *flag* is set to _GCURSOROFF, the cursor will be left off.

Return Value

The function returns the previous value of *flag*. There is no error return.

_gettextcursor
_settextcursor

Standards: None

16-Bit: MS-DOS, WIN, WIN DLL

```
/* DISCURS.C: This program changes the cursor
 * shape using _gettextcursor and _settextcursor,
 * and hides the cursor using _displaycursor.
 */
#include <conio.h>
#include <graph.h>
void main( void )
{
   short oldcursor;
   short newcursor = 0x007;        /* Full block cursor */
   /* Save old cursor shape and make sure cursor is on */
   oldcursor = _gettextcursor();
   _clearscreen( _GCLEARSCREEN );
   _displaycursor( _GCURSORON );
   _outtext( "\nOld cursor shape: " );
   _getch();
   /* Change cursor shape */
   _outtext( "\nNew cursor shape: " );
   _settextcursor( newcursor );
   _getch();
   /* Restore original cursor shape */
   _outtext( "\n" );
   _settextcursor( oldcursor );
}
```

div

#include <stdlib.h>

Syntax div_t div(int *numer*, int *denom*);

Parameter	Description
numer	Numerator
denom	Denominator

The div function divides *numer* by *denom*, computing the quotient and the remainder. The div_t structure contains the following elements:

Element	Description
int quot	Quotient
int rem	Remainder

The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer that is less than the absolute value of the mathematical quotient. If the denominator is 0, the program will terminate with an error message.

Return Value

The div function returns a structure of type div_t, comprising both the quotient and the remainder. The structure is defined in STDLIB.H.

ldiv

Standards: ANSI

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* DIV.C: This example takes two integers as command-line
 * arguments and displays the results of the integer
 * division. This program accepts two arguments on the
 * command line following the program name, then calls
 * div to divide the first argument by the second.
 * Finally, it prints the structure members quot and rem.
 */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
void main( int argc, char *argv[] )
{
   int x,y;
   div_t div_result;
   x = atoi( argv[1] );
   y = atoi( argv[2] );
   printf( "x is %d, y is %d\n", x, y );
   div_result = div( x, y );
   printf( "The quotient is %d, and the remainder is %d\n",
           div_result.quot, div_result.rem );
}
```

_dos_allocmem

#include <dos.h>, <errno.h>

Syntax unsigned _dos_allocmem(unsigned *size*, unsigned **seg*);

Parameter	Description
size	Block size to allocate
seg	Return buffer for segment descriptor

The _dos_allocmem function uses MS-DOS service 0x48 to allocate a block of memory *size* paragraphs long. (A paragraph is 16 bytes.) Allocated blocks are always paragraph aligned. The segment descriptor for the initial segment of the new block is returned in the word that *seg* points to. If the request cannot be satisfied, the maximum possible size (in paragraphs) is returned in this word instead.

Return Value

If successful, the _dos_allocmem function returns 0. Otherwise, it returns the MS-DOS error code and sets errno to ENOMEM, indicating insufficient memory or invalid arena (memory area) headers.

_alloca
calloc functions
_dos_freemem
_dos_setblock
_halloc
malloc functions

Standards: None

16-Bit: MS-DOS

```
/* DALOCMEM.C: This program allocates 20 paragraphs
 * of memory, increases the allocation to 40
 * paragraphs, and then frees the memory space.
 */
#include <dos.h>
#include <stdio.h>
void main( void )
{
   unsigned segment;
   unsigned maxsize;
   /* Allocate 20 paragraphs */
   if( _dos_allocmem( 20, &segment ) != 0 )
      printf( "allocation failed\n" );
   else
      printf( "allocation successful\n" );
   /* Increase allocation to 40 paragraphs */
   if( _dos_setblock( 40, segment, &maxsize ) != 0 )
      printf( "allocation increase failed\n" );
   else
      printf( "allocation increase successful\n" );
   /* free memory */
   if( _dos_freemem( segment ) != 0 )
      printf( "free memory failed\n" );
   else
      printf( "free memory successful\n" );
}
```

_dos_close

#include <dos.h>, <errno.h>

Syntax unsigned _dos_close(int *handle*);

Parameter	**Description**
handle	Target file handle

The _dos_close function uses system call 0x3E to close the file indicated by *handle*. The file's *handle* parameter is returned by the call that created or last opened the file.

Return Value

The function returns 0 if successful. Otherwise, it returns the MS-DOS error code and sets errno to EBADF, indicating an invalid file handle.

Do not use the MS-DOS interface I/O routines with the console, low-level, or stream I/O routines.

_close
_creat
_dos_close
_dos_creat functions
_dos_open
_dos_read
_dos_write
_dup
_open

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_dos_commit

#include <dos.h>, <errno.h>

Syntax unsigned _dos_commit(int *handle*);

Parameter	**Description**
handle	Target file handle

The _dos_commit function uses system call 0x68 to flush to disk the MS-DOS buffers associated with the file indicated by *handle*. It also forces an update on the corresponding disk directory and the file allocation table. System call 0x68 ensures that the specified file is flushed directly to disk and not flushed at the operating system's discretion.

The system call used to implement _dos_commit is only available in MS-DOS versions 3.3 and later. Using _dos_commit in earlier versions of MS-DOS results in undefined behavior.

Do not use the MS-DOS interface I/O routines with the console, low-level, or stream I/O routines.

Return Value

The function returns 0 if successful. Otherwise, it returns the MS-DOS error code and sets errno to EBADF, indicating an invalid file handle.

_close
_creat
_dos_creat functions
_dos_open
_dos_read
_dos_write
_dup
_open

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* DCOMMIT.C illustrates MS-DOS file I/O functions including:
 *  _dos_commit  dos_creatnew  _dos_write  _dos_creat  _dos_close
 */
#include <dos.h>
#include <errno.h>
#include <conio.h>
void main( void )
{
   char saveit[] = "Straight to disk. ",
        prompt[] = "File exists, overwrite? [y|n] ",
        err[] = "Error occured. ",
        newline[] = "\n\r";
   int hfile, ch;
   unsigned count;
   /* Open file and create, overwriting if necessary. */
   if( _dos_creatnew( "COMMIT.LOG", _A_NORMAL, &hfile ) != 0 )
   {
      if( errno == EEXIST )
      {
         /* Use _dos_write to display prompts. Use bdos to call
          * function 1 to get and echo keystroke.
          */
         _dos_write( 1, prompt, sizeof( prompt ) - 1, &count );
         ch = bdos( 1, 0, 0 ) & 0x00ff;
         if( (ch == 'y') || (ch == 'Y') )
            _dos_creat( "COMMIT.LOG", _A_NORMAL, &hfile );
         _dos_write( 1, newline, sizeof( newline ) - 1, &count );
      }
   }
   /* Write to file; output passes through operating system's buffers. */
   if( _dos_write( hfile, saveit, sizeof( saveit ), &count ) != 0 )
   {
      _dos_write( 1, err, sizeof( err ) - 1, &count );
      _dos_write( 1, newline, sizeof( newline ) - 1, &count );
   }
   /* Write directly to file with no intermediate buffering */
   if( _dos_commit( hfile ) != 0 )
   {
      _dos_write( 1, err, sizeof( err ) - 1, &count );
      _dos_write( 1, newline, sizeof( newline ) - 1, &count );
   }
   /* Close file. */
   if( _dos_close( hfile ) != 0 )
   {
      _dos_write( 1, err, sizeof( err ) - 1, &count );
      _dos_write( 1, newline, sizeof( newline ) - 1, &count );
   }
}
```

_dos_creat Functions

#include <dos.h>, <errno.h>

Syntax unsigned _dos_creat(const char *_filename_, unsigned _attrib_, int *_handle_);

unsigned _dos_creatnew(const char *_filename_, unsigned _attrib_, int *_handle_);

Parameter	Description
filename	Name of new file
attrib	File attributes
handle	Handle return buffer

The _dos_creat and _dos_creatnew routines create and open a new file named *filename*; this new file has the access attributes specified in the *attrib* parameter. The new file's handle is copied into the integer location pointed to by *handle*. The file is opened for both read and write access. If file sharing is installed, the file is opened in compatibility mode.

The _dos_creat routine uses system call 0x3C, and the _dos_creatnew routine uses system call 0x5B. If the file already exists, _dos_creat erases its contents and leaves its attributes unchanged; however, the _dos_creatnew routine fails if the file already exists.

Return Value

If successful, both routines return 0. Otherwise, they return the MS-DOS error code and set errno to one of the following values:

Constant	Meaning
EACCES	Access denied because the directory is full or, for _dos_creat only, the file exists and cannot be overwritten
EEXIST	File already exists (_dos_creatnew only)
EMFILE	Too many open file handles
ENOENT	Path or file not found

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```c
/* DCREAT.C: This program creates a file using the _dos_creat
 * function. The program cannot create a new file using the
 * _dos_creatnew function because it already exists.
 */
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
void main( void )
{
   int fh1, fh2;
   int result;
   if( ( result = _dos_creat( "data", _A_NORMAL, &fh1 ) ) != 0 )
      printf( "Couldn't create data file\n" );
   else
   {
      printf( "Created data file.\n" );
      /* If _dos_creat is successful, the _dos_creatnew call
       * will fail since the file exists
       */
      if( ( result = _dos_creatnew( "data", _A_RDONLY, &fh2 ) ) != 0 )
         printf( "Couldn't create data file\n" );
      else
      {
         printf( "Created data file.\n" );
        _dos_close( fh2 );
      }
      _dos_close( fh1 );
   }
}
```

_dos_find Functions

#include <dos.h>, <errno.h>

Syntax unsigned _dos_findfirst(const char *_filename_, unsigned _attrib_, struct _find_t *_fileinfo_);

unsigned _dos_findnext(struct _find_t *_fileinfo_);

Parameter	Description
filename	Target filename
attrib	Target attributes
fileinfo	File-information buffer

The _dos_findfirst routine uses system call 0x4E to return information about the first instance of a file whose name and attributes match *filename* and *attrib*.

The *filename* parameter may use wildcards (* and ?). The *attrib* parameter can be any of the following manifest constants:

_A_ARCH
Archive. Set whenever the file is changed, and cleared by the MS-DOS BACKUP command.

_A_HIDDEN
Hidden file. Cannot be found with the MS-DOS DIR command. Returns information about normal files as well as about files with this attribute.

_A_NORMAL
Normal. File can be read or written without restriction.

_A_RDONLY
Read-only. File cannot be opened for writing, and a file with the same name cannot be created. Returns information about normal files as well as about files with this attribute.

_A_SUBDIR
Subdirectory. Returns information about normal files as well as about files with this attribute.

_A_SYSTEM
System file. Cannot be found with the MS-DOS DIR command. Returns information about normal files as well as about files with this attribute.

_A_VOLID
Volume ID. Only one file can have this attribute, and it must be in the root directory.

Multiple constants can be combined (with the OR operator), using the vertical-bar (|) character.

If the *attrib* parameter to either of these functions is _A_RDONLY, _A_HIDDEN, _A_SYSTEM, or _A_SUBDIR, the function also returns any normal attribute files that match the *filename* parameter. That is, a normal file does not have a read-only, hidden, system, or directory attribute.

Information is returned in a _find_t structure, defined in DOS.H. The _find_t structure contains the following elements:

Element	Description
char reserved[21]	Reserved for use by MS-DOS
char attrib	Attribute byte for matched path
unsigned wr_time	Time of last write to file
unsigned wr_date	Date of last write to file
long size	Length of file in bytes

char name[13]	Null-terminated name of matched file/directory, without the path

The formats for the wr_time and wr_date elements are in MS-DOS format and are not usable by any other C run-time function. The time format is shown below:

Bits	Contents
0 - 4	Number of 2-second increments (0 - 29)
5 - 10	Minutes (0 - 59)
11 - 15	Hours (0 - 23)

The date format is shown below:

Bits	Contents
0 - 4	Day of month (1-31)
5 - 8	Month (1-12)
9 - 15	Year (relative to 1980)

Do not alter the contents of the buffer between a call to _dos_findfirst and a subsequent call to the _dos_findnext function. Also, the buffer should not be altered between calls to _dos_findnext.

The _dos_findnext routine uses system call 0x4F to find the next name, if any, that matches the *filename* and *attrib* parameters specified in a prior call to _dos_findfirst. The *fileinfo* parameter must point to a structure initialized by a previous call to _dos_findfirst. The contents of the structure will be altered as described above if a match is found.

Return Value

If successful, both functions return 0. Otherwise, they return the MS-DOS error code and set errno to ENOENT, indicating that *filename* could not be matched.

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* DFIND.C: This program finds and prints all files
 * in the current directory with the .c extension.
 */
#include <stdio.h>
#include <dos.h>
void main( void )
{
   struct _find_t  c_file;
   /* find first .c file in current directory */
   _dos_findfirst( "*.c", _A_NORMAL, &c_file );
   printf( "Listing of .c files:\n\n" );
   printf( "File: %s is %ld bytes\n", c_file.name, c_file.size );
   /* Find the rest of the .c files */
   while( _dos_findnext( &c_file ) == 0 )
      printf( "File: %s is %ld bytes\n", c_file.name, c_file.size );
}
```

_dos_freemem

#include <dos.h>, <errno.h>

Syntax unsigned _dos_freemem(unsigned *seg*);

Parameter	**Description**
seg	Block to be released

The _dos_freemem function uses system call 0x49 to release a block of memory previously allocated by _dos_allocmem. The *seg* parameter is a value returned by a previous call to _dos_allocmem. The freed memory may no longer be used by the application program.

Return Value

If successful, _dos_freemem returns 0. Otherwise, it returns the MS-DOS error code and sets errno to ENOMEM, indicating a bad segment value (one that does not correspond to a segment returned by a previous _dos_allocmem call) or invalid arena (memory area) headers.

_dos_allocmem
_dos_setblock
free functions

Standards: None

16-Bit: MS-DOS

_dos_getdate

#include <dos.h>

Syntax void _dos_getdate(struct _dosdate_t **date*);

Parameter	**Description**
date	Current system date

The _dos_getdate routine uses system call 0x2A to obtain the current system date. The date is returned in a _dosdate_t structure, defined in DOS.H.

The _dosdate_t structure contains the following elements:

Element	**Description**
unsigned char day	1-31
unsigned char month	1-12
unsigned int year	1980 - 2099
unsigned char dayofweek	0 - 6 (0 = Sunday)

Return Value

None.

_dos_gettime
_dos_setdate
_dos_settime
gmtime
localtime
mktime
_strdate
_strtime
time

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_dos_getdiskfree

#include <dos.h>, <errno.h>

Syntax unsigned _dos_getdiskfree(unsigned *drive*, struct _diskfree_t * *diskspace*);

Parameter	**Description**
drive	Drive number (default is 0)
diskspace	Buffer to hold disk information

The _dos_getdiskfree routine uses system call 0x36 to obtain information on the disk drive specified by *drive*. The default drive is 0, drive A is 1, drive B is 2, and so on. Information is returned in the _diskfree_t structure (defined in DOS.H) pointed to by *diskspace*.

The struct _diskfree_t structure contains the following elements:

Element	**Description**
unsigned total_clusters	Total clusters on disk
unsigned avail_clusters	Available clusters on disk
unsigned sectors_per_cluster	Sectors per cluster
unsigned bytes_per_sector	Bytes per sector

Return Value

If successful, the function returns 0. Otherwise, it returns a nonzero value and sets errno to EINVAL, indicating that an invalid drive was specified.

_dos_getdrive
_dos_setdrive

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* DGDISKFR.C: This program displays information
 * about the default disk drive.
 */
#include <stdio.h>
#include <dos.h>
void main( void )
{
   struct _diskfree_t drive;
   /* Get information on default disk drive 0 */
   _dos_getdiskfree( 0, &drive );
   printf( "total clusters: %u\n", drive.total_clusters );
   printf( "available clusters: %u\n", drive.avail_clusters );
   printf( "sectors per cluster: %u\n", drive.sectors_per_cluster );
   printf( "bytes per sector: %u\n", drive.bytes_per_sector );
}
```

_dos_getdrive

#include <dos.h>

Syntax void _dos_getdrive(unsigned **drive*);

Parameter	**Description**
drive	Current-drive return buffer

The _dos_getdrive routine uses system call 0x19 to obtain the current disk drive. The current drive is returned in the word that *drive* points to: 1 = drive A, 2 = drive B, and so on.

Return Value

None.

_dos_getdiskfree
_dos_setdrive
_getdrive

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* DGDRIVE.C: This program prints the letter of
 * the current drive, changes the default drive
 * to A, then returns the number of disk drives.
 */
#include <stdio.h>
#include <dos.h>
void main( void )
{
   unsigned olddrive, newdrive;
   unsigned number_of_drives;
   /* Print current default drive information */
   _dos_getdrive( &olddrive );
   printf( "The current drive is: %c\n", 'A' + olddrive - 1 );
   /* Set default drive to be drive A */
   printf( "Changing default drive to A\n");
   _dos_setdrive( 1, &number_of_drives );
   /* Get new default drive information and total number of drives */
   _dos_getdrive( &newdrive );
   printf( "The current drive is: %c\n", 'A' + newdrive - 1 );
   printf( "Number of logical drives: %d\n", number_of_drives );
   /* Restore default drive */
   _dos_setdrive( olddrive, &number_of_drives );
}
```

_dos_getfileattr

#include <dos.h>, <errno.h>

Syntax unsigned _dos_getfileattr(const char **path*, unsigned **attrib*);

Parameter	Description
path	Full path of target file/directory
attrib	Word to store attributes in

The _dos_getfileattr routine uses system call 0x43 to obtain the current attributes of the file or directory pointed to by *path*. The attributes are copied to the low-order byte of the *attrib* word. Attributes are represented by manifest constants, as described below:

_A_ARCH
Archive. Set whenever the file is changed, or cleared by the MS-DOS BACKUP command.

_A_HIDDEN
Hidden file. Cannot be found by a directory search.

_A_NORMAL
Normal. File can be read or written without restriction.

_A_RDONLY
Read-only. File cannot be opened for a write, and a file with the same name cannot be created.

_A_SUBDIR
Subdirectory.

_A_SYSTEM
System file. Cannot be found by a directory search.

_A_VOLID
Volume ID. Only one file can have this attribute, and it must be in the root directory.

Return Value

If successful, the function returns 0. Otherwise, it returns the MS-DOS error code and sets errno to ENOENT, indicating that the target file or directory could not be found.

_access
_chmod
_dos_setfileattr
_umask

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* DGFILEAT.C: This program creates a file with the
 * specified attributes, then prints this information
 * before changing the file attributes back to normal.
 */
#include <stdio.h>
#include <dos.h>
void main( void )
{
   unsigned oldattrib, newattrib;
   /* Get and display file attribute */
   _dos_getfileattr( "DGFILEAT.C", &oldattrib );
   printf( "Attribute: 0x%.4x\n", oldattrib );
   if( ( oldattrib & _A_RDONLY ) != 0 )
      printf( "Read only file\n" );
   else
      printf( "Not a read only file.\n" );
   /* Reset file attribute to normal file */
   _dos_setfileattr( "DGFILEAT.C", _A_RDONLY );
   _dos_getfileattr( "DGFILEAT.C", &newattrib );
   printf( "Attribute: 0x%.4x\n", newattrib );
   /* Restore file attribute */
   _dos_setfileattr( "DGFILEAT.C", oldattrib );
   _dos_getfileattr( "DGFILEAT.C", &newattrib );
   printf( "Attribute: 0x%.4x\n", newattrib );
}
```

_dos_getftime

#include <dos.h>, <errno.h>

Syntax unsigned _dos_getftime(int *handle*, unsigned **date*, unsigned **time*);

Parameter	Description
handle	Target file
date	Date-return buffer
time	Time-return buffer

The _dos_getftime routine uses system call 0x57 to get the date and time that the specified file was last written. The file must have been opened with a call to _dos_open or _dos_creat prior to calling _dos_getftime. The date and time are returned in the words pointed to by *date* and *time*. The values appear in the MS-DOS date and time format:

Time Bits	Meaning
0–4	Number of 2-second increments (0–29)
5–10	Minutes (0–59)
11–15	Hours (0–23)

Date Bits	Meaning
0–4	Day (1–31)
5–8	Month (1–12)
9–15	Year (1980–2099)

Return Value

If successful, the function returns 0. Otherwise, it returns the MS-DOS error code and sets errno to EBADF, indicating that an invalid file handle was passed.

_dos_setftime
_fstat_stat
_stat

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* DGFTIME.C: This program displays and modifies
 * the date and time fields of a file.
 */
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
void main( void )
{
                                /* FEDC BA98 7654 3210           */
   unsigned new_date = 0x26cf;  /* 0010 0110 1100 1111  12/15/99 */
   unsigned new_time = 0x48e0;  /* 0100 1000 1110 0000  9:07 AM  */
   unsigned old_date, old_time;
   int fh;
  /* Open file with _dos_open function */
   if( _dos_open( "dgftime.obj", _O_RDONLY, &fh ) != 0 )
      exit( 1 );
   /* Get file date and time */
   _dos_getftime( fh, &old_date, &old_time );
   printf( "Old date field: 0x%.4x\n", old_date );
   printf( "Old time field: 0x%.4x\n", old_time );
   system( "dir dgftime.obj" );
   /* Modify file date and time */
   if( !_dos_setftime( fh, new_date, new_time ) )
   {
      _dos_getftime( fh, &new_date, &new_time );
      printf( "New date field: 0x%.4x\n", new_date );
      printf( "New time field: 0x%.4x\n", new_time );
      system( "dir dgftime.obj" );
      /* Restore date and time */
      _dos_setftime( fh, old_date, old_time );
   }
   _dos_close( fh );
}
```

_dos_gettime

#include <dos.h>

Syntax void _dos_gettime(struct _dostime_t **time*);

Parameter	**Description**
time	Current system time

The _dos_gettime routine uses system call 0x2C to obtain the current system time. The time is returned in a _dostime_t structure, defined in DOS.H.

The dostime_t structure contains the following elements:

Element	**Description**
unsigned char hour	0–23
unsigned char minute	0–59
unsigned char second	0–59
unsigned char hsecond	1/100 second; 0–99

Return Value

None.

_dos_getdate
_dos_setdate
_dos_settime
gmtime
localtime
_strtime

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* DGTIME.C: This program gets and displays
 * current date and time values.
 */
#include <stdio.h>
#include <dos.h>
void main( void )
{
   struct _dosdate_t date;
   struct _dostime_t time;
   /* Get current date and time values */
   _dos_getdate( &date );
   _dos_gettime( &time );
   printf( "Today's date is %d-%d-%d\n", date.month, date.day, date.year );
   printf( "The time is %02d:%02d\n", time.hour, time.minute );
}
```

_dos_getvect

#include <dos.h>

Syntax void (__cdecl __interrupt __far *_dos_getvect(unsigned *intnum*))( );

Parameter	Description
intnum	Target interrupt vector

The _dos_getvect routine uses system call 0x35 to get the current value of the interrupt vector specified by *intnum*.

This routine is typically used in conjunction with the _dos_setvect function. To replace an interrupt vector, first save the current vector of the interrupt using _dos_getvect. Then set the vector to your own interrupt routine with _dos_setvect. The saved vector can later be restored, if necessary, using _dos_setvect. The user-defined routine may also need the original vector in order to call that vector or chain to it with _chain_intr.

Return Value

The function returns a far pointer for the *intnum* interrupt to the current handler, if there is one.

_chain_intr
_dos_keep
_dos_setvect

Standards:	None
16-Bit:	MS-DOS, QWIN, WIN, WIN DLL

_dos_keep

#include <dos.h>

Syntax void _dos_keep(unsigned *retcode*, unsigned *memsize*);

Parameter	Description
retcode	Exit status code
memsize	Allocated resident memory (in 16-byte paragraphs)

The _dos_keep routine installs TSRs (terminate-and-stay-resident programs) in memory, using system call 0x31.

The routine first exits the calling process, leaving it in memory. It then returns the low-order byte of *retcode* to the parent of the calling process. Before returning execution to the parent process, _dos_keep sets the allocated memory for the now-resident process to *memsize* 16-byte paragraphs. Any excess memory is returned to the system.

The _dos_keep function calls the same internal routines called by exit. It therefore takes the following actions:

- Calls any functions that have been registered by atexit or _onexit calls.
- Flushes all file buffers.
- Restores interrupt vectors replaced by the C startup code. The primary one is interrupt 0 (divide by zero). If the emulator math library is used and there is no coprocessor, interrupts 0x34 through 0x3D are restored. If there is a coprocessor, interrupt 2 is restored.

Do not use the emulator math library in TSRs unless you are familiar with the startup code and the coprocessor. Use the alternate math package if the TSR must do floating-point math.

The _dos_keep function should not be used in any program that shells out to an MS-DOS command line, since doing so causes subsequent memory problems. The Microsoft Programmer's WorkBench (PWB) environment is an example of such a program.

Return Value

None.

_cexit
_chain_intr
_dos_getvect
_dos_setvect
_exit

Standards: None
16-Bit: MS-DOS

_dos_lock

#include <dos.h>, <errno.h>

Syntax unsigned _dos_lock(int *handle*, int *mode,* unsigned long *origin*, unsigned long *length*);

Parameter	Description
handle	Target file handle
mode	Lock or unlock region
origin	Start of region to be locked or unlocked
length	Length of region to be locked or unlocked

The _dos_lock routine is used to synchronize files and records in a multitasking environment or network. The routine uses MS-DOS system call 0x5C to lock or unlock a region of the file specified by *handle*. The region is defined with the *origin* argument, which indicates where the region starts, and the *length* argument, which indicates the length of the region. If *mode* is 0, the specified region is to be locked; if *mode* is 1, the specified region is to be unlocked. The region to be locked can extend beyond the end of file. File sharing must be installed to use this routine. (See your system documentation for detailed information about file sharing.)

To determine the status of a region, use _dos_lock to lock the region, then examine the return value.

Return Value

If successful, the _dos_lock routine returns 0. Otherwise, it returns the MS-DOS error code and sets errno as follows:

Error Code	Condition
EBADF	This error code is set if *handle* is invalid.
EACCESS	This error code is set if all or part of the specified region is already locked or if the specified region to be unlocked is not identical to a region previously locked, or if another program tries to read from or write to a locked region.
EINVAL	This error code is set if file sharing is not installed or if *mode* is invalid.

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_dos_open

#include <dos.h>, <errno.h>

#include <fcntl.h> Access mode constants

#include <share.h> Sharing mode constants

Syntax unsigned _dos_open(const char **filename*, unsigned *mode*, int **handle*);

Parameter	Description
filename	Path to an existing file
mode	Permissions
handle	Pointer to integer

The _dos_open routine uses system call 0x3D to open the existing file pointed to by *filename*. The handle for the opened file is copied into the integer pointed to by *handle*. The *mode* parameter specifies the file's access, sharing, and inheritance modes by combining (with the OR operator) manifest constants from the three groups shown below. At most, one access mode and one sharing mode can be specified at a time.

Constant	Mode	Meaning
_O_RDONLY	Access	Read-only
_O_WRONLY	Access	Write-only
_O_RDWR	Access	Both read and write
_SH_COMPAT	Sharing	Compatibility
_SH_DENYRW	Sharing	Deny reading and writing
_SH_DENYWR	Sharing	Deny writing
_SH_DENYRD	Sharing	Deny reading
_SH_DENYNO	Sharing	Deny neither
_O_NOINHERIT	Inheritance by the child process	File is not inherited

Do not use the MS-DOS interface I/O routines in conjunction with the console, low-level, or stream I/O routines.

Return Value

If successful, the function returns 0. Otherwise, it returns the MS-DOS error code and sets errno to one of the following manifest constants:

Constant	Meaning
EACCES	Access denied (possible reasons include specifying a directory or volume ID for *filename*, or opening a read-only file for write access)
EINVAL	Sharing mode specified when file sharing not installed, or

	access-mode value is invalid
EMFILE	Too many open file handles
ENOENT	Path or file not found

_dos_close
_dos_read
_dos_write

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* DOPEN.C: This program uses MS-DOS I/O functions
 * to open and close a file. If the file does not
 * exist, display appropriate message.
 */
#include <fcntl.h>
#include <stdio.h>
#include <dos.h>
void main( void )
{
   int fh;

   /* Open file with _dos_open function */
   if( _dos_open( "data1", _O_RDONLY, &fh ) != 0 )
      perror( "Open failed on input file" );
   else
   {
      printf( "Open succeeded on input file\n" );

      /* Close file with _dos_close function */
      if( _dos_close( fh ) != 0 )
         perror( "Close failed" );
      else
         printf( "File successfully closed\n" );
   }
}
```

_dos_read

#include <dos.h>

Syntax unsigned _dos_read(int *handle*, void __far **buffer*, unsigned *count,* unsigned **numread*);

Parameter	Description
handle	File to read
buffer	Buffer to write to
count	Number of bytes to read
numread	Number of bytes actually read

The _dos_read routine uses system call 0x3F to read *count* bytes of data from the file specified by *handle*. The routine then copies the data to the buffer pointed to by *buffer*. The integer pointed to by *numread* will show the number of bytes actually read, which may be less than the number requested in *count*. If the number of bytes actually read is 0, it means the routine tried to read at end-of-file.

Do not use the MS-DOS interface I/O routines in conjunction with the console, low-level, or stream I/O routines.

Return Value

If successful, the function returns 0. Otherwise, it returns the MS-DOS error code and sets errno to one of the following constants:

Constant	Meaning
EACCES	Access denied (*handle* is not open for read access)
EBADF	File handle is invalid

_dos_close
_dos_open
_dos_write
_read

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* DREAD.C: This program uses the MS-DOS I/O
 * operations to read the contents of a file.
 */
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <dos.h>
void main( void )
{
   int fh; char buffer[50];
   unsigned number_read;
   /* Open file with _dos_open function */
   if( _dos_open( "dread.c", _O_RDONLY, &fh ) != 0 )
      perror( "Open failed on input file\n" );
   else
      printf( "Open succeeded on input file\n" );
   /* Read data with _dos_read function */
   _dos_read( fh, buffer, 50, &number_read );
   printf( "First 40 characters are: %.40s\n\n", buffer );
   /* Close file with _dos_close function */
   _dos_close( fh );
}
```

_dos_seek

#include <dos.h>, <errno.h>

Syntax unsigned long _dos_seek(int *handle*, unsigned long *offset,* int *origin*);

Parameter	Description
handle	Target file handle
offset	Offset of new position relative to *origin*
origin	Position in target file to begin seeking

The _dos_seek routine uses MS-DOS system call 0x42 to move the file pointer to the specified position in the open file specified by *handle*. The file pointer is maintained by the system; it points to the next byte to be read from a file or to the next position in the file to receive a byte. The *offset* specifies the number of bytes to move the file pointer. The *origin* specifies the origin of the move; to move the pointer from the beginning of the file, set *origin* to 0; to move the pointer from the current position, set *origin* to 1; to move the pointer from the end of the file, set *origin* to 2.

A program should never move the file pointer to a position before the beginning of the file. Although this does not generate an error during the move, it generates an error on a subsequent read or write operation. However, a program can move the file pointer past the end of file. A subsequent write operation writes data to the given position in the file, filling the gap between the previous end of file and the given position with undefined data. This is a common way to reserve file space without writing to the file.

Return Value

If successful, the routine returns the offset, in bytes, from the beginning of the file. Otherwise, it returns a value of -1L for offest, and sets errno to EBADF if *handle* is invalid, or to EINVAL if the value proivded for *origin* is invalid.

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_dos_setblock

#include <dos.h>

Syntax unsigned _dos_setblock(unsigned *size*, unsigned *seg*, unsigned **maxsize*);

Parameter	**Description**
size	New segment size
seg	Target segment
maxsize	Maximum-size buffer

The _dos_setblock routine uses system call 0x4A to change the size of *seg*, previously allocated by _dos_allocmem, to *size* paragraphs. If the request cannot be satisfied, the maximum possible segment size is copied to the buffer pointed to by *maxsize*.

Return Value

The function returns 0 if successful. If the call fails, it returns the MS-DOS error code and sets errno to ENOMEM, indicating a bad segment value was passed. A bad segment value is one that does not correspond to a segment returned from a previous _dos_allocmem call, or one that contains invalid arena headers.

_dos_allocmem
_dos_freemem
realloc functions

Standards: None

16-Bit: MS-DOS

_dos_setdate

#include <dos.h>

Syntax unsigned _dos_setdate(struct _dosdate_t **date*);



Parameter	Description
date	New system date

The _dos_setdate routine uses system call 0x2B to set the current system date. The date is stored in the _dosdate_t structure pointed to by *date*, defined in DOS.H. The _dosdate_t structure contains the following elements:

Element	Description
unsigned char day	1-31
unsigned char month	1-12
unsigned int year	1980-2099
unsigned char dayofweek	0-6 (0 = Sunday)

Return Value

If successful, the function returns 0. Otherwise, it returns a nonzero value and sets errno to EINVAL, indicating an invalid date was specified.

_dos_getdate
_dos_gettime
_dos_settime
gmtime
localtime
mktime
_strdate
_strtime
time

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* DSTIME.C: This program changes the time and date
 * values and displays the new date and time values.
 */
#include <dos.h>
#include <conio.h>
#include <stdio.h>
#include <time.h>
void main( void )
{
   struct _dosdate_t olddate, newdate = { { 4 }, { 7 }, { 1999 } };
   struct _dostime_t oldtime, newtime = { { 3 }, { 45 }, { 30 }, { 0 } };
   char   datebuf[40], timebuf[40];
   /* Get current date and time values */
   _dos_getdate( &olddate );
   _dos_gettime( &oldtime );
   printf( "%s    %s\n" , _strdate( datebuf ), _strtime( timebuf ) );
   /* Modify date and time structures */
   _dos_setdate( &newdate ); _dos_settime( &newtime );
   printf( "%s    %s\n" , _strdate( datebuf ), _strtime( timebuf ) );
   /* Restore old date and time */
   _dos_setdate( &olddate );
   _dos_settime( &oldtime );
}
```

_dos_setdrive

#include <dos.h>

Syntax void _dos_setdrive(unsigned *drive*, unsigned **numdrives*);

Parameter	**Description**
drive	New default drive
numdrives	Total drives available

The _dos_setdrive routine uses system call 0x0E to set the current default drive to the *drive* parameter: 1 = drive A, 2 = drive B, and so on. The *numdrives* parameter indicates the total number of drives in the system. If this value is 4, for example, it does not mean the drives are designated A, B, C, and D; it means only that four drives are in the system.

Return Value

There is no return value. If an invalid drive number is passed, the function fails without indication. Use the _dos_getdrive routine to verify whether the desired drive has been set.

_dos_getdiskfree
_dos_getdrive

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_dos_setfileattr

#include <dos.h>

Syntax unsigned _dos_setfileattr(const char **path,* unsigned *attrib*);

Parameter	Description
path	Full path of target file or directory
attrib	New attributes

The _dos_setfileattr routine uses system call 0x43 to set the attributes of the file or directory pointed to by *path*. The actual attributes are contained in the low-order byte of the *attrib* word. Attributes are represented by manifest constants, as described below:

Constant	Meaning
_A_ARCH	Archive. Set whenever the file is changed, or cleared by the MS-DOS BACKUP command.
_A_HIDDEN	Hidden file. Cannot be found by a directory search.
_A_NORMAL	Normal. File can be read or written to without restriction.
_A_RDONLY	Read-only. File cannot be opened for writing, and a file with the same name cannot be created.
_A_SUBDIR	Subdirectory.
_A_SYSTEM	System file. Cannot be found by a directory search.
_A_VOLID	Volume ID. Only one file can have this attribute, and it must be in the root directory.

Return Value

The function returns 0 if successful. Otherwise, it returns the MS-DOS error code and sets errno to one of the following:

Constant	Meaning
EACCES	Access denied; cannot change the volume ID or the subdirectory.
ENOENT	No file or directory matching

the target was found.

_dos_getfileattr

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_dos_setftime

#include <dos.h>

Syntax unsigned _dos_setftime(int *handle*, unsigned *date*, unsigned *time*);

Parameter	Description
handle	Target file
date	Date of last write
time	Time of last write

The _dos_setftime routine uses system call 0x57 to set the *date* and *time* at which the file identified by *handle* was last written to. These values appear in the MS-DOS date and time format, described in the following lists:

Time Bits	Meaning
0 - 4	Number of two-second increments (0 - 29)
5 - 10	Minutes (0 - 59)
11 - 15	Hours (0 - 23)

Date Bits	Meaning
0 - 4	Day (1 - 31)
5 - 8	Month (1 - 12)
9 - 15	Year since 1980 (for example, 1994 is stored as 14)

Return Value

If successful, the function returns 0. Otherwise, it returns the MS-DOS error code and sets errno to EBADF, indicating that an invalid file handle was passed.

_dos_getftime
_fstat
_stat

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_dos_settime

#include <dos.h>

Syntax unsigned _dos_settime(struct _dostime_t **time*);

Parameter	**Description**
time	New system time

The _dos_settime routine uses system call 0x2D to set the current system time to the value stored in the _dostime_t structure that *time* points to, as defined in DOS.H. The _dostime_t structure contains the following elements:

Element	**Description**
unsigned char hour	0–23
unsigned char minute	0–59
unsigned char second	0–59
unsigned char hsecond	Hundredths of a second; 0–99

Return Value

If successful, the function returns 0. Otherwise, it returns a nonzero value and sets errno to EINVAL, indicating an invalid time was specified.

_dos_getdate
_dos_gettime
_dos_setdate
gmtime
localtime
mktime
_strdate
_strtime

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* DSTIME.C: This program changes the time and date
 * values and displays the new date and time values.
 */
#include <dos.h>
#include <conio.h>
#include <stdio.h>
#include <time.h>
void main( void )
{
   struct _dosdate_t olddate, newdate = { { 4 }, { 7 }, { 1999 } };
   struct _dostime_t oldtime, newtime = { { 3 }, { 45 }, { 30 }, { 0 } };
   char   datebuf[40], timebuf[40];
   /* Get current date and time values */
   _dos_getdate( &olddate );
   _dos_gettime( &oldtime );
   printf( "%s    %s\n" , _strdate( datebuf ), _strtime( timebuf ) );
   /* Modify date and time structures */
   _dos_setdate( &newdate ); _dos_settime( &newtime );
   printf( "%s    %s\n" , _strdate( datebuf ), _strtime( timebuf ) );
   /* Restore old date and time */
   _dos_setdate( &olddate );
   _dos_settime( &oldtime );
}
```

_dos_setvect

#include <dos.h>

Syntax void _dos_setvect(unsigned *intnum*, void(__cdecl __interrupt __far **handler*)( ));

Parameter	Description
intnum	Target-interrupt vector
handler	Interrupt handler for which to assign *intnum*

The _dos_setvect routine uses system call 0x25 to set the current value of the interrupt vector *intnum* to the function pointed to by *handler*. Subsequently, whenever the *intnum* interrupt is generated, the *handler* routine will be called. If *handler* is a C function, it must have been previously declared with the interrupt attribute. Otherwise, you must make sure that the function satisfies the requirements for an interrupt-handling routine. For example, if *handler* is an assembler function, it must be a far routine that returns with an IRET instead of a RET.

The interrupt attribute indicates that the function is an interrupt handler. The compiler generates appropriate entry and exit sequences for the interrupt-handling function, including saving and restoring all registers and executing an IRET instruction to return.

The _dos_setvect routine is generally used with the _dos_getvect function. To replace an interrupt vector, first save the current vector of the interrupt using _dos_getvect. Then set the vector to your own interrupt routine with _dos_setvect. The saved vector can later be restored, if necessary, using _dos_setvect. The user-defined routine may also need the original vector in order to call it or to chain to it with _chain_intr.

Registers and Interrupt Functions

When you call an interrupt function, the DS register is initialized to the C data segment. This allows you to access global variables from within an interrupt function.

In addition, all registers except SS are saved on the stack. You can access these registers within the function if you declare a function parameter list containing a formal parameter for each saved register. The following example illustrates such a declaration:

```
void __interrupt __far int_handler( unsigned _es, unsigned _ds,
                                    unsigned _di, unsigned _si,
                                    unsigned _bp, unsigned _sp,
                                    unsigned _bx, unsigned _dx,
                                    unsigned _cx, unsigned _ax,
                                    unsigned _ip, unsigned _cs,
                                    unsigned _flags )
{
.
.
.
}
```

The formal parameters must appear in the opposite order from which they are pushed onto the stack. You can omit parameters from the end of the list in a declaration, but not from the beginning. For example, if your handler needs to use only DI and SI, you must still provide ES and DS, but not necessarily BX or DX.

You can pass additional parameters if your interrupt handler will be called directly from C rather than by an INT instruction. To do this, you must declare all register parameters and then declare your parameter at the end of the list.

The compiler always saves and restores registers in the same, fixed order. Thus, no matter what

names you use in the formal parameter list, the first parameter in the list refers to ES, the second refers to DS, and so on. If your interrupt routines will use inline assembler, you should distinguish the parameter names so that they will not be the same as the real register names.

If you change any of the register parameters of an interrupt function while the function is executing, the corresponding register contains the changed value when the function returns. For example:

```
void __interrupt __far int_handler( unsigned _es, unsigned _ds,
                                    unsigned _di, unsigned _si )
{
   _di = -1;
}
```

This code causes the DI register to contain -1 when the *handler* function returns. It is not a good idea to modify the values of the parameters representing the IP and CS registers in interrupt functions. If you must modify a particular flag (such as the carry flag for certain MS-DOS and BIOS interrupt routines), use the OR operator (|) so that other bits in the flag register are not changed.

When an interrupt function is called by an INT instruction, the interrupt-enable flag is cleared. If your interrupt function needs to do significant processing, you should use the _enable function to set the interrupt flag so that interrupts can be handled.

Precautions for Interrupt Functions

Because MS-DOS is not reentrant (an MS-DOS interrupt cannot be called from inside an MS-DOS interrupt), it is usually not safe to call from inside an interrupt function any standard library function that calls DOS INT 21H. Similar precautions apply to many BIOS functions. Functions that rely on INT 21H calls include I/O functions and the _dos family of functions. Functions that rely on the machine's BIOS include graphics functions and the _bios family of functions. It is usually safe to use functions that do not rely on INT 21H or BIOS, such as string-handling functions. Before using a standard library function in an interrupt function, be sure that you are familiar with the action of the library function.

Return Value

None.

_chain_intr
_dos_getvect
_dos_keep

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_dos_write

#include <dos.h>

Syntax unsigned _dos_write(int *handle*, void __far **buffer*, unsigned *count,* unsigned **numwrt*);

Parameter	Description
handle	File to write to
buffer	Buffer to write from
count	Number of bytes to write
numwrt	Number of bytes actually written

The _dos_write routine uses system call 0x40 to write data to the file that *handle* references; *count* bytes of data from the buffer to which *buffer* points are written to the file. The integer pointed to by *numwrt* will be the number of bytes actually written, which may be less than the number requested.

Do not use the MS-DOS interface routines with the console, low-level, or stream I/O routines.

Return Value

If successful, the function returns 0. Otherwise, it returns the MS-DOS error code and sets errno to one of the following manifest constants:

Constant	Meaning
EACCES	Access denied (*handle* references a file not open for write access)
EBADF	Invalid file handle

_dos_close
_dos_open
_dos_read
_write

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* DWRITE.C: This program uses MS-DOS I/O
 * functions to write to a file.
 */
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
void main( void )
{
   char out_buffer[] = "Hello";
   int  fh;
   unsigned n_written;
   /* Open file with _dos_creat function */
   if( _dos_creat( "data", _A_NORMAL, &fh ) == 0 )
   {
      /* Write data with _dos_write function */
      _dos_write( fh, out_buffer, 5, &n_written );
      printf( "Number of characters written: %d\n", n_written );
      _dos_close( fh );
      printf( "Contents of file are:\n" );
      system( "type data" );
   }
}
```

_dosexterr

#include <dos.h>

Syntax int _dosexterr(struct _DOSERROR **errorinfo*);

Parameter	Description
errorinfo	Extended MS-DOS error information

The _dosexterr function obtains the extended error information returned by MS-DOS system call 0x59 and stores the values in the structure pointed to by *errorinfo*. This function is useful when making system calls with MS-DOS versions 3.0 and later, which offer extended error handling.

The structure type _DOSERROR is defined in DOS.H. The _DOSERROR structure contains the following elements:

Element	Description
int exterror	AX register contents
char errclass	BH register contents
char action	BL register contents
char locus	CH register contents

Giving a NULL pointer parameter causes _dosexterr to return the value in AX without filling in the structure fields. See *MS-DOS Encyclopedia* (Duncan, ed.; Redmond, WA: Microsoft Press, 1988) or *Programmer's PC Sourcebook* 2nd ed. (Hogan; Redmond, WA: Microsoft Press, 1991) for more information on the register contents.

Return Value

The _dosexterr function returns the value in the AX register (identical to the value in the exterror structure field).

The _dosexterr function should be used only with MS-DOS version 3.0 or later.

perror

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* DOSEXERR.C: This program tries to open the file test.dat.
 * If the attempted open operation fails, the program uses
 * _dosexterr to display extended error information.
 */
#include <dos.h>
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
void main( void )
{
   struct _DOSERROR doserror;
   int fd;
   /* Attempt to open a non-existent file */
   if( (fd = _open( "TEST.DAT", _O_RDONLY )) == -1 )
   {
      _dosexterr( &doserror );
      printf( "Error: %d  Errclass: %d  Action: %d  Locus: %d\n",
               doserror.exterror, doserror.errclass,
               doserror.action,   doserror.locus );
   }
   else
   {
      printf( "Open succeeded so no extended information printed\n" );
      _close( fd );
   }
}
```

_dup, _dup2

#include <io.h> Required only for function declarations

Syntax int _dup(int *handle*);

int _dup2(int *handle1*, int *handle2*);

Parameter	Description
handle, *handle1*	Handle referring to open file
handle2	Any handle value

The _dup and _dup2 functions cause a second file handle to be associated with a currently open file. Operations on the file can be carried out using either file handle. The type of access allowed for the file is unaffected by the creation of a new handle.

The _dup function returns the next available file handle for the given file. The _dup2 function forces *handle2* to refer to the same file as *handle1*. If *handle2* is associated with an open file at the time of the call, that file is closed.

Both _dup and _dup2 accept file handles as parameters. If you want to pass a stream (FILE *) to either of these functions, use the _fileno function. This function associates a stream with a file handle. The following example shows how to associate stderr (defined as FILE * in STDIO.H) with a handle:

```
cstderr = _dup( _fileno( stderr ));
```

Note that in a QuickWin application, you cannot use either _dup and _dup2 with _fileno on the stdin, stdout, or stderr streams. You can, however, use the combination of _dup functions and the _fileno function on other streams.

Return Value

The _dup function returns a new file handle. The _dup2 function returns 0 to indicate success. Both functions return -1 if an error occurs and set errno to one of the following values:

Value	Meaning
EBADF	Invalid file handle
EMFILE	No more file handles available (too many open files)

Use _dup and _dup2 for compatibility with ANSI naming conventions of non-ANSI functions. Use dup and dup2 and link with OLDNAMES.LIB for UNIX compatibility.

_close
_creat
_open

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* DUP.C: This program uses the variable old to save
 * the original stdout. It then opens a new file named
 * new and forces stdout to refer to it. Finally, it
 * restores stdout to its original state.
 */
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   int old;
   FILE *new;
   old = _dup( 1 );   /* "old" now refers to "stdout" */
                      /* Note:  file handle 1 == "stdout" */
   if( old == -1 )
   {
      perror( "_dup( 1 ) failure" );
      exit( 1 );
   }
   write( old, "This goes to stdout first\r\n", 27 );
   if( ( new = fopen( "data", "w" ) ) == NULL )
   {
      puts( "Can't open file 'data'\n" );
      exit( 1 );
   }
   /* stdout now refers to file "data" */
   if( -1 == _dup2( _fileno( new ), 1 ) )
   {
      perror( "Can't _dup2 stdout" );
      exit( 1 );
   }
   puts( "This goes to file 'data'\r\n" );
   /* Flush stdout stream buffer so it goes to correct file */
   fflush( stdout );
   fclose( new );
   /* Restore original stdout */
   _dup2( old, 1 );
   puts( "This goes to stdout\n" );
   puts( "The file 'data' contains:" );
   system( "type data" );
}
```

_ecvt

#include <stdlib.h> Required only for function declarations

Syntax char *_ecvt(double *value*, int *count*, int **dec*, int **sign*);

Parameter	Description
value	Number to be converted
count	Number of digits stored
dec	Stored decimal-point position
sign	Sign of converted number

The _ecvt function converts a floating-point number to a character string. The *value* parameter is the floating-point number to be converted. The _ecvt function stores up to *count* digits of *value* as a string and appends a null character ('\0'). If the number of digits in *value* exceeds *count,* the low-order digit is rounded. If there are fewer than *count* digits, the string is padded with zeros.

Only digits are stored in the string. The position of the decimal point and the sign of *value* can be obtained from *dec* and *sign* after the call. The *dec* parameter points to an integer value giving the position of the decimal point with respect to the beginning of the string. A 0 or negative integer value indicates that the decimal point lies to the left of the first digit. The *sign* parameter points to an integer indicating the sign of the converted number. If the integer value is 0, the number is positive. Otherwise, the number is negative.

The _ecvt and _fcvt functions use a single statically allocated buffer for the conversion. Each call to one of these routines destroys the result of the previous call.

Return Value

The _ecvt function returns a pointer to the string of digits. There is no error return.

Use _ecvt for compatibility with ANSI naming conventions of non-ANSI functions. Use ecvt and link with OLDNAMES.LIB for UNIX compatibility.

atof
atoi
atol
_fcvt
_gcvt

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* ECVT.C: This program uses _ecvt to convert a
 * floating-point number to a character string.
 */
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   int     decimal,   sign;
   char    *buffer;
   int     precision = 10;
   double  source = 3.1415926535;
   buffer = _ecvt( source, precision, &decimal, &sign );
   printf( "source: %2.10f   buffer: '%s'  decimal: %d   sign: %d\n",
           source, buffer, decimal, sign );
}
```

_ellipse Functions

#include <graph.h>

Syntax short __far _ellipse(short *control*, short *x1*, short *y1*, short *x2*, short *y2*);

short __far _ellipse_w(short *control*, double *wx1*, double *wy1*, double *wx2*, double *wy2*);

short __far _ellipse_wxy(short *control*, struct _wxycoord __far **pwxy1* , struct _wxycoord __far **pwxy2*);

Parameter	Description
control	Fill flag
x1, *y1*	Upper-left corner of bounding rectangle
x2, *y2*	Lower-right corner of bounding rectangle
wx1, *wy1*	Upper-left corner of bounding rectangle
wx2, *wy2*	Lower-right corner of bounding rectangle
pwxy1	Upper-left corner of bounding rectangle
pwxy2	Lower-right corner of bounding rectangle

The _ellipse functions draw ellipses or circles. The borders are drawn in the current color. In the _ellipse function, the center of the ellipse is the center of the bounding rectangle defined by the view-coordinate points (*x1*, *y1*) and (*x2*, *y2*).

In the _ellipse_w function, the center of the ellipse is the center of the bounding rectangle defined by the window-coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*).

In the _ellipse_wxy function, the center of the ellipse is the center of the bounding rectangle defined by the window-coordinate points (*pwxy1*) and (*pwxy2*).

If the bounding-rectangle arguments define a point or a vertical or horizontal line, no figure is drawn.

The *control* parameter can be one of the following manifest constants:

Constant	Action
_GFILLINTERIOR	Uses _floodfill to fill the ellipse using the current fill mask
_GBORDER	Does not fill the ellipse

The control option given by _GFILLINTERIOR is equivalent to a subsequent call to the _floodfill function, using the center of the ellipse as the starting point and the current color (set by _setcolor) as the boundary color.

Return Value

The _ellipse functions return a nonzero value if the ellipse is drawn successfully; otherwise, they return 0.

_arc functions
_floodfill
_grstatus
_lineto functions
_pie functions
_polygon functions
_rectangle functions
_setcolor
_setfillmask

Standards: None
16-Bit: MS-DOS, QWIN

```
/* ELLIPSE.C: This program draws a simple ellipse. */
#include <conio.h>
#include <stdlib.h>
#include <graph.h>
void main( void )
{
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   _ellipse( _GFILLINTERIOR, 80, 50, 240, 150 );
   /* Strike any key to clear screen. */
   _getch();
   _setvideomode( _DEFAULTMODE );
}
```

_enable

#include <dos.h>

Syntax void _enable(void)**;**

The _enable routine enables interrupts by executing an 8086 STI machine instruction**.**

Return Value

None**.**

_disable

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_eof

#include <io.h> Required only for function declarations

Syntax int _eof(int *handle*);

Parameter	**Description**
handle	Handle referring to open file

The _eof function determines whether the end of the file associated with *handle* has been reached.

Return Value

The _eof function returns the value 1 if the current position is end-of-file, or 0 if it is not. A return value of -1 indicates an error; in this case, errno is set to EBADF, indicating an invalid file handle.

clearerr
feof
ferror
perror

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* EOF.C: This program reads data from a file
 * ten bytes at a time until the end of the
 * file is reached or an error is encountered.
 */
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
void main( void )
{
   int  fh, count, total = 0;
   char buf[10];
   if( (fh = _open( "_eof.c", _O_RDONLY )) == - 1 )
   {
        perror( "Open failed");
        exit( 1 );
   }
   /* Cycle until end of file reached: */
   while( !_eof( fh ) )
   {
      /* Attempt to read in 10 bytes: */
      if( (count = _read( fh, buf, 10 )) == -1 )
      {
         perror( "Read error" );
         break;
      }
      /* Total up actual bytes read */
      total += count;
   }
   printf( "Number of bytes read = %d\n", total );
   _close( fh );
}
```

_exec Functions

#include <process.h> Required only for function declarations

Syntax int _execl(const char *_cmdname_, const char *_arg0, ..._ const char *_argn_, NULL);

int _execle(const char *_cmdname_, const char *_arg0, ..._ const char *_argn_, NULL, const char * const *_envp_);

int _execlp(const char *_cmdname_, const char *_arg0, ..._ const char *_argn_, NULL);

int _execlpe(const char *_cmdname_, const char *_arg0, ..._ const char *_argn_, NULL, const char * const *_envp_);

int _execv(const char *_cmdname_, const char *const *_argv_);

int _execve(const char *_cmdname_, const char *const *_argv_, const char * const *_envp_);

int _execvp(const char *_cmdname_, const char *const *_argv_);

int _execvpe(const char *_cmdname_, const char *const *_argv_, const char * const *_envp_);

Parameter	Description
cmdname	Path of file to be executed
arg0, ... argn	List of pointers to parameters
argv	Array of pointers to parameters
envp	Array of pointers to environment settings

The _exec functions load and execute new child processes. When the call is successful in MS-DOS, the child process is placed in the memory previously occupied by the calling process. Sufficient memory must be available for loading and executing the child process.

All of the _exec functions use the same operating system function. The letter(s) at the end of the function name determine the specific variation, as shown in the following list:

Letter	Variation
e	An array of pointers to environment parameters is explicitly passed to the child process.
l	Command-line arguments are passed individually to the _exec function.
p	Uses the PATH environment variable to find the file to be executed.
v	Command-line arguments are passed to the _exec function as an array of pointers.

The *cmdname* parameter specifies the file to be executed as the child process. It can specify a full path (from the root), a partial path (from the current working directory), or just a filename. If *cmdname* does not have a filename extension or does not end with a period (.), the _exec function searches for the named file; if the search is unsuccessful, it tries the same base name, first with the extension .COM, then with the extension .EXE. If *cmdname* has an extension, only that extension is used in the search. If *cmdname* ends with a period, the _exec calls search for *cmdname* with no extension. The _execlp, _execlpe, _execvp, and _execvpe routines search for *cmdname* (using the same procedures) in the directories specified by the PATH environment variable.

If *cmdname* contains a drive specifier or any slashes (that is, if it is a relative path), the _exec call searches only for the specified file; the path is not searched. Note that the MS-DOS APPEND command cannot be used with the _exec functions.

Parameters are passed to the new process by giving one or more pointers to character strings as parameters in the _exec call. These character strings form the parameter list for the child process. The combined length of the strings forming the parameter list for the new process must not exceed 128 bytes (in real mode only). The terminating null character ('\0') for each string is not included in the count, but space characters (inserted automatically to separate the parameters) are counted.

The argument pointers can be passed as separate parameters (_execl, _execle, _execlp, and _execlpe) or as an array of pointers (_execv, _execve, _execvp, and _execvpe). At least one parameter, *arg0*, must be passed to the child process; this parameter is *argv*[0] of the child process. Usually, this parameter is a copy of the *cmdname* parameter. (A different value will not produce an error.) With versions of MS-DOS earlier than 3.0, the passed value of *arg0* is not available for use in the child process. However, with MS-DOS versions 3.0 and later, *cmdname* is available as *arg0*.

The _execl, _execle, _execlp, and _execlpe calls are typically used when the number of parameters is known in advance. The parameter *arg0* is usually a pointer to *cmdname*. The parameters *arg1* through *argn* point to the character strings forming the new parameter list. A null pointer must follow *argn* to mark the end of the parameter list.

The _execv, _execve, _execvp, and _execvpe calls are useful when the number of parameters to the new process is variable. Pointers to the parameters are passed as an array, *argv*. The parameter *argv*[0] is usually a pointer to *cmdname*. The parameters *argv*[1] through *argv*[*n*] point to the character strings forming the new parameter list. The parameter *argv*[*n*+1] must be a NULL pointer to mark the end of the parameter list.

Files that are open when an _exec call is made remain open in the new process. In the _execl, _execlp, _execv, and _execvp calls, the child process inherits the environment of the parent. The _execle, _execlpe, _execve, and _execvpe calls allow the user to alter the environment for the child process by passing a list of environment settings through the *envp* parameter. The parameter *envp* is an array of character pointers, each element of which (except for the final element) points to a null-terminated string defining an environment variable. Such a string usually has the form

NAME=*value*

where NAME is the name of an environment variable and *value* is the string value to which that variable is set. (Note that *value* is not enclosed in double quotation marks.) The final element of the *envp* array should be NULL. When *envp* itself is NULL, the child process inherits the environment settings of the parent process.

A program executed with one of the _exec family of functions is always loaded into memory as if the "maximum allocation" field in the program's .EXE file header is set to the default value of 0xFFFFH. You can use the EXEHDR utility to change the maximum allocation field of a program; however, such a program invoked with one of the _exec functions may behave differently from a program invoked directly from the operating-system command line or with one of the _spawn functions.

Note that COMMAND.COM checks the first two bytes of a file to determine whether it is an .EXE file or a .COM file—you can execute a file named by any extension, as long as its content is truly executable.

The _exec calls do not preserve the translation modes of open files. If the child process must use files inherited from the parent, the _setmode routine should be used to set the translation mode of these files to the desired mode.

You must explicitly flush (using fflush or _flushall) or close any stream prior to the _exec function call.

Signal settings are not preserved in child processes that are created by calls to _exec routines. The signal settings are reset to the default in the child process.

Return Value

The _exec functions do not normally return to the calling process. If an _exec function returns, an error has occurred and the return value is -1. The errno variable is set to one of the following values:

Value	Meaning

E2BIG	The parameter list exceeds 128 bytes, or the space required for the environment information exceeds 32K.
EACCES	The specified file has a locking or sharing violation (MS-DOS version 3.0 or later).
EMFILE	Too many files open (the specified file must be opened to determine whether it is executable).
ENOENT	File or path not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the child process; or the available memory has been corrupted; or an invalid block exists, indicating that the parent process was not allocated properly.

Use _exec for compatibility with ANSI naming conventions of non-ANSI functions. Use exec and link with OLDNAMES.LIB for UNIX compatibility.

Because of differences in MS-DOS versions 2.0 and 2.1, child processes generated by the _exec family of functions (or by the equivalent _spawn functions with the _P_OVERLAY parameter) may cause fatal system errors when they exit. If you are running MS-DOS 2.0 or 2.1, you must upgrade to MS-DOS version 3.0 or later to use these functions.

Bound programs cannot use the _exec family of functions in real mode.

abort
atexit
exit
_exit
_onexit
_spawn functions
system

Standards: UNIX
16-Bit: MS-DOS

```
/* EXEC.C: This program accepts a number in the range 1
 * through 8 from the command line. Based on the number it
 * receives, it executes one of the eight different
 * procedures that spawn the process named child. For some
 * of these procedures, the child.exe file must be in the
 * same directory; for others, it need only be in the same path.
 */
#include <stdio.h>
#include <process.h>
char *my_env[] = {
   "THIS=environment will be",
   "PASSED=to child.exe by the",
   "_EXECLE=and",
   "_EXECLPE=and",
   "_EXECVE=and",
   "_EXECVPE=functions",
   NULL
   };
void main( int argc, char *argv[] )
{
   char *args[4];
   args[0] = "child";     /* Set up parameters to send */
   args[1] = "_execv??";
   args[2] = "two";
   args[3] = NULL;
   switch( argv[1][0] )  /* Based on first letter of argument */
   {
   case '1':
      _execl( argv[2], argv[2], "_execl", "two", NULL );
      break;
   case '2':
      _execle( argv[2], argv[2], "_execle", "two", NULL, my_env );
      break;
   case '3':
      _execlp( argv[2], argv[2], "_execlp", "two", NULL );
      break;
   case '4':
      _execlpe( argv[2], argv[2], "_execlpe", "two", NULL, my_env );
      break;
   case '5':
      _execv( argv[2], args );
      break;
   case '6':
      _execve( argv[2], args, my_env );
      break;
   case '7':
      _execvp( argv[2], args );
      break;
   case '8':
      _execvpe( argv[2], args, my_env );
      break;
   default:
      printf( "SYNTAX: EXEC <1-8> <childprogram>\n" );
      exit( 1 );
   }
   printf( "Process was not spawned.\n" );
   printf( "Program 'child' was not found." );
}
```

exit, _exit

#include <process.h> Required only for function declarations

#include <stdlib.h> Use either PROCESS.H or STDLIB.H

Syntax void exit(int *status*);

void _exit(int *status*);

Parameter	**Description**
status	Exit status

The exit and _exit functions terminate the calling process. The exit function first calls, in LIFO (last-in-first-out) order, the functions registered by atexit and _onexit, then flushes all file buffers before terminating the process. The _exit function terminates the process without processing atexit or _onexit functions or flushing stream buffers. The *status* value is typically set to 0 to indicate a normal exit and set to some other value to indicate an error.

Although the exit and _exit calls do not return a value, the low-order byte of *status* is made available to the waiting parent process, if one exists, after the calling process exits. The *status* value is available to the operating-system batch command ERRORLEVEL.

The behavior of the exit, _exit, _cexit, and _c_exit functions is as follows:

Function	**Action**
exit	Performs complete C library termination procedures, terminates the process, and exits with the supplied status code.
_exit	Performs "quick" C library termination procedures, terminates the process, and exits with the supplied status code.
_cexit	Performs complete C library termination procedures and returns to caller, but does not terminate the process.
_c_exit	Performs "quick" C library termination procedures and returns to caller, but does not terminate the process.

Return Value

None.

abort
atexit
_cexit
_exec functions
_onexit
_spawn functions
system

exit

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN

_exit

Standards: None
16-Bit: MS-DOS, QWIN, WIN

```
/* EXITER.C: This program prompts the user for a yes
 * or no and returns an exit code of 1 if the
 * user answers Y or y; otherwise it returns 0. The
 * error code could be tested in a batch file.
 */
#include <conio.h>
#include <stdlib.h>
void main( void )
{
   int ch;
   _cputs( "Yes or no? " );
   ch = _getch();
   _cputs( "\r\n" );
   if( toupper( ch ) == 'Y' )
      exit( 1 );
   else
      exit( 0 );
}
```

exp, expl

#include <math.h>

Syntax double exp(double *x*);

long double expl(long double *x*);

Parameter	Description
x	Floating-point value

The exp and expl functions return the exponential function of their floating-point parameters (*x*).

The expl function is the 80-bit counterpart; it uses an 80-bit, 10-byte coprocessor form of parameters and return values. See the long double functions for more details on this data type.

Return Value

These functions return the exponential value of *x* if successful. On overflow, exp and expl return _LHUGE_VAL. On underflow, they return 0. The errno variable is set to ERANGE on overflow but is not set on underflow.

log functions

exp

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

expl

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* EXP.C */
#include <math.h>
#include <stdio.h>
void main( void )
{
   double x = 2.302585093, y;
   y = exp( x );
   printf( "exp( %f ) = %f\n", x, y );
}
```

_expand Functions

#include <malloc.h> Required only for function declarations

Syntax void *_expand(void **memblock*, size_t *size*);

void __based(void) *_bexpand(__segment *seg*, void __based(void) **memblock*, size_t *size*);

void __far *_fexpand(void __far **memblock*, size_t *size*);

void __near *_nexpand(void __near **memblock*, size_t *size*);

Parameter	Description
memblock	Pointer to previously allocated memory block
size	New size in bytes
seg	Value of base segment

The _expand family of functions changes the size of a previously allocated memory block by attempting to expand or contract the block without moving its location in the heap. The *memblock* parameter points to the beginning of the block. The *size* parameter gives the new size of the block, in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes.

The *memblock* parameter can also point to a block that has been freed, as long as there has been no intervening call to calloc, _expand, malloc, or realloc. If *memblock* points to a freed block, the block remains free after a call to one of the _expand functions.

The *seg* parameter is the segment address of the __based heap.

In large data models (compact-, large-, and huge-model programs), _expand maps to _fexpand. In small data models (tiny-, small-, and medium-model programs), _expand maps to _nexpand.

The various _expand functions change the size of the storage block in the data segments shown in the list below:

Function	Data Segment
_expand	Depends on data model of program
_bexpand	Based heap specified by *seg*, or in all based heaps if *seg* is zero
_fexpand	Far heap (outside default data segment)
_nexpand	Near heap (inside default data segment)

Return Value

The _expand family of functions returns a void pointer to the reallocated memory block. Unlike realloc, _expand cannot move a block to change its size. This means the *memblock* parameter to _expand is the same as the return value if there is sufficient memory available to expand the block without moving it.

With the exception of the _bexpand function, these functions return NULL if there is insufficient memory available to expand the block to the given size without moving it. The _bexpand function returns _NULLOFF if insufficient memory is available. The item pointed to by *memblock* will have been expanded as much as possible in its current location.

The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. The new size of the item can be checked with one of the _msize functions. To get a pointer to a type other than void, use a type cast on the return value.

calloc functions
free functions
malloc functions
_msize functions
realloc functions

_expand

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_bexpand, **_fexpand**, **_nexpand**

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* EXPAND.C */
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
void main( void )
{
   char *bufchar;
   printf( "Allocate a 512 element buffer\n" );
   if( (bufchar = (char *)calloc( 512, sizeof( char ) )) == NULL )
      exit( 1 );
   printf( "Allocated %d bytes at %Fp\n", _msize( bufchar ), (void __far
*)bufchar );
   if( (bufchar = (char *)_expand( bufchar, 1024 )) == NULL )
      printf( "Can't expand" );
   else
      printf( "Expanded block to %d bytes at %Fp\n", _msize( bufchar ), (void __far
*)bufchar );
   /* Free memory */
   free( bufchar );
   exit( 0 );
}
```

fabs, fabsl

#include <math.h>

Syntax double fabs(double *x*);

long double fabsl(long double *x*);

Parameter	Description
x	Floating-point value

The fabs and fabsl functions calculate the absolute value of their floating-point parameters.

The fabsl function is the 80-bit counterpart; it uses an 80-bit, 10-byte coprocessor form of parameters and return values. See the long double functions for more details on this data type.

Return Value

These functions return the absolute value of their arguments. There is no error return.

abs
_cabs
labs

fabs

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

fabsl

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* ABS.C: This program computes and displays
 * the absolute values of several numbers.
 */
#include  <stdio.h>
#include  <math.h>
#include  <stdlib.h>
void main( void )
{
   int    ix = -4, iy;
   long   lx = -41567L, ly;
   double dx = -3.141593, dy;
   iy = abs( ix );
   printf( "The absolute value of %d is %d\n", ix, iy);
   ly = labs( lx );
   printf( "The absolute value of %ld is %ld\n", lx, ly);
   dy = fabs( dx );
   printf( "The absolute value of %f is %f\n", dx, dy );
}
```

fclose, _fcloseall

#include <stdio.h>

Syntax int fclose(FILE **stream*);

int _fcloseall(void);

Parameter	**Description**
stream	Pointer to FILE structure

The fclose function closes *stream*. The _fcloseall function closes all open streams except stdin, stdout, stderr (and in MS-DOS, _stdaux and _stdprn). It also closes and deletes any temporary files created by tmpfile.

In both functions, all buffers associated with the stream are flushed prior to closing. System-allocated buffers are released when the stream is closed. Buffers assigned by the user with setbuf and setvbuf are not automatically released.

Return Value

The fclose function returns 0 if the stream is successfully closed. The _fcloseall function returns the total number of streams closed. Both functions return EOF to indicate an error.

_close
_fdopen
fflush
fopen
freopen

fclose

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fcloseall

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FOPEN.C: This program opens files named "data"
 * and "data2".It  uses fclose to close "data" and
 * _fcloseall to close all remaining files.
 */
#include <stdio.h>
FILE *stream, *stream2;
void main( void )
{
   int numclosed;
   /* Open for read (will fail if 'data does not exist) */
   if( (stream  = fopen( "data", "r" )) == NULL )
      printf( "The file 'data' was not opened\n" );
   else
      printf( "The file 'data' was opened\n" );
   /* Open for write */
   if( (stream2 = fopen( "data2", "w+" )) == NULL )
      printf( "The file 'data2' was not opened\n" );
   else
      printf( "The file 'data2' was opened\n" );
   /* Close stream */
   if( fclose( stream ) )
      printf( "The file 'data' was not closed\n" );
   /* All other files are closed: */
   numclosed = _fcloseall( );
   printf( "Number of files closed by _fcloseall: %u\n", numclosed );
}
```

_fcvt

#include <stdlib.h> Required only for function declarations

Syntax char *_fcvt(double *value*, int *count*, int **dec*, int **sign*);

Parameter	Description
value	Number to be converted
count	Number of digits after decimal point
dec	Pointer to stored decimal-point position
sign	Pointer to stored sign indicator

The _fcvt function converts a floating-point number to a null-terminated character string. The *value* parameter is the floating-point number to be converted. The _fcvt function stores the digits of *value* as a string and appends a null character ('\0'). The *count* parameter specifies the number of digits to be stored after the decimal point. Excess digits are rounded off to *count* places. If there are fewer than *count* digits of precision, the string is padded with zeros.

Only digits are stored in the string. The position of the decimal point and the sign of *value* can be obtained from *dec* and *sign* after the call. The *dec* parameter points to an integer value; this integer value gives the position of the decimal point with respect to the beginning of the string. A zero or negative integer value indicates that the decimal point lies to the left of the first digit. The parameter *sign* points to an integer indicating the sign of *value*. The integer is set to 0 if *value* is positive and is set to a nonzero number if *value* is negative.

The _ecvt and _fcvt functions use a single statically allocated buffer for the conversion. Each call to one of these routines destroys the results of the previous call.

Return Value

The _fcvt function returns a pointer to the string of digits. There is no error return.

Use _fcvt for compatibility with ANSI naming conventions of non-ANSI functions. Use fcvt and link with OLDNAMES.LIB for UNIX compatibility.

atof
atoi
atol
_ecvt
_gcvt

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FCVT.C: This program converts the constant
 * 3.1415926535 to a string and sets the pointer
 * *buffer to point to that string.
 */
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   int  decimal, sign;
   char *buffer;
   double source = 3.1415926535;
   buffer = _fcvt( source, 7, &decimal, &sign );
   printf( "source: %2.10f   buffer: '%s'   decimal: %d   sign: %d\n",
            source, buffer, decimal, sign );
}
```

_fdopen

#include <stdio.h>

Syntax FILE *_fdopen(int *handle*, const char **mode*);

Parameter	**Description**
handle	Handle referring to open file
mode	Type of access permitted

The _fdopen function associates an input/output stream with the file identified by *handle*, thus allowing a file opened for low-level I/O to be buffered and formatted. The *mode* character string specifies the type of access requested for the file, as shown below. The following list gives the *mode* string used in the fopen and _fdopen functions and the corresponding *oflag* arguments used in the _open and _sopen functions. A complete description of the *mode* string argument is given in the fopen function.

"r"
  _O_RDONLY
"w"
  _O_WRONLY (usually _O_WRONLY | _O_CREAT | _O_TRUNC)
"a"
  _O_WRONLY | _O_APPEND (usually _O_WRONLY | _O_CREAT | _O_APPEND)
"r+"
  _O_RDWR
"w+"
  _O_RDWR (usually _O_RDWR | _O_CREAT | _O_TRUNC)
"a+"
  _O_RDWR | _O_APPEND (usually _O_RDWR | _O_APPEND | _O_CREAT)

In addition to the values listed above, one of the following characters can be included in the *mode* string to specify the translation mode for new lines. These characters correspond to the constants used in the _open and _sopen functions, as shown below:

t
  _O_TEXT
b
  _O_BINARY

If t or b is not given in the *mode* string, the translation mode is defined by the default-mode variable _fmode.

In addition to the file attribute and the text or binary mode listed above, the *mode* string accepts either c or n to specify commit to disk, or do not commit to disk, respectively. These characters have no correspondence to constants used in the _open and _sopen functions.

c
  Commit to disk, no _open/_sopen equivalent.
n
  No commit, no _open/_sopen equivalent. Default.

If c or n is not given in the *mode* string, n is the default mode.

Return Value

The _fdopen function returns a pointer to the open stream. A null pointer value indicates an error.

Use _fdopen for compatibility with ANSI naming conventions of non-ANSI functions. Use fdopen and link with OLDNAMES.LIB for UNIX compatibility.

The t, c, and n options are not part of the ANSI standard for fopen and _fdopen but are instead Microsoft extensions and should not be used where ANSI portability is desired.

_dup
_dup2
fclose
_fcloseall
fopen
freopen
_open

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* _FDOPEN.C: This program opens a file using low-
 * level I/O, then uses _fdopen to switch to stream
 * access. It counts the lines in the file.
 */
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
void main( void )
{
   FILE *stream;
   int  fh, count = 0;
   char inbuf[128];
   /* Open a file handle. */
   if( (fh = _open( "_fdopen.c", _O_RDONLY )) == -1 )
      exit( 1 );
   /* Change handle access to stream access. */
   if( (stream = _fdopen( fh, "r" )) == NULL )
      exit( 1 );
   while( fgets( inbuf, 128, stream ) != NULL )
      count++;
   /* After _fdopen, close with fclose, not _close. */
   fclose( stream );
   printf( "Lines in file: %d\n", count );
}
```

// Example: ALARM.C

```
/* ALARM.C illustrates inline assembly and functions or keywords
 * related to terminate-and-stay-resident programs. Functions include:
 *      _dos_setvect    _dos_getvect    _dos_keep           (MS-DOS-only)
 *      _enable         _disable        _chain_intr
 *
 * Keywords:
 *      __interrupt
 * Directive:
 *      #pragma
 * Pragmas:
 *      check_stack     check_pointer   intrinsic
 * Global variables:
 *      _psp
 *
 * WARNING: You should run ALARM from the MS-DOS command line.
 *
 * See MOVEMEM.C for another pragma example.
 */

#include <dos.h>
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* Stack and pointer checking off */
#pragma check_stack( off )
#pragma check_pointer( off )
#pragma intrinsic( _enable, _disable )

/* In-line assembler macro to sound a bell. Note that comments in macros must
 * be in the C format, not the assembler format.
 */
#define BEEP() __asm { \
   __asm   sub  bx, bx    /* Page 0      */ \
   __asm   mov  ax, 0E07h /* TTY bell    */ \
   __asm   int  10h       /* BIOS 10     */ \
   }

#define TICKPERMIN  1092L
#define MINPERHOUR  60L
enum BOOLEAN { FALSE, TRUE };

/* Prototypes for interrupt functions */
void (__interrupt __far *oldtimer)();
void (__interrupt __far *oldvideo)();
void __interrupt __far newtimer();
void __interrupt __far newvideo();

/* Variables that will be accessed inside TSR must be global. */
int  ftimesup = FALSE, finvideo = FALSE;
long goaltick;
long __far *pcurtick = (long __far *)0x0000046cL;

/* Huge pointers force compiler to do segment arithmetic for us. */
char __huge *tsrstack;
char __huge *appstack;
char __huge *tsrbottom;

void main( int argc, char **argv )
```

```
{
    long minute, hour;
    unsigned tsrsize;

    /* Initialize stack and bottom of program. */
    __asm mov  WORD PTR tsrstack[0], sp
    __asm mov  WORD PTR tsrstack[2], ss
    _FP_SEG( tsrbottom ) = _psp;
    _FP_OFF( tsrbottom ) = 0;

    /* Program size is:
     *     top of stack
     *   - bottom of program (converted to paragraphs)
     *   + one extra paragraph
     */
    tsrsize = ((tsrstack - tsrbottom) >> 4) + 1;

    /* If command line not given, show syntax and quit. */
    if( argc < 2 )
    {
       puts( "  Syntax: ALARM <hhmm> " );
       puts( "     where <hhmm> is time (24-hour format) to ring alarm" );
       exit( 1 );
    }

    /* Convert time to ticks past midnight. Time must include 0 in first
     * position (0930, not 930). Time must be later than current time.
     */
    minute = atol( argv[1] + 2 );
    argv[1][2] = 0;
    hour = atol( argv[1] );
    goaltick = (hour * MINPERHOUR * TICKPERMIN) + (minute * TICKPERMIN);
    if( *pcurtick > goaltick )
    {
       puts( "It's past that time now" );
       exit( 1 );
    }

    /* Replace existing timer and video routines with ours. */
    oldtimer = _dos_getvect( (unsigned)0x1c );
    _dos_setvect( 0x1c, newtimer );
    oldvideo = _dos_getvect( 0x10 );
    _dos_setvect( 0x10, newvideo );

    /* Free the PSP segment and terminate with program resident. */
    _dos_freemem( _psp );
    _dos_keep( 0, tsrsize );
}

/* Our timer interrupt compares current time to goal. If earlier,
 * it just continues. If later, it beeps and sets a flag to quit checking.
 */
void __interrupt __far newtimer()
{
    if( ftimesup )
       _chain_intr( oldtimer );
    else
    {
       /* First, execute the original timer interrupt. */
       (*oldtimer)();
```

```
        /* Activate if two conditions are met: First, it's past time for
         * the alarm. Second, we are not in a video interrupt. Checking
         * the video interrupt prevents the rare but potentially
         * dangerous case of calling INT 10 to beep while INT 10 is
         * already running.
         */
        if( (*pcurtick > goaltick) && !finvideo )
        {
            /* Set flag so we'll never return. */
            ftimesup = TRUE;

            /* Save current stack of application, and set old stack of TSR.
             * This is for safety since we don't know the state of the
             * application stack, but we do know the state of our own stack.
             * Turn off interrupts during the stack switch.
             */
            _disable();
            __asm
            {
               mov  WORD PTR appstack[0], sp   ; Save current stack
               mov  WORD PTR appstack[2], ss
               mov  sp, WORD PTR tsrstack[0]   ; Load new stack
               mov  ss, WORD PTR tsrstack[2]
            }
            _enable();

            /* Restore application stack. */
            _disable();
            __asm
            {
               mov  sp, WORD PTR appstack[0]
               mov  ss, WORD PTR appstack[2]
            }
            _enable();
        }
    }
}

/* Protects against reentering INT 10 while it is already executing.
 * Although rare, this could be disastrous if the interrupt routine was
 * interrupted while it was accessing a hardware register.
 */
void __interrupt __far newvideo( unsigned _es, unsigned _ds, unsigned _di,
   unsigned _si, unsigned _bp, unsigned _sp,
   unsigned _bx, unsigned _dx, unsigned _cx,
   unsigned _ax, unsigned _ip, unsigned _cs,
   unsigned _flags )
{
    static unsigned save_bp;

    /* If not already inside interrupt, chain to original. */
    if( !finvideo )
       _chain_intr( oldvideo );
    else
    {

       /* Set the inside flag, then make sure all the real registers
        * that might be passed to an interrupt 10h match the parameter
        * registers. Some of the real registers may be modified by the
```

```
         * preceding code. Note that BP must be saved in a static (nonstack)
         * variable so that it can be retrieved without modifying the stack.
         */
        ++finvideo;
        __asm
        {
            mov ax, _ax
            mov bx, _bx
            mov cx, _cx
            mov dx, _dx
            mov es, _es
            mov di, _di
            mov save_bp, bp
            mov bp, _bp
        }

        /* Call the original interrupt. */
        (*oldvideo)();

        /* Make sure that any values returned in real registers by the
         * interrupt are updated in the parameter registers. Reset the flag.
         */
        __asm
        {
            mov bp, save_bp
            mov _bp, bp
            mov _di, di
            mov _es, es
            mov _dx, dx
            mov _cx, cx
            mov _bx, bx
            mov _ax, ax
        }
        --finvideo;
    }
}
```

// Example: ANALYZE.C

```
/* ANALYZE.C illustrates presentation graphics analyze functions.
 * The example uses function:                        (MS-DOS-only)
 *      _pg_analyzechartms
 *
 * The same principles apply for:
 *      _pg_analyzepie        _pg_analyzechart
 *      _pg_analyzescatter    _pg_analyzescatterms
 */

#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include <graph.h>
#include <pgchart.h>

#define FALSE  0
#define TRUE   1

/* Note that data are declared as a single-dimension array. The multiseries
 * chart functions expect only one dimension. See the _pg_chartms example
 * for an alternate method using multidimension array.
 */
#define TEAMS  4
#define MONTHS 3
float __far values[TEAMS * MONTHS] = { .435F,   .522F,   .671F,
                                       .533F,   .431F,   .590F,
                                       .723F,   .624F,   .488F,
                                       .329F,   .226F,   .401F };
char __far *months[MONTHS] =         { "May",   "June",  "July" };
char __far *teams[TEAMS] = { "Reds", "Sox", "Cubs", "Mets" };

void main()
{
    _chartenv env;

    if( !_setvideomode( _MAXRESMODE ) )
        exit( 1 );

    _pg_initchart();                    /* Initialize chart system     */

    /* Default multiseries bar chart */
    _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Little League Records - Default" );
    _pg_chartms( &env, months, values, TEAMS, MONTHS, MONTHS, teams );
    _getch();
    _clearscreen( _GCLEARSCREEN );

    /* Analyze multiseries bar chart with autoscale. This sets all
     * default scale values. We want some x values to be automatic,
     * including scalemin and scalefactor.
     */
    _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Little League Records - Customized" );
    env.xaxis.autoscale = TRUE;

    _pg_analyzechartms( &env, months, values,
                        TEAMS, MONTHS, MONTHS, teams );

    /* Now customize some of the x axis values. Then draw the chart. */
    env.xaxis.autoscale = FALSE;
```

```
    env.xaxis.scalemax = 1.0F;          /* Make scale show 0.0 to 1.0   */
    env.xaxis.ticinterval = 0.2F;       /* Don't make scale too crowded */
    env.xaxis.ticdecimals = 3;          /* Show three decimals          */
    strcpy( env.xaxis.scaletitle.title, "Win/Loss Percentage" );
    _pg_chartms( &env, months, values, TEAMS, MONTHS, MONTHS, teams );
    _getch();

    _setvideomode( _DEFAULTMODE );
}
```

// Example: ANIMATE.C

```
/* ANIMATE.C illustrates animation functions including:
 *      _getimage       _putimage         (MS-DOS-only)
 *      _imagesize      _grstatus
 */

#include <conio.h>
#include <stddef.h>
#include <stdlib.h>
#include <malloc.h>
#include <graph.h>

short action[5]  = { _GPSET,   _GPRESET, _GXOR,    _GOR,     _GAND     };
char *descrip[5] = {  "PSET  ", "PRESET", "XOR   ", "OR    ", "AND   " };

void exitfree( char __huge *buffer );

void main()
{
   char  __huge *buffer;
   long  imsize;
   short i, x, y = 30;

   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );

   /* Measure the image to be drawn and allocate memory for it. */
   imsize = _imagesize( -16, -16, +16, +16 );
   buffer = _halloc( imsize, 1 );
   if( buffer == NULL )
      exit( 1 );

   _setcolor( 3 );
   for ( i = 0; i < 5; i++ )
   {
      /* Draw ellipse at new position and get a copy of it. */
      x = 50; y += 40;
      _ellipse( _GFILLINTERIOR, x - 15, y - 15, x + 15, y + 15 );
      _getimage( x - 16, y - 16, x + 16, y + 16, buffer );
      if( _grstatus() )
         exitfree( buffer );        /* Quit on error                   */

      /* Display action type and copy a row of ellipses with that type. */
      _settextposition( 1, 1 );
      _outtext( descrip[i] );
      while( x < 260 )
      {
         x += 5;
         _putimage( x - 16, y - 16, buffer, action[i] );
         if( _grstatus() < 0 )      /* Ignore warnings, quit on errors */
            exitfree( buffer );
      }
      _getch();
   }
   exitfree( buffer );
}

void exitfree( char __huge *buffer )
{
   _hfree( buffer );
   exit( !_setvideomode( _DEFAULTMODE ) );
```

```
}
```

// Example: ARGS.C

```
/* ARGS.C illustrates the following variables used for accessing
 * command-line arguments and environment variables:
 *      argc            argv            envp
 */

#include <stdio.h>
#include <process.h>

void main( int argc,        /* Number of strings in array argv          */
           char *argv[],    /* Array of command-line argument strings   */
           char **envp )    /* Array of environment variable strings    */
{
    int count;

    /* Display each command-line argument. */
    printf( "\nCommand-line arguments:\n" );
    for( count = 0; count < argc; count++ )
        printf( "  argv[%d]   %s\n", count, argv[count] );

    /* Display each environment variable. */
    printf( "\nEnvironment variables:\n" );
    while( *envp != NULL )
        printf( "  %s\n", *(envp++) );

    return;
}
```

// Example: ASCII.C

```
/* ASCII.C illustrates function:
 *     _outmem
 */

#include <stdio.h>
#include <graph.h>

void main()
{
   short i, len;
   char tmp[10];

   _clearscreen( _GCLEARSCREEN );
   for( i = 0; i < 256; i++ )
   {
      _settextposition( (short)((i % 24) + 1), (short)((i / 24) * 7) );
      len = (short)sprintf( tmp, "%3d %c", i, i );
      _outmem( tmp, len );
   }
   _settextposition( 24, 1 );
}
```

// Example: ASSERT.C

```
/* ASSERT.C illustrates:
 *      assert
 */

#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <assert.h>

#define MAXSTR 120

void chkstr( char *string );    /* Prototype */

void main()
{
   char string1[MAXSTR], string2[MAXSTR];

   /* Do various processes on strings and check the results. If
    * none cause errors, force an error with an empty string.
    */
   printf( "Enter a string: " );
   gets( string1 );
   chkstr( string1 );

   printf( "Enter another string: " );
   gets( string2 );
   chkstr( string2 );

   strcat( string1, string2 );
   chkstr( string1 );
   printf( "string1 + string2 = %s\n", string1 );

   chkstr( "" );
   printf( "You'll never get here\n" );
}

/* Tests a string to see if it is NULL, empty, or longer than MAXSTR. */
void chkstr( char *string )
{
   assert( string != NULL );               /* Cannot be NULL           */
   assert( *string != '\0' );              /* Cannot be empty          */
   assert( strlen( string ) < MAXSTR );    /* Length less than maximum */
}
```

// Example: ATEXIT.C

```
/* ATEXIT.C illustrates function:
 *      atexit          _onexit
 */

#include <stdlib.h>
#include <stdio.h>
#define ANSI                /* Comment out to try _onexit     */

/* Prototypes */
void fn1( void ), fn2( void ), fn3( void ), fn4( void );

void main()
{

   /* atexit is the ANSI standard. It returns 0 for success, nonzero
    * for fail.
    */
#if defined( ANSI )
   atexit( fn1 );
   atexit( fn2 );
   atexit( fn3 );
   atexit( fn4 );

   /* _onexit is a Microsoft extension. It returns a pointer to a function
    * for success, NULL for fail.
    */
#else
   _onexit( fn1 );
   _onexit( fn2 );
   _onexit( fn3 );
   _onexit( fn4 );
#endif

   printf( "This is executed first.\n" );
}

void fn1()
{
   printf( "next.\n" );
}

void fn2()
{
   printf( "executed " );
}

void fn3()
{
   printf( "is " );
}

void fn4()
{
   printf( "This " );
}
```

// Example: ATONUM.C

```
/* ATONUM.C illustrates string-to-number conversion functions including:
 *      atof            atoi            atol            _gcvt
 *
 * It also illustrates:
 *      _cgets          _cputs
 */

#include <stdlib.h>
#include <string.h>
#include <conio.h>

#define MAXSTR 100

char cnumbuf[MAXSTR] = { MAXSTR + 2, 0 };
char tmpbuf[MAXSTR];

/* Numeric input and output without printf */
void main()
{
    int     integer;
    long    longint;
    float   real;
    char    *numbuf;

    /* Using _cgets (rather than gets) allows use of MS-DOS editing keys
     * (or of editing keys from MS-DOS command-line editors).
     */
    _cputs( "Enter an integer: " );
    numbuf = _cgets( cnumbuf );
    _cputs( "\r\n" );                /* _cputs doesn't translate \n     */
    integer = atoi( numbuf );
    strcpy( tmpbuf, numbuf );
    strcat( tmpbuf, " + " );

    _cputs( "Enter a long integer: " );
    numbuf = _cgets( cnumbuf );
    _cputs( "\r\n" );
    longint = atol( numbuf );
    strcat( tmpbuf, numbuf );
    strcat( tmpbuf, " + " );

    _cputs( "Enter a floating-point number: " );
    numbuf = _cgets( cnumbuf );
    _cputs( "\r\n" );
    real = (float)atof( numbuf );
    strcat( tmpbuf, numbuf );
    strcat( tmpbuf, " = " );

    _gcvt( integer + longint + real, 4, numbuf );
    strcat( tmpbuf, numbuf );
    strcat( tmpbuf, "\r\n" );

    _cputs( tmpbuf );
}
```

// Example: BARCOL.C

```
/* BARCOL.C illustrates presentation graphics support routines and
 * single-series chart routines including:           (MS-DOS-only)
 *    _pg_initchart   _pg_chart   _pg_chartpie   _pg_defaultchart
 */

#include <conio.h>
#include <stdlib.h>
#include <graph.h>
#include <string.h>
#include <pgchart.h>

#define COUNTRIES 5
float __far value[COUNTRIES] =     { 42.5F,  14.3F, 35.2F,  21.3F, 32.6F   };
char  __far *category[COUNTRIES] = { "CAN",  "JAP", "USA",  "UK",  "Other" };
short __far explode[COUNTRIES] =   { 0,      1,     0,      1,     0       };

void main()
{
    _chartenv env;

    if( !_setvideomode( _MAXRESMODE ) )
        exit( 1 );

    _pg_initchart();                    /* Initialize chart system    */

    /* Single-series bar chart */
    _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Widget Production" );
    _pg_chart( &env, category, value, COUNTRIES );
    _getch();
    _clearscreen( _GCLEARSCREEN );

    /* Single-series column chart */
    _pg_defaultchart( &env, _PG_COLUMNCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Widget Production" );
    _pg_chart( &env, category, value, COUNTRIES );
    _getch();
    _clearscreen( _GCLEARSCREEN );

    /* Pie chart */
    _pg_defaultchart( &env, _PG_PIECHART, _PG_PERCENT );
    strcpy( env.maintitle.title, "Widget Production" );
    _pg_chartpie( &env, category, value, explode, COUNTRIES );
    _getch();

    _setvideomode( _DEFAULTMODE );
}
```

// Example: BEEP.C

```
/* BEEP.C illustrates timing and port input and output functions
 * including:
 *      _inp            _outp           clock
 *
 * Also keyword:
 *      enum
 *
 */

#include <time.h>
#include <conio.h>

/* Use intrinsic versions of _outp and _inp */
#pragma intrinsic( _outp, _inp )

/* Prototypes */
void Beep( int frequency, int duration );
void Sleep( clock_t wait );

enum NOTES      /* Enumeration of notes and frequencies     */
{
    C0 = 262, D0 = 296, E0 = 330, F0 = 349, G0 = 392, A0 = 440, B0 = 494,
    C1 = 523, D1 = 587, E1 = 659, F1 = 698, G1 = 784, A1 = 880, B1 = 988,
    EIGHTH = 125, QUARTER = 250, HALF = 500, WHOLE = 1000, END = 0
} song[] =      /* Array initialized to notes of song       */
{
    C1, HALF, G0, HALF, A0, HALF, E0, HALF, F0, HALF, E0, QUARTER,
    D0, QUARTER, C0, WHOLE, END
};

void main ()
{
    int note;

    for( note = 0; song[note]; note += 2 )
        Beep( song[note], song[note + 1] );
}

/* Sounds the speaker for a time specified in microseconds by duration
 * at a pitch specified in hertz by frequency.
 */
void Beep( int frequency, int duration )
{
    int control;

    /* If frequency is 0, Beep doesn't try to make a sound. It
     * just sleeps for the duration.
     */
    if( frequency )
    {
        /* 75 is about the shortest reliable duration of a sound. */
        if( duration < 75 )
            duration = 75;

        /* Prepare timer by sending 10111100 to port 43. */
        _outp( 0x43, 0xb6 );

        /* Divide input frequency by timer ticks per second and
         * write (byte by byte) to timer.
         */
```

```
        frequency = (unsigned)(1193180L / frequency);
        _outp( 0x42, (char)frequency );
        _outp( 0x42, (char)(frequency >> 8) );

        /* Save speaker control byte. */
        control = inp( 0x61 );

        /* Turn on the speaker (with bits 0 and 1). */
        _outp( 0x61, control | 0x3 );
    }

    Sleep( (clock_t)duration );

    /* Turn speaker back on if necessary. */
    if( frequency )
        _outp( 0x61, control );

}

/* Pauses for a specified number of microseconds. */
void Sleep( clock_t wait )
{
    clock_t goal;

    goal = wait + clock();
    while( goal > clock() )
        ;
}
```

// Example: BESSEL.C

```
/* BESSEL.C illustrates Bessel functions including:
 * _j0  _jn  _y0  _y1  _yn
 */

#include <math.h>
#include <stdio.h>

void main()
{
    double x = 2.387;
    int n = 3, c;

    printf( "Bessel functions for x = %f:\n", x );
    printf( "  Kind\t\tOrder\tFunction\tResult\n\n" );
    printf( "  First\t\t0\t_j0( x )\t%f\n", _j0( x ) );
    printf( "  First\t\t1\t_j1( x )\t%f\n", _j1( x ) );
    for( c = 2; c < 10; c++ )
        printf( "  First\t\t%d\t_jn( n, x )\t%f\n", c, _jn( c, x ) );

    printf( "  Second\t0\t_y0( x )\t%f\n", _y0( x ) );
    printf( "  Second\t1\t_y1( x )\t%f\n", _y1( x ) );
    for( c = 2; c < 10; c++ )
        printf( "  Second\t%d\t_yn( n, x )\t%f\n", c, _yn( c, x ) );
}
```

// Example: BUFTEST.C

```
/* BUFTEST.C illustrates buffer control for stream I/O, using function:
 *      setvbuf
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

long countln( FILE *stream );           /* Prototype        */
char buf[BUFSIZ * 4];                   /* File buffer      */

void main( int argc, char *argv[] )
{
    time_t start, end;
    FILE *stream;
    long c;

    /* Use standard buffer. */
    if( (stream = fopen( argv[1], "rt" )) == NULL )
    {
        printf( "SYNTAX: BUFTEST filename\n" );
        exit( 1 );
    }
    start = clock();
    c = countln( stream );
    end = clock();
    printf( "Time: %5.1f\tBuffering: Normal\tBuffer size: %d\n",
             ((float)end - start) / CLOCKS_PER_SEC, BUFSIZ );

    /* Use a larger buffer. */
    if( (stream = fopen( argv[1], "rt" )) == NULL )
        exit( 1 );
    setvbuf( stream, buf, _IOFBF, sizeof( buf ) );
    start = clock();
    c = countln( stream );
    end = clock();
    printf( "Time: %5.1f\tBuffering: Normal\tBuffer size: %d\n",
             ((float)end - start) / CLOCKS_PER_SEC, BUFSIZ * 4 );

    /* Try it with no buffering. */
    if( (stream = fopen( argv[1], "rt" )) == NULL )
        exit( 1 );
    setvbuf( stream, NULL, _IONBF, 0 );
    start = clock();
    c = countln( stream );
    end = clock();
    printf( "Time: %5.1f\tBuffering: None\tBuffer size: %d\n",
             ((float)end - start) / CLOCKS_PER_SEC, 0 );

    printf( "File %s has %ld lines\n", argv[1], c );
    exit( 0 );
}

/* Counts lines in a text file and closes file */
long countln( FILE *stream )
{
    char linebuf[256];
    long c = 0;

    while( !feof( stream ) )
```

```
    {
        if( fgets( linebuf, 255, stream ) == NULL )
            break;
        ++c;
    }
    fclose( stream );
    return c;
}
```

// Example: CABS.C

```
/* CABS.C illustrates functions:
 *      _cabs            _hypot
 */

#include <math.h>
#include <stdio.h>

void main()
{
    struct _complex ne = {  3.0,  4.0 }, se = { -3.0, -4.0 },
                    sw = { -3.0, -4.0 }, nw = { -3.0,  4.0 };

    printf( "Absolute %4.1lf + %4.1lfi:\t\t%4.1f\n",
            ne.x, ne.y, _cabs( ne ) );
    printf( "Absolute %4.1lf + %4.1lfi:\t\t%4.1f\n",
            sw.x, sw.y, _cabs( sw ) );

    printf( "Hypotenuse of %4.1lf and %4.1lf:\t%4.1f\n",
            se.x, se.y, _hypot( se.x, se.y ) );
    printf( "Hypotenuse of %4.1lf and %4.1lf:\t%4.1f\n",
            nw.x, nw.y, _hypot( nw.x, nw.y ) );
}
```

// Example: CASE.C

```
/* CASE.C illustrates case conversion and other conversions.
 * Functions illustrated include:
 *      _strupr          toupper         _toupper
 *      _strlwr          tolower         _tolower
 *      _strrev          __toascii
 */

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

char mstring[] = "Dog Saw Dad Live On";
char *ustring, *tstring, *estring;
char *p;

void main()
{
    printf( "Original:\t%s\n", mstring );

    /* Uppercase and lowercase */
    ustring = _strupr( _strdup( mstring ) );
    printf( "Uppercase:\t%s\n", ustring );

    printf( "Lowercase:\t%s\n", _strlwr( ustring ) );

    /* Reverse case of each character. */
    tstring = _strdup( mstring );
    for( p = tstring; *p; p++ )
    {
        if( isupper( *p ) )
            *p = tolower( *p );
        else
            *p = toupper( *p );

        /* This alternate code (commented out) shows how to use _tolower
         * and _toupper for the same purpose.
        if( isupper( *p ) )
            *p = _tolower( *p );
        else if( islower( *p ) )
            *p = _toupper( *p );
         */
    }
    printf( "Toggle case:\t%s\n", tstring );

    /* Encode and decode string. The decoding technique will convert
     * strings with some high bits set (produced by some word processors).
     */
    estring = _strdup( mstring );
    for( p = estring; *p; p++ )
        *p = *p | 0x80;
    printf( "Encoded:\t%s\n", estring );

    for( p = estring; *p; p++ )
        *p = __toascii( *p );
    printf( "Decoded:\t%s\n", estring );

    printf( "Reversed:\t%s\n", _strrev( ustring ) );
}
```

// Example: CGAPAL.C

```
/* CGAPAL.C illustrates CGA palettes using function:
 *   _selectpalette   (MS-DOS-only)
 */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <graph.h>

long bkcolor[8] = { _BLACK,  _BLUE,     _GREEN,  _CYAN,
                    _RED,    _MAGENTA,  _BROWN,  _WHITE };
char *bkname [] = { "BLACK", "BLUE",    "GREEN", "CYAN",
                    "RED",   "MAGENTA", "BROWN", "WHITE" };
void main()
{
    int i, j, k;

    if ( !_setvideomode( _MRES4COLOR ) )
        exit( 1 );
    for( i = 0; i < 4; i++ )                    /* Palette loop          */
    {
        _selectpalette( i );
        for( k = 0; k < 8; k++ )                /* Background color loop */
        {
            _clearscreen( _GCLEARSCREEN );
            _setbkcolor( bkcolor[k] );
            _settextposition( 1, 1 );
            printf( "Background: %s\tPalette: %d", bkname[k], i );
            for( j = 1; j < 4; j++ )            /* Foreground color loop */
            {
                _setcolor( j );
                _ellipse( _GFILLINTERIOR, 100, j * 30, 220, 80 + (j * 30) );
            }
            _getch();
        }
    }
    _setvideomode( _DEFAULTMODE );
}
```

// Example: CHMOD1.C

```
/* CHMOD1.C illustrates reading and changing file attribute and time
 * using functions:
 *      _access          _chmod           _utime
 *
 * See CHMOD2.C for a more powerful variation of this program using
 * _dos_ functions.
 */

#include <io.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <sys\utime.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

enum FILEATTRIB { EXIST, WRITE = 2, READ = 4, READWRITE = 6 };

/* Macro uses access */
#define EXIST( name ) !_access( name, EXIST )

void main( int argc, char *argv[] )
{
    if( !EXIST( argv[1] ) )
    {
        printf( "Syntax:  CHMOD1 <filename>" );
        exit( 1 );
    }

    if( !_access( argv[1], WRITE ) )
    {
        printf( "File %s is read/write. Change to read only? ", argv[1] );

        /* NOTE: Use stdlib.h for function definition of toupper rather
         * than macro version in ctype.h. Macro side effects would cause
         * macro version to read two keys.
         */
        if( toupper( _getche() ) == 'Y' )
            _chmod( argv[1], _S_IREAD );
    }
    else
    {
        printf( "File %s is read only. Change to read/write? ", argv[1] );
        if( toupper( _getch() ) == 'Y' )
            _chmod( argv[1], _S_IREAD | _S_IWRITE );
    }

    printf( "\nUpdate file time to current time? " );
    if( toupper( _getche() ) == 'Y' )
        _utime( argv[1], NULL );
    exit( 0 );
}
```

// Example: CHMOD2.C

```
/* CHMOD2.C illustrates reading and changing file attributes and file
 * time using functions:
 *   _dos_getftime       _dos_setftime                  (MS-DOS-only)
 *   _dos_getfileattr    _dos_setfileattr
 *
 * See CHMOD1.C for a simpler variation of this program using the _utime,
 * _access, and _chmod functions.
 */

#include <dos.h>
#include <fcntl.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <sys\types.h>
#include <sys\stat.h>

char *datestr( unsigned d, char *buf );     /* Prototypes */
char *timestr( unsigned t, char *buf );

void main( int argc, char *argv[] )
{
    unsigned fdate, ftime, fattr;
    struct _dosdate_t ddate;
    struct _dostime_t dtime;
    int hsource;
    char timebuf[10], datebuf[10], *pkind;

    /* Open to get handle and test for errors (such as nonexistence). */
    if( _dos_open( argv[1], _O_RDONLY, &hsource ) )
    {
        printf( "Can't open\n" );
        exit( 1 );
    }

    /* Get time, date, and attribute of file. */
    _dos_getftime( hsource, &fdate, &ftime );
    _dos_getfileattr( argv[1], &fattr );

    /* Convert information into formatted strings. */
    datestr( fdate, datebuf );
    timestr( ftime, timebuf );

    printf( "%-12s   %-8s   %-9s   %-9s    %s %s %s\n",
            "FILE", "TIME", "DATE", "KIND", "RDO", "HID", "SYS" );
    printf( "%-12s   %8s   %8s    %-9s     %c   %c   %c\n",
            argv[1], timebuf, datebuf, pkind,
            (fattr & _A_RDONLY) ? 'Y' : 'N',
            (fattr & _A_HIDDEN) ? 'Y' : 'N',
            (fattr & _A_SYSTEM) ? 'Y' : 'N' );

    /* Update file time or attribute. */
    printf( "Change: (T)ime  (R)ead only  (H)idden  (S)ystem\n" );
    switch( toupper( _getch() ) )       /* Use stdlib.h, not ctype.h */
    {
    case 'T':                       /* Set to current time */
        _dos_gettime( &dtime );
        _dos_getdate( &ddate );
        ftime = (dtime.hour << 11) | (dtime.minute << 5);
        fdate = ((ddate.year - 1980) << 9) | (ddate.month << 5) |
```

```
                ddate.day;
        _dos_setftime( hsource, fdate, ftime );
        break;
    case 'R':               /* Toggle read only */
        _dos_setfileattr( argv[1], fattr ^ _A_RDONLY );
        break;
    case 'H':               /* Toggle hidden    */
        _dos_setfileattr( argv[1], fattr ^ _A_HIDDEN );
        break;
    case 'S':               /* Toggle system    */
        _dos_setfileattr( argv[1], fattr ^ _A_SYSTEM );
        break;
    }
    _dos_close( hsource );
    exit( 1 );
}

/* Takes unsigned time in the format:              fedcba9876543210
 * s=2 sec incr, m=0-59, h=23                      hhhhhmmmmmmssssss
 * Changes to a 9-byte string (ignore seconds):    hh:mm ?m
 */
char *timestr( unsigned t, char *buf )
{
    int h = (t >> 11) & 0x1f, m = (t >> 5) & 0x3f;

    sprintf( buf, "%2.2d:%02.2d %cm", h % 12, m,  h > 11 ? 'p' : 'a' );
    return buf;
}

/* Takes unsigned date in the format:              fedcba9876543210
 * d=1-31, m=1-12, y=0-119 (1980-2099)             yyyyyyymmmmddddd
 * Changes to a 9-byte string:                     mm/dd/yy
 */
char *datestr( unsigned d, char *buf )
{
    sprintf( buf, "%2.2d/%02.2d/%02.2d",
             (d >> 5) & 0x0f, d & 0x1f, (d >> 9) + 80 );
    return buf;
}
```

// Example: CMPSTR.C

```
/* CMPSTR.C illustrates string and memory comparison functions including:
 *       memcmp        _memicmp
 *       strncmp       _strnicmp
 *       strcmp        _stricmp          _strcmpi
 */

#include <memory.h>
#include <string.h>
#include <stdio.h>

char string1[] = "The quick brown dog jumps over the lazy fox";
char string2[] = "The QUICK brown fox jumps over the lazy dog";

void main()
{
    char tmp[20];
    int result;

    printf( "Compare strings:\n\t\t%s\n\t\t%s\n\n", string1, string2 );

    printf( "Function:\tmemcmp\n" );
    result = memcmp( string1, string2 , 42 );
    if( result > 0 )
        strcpy( tmp, "greater than" );
    else if( result < 0 )
        strcpy( tmp, "less than" );
    else
        strcpy( tmp, "equal to" );
    printf( "Result:\t\tString 1 is %s string 2\n\n", tmp );

    printf( "Function:\t_memicmp\n" );
    result = _memicmp( string1, string2, 42 );
    if( result > 0 )
        strcpy( tmp, "greater than" );
    else if( result < 0 )
        strcpy( tmp, "less than" );
    else
        strcpy( tmp, "equal to" );
    printf( "Result:\t\tString 1 is %s string 2\n\n", tmp );

    printf( "Function:\tstrncmp\n" );
    result = strncmp( string1, string2 , 42 );
    if( result > 0 )
        strcpy( tmp, "greater than" );
    else if( result < 0 )
        strcpy( tmp, "less than" );
    else
        strcpy( tmp, "equal to" );
    printf( "Result:\t\tString 1 is %s string 2\n\n", tmp );

    printf( "Function:\t_strnicmp\n" );
    result = _strnicmp( string1, string2, 42 );
    if( result > 0 )
        strcpy( tmp, "greater than" );
    else if( result < 0 )
        strcpy( tmp, "less than" );
    else
        strcpy( tmp, "equal to" );
    printf( "Result:\t\tString 1 is %s string 2\n\n", tmp );
```

```
    printf( "Function:\tstrcmp\n" );
    result = strcmp( string1, string2 );
    if( result > 0 )
        strcpy( tmp, "greater than" );
    else if( result < 0 )
        strcpy( tmp, "less than" );
    else
        strcpy( tmp, "equal to" );
    printf( "Result:\t\tString 1 is %s string 2\n\n", tmp );

    printf( "Function:\t_stricmp or _strcmpi\n" );
    result = _stricmp( string1, string2 );
    /* _strcmpi (commented out) is the same as _stricmp.
    result = _strcmpi( string1, string2 );
     */
    if( result > 0 )
        strcpy( tmp, "greater than" );
    else if( result < 0 )
        strcpy( tmp, "less than" );
    else
        strcpy( tmp, "equal to" );
    printf( "Result:\t\tString 1 is %s string 2\n", tmp );
}
```

// Example: COM.C

```
/* COM.C illustrates serial port access using function:
 *      _bios_serialcom                     (MS-DOS-only)
 */

#include <bios.h>
#include <stdio.h>

void main()
{
    unsigned status, port;

    for( port = 0; port < 3; port++ )
    {
        status = _bios_serialcom( _COM_STATUS, port, 0 );

        /* Report status of each serial port and test whether there is a
         * responding device (such as a modem) for each. If data-set-ready
         * and clear-to-send bits are set, a device is responding.
         */
        printf( "COM%c status: %.4X\tActive: %s\n",
                (char)port + '1', status,
                (status & 0x0030) ? "YES" : "NO" );
    }
}
```

// Example: COMMIT.C

```
/* COMMIT.C illustrates low-level file I/O functions including:
 *
 *     _close    _commit    memset    _open    _write
 *
 * This is example code; to keep the code simple and readable
 * return values are not checked.
 */


#include <io.h>
#include <stdio.h>
#include <fcntl.h>
#include <memory.h>

#define MAXBUF 32

int log_receivable( int );

void main( void )
{
    int fhandle;
    fhandle = _open( "TRANSACT.LOG", _O_APPEND | _O_CREAT |
                                     _O_BINARY | _O_RDWR );
    log_receivable( fhandle );
    _close( fhandle );
}

int log_receivable( int fhandle )
{
/* The log_receivable function prompts for a name and a monetary amount
 * and places both values into a buffer (buf). The _write function
 * writes the values to the operating system and the _commit function
 * ensures that they are written to a disk file.
 */

    int i;
    char  buf[MAXBUF];

    memset( buf, '\0', MAXBUF );
    /* Begin Transaction. */
    printf( "Enter name: " );
    gets( buf );
    for( i = 1; buf[i] != '\0'; i++ );
    /* Write the value as a '\0' terminated string. */
    _write( fhandle, buf, i+1 );
    printf( "\n" );

    memset( buf, '\0', MAXBUF );
    printf( "Enter amount: $" );
    gets( buf );
    for( i = 1; buf[i] != '\0'; i++ );
    /* Write the value as a '\0' terminated string. */
    _write( fhandle, buf, i+1 );
    printf( "\n" );

    /* The _commit function ensures that two important pieces of data are
     * safely written to disk. The return value of the _commit function
     * is returned to the calling function.
     */
    return _commit( fhandle );
```

```
}
```

// Example: COPROC.C

```
/* COPROC.C illustrates use of the status and control words of a floating-
 * point coprocessor (or emulator). Functions illustrated include:
 *      _clear87            _status87           _control87
 */

#include <stdio.h>
#include <conio.h>
#include <float.h>
#include <stdlib.h>
#include <string.h>

double dx = 1e-40, dy;
float fx, fy;
unsigned status, control;
char tmpstr[20];
char *binstr( int num, char *buffer );

void main()
{
    printf( "Status Word Key:\n" );
    printf( "B\tBusy flag\n0-3\tCondition codes\nS\tStack top pointer\n" );
    printf( "E\tError summary\nF\tStack flag\nP\tPrecision exception\n" );
    printf( "U\tUnderflow exception\nO\tOverflow exception\n" );
    printf( "Z\tZero divide exception\nD\tDenormalized exception\n" );
    printf( "I\tInvalid operation exception\n\n" );

    binstr(  _clear87(), tmpstr );
    printf( "B3SSS210EFPUOZDI  Function\tCondition\n\n" );
    printf( "%16s  _clear87\tAfter clearing\n", tmpstr );

    /* Storing double to float that hasn't enough precision for it
     * causes underflow and precision exceptions.
     */
    fx = (float)dx;
    binstr(  _status87(), tmpstr );
    printf( "%16s  _status87\tAfter moving double to float\n", tmpstr );

    /* Storing float with lost precision back to double adds denormalized
     * exception (previous exceptions remain).
     */
    dy = fx;
    binstr(  _clear87(), tmpstr );
    printf( "%16s  _clear87\tAfter moving float to double\n", tmpstr );

    /* Using _clear87() erases previous exceptions. */
    fy = (float)dy;
    binstr(  _status87(), tmpstr );
    printf( "%16s  _status87\tAfter moving double to float\n\n", tmpstr );

    _getch();
    printf( "Control Word Key:\n" );
    printf( "i\tInfinity control\nr\tRounding control\n" );
    printf( "p\tPrecision control\ne\tInterrupt enable mask\n" );
    printf( "U\tUnderflow mask\nO\tOverflow mask\n" );
    printf( "Z\tZero divide mask\nD\tDenormalized mask\n" );
    printf( "I\tInvalid operation mask\n\n" );
    printf( "???irrppe?PUOZDI  Result\n" );
    fy = .1f;

    /* Show current control word. */
```

```
    binstr( _control87( 0, 0 ), tmpstr );
    printf( "%16s  %.1f * %.1f = %.15e with initial precision\n",
            tmpstr, fy, fy, fy * fy );

    /* Set precision to 24 bits. */
    binstr( _control87( _PC_24, _MCW_PC ), tmpstr );
    printf( "%16s  %.1f * %.1f = %.15e with 24-bit precision\n",
            tmpstr, fy, fy, fy * fy );

    /* Restore default. */
    binstr( _control87( _CW_DEFAULT, 0xffff ), tmpstr );
    printf( "%16s  %.1f * %.1f = %.15e with default precision\n",
            tmpstr, fy, fy, fy * fy );
}

/* Converts integer to string of 16 binary characters. */
char *binstr( int num, char *buffer )
{
    char tmp[17];
    int  len;

    memset( buffer, '0', 16 );
    len = strlen( _itoa( num, tmp, 2 ) );
    strcpy( buffer + 16 - len, tmp );
    return buffer;
}
```

// Example: COPY1.C

```
/* COPY1.C illustrates low-level file I/O and dynamic memory allocation
 * functions including:
 *      _open           _close
 *      _read           _write          _eof
 *      malloc          free            _memmax
 *
 * Also the global variable:
 *      errno
 *
 * See COPY2.C for another version of the copyfile function and
 * HEAPBASE.C for an example of based heap management.
 */

#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <fcntl.h>          /* _O_ constant definitions */
#include <sys\types.h>
#include <sys\stat.h>       /* _S_ constant definitions */
#include <malloc.h>
#include <errno.h>

int copyfile( char *source, char *destin );     /* Prototype */

void main( int argc, char *argv[] )
{
    if( argc == 3 )
        if( copyfile( argv[1], argv[2] ) )
            printf( "Copy failed\n" );
        else
            printf( "Copy successful\n" );
    else
        printf( "  SYNTAX: COPY1 <source> <target>\n" );
}

/* Copies one file to another (both specified by path). Dynamically
 * allocates memory for the file buffer. Prompts before overwriting
 * existing file. Returns 0 if successful, otherwise an error number.
 */
int copyfile( char *source, char *target )
{
    char *buf;
    int hsource, htarget, ch;
    unsigned count = 0xff00;

    /* Open source file and create target, overwriting if necessary. */
    if( (hsource = _open( source, _O_BINARY | _O_RDONLY )) == - 1 )
        return errno;
    htarget = _open( target, _O_BINARY | _O_WRONLY | _O_CREAT | _O_EXCL,
                             _S_IREAD | _S_IWRITE );
    if( errno == EEXIST )
    {
        _cputs( "Target exists. Overwrite? " );
        ch = _getch();
        if( (ch == 'y') || (ch == 'Y') )
            htarget = _open( target, _O_BINARY | _O_WRONLY | _O_CREAT |
                                     _O_TRUNC, _S_IREAD | _S_IWRITE );
        printf( "\n" );
    }
    if( htarget == -1 )
```

```
        return errno;

    if( (unsigned)_filelength( hsource ) < count )
        count = (int)_filelength( hsource );

    /* Dynamically allocate a large file buffer. If there's not enough
     * memory for it, find the largest amount available on the near heap
     * and allocate that. This can't fail, no matter what the memory model.
     */
    if( (buf = (char *)malloc( (size_t)count )) == NULL )
    {
        count = _memmax();
        if( (buf = (char *)malloc( (size_t)count )) == NULL )
            return ENOMEM;
    }

    /* Read-write until there's nothing left. */
    while( !_eof( hsource ) )
    {
        /* Read and write input. */
        if( (count = _read( hsource, buf, count )) == -1 )
            return errno;
        if( (count = _write( htarget, buf, count )) == - 1 )
            return errno;
    }

    /* Close files and release memory. */
    _close( hsource );
    _close( htarget );
    free( buf );
    return 0;
}
```

// Example: COPY2.C

```
/* COPY2.C illustrates MS-DOS file I/O and MS-DOS memory allocation functions
 * including:                                          (MS-DOS-only)
 *      _dos_open       _dos_close      _dos_creatnew   _dos_creat
 *      _dos_read       _dos_write      _dos_allocmem   _dos_free
 *
 * See COPY1.C for another version of the copyfile function.
 */

#include <dos.h>
#include <fcntl.h>
#include <conio.h>
#include <stdio.h>

int copyfile( char *source, char *destin );     /* Prototype */

void main( int argc, char *argv[] )
{
    if( argc == 3 )
        if( copyfile( argv[1], argv[2] ) )
            printf( "Copy failed\n" );
        else
            printf( "Copy successful\n" );
    else
        printf( "  SYNTAX: COPY2 <source> <target>\n" );
}

/* Copies one file to another (both specified by path). Dynamically
 * allocates memory for the file buffer. Prompts before overwriting
 * existing file. Returns 0 if successful, or an error number if
 * unsuccessful. This function uses _dos_ functions only; standard
 * C functions are not used.
 */
#define EXIST 80
enum ATTRIB { NORMAL, RDONLY, HIDDEN, SYSTEM = 4 };
int copyfile( char *source, char *target )
{
    char __far *buf = NULL;
    char prompt[] = "Target exists. Overwrite? ", newline[] = "\n\r";
    int hsource, htarget, ch;
    unsigned ret, segbuf, count;

    /* Attempt to dynamically allocate all of memory (0xffff paragraphs).
     * This will fail, but will return the amount actually available
     * in segbuf. Then allocate this amount.
     */
    _dos_allocmem( 0xffff, &segbuf );
    count = segbuf;
    if( ret = _dos_allocmem( count, &segbuf ) )
        return ret;
    _FP_SEG( buf ) = segbuf;

    /* Open source file and create target, overwriting if necessary. */
    if( ret = _dos_open( source, _O_RDONLY, &hsource ) )
        return ret;
    ret = _dos_creatnew( target, NORMAL, &htarget );
    if( ret == EXIST )
    {
        /* Use _dos_write to display prompts. Use _bdos to call
         * function 1 to get and echo keystroke.
         */
```

```
            _dos_write( 1, prompt, sizeof( prompt ) - 1, &ch );
            ch = _bdos( 1, 0, 0 ) & 0x00ff;
            if( (ch == 'y') || (ch == 'Y') )
                ret = _dos_creat( target, NORMAL, &htarget );
            _dos_write( 1, newline, sizeof( newline ) - 1, &ch );
        }
        if( ret )
            return ret;

        /* Read and write until there is nothing left. */
        while( count )
        {
            /* Read and write input. */
            if( (ret = _dos_read( hsource, buf, count, &count )) )
                return ret;
            if( (ret = _dos_write( htarget, buf, count, &count )) )
                return ret;
        }

        /* Close files and free memory. */
        _dos_close( hsource );
        _dos_close( htarget );
        _dos_freemem( segbuf );
        return 0;
    }
```

// Example: CPYSTR.C

```
/* CPYSTR.C illustrates memory and string copy functions including:
 *      _memccpy        memcpy          memmove
 *      strncpy         strcpy          _strdup          strlen
 */

#include <memory.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <dos.h>

char string1[60] = "The quick brown dog jumps over the lazy fox";
char string2[60] = "The quick brown fox jumps over the lazy dog";
/*                           1         2         3         4         5 *
 *                 12345678901234567890123456789012345678901234567890 */
void main()
{
    char buffer[61];
    char *pdest, *newstring;
    int  pos;

    printf( "Function:\t_memccpy 60 characters or to character 's'\n" );
    printf( "Source:\t\t%s\n", string1 );
    pdest = _memccpy( buffer, string1, 's', 60 );
    *pdest = '\0';
    printf( "Result:\t\t%s\n", buffer );
    printf( "Length:\t\t%d characters\n\n", strlen( buffer ) );

    pos = pdest - buffer;
    printf( "Function:\tstrcpy\n" );
    printf( "Source:\t\t%s\n", string2 + pos );
    pdest = strcpy( buffer + pos, string2 + pos );
    printf( "Result:\t\t%s\n", buffer );
    printf( "Length:\t\t%d characters\n\n", strlen( buffer ) );

    printf( "Function:\tmemcpy 20 characters\n" );
    printf( "Source:\t\t%s\n", string2 );
    memcpy( buffer, string2, 20 );
    printf( "Result:\t\t%s\n", buffer );
    printf( "Length:\t\t%d characters\n\n", strlen( buffer ) );

    printf( "Function:\tstrncpy 30 characters\n" );
    printf( "Source:\t\t%s\n", string1 + 20 );
    pdest = strncpy( buffer + 20, string1 + 20, 30 );
    printf( "Result:\t\t%s\n", buffer );
    printf( "Length:\t\t%d characters\n\n", strlen( buffer ) );

    _getch();

    printf( "Function:\t_strdup\n" );
    printf( "Source:\t\t%s\n", buffer );
    newstring = _strdup( buffer );
    printf( "Result:\t\t%s\n", newstring );
    printf( "Length:\t\t%d characters\n\n", strlen( newstring ) );

    /* Illustrate overlapping copy: memmove handles it correctly,
     * memcpy does not.
     */
    printf( "Function:\tmemcpy with overlap\n" );
    printf( "Source:\t\t%s\n", string1 + 4 );
```

```
    printf( "Destination:\t%s\n", string1 + 10 );
    memcpy( string1 + 10, string1 + 4, 40 );
    printf( "Result:\t\t%s\n", string1 );
    printf( "Length:\t\t%d characters\n\n", strlen( string1 ) );

    printf( "Function:\tmemmove with overlap\n" );
    printf( "Source:\t\t%s\n", string2 + 4 );
    printf( "Destination:\t%s\n", string2 + 10 );
    memmove( string2 + 10, string2 + 4, 40 );
    printf( "Result:\t\t%s\n", string2 );
    printf( "Length:\t\t%d characters\n\n", strlen( string2 ) );
}
```

// Example: CURSOR.C

```
/* CURSOR.C illustrates cursor functions including:
 *      _settextcursor  _gettextcursor  _displaycursor
 */

#include <conio.h>
#include <stdio.h>
#include <graph.h>

/* Macro to define cursor lines */
#define CURSOR(top,bottom) (((top) << 8) | (bottom))

void main()
{
    short oldcursor, newcursor;
    unsigned char top, bottom;
    char buffer[80];

    /* Save old cursor shape and make sure cursor is on. */
    oldcursor = _gettextcursor();
    _clearscreen( _GCLEARSCREEN );
    _displaycursor( _GCURSORON );

    /* Change cursor shape. */
    for( top = 7, bottom = 7; top; top-- )
    {
        _settextposition( 1, 1 );
        sprintf( buffer, "Top line: %d   Bottom line: %d  ", top, bottom );
        _outtext( buffer );
        newcursor = CURSOR( top, bottom );
        _settextcursor( newcursor );
        _getch();
    }

    _outtext( "\nCursor off" );
    _displaycursor( _GCURSOROFF );
    _getch();
    _outtext( "\nCursor on" );
    _displaycursor( _GCURSORON );
    _getch();

    /* Restore original cursor shape. */
    _settextcursor( oldcursor );
    _clearscreen( _GCLEARSCREEN );
}
```

// Example: DCOMMIT.C

```
/* DCOMMIT.C illustrates MS-DOS file I/O functions including:
 *      _dos_commit     _dos_creatnew     _dos_write
 *      _dos_creat      _dos_close
 *
 */

#include <dos.h>
#include <errno.h>
#include <conio.h>

void main()
{
    char saveit[] = "Straight to disk. ",
         prompt[] = "File exists, overwrite? [y|n] ",
         err[] = "Error occurred. ",
         newline[] = "\n\r";
    int hfile, ch;
    unsigned count;

    /* Open file and create, overwriting if necessary. */
    if( _dos_creatnew( "COMMIT.LOG", _A_NORMAL, &hfile ) != 0 )
    {
        if( errno == EEXIST )
        {
            /* Use _dos_write to display prompts. Use _bdos to call
             * function 1 to get and echo keystroke.
             */
            _dos_write( 1, prompt, sizeof( prompt ) - 1, &count );
            ch = _bdos( 1, 0, 0 ) & 0x00ff;
            if( (ch == 'y') || (ch == 'Y') )
                _dos_creat( "COMMIT.LOG", _A_NORMAL, &hfile );
            _dos_write( 1, newline, sizeof( newline ) - 1, &count );
        }
    }
    /* Write to file; output passes through
       operating system's buffers. */
    if( _dos_write( hfile, saveit, sizeof( saveit ), &count ) != 0 )
    {
        _dos_write( 1, err, sizeof( err ) - 1, &count );
        _dos_write( 1, newline, sizeof( newline ) - 1, &count );
    }
    /* Write directly to file with no intermediate buffering */
    if( _dos_commit( hfile ) != 0 )
    {
        _dos_write( 1, err, sizeof( err ) - 1, &count );
        _dos_write( 1, newline, sizeof( newline ) - 1, &count );
    }
     /* Close file. */
    if( _dos_close( hfile ) != 0 )
    {
        _dos_write( 1, err, sizeof( err ) - 1, &count );
        _dos_write( 1, newline, sizeof( newline ) - 1, &count );
    }
}
```

// Example: DIRECT.C

```
/* DIRECT.C illustrates directory functions and miscellaneous file
 * functions including:
 *      _getcwd         _mkdir          _chdir          _rmdir
 *      system          _mktemp         remove          _unlink
 *      _stat
 *
 * See NULLFILE.C for an example of _fstat, which is similar to _stat.
 */

#include <direct.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <io.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#define ANSI                /* Delete for UNIX version. */

void main()
{
    char cwd[_MAX_DIR];
    char tmpdir[] = "DRXXXXXX";
    struct _stat filestat;

    /* Get the current working directory. */
    _getcwd( cwd, _MAX_DIR );

    /* Try to make temporary name for directory. */
    if( _mktemp( tmpdir ) == NULL )
    {
        perror( "Can't make temporary directory" );
        exit( 1 );
    }

    /* Try to create a new directory, and if successful, change to it. */
    if( !_mkdir( tmpdir ) )
    {
        _chdir( tmpdir );

        /* Create and display a file to prove it. */
        system( "echo This is a test. > TEST.TXT" );
        system( "type test.txt" );

        /* Display some file statistics. */
        if( !_stat( "TEST.TXT", &filestat ) )
        {
            printf( "File: TEST.TXT\n" );
            printf( "Drive %c:\n", filestat.st_dev + 'A' );
            printf( "Directory: %s\\%s\n", cwd + 2, tmpdir );
            printf( "Size: %ld\n", filestat.st_size );
            printf( "Created: %s", ctime( &filestat.st_atime ) );
        }
        _getch();

        /* Delete file, go back to original directory, and remove
         * directory.
         */
#if defined( ANSI )
        remove( "TEST.TXT" );       /* ANSI compatible */
```

```
#else
        _unlink( "TEST.TXT" );      /* UNIX compatible */
#endif
        _chdir( cwd );
        _rmdir( tmpdir );
    }
}
```

// Example: DISK.C

```
/* DISK.C illustrates low-level disk access using functions:
 *   _bios_disk   _dos_getdiskfree   (MS-DOS-only)
 *
 */

#include <stdio.h>
#include <conio.h>
#include <bios.h>
#include <dos.h>
#include <stdlib.h>

char __far diskbuf[512];

void main( int argc, char *argv[] )
{
    unsigned status = 0, i;
    struct _diskinfo_t di;
    struct _diskfree_t df;
    unsigned char __far *p, linebuf[17];

    if( argc != 5 )
    {
        printf( "  SYNTAX: DISK <driveletter> <head> <track> <sector>" );
        exit( 1 );
    }

    if( (di.drive = toupper( argv[1][0] ) - 'A' ) > 1 )
    {
        printf( "Must be floppy drive" );
        exit( 1 );
    }
    di.head     = atoi( argv[2] );
    di.track    = atoi( argv[3] );
    di.sector   = atoi( argv[4] );
    di.nsectors = 1;
    di.buffer   = diskbuf;

    /* Get information about disk size. */
    if( _dos_getdiskfree( di.drive + 1, &df ) )
        exit( 1 );

    /* Try reading disk three times before giving up. */
    for( i = 0; i < 3; i++ )
    {
        status = _bios_disk( _DISK_READ, &di ) >> 8;
        if( !status )
            break;
    }

    /* Display one sector. */
    if( status )
        printf( "Error: 0x%.2x\n", status );
    else
    {
        for( p = diskbuf, i = 0; p < (diskbuf + df.bytes_per_sector); p++ )
        {
            linebuf[i++] = (*p > 32) ? *p : '.';
            printf( "%.2x ", *p );
            if( i == 16 )
            {
```

```
                linebuf[i] = '\0';
                printf( " %16s\n", linebuf );
                i = 0;
            }
        }
    }
    exit( 1 );
}
```

// Example: DOSMEM.C

```
/* DOSMEM.C illustrates functions:
 *      _dos_allocmem     _dos_setblock     _dos_freemem    (MS-DOS-only)
 *
 * See COPY2.C for another example of _dos_allocmem and _dos_freemem.
 */

#include <dos.h>
#include <stdio.h>
#include <stdlib.h>

void main()
{
    char __far *buf = NULL, __far *p;
    unsigned segbuf, maxbuf, size = 512;

    /* Allocate 512-byte buffer. Convert the size to paragraphs.
     * Assign the segment to the buffer. Fill with A's.
     */
    if( _dos_allocmem( size >> 4, &segbuf ) )
        exit( 1 );
    _FP_SEG( buf ) = segbuf;
    for( p = buf; p < (buf + size); p++ )
        *p = 'A';

    /* Double the allocation. Fill the second half with B's. */
    size *= 2;
    if( _dos_setblock( size >> 4, segbuf, &maxbuf ) )
        exit( 1 );
    _FP_SEG( buf ) = segbuf;
    for( p = buf + (size / 2); p < (buf + size); p++ )
        *p = 'B';
    *(--p) = '\0';

    printf( "Memory available: %u paragraphs\n", maxbuf );
    printf( "Buffer at %Fp contains:\n%Fs", (int __far *)buf, buf );

    /* Free memory */
    exit( !_dos_freemem( segbuf ) );
}
```

// Example: DRIVES.C

```
/* DRIVES.C illustrates drive functions including:
 *      _getdrive       _chdrive        _getdcwd
 *
 * See DIRECT.C for an example of _getcwd.
 */

#include <stdio.h>
#include <conio.h>
#include <direct.h>
#include <stdlib.h>

void main()
{
    int ch, drive, curdrive;
    static char path[_MAX_PATH];

    /* Save current drive. */
    curdrive = _getdrive();

    printf( "Available drives are: \n" );

    /* If we can switch to the drive, it exists. */
    for( drive = 1; drive <= 26; drive++ )
        if( !_chdrive( drive ) )
            printf( "%c: ", drive + 'A' - 1 );

    while( 1 )
    {
        printf( "\nType drive letter to check or ESC to quit: " );
        ch = _getch();
        if( ch == 27 )
            break;
        if( _getdcwd( toupper( ch ) - 'A' + 1, path, _MAX_PATH ) != NULL )
            printf( "\nCurrent directory on that drive is %s\n", path );
    }

    /* Restore original drive.
     */
    _chdrive( curdrive );
}
```

// Example: ENVIRON.C

```
/* ENVIRON.C illustrates environment variable functions including:
 *      getenv          _putenv          _searchenv
 */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

void main( void )
{
    char *pathvar, pathbuf[256], filebuf[256];

    /* Get the PATH environment variable and save a copy of it. */
    pathvar = getenv( "PATH" );
    strcpy( pathbuf, pathvar );
    printf( "Old PATH: %s\n", pathvar ? pathvar : "variable not set");

    /* Add a new directory to the path. */
    strcpy( pathbuf, "PATH=" );
    strcat( pathbuf, pathvar );
    strcat( pathbuf, ";\\C700\\INIT;" );
    if( _putenv( pathbuf ) == -1 )
    {
        printf( "Failed\n");
        exit( 1 );
    }
    else
        printf( "New PATH: %s\n", pathbuf );

    /* Search for file in the new path. */
    _searchenv( "TOOLS.INI", "PATH", filebuf );
    if( *filebuf )
        printf( "TOOLS.INI found at %s\n", filebuf );
    else
        printf( "TOOLS.INI not found\n" );

    /* Restore original path. */
    strcpy( pathbuf, "PATH=" );
    strcat( pathbuf, pathvar );
    if( _putenv( pathbuf ) == -1 )
        printf( "Failed\n");
    else
        printf( "Old PATH: %s\n", pathvar );
    exit( 0 );
}
```

// Example: ERROR.C

```
/* ERROR.C illustrates stream file error handling. Functions illustrated
 * include:
 *      ferror          clearerr        exit            _exit
 *      perror          strerror        _strerror
 *
 * The _exit routine is not specifically illustrated, but it is the same
 * as exit except that file buffers are not flushed and atexit and _onexit
 * are not called.
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
enum BOOL { FALSE, TRUE };

void errortest( FILE *stream, char *msg, int fterm );   /* Prototype */
FILE *stream;
char string[] = "This should never be written";

void main( int argc, char *argv[] )
{
    /* Open file and test to see if open caused error. If so, terminate. */
    stream = fopen( argv[1], "r" );
    errortest( stream, "Can't open file", TRUE );

    /* Try to write to a read-only file, then test to see if write
     * caused error. If so, clear error, but don't terminate.
     */
    fprintf( stream, "%s\n", string );
    errortest( stream, "Can't write file", FALSE );
    exit( 0 );
}

/* Tests a stream to see if there is an error on it. If so, prints the
 * msg argument and terminates if the fterm flag is set.
 */
void errortest( FILE *stream, char *msg, int fterm )
{
    /* If stream doesn't exist (failed fopen) or if there is an error
     * on the stream, handle error.
     */
    if( (stream == NULL) || (ferror( stream )) )
    {
        perror( msg );
        /* _strerror and strerror can be used to get the same result
         * as perror, as illustrated by these lines (commented out).
        printf( "%s: %s\n", msg, strerror( errno ) );
        printf( _strerror( msg ) );
         */

        /* Terminate or clear error, depending on terminate flag. */
        if( fterm )
            exit( errno );
        else
            clearerr( stream );
    }
}
```

// Example: EXEC.C

```
/* EXEC.C illustrates the different versions of exec including:
 *      _execl          _execle         _execlp         _execlpe
 *      _execv          _execve         _execvp         _execvpe
 *
 * Although EXEC.C can exec any program, you can verify how different
 * versions handle arguments and environment by compiling and
 * specifying the sample program ARGS.C. See SPAWN.C for examples
 * of the similar spawn functions.
 */

#include <stdio.h>
#include <conio.h>
#include <process.h>

char *my_env[] =                /* Environment for exec?e */
{
    "THIS=environment will be",
    "PASSED=to child by the",
    "SPAWN=functions",
    NULL
};

void main()
{
    char *args[4], prog[80];
    int ch;

    printf( "Enter name of program to exec: " );
    gets( prog );
    printf( " 1. _execl   2. _execle   3. _execlp   4. _execlpe\n" );
    printf( " 5. _execv   6. _execve   7. _execvp   8. _execvpe\n" );
    printf( "Type a number from 1 to 8 (or 0 to quit): " );
    ch = _getche();
    if( (ch < '1') || (ch > '8') )
        exit( 1 );
    printf( "\n\n" );

    /* Arguments for _execv? */
    args[0] = prog;
    args[1] = "exec??";
    args[2] = "two";
    args[3] = NULL;

    switch( ch )
    {
    case '1':
        _execl( prog, prog, "_execl", "two", NULL );
        break;
    case '2':
        _execle( prog, prog, "_execle", "two", NULL, my_env );
        break;
    case '3':
        _execlp( prog, prog, "_execlp", "two", NULL );
        break;
    case '4':
        _execlpe( prog, prog, "_execlpe", "two", NULL, my_env );
        break;
    case '5':
        _execv( prog, args );
        break;
```

```
    case '6':
        _execve( prog, args, my_env );
        break;
    case '7':
        _execvp( prog, args );
        break;
    case '8':
        _execvpe( prog, args, my_env );
        break;
    default:
        break;
    }

    /* This point is reached only if exec fails. */
    printf( "\nProcess was not execed." );
    exit( 0 );
}
```

// Example: EXTDIR.C

```
/* EXTDIR.C illustrates wildcard handling using functions:
 *    _dos_findfirst    _dos_findnext     sprintf       (MS-DOS-only)
 */

#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <dos.h>
#include <io.h>
#include <sys\types.h>
#include <sys\utime.h>
#include <sys\stat.h>

long fileinfo( struct _find_t *find );       /* Prototypes */
char *timestr( unsigned d, char *buf );
char *datestr( unsigned d, char *buf );

void main( int argc, char *argv[] )
{
    struct _find_t find;
    long size;

    /* Find first matching file, then find additional matches. */
    if( !_dos_findfirst( argv[1], 0xffff, &find ) )
    {
        printf( "%-12s   %-8s    %-8s   %-8s   %-9s   %s %s %s %s\n",
                "FILE", "SIZE", "TIME", "DATE", "KIND",
                "RDO", "HID", "SYS", "ARC" );
        size = fileinfo( &find );
    }
    else
    {
        printf( "  SYNTAX: EXTDIR <wildfilespec>" );
        exit( 1 );
    }
    while( !_dos_findnext( &find ) )
        size += fileinfo( &find );
    printf( "%-12s   %8ld\n\n", "Total", size );
    exit( 0 );
}

/* Displays information about a file. */
long fileinfo( struct _find_t *pfind )
{
    char timebuf[10], datebuf[10], *pkind;

    datestr( pfind->wr_date, datebuf );
    timestr( pfind->wr_time, timebuf );

    if( pfind->attrib & _A_SUBDIR )
        pkind = "Directory";
    else if( pfind->attrib & _A_VOLID )
        pkind = "Label";
    else
        pkind = "File";

    printf( "%-12s   %8ld    %8s   %8s   %-9s    %c   %c   %c   %c\n",
            pfind->name, pfind->size, timebuf, datebuf, pkind,
            (pfind->attrib & _A_RDONLY) ? 'Y' : 'N',
```

```
            (pfind->attrib & _A_HIDDEN) ? 'Y' : 'N',
            (pfind->attrib & _A_SYSTEM) ? 'Y' : 'N',
            (pfind->attrib & _A_ARCH)   ? 'Y' : 'N' );
    return pfind->size;
}

/* Takes unsigned time in the format:              fedcba9876543210
 * s=2 sec incr, m=0-59, h=23                      hhhhhmmmmmmsssss
 * Changes to a 9-byte string (ignore seconds):    hh:mm ?m
 */
char *timestr( unsigned t, char *buf )
{
    int h = (t >> 11) & 0x1f, m = (t >> 5) & 0x3f;

    sprintf( buf, "%2.2d:%02.2d %cm", h % 12, m,  h > 11 ? 'p' : 'a' );
    return buf;
}

/* Takes unsigned date in the format:              fedcba9876543210
 * d=1-31, m=1-12, y=0-119 (1980-2099)             yyyyyyymmmmddddd
 * Changes to a 9-byte string:                     mm/dd/yy
 */
char *datestr( unsigned d, char *buf )
{
    sprintf( buf, "%2.2d/%02.2d/%02.2d",
             (d >> 5) & 0x0f, d & 0x1f, (d >> 9) + 80 );
    return buf;
}
```

// Example: EXTERR.C

```
/* EXTERR.C illustrates function:
 *    _dosexterr                    (MS-DOS-only)
 */

#include <dos.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

void errorinfo( void );     /* Prototype */

void main( int argc, char *argv[] )
{
    int hsource;

    /* Open to get file handle and test for errors. Try specifying
     * invalid files to show different errors.
     */
    if( _dos_open( argv[1], _O_RDWR, &hsource ) )
        errorinfo();
    printf( "No error\n" );
    _dos_close( hsource );
    exit( 0 );
}

/* Displays an extended error message. */
void errorinfo()
{
    struct _DOSERROR err;
    static char *eclass[] =
    {
        "", "Out of Resource", "Temporary Situation", "Authorization",
        "Internal", "Hardware Failure", "System Failure",
        "Application Error", "Not Found", "Bad Format", "Locked",
        "Media", "Already Exists", "Unknown"
    };
    static char *eaction[] =
    {
        "", "Retry", "Delay Retry", "User", "Abort", "Immediate Exit",
        "Ignore", "Retry After User Intervention"
    };
    static char *elocus[] =
    {
        "", "Unknown", "Block Device", "Net", "Serial Device", "Memory"
    };


    /* Get error information and display class, action, and locus. */
    _dosexterr( &err );
    printf( "Class:\t%s\nAction:\t%s\nLocus:\t%s\nAction\t",
            eclass[err.errclass], eaction[err.action], elocus[err.locus] );

    /* Errors that could be caused by sample _dos_open. You can expand
     * this list to handle others.
     */
    switch( err.exterror )
    {
    case 2:
        printf( "File not found\n" );
        break;
```

```
    case 3:
        printf( "Path not found\n" );
        break;
    case 5:
        printf( "Access denied\n" );
        break;
    }
    exit( err.exterror );
}
```

// Example: FCVT.C

```
/* FCVT.C illustrates floating-point-to-string conversion functions:
 *      _gcvt           _ecvt           _fcvt
 *
 * See MKFPSTR.C for an example of using the data returned by _fcvt
 * to build a formatted string. See ATONUM.C for an example of using
 * the string returned by _gcvt.
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void main()
{
    int decimal, sign;
    char *pnumstr;
    int precision = 7;
    char numbuf[50];
    double number1, number2;

    printf( "Enter two floating-point numbers: " );
    scanf( "%lf %lf", &number1, &number2 );

    /* With _gcvt, precision specifies total number of digits.
     * The decimal place and sign are inserted in the string.
     */
    _gcvt( number1 + number2, precision, numbuf );
    printf( "\nString produced by _gcvt: %s\n", numbuf );
    printf( "Total digits: %d\n", precision );

    /* With _ecvt, precision specifies total number of digits.
     * The decimal place and sign are provided for use in formatting.
     */
    pnumstr = _ecvt( number1 + number2, precision, &decimal, &sign );
    printf( "\nString produced by _ecvt: %s\nSign: %s\n",
             pnumstr, sign ? "-" : "+" );
    printf( "Digits left of decimal: %d\nTotal digits: %d\n",
             decimal, precision );

    /* With _fcvt, precision specifies digits after decimal place.
     * The decimal place and sign are provided for use in formatting.
     */
    pnumstr = _fcvt( number1 + number2, precision, &decimal, &sign );
    printf( "\nString produced by _fcvt: %s\nSign: %s\n",
             pnumstr, sign ? "-" : "+"  );
    printf( "Digits left of decimal: %d\nDigits after decimal: %d\n",
             decimal, precision );
}
```

// Example: FIGURE.C

```
/* FIGURE.C illustrates graphics drawing functions including:
 *      _setpixel   _lineto     _moveto     _rectangle      _ellipse
 *      _pie        _arc        _getarcinfo              (MS-DOS-only)
 *
 * Window versions of graphics drawing functions (such as _rectangle_w,
 * _ellipse_wxy, and _lineto_w) are illustrated in WINDOW.C and GEDIT.C.
 */

#include <conio.h>
#include <stdlib.h>
#include <graph.h>

void main()
{
    short x, y;
    struct _xycoord xystart, xyend, xyfill;

    if( !_setvideomode( _MAXRESMODE ) ) /* Find a valid graphics mode */
        exit( 1 );                      /* No graphics available      */

    for( x = 10, y = 50; y < 90; x += 2, y += 3 )/* Draw pixels       */
        _setpixel( x, y );
    _getch();

    for( x = 60, y = 50; y < 90; y += 3 )        /* Draw lines        */
    {
        _moveto( x, y );
        _lineto( x + 20, y );
    }
    _getch();

    x = 110; y = 70;                             /* Draw rectangles   */
    _rectangle( _GBORDER,       x - 20, y - 20, x, y );
    _rectangle( _GFILLINTERIOR, x + 20, y + 20, x, y );
    _getch();

    x = 160;                                     /* Draw ellipses     */
    _ellipse( _GBORDER,       x - 20, y - 20, x, y );
    _ellipse( _GFILLINTERIOR, x + 20, y + 20, x, y );
    _getch();

    x = 210;                                    /* Draw pies          */
    _pie( _GBORDER,       x - 20, y - 20, x, y,
                          x - 10, y - 20, x - 20, y - 10 );
    _pie( _GFILLINTERIOR, x + 20, y + 20, x,      y,
                          x,      y + 10, x + 10, y );

    x = 260;                                     /* Draw arcs         */
    _arc( x - 20, y - 20, x, y, x - 10, y - 20, x - 20, y - 10 );
    _arc( x + 20, y + 20, x, y, x,      y + 10, x + 10, y );

    /* Get endpoints of arc and enclose the figure, then fill it. */
    _getarcinfo( &xystart, &xyend, &xyfill );
    _moveto( xystart.xcoord, xystart.ycoord );
    _lineto( xyend.xcoord,   xyend.ycoord );
    _floodfill( xyfill.xcoord, xyfill.ycoord, _getcolor() );

    _getch();

    _setvideomode( _DEFAULTMODE );
```

```
}
```

// Example: FINDSTR.C

```
/* FINDSTR.C illustrates memory and string search functions including:
 *       memchr        strchr        strrchr            strstr
 */

#include <memory.h>
#include <string.h>
#include <stdio.h>

int  ch = 'r';
char str[] =    "lazy";
char string[] = "The quick brown dog jumps over the lazy fox";
char fmt1[] =   "         1         2         3         4         5";
char fmt2[] =   "12345678901234567890123456789012345678901234567890";

void main()
{
    char *pdest;
    int result;

    printf( "String to be searched:\n\t\t%s\n", string );
    printf( "\t\t%s\n\t\t%s\n\n", fmt1, fmt2 );

    printf( "Function:\tmemchr\n" );
    printf( "Search char:\t%c\n", ch );
    pdest = memchr( string, ch, sizeof( string ) );
    result = pdest - string + 1;
    if( pdest != NULL )
        printf( "Result:\t\t%c found at position %d\n\n", ch, result );
    else
        printf( "Result:\t\t%c not found\n" );

    printf( "Function:\tstrchr\n" );
    printf( "Search char:\t%c\n", ch );
    pdest = strchr( string, ch );
    result = pdest - string + 1;
    if( pdest != NULL )
        printf( "Result:\t\t%c found at position %d\n\n", ch, result );
    else
        printf( "Result:\t\t%c not found\n" );

    printf( "Function:\tstrrchr\n" );
    printf( "Search char:\t%c\n", ch );
    pdest = strrchr( string, ch );
    result = pdest - string + 1;
    if( pdest != NULL )
        printf( "Result:\t\t%c found at position %d\n\n", ch, result );
    else
        printf( "Result:\t\t%c not found\n" );

    printf( "Function:\tstrstr\n" );
    printf( "Search string:\t%s\n", str );
    pdest = strstr( string, str );
    result = pdest - string + 1;
    if( pdest != NULL )
        printf( "Result:\t\t%s found at position %d\n\n", str, result );
    else
        printf( "Result:\t\t%c not found\n" );
}
```

// Example: FILL.C

```
/* FILL.C illustrates color, filling, and linestyle functions including:
 *    _setlinestyle      _setfillmask       _setcolor
 *    _getlinestyle      _floodfill                        (MS-DOS-only)
 *
 * The _getfillmask function is not shown, but its use is similar to
 * _getlinestyle.
 */

#include <conio.h>
#include <graph.h>
#include <time.h>
#include <stdlib.h>
#include <stddef.h>

void main()
{
    short x, y, xinc, yinc, xwid, ywid;
    unsigned char fill[8];
    struct _videoconfig vc;
    unsigned seed = (unsigned)time( NULL ); /* Different seed each time   */
    short i, color;

    if( !_setvideomode( _MAXRESMODE ) )     /* Find a valid graphics mode */
        exit( 1 );                          /* No graphics available      */
    _getvideoconfig( &vc );

    xinc = vc.numxpixels / 8;               /* Size variables to mode     */
    yinc = vc.numypixels / 8;
    xwid = (xinc / 2) - 4;
    ywid = (yinc / 2) - 4;

    /* Draw circles and lines with different patterns. */
    for( x = xinc; x <= (vc.numxpixels - xinc); x += xinc )
    {
        for( y = yinc; y <= (vc.numypixels - yinc); y += yinc )
        {
            /* Vary random seed, randomize fill and color. */
            srand( seed = (seed + 431) * 5 );
            for( i = 0; i < 8; i++ )
                fill[i] = (unsigned char)rand();
            _setfillmask( fill );
            color = (rand() % vc.numcolors) + 1;
            _setcolor( color );

            /* Draw ellipse and fill with random color. */
            _ellipse( _GBORDER, x - xwid, y - ywid, x + xwid, y + ywid );
            _setcolor( (rand() % vc.numcolors) + 1 );
            _floodfill( x, y, color );

            /* Draw vertical and horizontal lines. Vertical line style
             * is the opposite of (NOT) horizontal style. Since lines are
             * overdrawn with several line styles, this has the effect of
             * combining colors and styles.
             */
            _setlinestyle( rand() );
            _moveto( 0, y + ywid + 4 );
            _lineto( vc.numxpixels - 1, y + ywid + 4 );
            _setlinestyle( ~_getlinestyle() );
            _moveto( x + xwid + 4, 0 );
            _lineto( x + xwid + 4, vc.numypixels - 1 );
```

```
        }
    }
    _getch();
    _setvideomode( _DEFAULTMODE );
}
```

// Example: FONTS.C

```
/* FONTS.C illustrates font functions including:          (MS-DOS-only)
 *   _registerfonts        _setfont            _outgtext
 *   _unregisterfonts      _getfontinfo        _getgtextextent
 *   _setgtextvector
 */

#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <graph.h>

#define NFONTS 6

unsigned char *face[NFONTS] =
{
    "Courier", "Helvetica", "Times Roman", "Modern", "Script", "Roman"
};
unsigned char *options[NFONTS] =
{
    "courier", "helv", "tms rmn", "modern", "script", "roman"
};

void main()
{
    unsigned char list[20];
    char fondir[_MAX_PATH];
    struct _videoconfig vc;
    struct _fontinfo fi;
    short nfont, x, y;

    /* Read header info from .FON files in current or given directory. */
    if( _registerfonts( "*.FON" ) <= 0 )
    {
        _outtext( "Enter full path where .FON files are located: " );
        gets( fondir );
        strcat( fondir, "\\*.FON" );
        if( _registerfonts( fondir ) <= 0 )
        {
            _outtext( "Error: can't register fonts" );
            exit( 1 );
        }
    }

    /* Set highest available graphics mode and get configuration. */
    if( !_setvideomode( _MAXRESMODE ) )
        exit( 1 );
    _getvideoconfig( &vc );

    /* Display each font name centered on screen. */
    for( nfont = 0; nfont < NFONTS; nfont++ )
    {
        /* Build options string. */
        strcat( strcat( strcpy( list, "t'" ), options[nfont] ), "'");
        strcat( list, "h30w24b" );

        _clearscreen( _GCLEARSCREEN );
        if( _setfont( list ) >= 0 )
        {
            /* Use length of text and height of font to center text. */
```

```
            x = (vc.numxpixels / 2) - (_getgtextextent( face[nfont] ) / 2);
            y = (vc.numypixels / 2) + (_getgtextextent( face[nfont] ) / 2);
            if( _getfontinfo( &fi ) )
            {
                _outtext( "Error: Can't get font information" );
                break;
            }
            _moveto( x, y );
            if( vc.numcolors > 2 )
                _setcolor( nfont + 2 );

            /* Rotate and display text. */
            _setgtextvector( 1, 0 );
            _outgtext( face[nfont] );
            _setgtextvector( 0, 1 );
            _outgtext( face[nfont] );
            _setgtextvector( -1, 0 );
            _outgtext( face[nfont] );
            _setgtextvector( 0, -1 );
            _outgtext( face[nfont] );
        }
        else
        {
            _outtext( "Error: Can't set font: " );
            _outtext( list );
        }
        _getch();
    }
    _unregisterfonts();
    _setvideomode( _DEFAULTMODE );
}
```

// Example: FREECT.C

```
/* FREECT.C illustrates the following heap functions:
 *      _freect         _memavl
 */

#include <malloc.h>
#include <stdio.h>

void main()
{
    char __near *bufs[64];
    unsigned request, avail, i;

    printf( "Near heap bytes free: %u\n\n", _memavl() );
    printf( "How many 1K buffers do you want from the near heap? " );
    scanf( "%d", &request );
    if( request > 64 )
    {
        printf( "There are only 64K in a segment.\n" );
        request = 64;
    }

    avail = _freect( 1024 );
    request = (avail > request) ? request : avail;
    printf( "You can have %d buffers\n", request );

    printf( "They are available at:\n");
    for( i = 0; i < request; i++ )
    {
        bufs[i] = (char __near *)_nmalloc( 1024 );
        printf( "%2d %Fp   ", i + 1, (char __far *)bufs[i] );
        if( (i % 5) == 4 )
            printf( "\n" );
    }
    printf( "\n\nNear heap bytes free: %u\n\n", _memavl() );
    printf( "Freeing buffers . . ." );
    for( i = request; i; i-- )
        _nfree( bufs[i] );
    printf( "\n\nNear heap bytes free: %u", _memavl() );
}
```

// Example: FULL.C

```
/* FULL.C illustrates:
 *      _fullpath
 */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <direct.h>

char full[_MAX_PATH], part[_MAX_PATH];

void main()
{
    while( 1 )
    {
        printf( "Enter partial path or ENTER to quit: " );
        gets( part );
        if( part[0] == 0 )
            break;

        if( _fullpath( full, part, _MAX_PATH ) != NULL )
            printf( "Full path is: %s\n", full );
        else
            printf( "Invalid path\n" );
    }
}
```

// Example: FUNGET.C

```
/* FUNGET.C illustrates getting and ungetting characters from a file.
 * Functions illustrated include:
 *      getc            getchar         ungetc
 *      fgetc           _fgetchar
 *
 * Although getchar and _fgetchar are not specifically used in the example,
 * they are equivalent to using getc or fgetc with stdin. See HEXDUMP.C
 * for another example of getc and fgetc.
 */

#include <conio.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

void getword( FILE *stream, char *buf );    /* Prototypes */
void skiptoword( FILE *stream );

void main( void )
{
    char buffer[128];
    FILE *infile;

    printf( "Enter file name: " );
    gets( buffer );
    if( (infile = fopen( buffer, "rb" )) == NULL )
    {
        perror( "Can't open file" );
        exit( 1 );
    }

    /* Read each word and print reversed version. */
    while( 1 )
    {
        skiptoword( infile );
        getword( infile, buffer );
        puts( _strrev( buffer ) );
    }
}

/* Reads one word (defined as a string of alphanumeric characters). */
void getword( FILE *stream, char *p )
{
    int  ch;

    do
    {
        /* Macro version used here, but function version could be used:
        ch = fgetc( stream );
         */
        ch = getc( stream );        /* Get characters until EOF  */
        if( ch == EOF )             /*   or nonalphanumeric     */
            exit( 0 );
        *(p++) = (char)ch;
    } while( isalnum( ch ) );
    ungetc( ch, stream );           /* Put nonalphanumeric back */
    *(--p) = '\0';                  /* Null-terminate            */
}
```

```
/* Throws away nondigit characters */
void skiptoword( FILE *stream )
{
    int  ch;

    do
    {
        ch = getc( stream );
        if( ch == EOF )
            if( feof( stream ) )
                exit( 0 );          /* End of file */
            else if( ferror( stream ) )
                exit( 1 );          /* Error */
            /* else EOF character in file */
    } while( !isalnum( ch ) );
    ungetc( ch, stream );
}
```

// Example: FWOPEN.C

```
/* FWOPEN.C - Demonstrate opening QuickWin windows
 * with _fwopen
 */

#include <io.h>
#include <stdio.h>
#include <stdlib.h>

#define OPENFLAGS "w"              /* Access permission */

void main()
{
    struct _wopeninfo wininfo;     /* Open information */
    char wintitle[32]="QuickWin "; /* Title for window */
    FILE *wp;                      /* FILE ptr to window */
    int nRes;                      /* I/O result */

    /* Set up window info structure for _fwopen */
    wininfo._version = _QWINVER;
    wininfo._title = wintitle;
    wininfo._wbufsize = _WINBUFDEF;

    /* Check current 'exit behavior' setting */
    /* Test should be true, since default is _WINEXITPERSIST */
    /* So set new behavior to prompt user */
    if( _wgetexit() == _WINEXITPERSIST )
       _wsetexit( _WINEXITPROMPT );

    /* Create a new window */
    /* NULL second argument accepts default size/position */
    wp = _fwopen( &wininfo, NULL, OPENFLAGS );
    if( wp == NULL )
    {
        printf( "***ERROR: _fwopen\n" );
        exit( -1 );
    }

    /* Write in the window */
    nRes = fprintf( wp, "Hello, QuickWin!\n" );

    /* Close the window */
    nRes = fclose( wp );

    /* On exiting anywhere, user is prompted
     * to keep window on screen or not
     */
    exit( 0 );

}
```

// Example: GEDIT.C

```
/* GEDIT.C illustrates translation between window, view, and
 * physical coordinates. Functions used include:         (MS-DOS-only)
 *      _getwindowcoord     _getviewcoord_wxy       _getphyscoord
 *      _getcolor           _getcurrentposition_w   _getpixel_w
 *      _setcolor           _setwindow              _setpixel_w
 *      _lineto_w           _moveto_w
 *
 * Similar functions not used in the example include:
 *      _getviewcoord_w     _getcurrentposition     _getpixel
 *
 * See WINDOW.C for another example of _setwindow.
 */

#include <conio.h>
#include <stdlib.h>
#include <graph.h>

/* Enums set constants to 0, 1, 2, etc. */
enum BOOL { FALSE, TRUE };
enum DISPLAY { MOVE, DRAW, ERASE };

void main()
{
    struct _xycoord view, phys;
    struct _wxycoord oldwin, newwin;
    struct _videoconfig vc;
    double xunit, yunit, xinc, yinc;
    int key, fintersect =FALSE, fdisplay = TRUE;
    short color;

    if( !_setvideomode( _MAXRESMODE ) ) /* Find a valid graphics mode  */
        exit( 1 );                      /* No graphics available       */
    _getvideoconfig( &vc );

    /* Set a window using real numbers. */
    _setwindow( FALSE, -125.0, -100.0, 125.0, 100.0 );

    /* Calculate the size of one pixel in window coordinates. Then get
     * the current window coordinates and color.
     */
    oldwin = _getwindowcoord( 1, 1 );
    newwin = _getwindowcoord( 2, 2 );
    xunit = xinc = newwin.wx - oldwin.wx;
    yunit = yinc = newwin.wy - oldwin.wy;
    newwin = oldwin = _getcurrentposition_w();
    color = _getcolor();

    while( 1 )
    {
        /* Set flag according to whether current pixel is on, then
         * turn pixel on.
         */
        if( _getpixel_w( oldwin.wx, oldwin.wy ) == color )
            fintersect = TRUE;
        else
            fintersect = FALSE;
        _setcolor( color );
        _setpixel_w( oldwin.wx, oldwin.wy );

        /* Get and test key. */
```

```
key = _getch();
switch( key )
{
case 27:                    /* ESC Quit                 */
    exit( !_setvideomode( _DEFAULTMODE ) );
case 45:                    /* MINUS More granularity   */
    if( xinc > xunit )
    {
        yinc -= yunit;
        xinc -= xunit;
    }
    continue;
case 43:                    /* PLUS Less granularity    */
    yinc += yunit;
    xinc += xunit;
    continue;
case 32:                    /* SPACE    Move no color   */
    fdisplay = MOVE;
    continue;
case 0:                     /* Extended code - get next */
    key = _getch();
    switch( key )
    {
    case 71:                /* HOME     -x -y           */
        newwin.wx -= xinc;  /*   (note fall through)    */
    case 72:                /* UP          -y           */
        newwin.wy -= yinc;
        break;
    case 73:                /* PGUP     +x -y           */
        newwin.wy -= yinc;
    case 77:                /* RIGHT    +x              */
        newwin.wx += xinc;
        break;
    case 81:                /* PGDN     +x +y           */
        newwin.wx += xinc;
    case 80:                /* DOWN        +y           */
        newwin.wy += yinc;
        break;
    case 79:                /* END      -x +y           */
        newwin.wy += yinc;
    case 75:                /* LEFT     -x              */
        newwin.wx -= xinc;
        break;
    case 82:                /* INS      Draw white      */
        fdisplay = DRAW;
        continue;
    case 83:                /* DEL      Draw black      */
        fdisplay = ERASE;
        continue;
    }
    break;
}

/* Translate window coordinates to view, view to physical. Then
 * check physical to make sure we're on screen. Update screen and
 * position if we are. Ignore if not.
 */
view = _getviewcoord_wxy( &newwin );
phys = _getphyscoord( view.xcoord, view.ycoord );
if( (phys.xcoord >= 0) && (phys.xcoord < vc.numxpixels) &&
```

```
            (phys.ycoord >= 0) && (phys.ycoord < vc.numypixels) )
        {
            /* If display on, draw to new position, else move to new. */
            if( fdisplay )
            {
                if( fdisplay == ERASE )
                    _setcolor( 0 );
                _lineto_w( newwin.wx, newwin.wy );
            }
            else
            {
                _setcolor( 0 );
                _moveto_w( newwin.wx, newwin.wy );

                /* If there was no intersect, erase old pixel. */
                if( !fintersect )
                    _setpixel_w( oldwin.wx, oldwin.wy );
            }
            oldwin = newwin;
        }
        else
            newwin = oldwin;
    }
}
```

// Example: GETCH.C

```
/* GETCH.C illustrates how to process ASCII or extended keys.
 * Functions illustrated include:
 *      _getch          _getche
 */

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

void main()
{
    int key;

    /* Read and display keys until ESC is pressed. */
    while( 1 )
    {
        /* If first key is 0, then get second extended. */
        key = _getch();
        if( (key == 0) || (key == 0xe0) )
        {
            key = _getch();
            printf( "ASCII: no\tChar: NA\t" );
        }

        /* Otherwise, there's only one key. */
        else
            printf( "ASCII: yes\tChar: %c \t", isgraph( key ) ? key : ' ' );

        printf( "Decimal: %d\tHex: %X\n", key, key );

        /* Echo character response to prompt. */
        if( key == 27)
        {
            printf( "Do you really want to quit? (Y/n) " );
            key = _getche();
            printf( "\n" );
            if( (toupper( key ) == 'Y') || (key == 13) )
                break;
        }
    }
}
```

// Example: HALLOC.C

```
/* HALLOC.C illustrates dynamic allocation of huge memory using functions:
 *      _halloc          _hfree
 */

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

void main()
{
    char __huge *bigbuf, __huge *p;
    long count = 100000L;

    /* Allocate huge buffer (100,000 bytes). */
    bigbuf = (char __huge *)_halloc( count, sizeof( char ) );
    if( bigbuf == NULL )
    {
        printf( "Insufficient memory" );
        exit( 1 );
    }

    /* Fill the buffer with characters. */
    for( p = bigbuf; count; count--, p++ )
        *p = (char)(count % 10) + '0';

    /* Free huge buffer. */
    _hfree( bigbuf );
    exit( 0 );
}
```

// Example: HANDLER.CPP

```
/* HANDLER.CPP: This program uses _set_new_handler to
 * print an error message if the new operator fails.
 */
#include <stdio.h>
#include <new.h>
/* Allocate memory in chunks of size MemBlock. */
const size_t MemBlock = 1024;
/* Allocate a memory block for the printf function to use in case
 * of memory allocation failure; the printf function uses malloc.
 * The failsafe memory block must be visible globally because the
 * handle_program_memory_depletion function can take one
 * argument only.
 */
char * failsafe = new char[128];
/* Declare a customized function to handle memory-allocation failure.
 * Pass this function as an argument to _set_new_handler.
 */
int handle_program_memory_depletion( size_t );
void main( void )
{
   // Register existence of a new memory handler.
   _set_new_handler( handle_program_memory_depletion );
   size_t *pmemdump = new size_t[MemBlock];
   for( ; pmemdump != 0; pmemdump = new size_t[MemBlock] );
}
int handle_program_memory_depletion( size_t size )
{
   // Release character buffer memory.
   delete failsafe;
   printf( "Allocation failed, " );
   printf( "%u bytes not available.\n", size );
   // Tell new to stop allocation attempts.
   return 0;
}
```

// Example: HARDERR.C

```
/* HARDERR.C illustrates handling of hardware errors using functions:
 *      _harderr            _hardresume         _hardretn  (MS-DOS-only)
 */

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <direct.h>
#include <string.h>
#include <dos.h>
#include <bios.h>

void __far hhandler( unsigned deverr, unsigned doserr, unsigned __far *hdr );
int _bios_str( char *p );

void main()
{
    /* Install our hard error handler. */
    _harderr( hhandler );

    /* Test it. */
    printf( "Make sure there is no disk in drive A:\n" );
    printf( "Press a key when ready...\n" );
    _getch();
    if( _mkdir( "a:\\test" ) )
    {
        printf( "Failed" );
        exit( 1 );
    }
    else
    {
        printf( "Succeeded" );
        _rmdir( "a:test" );
        exit( 0 );
    }
}

/* Handler to deal with hard error codes. Since MS-DOS is not reentrant,
 * it is not safe to use MS-DOS calls for doing I/O within the MS-DOS
 * Critical Error Handler (int 24h) used by _harderr. Therefore, screen
 * output and keyboard input must be done through the BIOS.
 */
void __far hhandler( unsigned deverr, unsigned doserr, unsigned __far *hdr )
{
    int ch;
    static char buf[200], tmpbuf[10];

    /* Copy message to buffer, then use BIOS to print it. */
    strcpy( buf, "\n\rDevice error code: " );
    strcat( buf, _itoa( deverr, tmpbuf, 10 ) );
    strcat( buf, "\n\rDOS error code:    " );
    strcat( buf, _itoa( doserr, tmpbuf, 10 ) );
    strcat( buf, "\n\r(R)etry, (F)ail, or (Q)uit? " );

    /* Use BIOS to write strings and get a key. */
    _bios_str( buf );
    ch = _bios_keybrd( _KEYBRD_READ ) & 0x00ff;
    _bios_str( "\n\r" );

    switch( ch )
```

```
    {
    case 'R':
    case 'r':       /* Try again */
    default:
        _hardresume( _HARDERR_RETRY );
    case 'Q':
    case 'q':       /* Quit program */

        /* The following statement may fail in the PWB environment
         * because of conflicts with PWB's hard error handler. The
         * Quit selection falls through to the Fail selection. You
         * can remove the comment to enable this line if you wish to
         * test from the command line.
        _hardresume( _HARDERR_ABORT );
         */
    case 'F':
    case 'f':       /* Return to DOS with error code */
        _hardretn( doserr );

    }
}

/* Display a string using BIOS interrupt 0x0e (Write TTY). Return length
 * of string displayed.
 */
int _bios_str( char *p )
{
    union _REGS inregs, outregs;
    char *start = p;

    inregs.h.ah = 0x0e;
    for( ; *p; p++ )
    {
        inregs.h.al = *p;
        _int86( 0x10, &inregs, &outregs );
    }
    return p - start;
}
```

// Example: HEAPBASE.C

```
/* HEAPBASE.C illustrates dynamic allocation of based memory using
 * functions:
 *      _bheapseg       _bmalloc        _bfree          _bfreeseg
 */

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>

void main()
{
    __segment seg;
    char __based( seg ) *outstr, __based( seg ) *instr;
    char __based( seg ) *pout,   __based( seg ) *pin;
    char tmpstr[80];
    int  len;

    printf( "Enter a string: " );
    gets( tmpstr );

    /* Request a based heap. Use based so that memory won't be taken from
     * near heap.
     */
    if( (seg = _bheapseg( 1000 )) == _NULLSEG )
        exit( 1 );

    /* Allocate based memory for two strings. */
    len = strlen( tmpstr );
    if( ((instr  = _bmalloc( seg, len + 1 )) == _NULLOFF) ||
        ((outstr = _bmalloc( seg, len + 1 )) == _NULLOFF) )
        exit( 1 );

    /* Copy a lowercased string to dynamic memory. The based memory is
     * far when addressed as a whole.
     */
    _fstrlwr( _fstrcpy( (char __far *)instr, (char __far *)tmpstr ) );

    /* Copy input string to output string in reversed order. When reading
     * and writing individual characters from a based heap, the compiler
     * will try to process them as near, thus speeding up the processing.
     */
    for( pin = instr + len - 1, pout = outstr;
                pout < outstr + len; pin--, pout++ )
        *pout = *pin;
    *pout = '\0';

    /* Display strings. Again, strings as a whole are far. */
    printf( "Input:  %Fs\n", (char __far *)instr );
    printf( "Output: %Fs\n", (char __far *)outstr );

    /* Free blocks and release based heap. */
    _bfree( seg, instr );
    _bfree( seg, outstr );
    _bfreeseg( seg );
    exit( 0 );
}
```

// Example: HEAPWALK.C

```
/* HEAPWALK.C illustrates heap testing functions including:
 *      _heapchk        _heapset        _heapwalk       _msize
 */

#include <stdio.h>
#include <conio.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>

/* Macro to get a random integer within a specified range */
#define getrandom( min, max ) ((rand() % (int)(((max)+1) - (min))) + (min))

int __far *heapvalidate( unsigned fill );    /* Prototypes */
void heapstat( int status );

void main()
{
    unsigned *p[10];
    int i, __far *perr;

    srand( (unsigned)time( NULL ) );  /* Seed with current time */

    /* Check heap status. Should be OK at start of heap. */
    heapstat( _heapchk() );

    /* Now do some operations that affect the heap. In this example,
     * allocate random-size blocks.
     */
    for( i = 0; i < 10; i++ )
    {
        if( (p[i] = (unsigned *)calloc( getrandom( 1, 10000 ),
                                        sizeof( unsigned ) )) == NULL )
        {
            --i;
            break;
        }
        printf( "Allocated %u at %Fp\n",
                _msize( p[i] ), (void __far *)p[i] );
    }

    /* Fill all free blocks with the test value. */
    heapstat( _heapset( 254 ) );

    /* In a real program, you might do operations here on the allocated
     * buffers. Then do heapvalidate to make sure none of the operations
     * wrote to free blocks.
     */
    perr = heapvalidate( 254 );
    if( perr == NULL )
        printf( "Free entries are unchanged.\n\n" );
    else
        printf( "Free entry at %Fp modified.\n\n", perr );

    /* Do some more heap operations. */
    for( i = 9; i >= 0; i-- )
    {
        free( p[i] );
        printf( "Deallocating %u at %Fp\n",
                _msize( p[i] ), (void __far *)p[i] );
```

```
    }

    /* Check heap again. */
    heapstat( _heapchk() );
}

/* Tests each block in the heap. If a modified free block is found,
 * returns its address. Otherwise returns NULL.
 */
int __far *heapvalidate( unsigned fill )
{
    _HEAPINFO hi;
    unsigned heapstatus, i;
    unsigned __far *p;

    /* Walk through entries, checking free blocks. */
    hi._pentry = NULL;
    while( (heapstatus = _heapwalk( &hi )) == _HEAPOK )
    {
        /* For free entries, check each byte to see that it still
         * has only the fill value. Return address if changed.
         */
        if( hi._useflag != _USEDENTRY )
        {
            for( p = (unsigned __far *)hi._pentry, i = 0; i < hi._size; p++,i++ )
                if( (unsigned)*p != fill )
                    return hi._pentry;
        }
    }
    return NULL;
}

/* Reports on the status returned by _heapwalk, _heapset, or _heapchk */
void heapstat( int status )
{
    printf( "\nHeap status: " );
    switch( status )
    {
    case _HEAPOK:
        printf( "OK - heap is fine" );
        break;
    case _HEAPEMPTY:
        printf( "OK - empty heap" );
        break;
    case _HEAPEND:
        printf( "OK - end of heap" );
        break;
    case _HEAPBADPTR:
        printf( "ERROR - bad pointer to heap" );
        break;
    case _HEAPBADBEGIN:
        printf( "ERROR - bad start of heap" );
        break;
    case _HEAPBADNODE:
        printf( "ERROR - bad node in heap" );
        break;
    }
    printf( "\n\n" );
}
```

// Example: HEXDUMP.C

```
/* HEXDUMP.C illustrates directory splitting and character stream I/O.
 * Functions illustrated include:
 *      _splitpath      _makepath       _getw           _putw
 *      fgetc           fputc           _fgetchar       _fputchar
 *      getc            putc            getchar         putchar
 *      _fsopen
 *
 * While _fgetchar, getchar, _fputchar, and putchar are not specifically
 * used in the example, they are equivalent to using fgetc or getc with
 * stdin, or to using fputc or putc with stdout. See FUNGET.C for another
 * example of fgetc and getc.
 */

#include <stdio.h>
#include <conio.h>
#include <share.h>
#include <dos.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

void main()
{
    FILE *infile, *outfile;
    static char inpath[_MAX_PATH], outpath[_MAX_PATH];
    static char drive[_MAX_DRIVE], dir[_MAX_DIR];
    static char fname[_MAX_FNAME], ext[_MAX_EXT];
    int  in, size;
    long i = 0L;

    /* Query for and open input file. */
    printf( "Enter input file name: " );
    gets( inpath );
    strcpy( outpath, inpath );
    if( (infile = fopen( inpath, "rb" )) == NULL )
    {
        printf( "Can't open input file: %d", errno );
        exit( 1 );
    }

    /* Build output file by splitting path and rebuilding with
     * new extension.
     */
    _splitpath( outpath, drive, dir, fname, ext );
    strcpy( ext, "hx" );
    _makepath( outpath, drive, dir, fname, ext );

    /* Open output file for writing. Using _fsopen allows use to ensure
     * that no one else writes to the file while we are writing to it.
     */
    if( (outfile = _fsopen( outpath, "wb", _SH_DENYWR )) == NULL )
    {
        printf( "Can't open output file: %d", errno );
        exit( 1 );
    }

    printf( "Creating %s from %s . . .\n", outpath, inpath );
    printf( "(B)yte or (W)ord: " );
    size = _getche();
```

```
    /* Get each character from input and write to output. */
    while( 1 )
    {
        if( (size == 'W') || (size == 'w') )
        {
            in = _getw( infile );
            if( (in == EOF) && (feof( infile ) || ferror( infile )) )
                break;
            fprintf( outfile, " %.4X", in );
            if( !(++i % 8) )
                _putw( 0x0A0D, outfile );        /* New line      */
        }
        else
        {
            /* This example uses the fgetc and fputc functions. You
             * could also use the macro versions:
            in = getc( infile );
             */
            in = fgetc( infile );
            if( (in == EOF) && (feof( infile ) || ferror( infile )) )
                break;
            fprintf( outfile, " %.2X", in );
            if( !(++i % 16) )
            {
                /* Macro version:
                putc( 13, outfile );
                putc( 10, outfile );
                 */
                fputc( 13, outfile );           /* New line      */
                fputc( 10, outfile );
            }
        }
    }
    _fcloseall();
    exit( 0 );
}
```

// Example: HMANAGE.C

```
/* HMANAGE.C illustrates heap management using:
 *      _heapadd        _heapmin
 */

#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <malloc.h>

void heapdump( char *msg );     /* Prototype */

char s1[] = { "Here are some strings that we use at first, then don't\n" };
char s2[] = { "need any more. We'll give their space to the heap. \n\n" };

void main()
{
    int *p[5], i;

    printf( "%s%s", s1, s2 );
    heapdump( "Initial heap" );

    /* Give space of used strings to heap. */
    _heapadd( s1, sizeof( s1 ) );
    _heapadd( s2, sizeof( s2 ) );
    heapdump( "After adding used strings" );

    /* Allocate some blocks. Some may use used string blocks. */
    for( i = 0; i < 4; i++ )
        if( (p[i] = (int *)calloc( 10 * (i + 1), sizeof( int ) )) == NULL )
        {
            --i;
            break;
        }
    heapdump( "After allocating memory" );

    /* Free some of the blocks. */
    free( p[1] );
    free( p[2] );
    heapdump( "After freeing memory" );

    /* Minimize heap. */
    _heapmin();
    heapdump( "After compacting heap" );

    exit( 0 );
}

/* Walk through heap entries, displaying information about each block. */
void heapdump( char *msg )
{
    _HEAPINFO hi;

    printf( "\n%s\n", msg );
    hi._pentry = NULL;
    while( _heapwalk( &hi ) == _HEAPOK )
        printf( "\t%s block at %Fp of size %u\t\n",
                hi._useflag == _USEDENTRY ? "USED" : "FREE",
                hi._pentry,
                hi._size );
    _getch();
```

```
}
```

// Example: INTMATH.C

```
/* INTMATH.C illustrates integer math functions including:
 *    abs   labs   __min   __max   div   ldiv
 *
 * See MATH.C for an example of fabs and other floating-point functions.
 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

void main()
{
    int x, y;
    long lx, ly;
    div_t divres;
    ldiv_t ldivres;

    printf( "Enter two integers: " );
    scanf( "%d %d", &x, &y );

    printf("Function\tResult\n\n" );
    printf( "abs\t\tThe absolute value of %d is %d\n", x, abs( x ) );
    printf( "__min\t\tThe lesser of %d and %d is %d\n", x, y, __min( x, y ) );
    printf( "__max\t\tThe greater of %d and %d is %d\n", x, y, __max( x, y ) );
    divres = div( x, y );
    printf( "div\t\tFor %d / %d, quotient is %d and remainder is %d\n\n",
            x, y, divres.quot, divres.rem );

    printf( "Enter two long integers: " );
    scanf( "%ld %ld", &lx, &ly );

    printf("Function\tResult\n\n" );
    ldivres = ldiv( lx, ly );
    printf( "labs\t\tThe absolute value of %ld is %ld\n", lx, labs( lx ) );
    printf( "ldiv\t\tFor %ld / %ld, quotient is %ld and remainder is %ld\n",
            lx, ly, ldivres.quot, ldivres.rem );
}
```

// Example: IOTEST.C

```
/* IOTEST.C compares low-level and stream I/O using function:
 *      _fdopen
 */

#include <io.h>
#include <fcntl.h>          /* _O_ constant definitions */
#include <sys\types.h>
#include <sys\stat.h>       /* _S_ constant definitions */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void main( int argc, char *argv[] )
{
    int  handle;
    FILE *stream;
    clock_t start;
    char tmp[16];

    /* Open for low-level I/O and read file in 16-byte chunks. */
    if( (handle = _open( argv[1], _O_BINARY | _O_RDONLY )) == - 1 )
        exit( 1 );
    start = clock();
    while( !_eof( handle ) )
        _read( handle, tmp, 16 );
    printf( "I/O Type: Low level\tTime: %5.1f\n",
             ((float)clock() - start) / CLOCKS_PER_SEC );

    /* Change to stream I/O and read file in 16-byte chunks. */
    if( (stream = _fdopen( handle, "rb" )) == NULL )
        exit( 1 );
    while( !feof( stream ) )
        fread( tmp, sizeof( char ), 16, stream );
    printf( "I/O Type: Stream\tTime: %5.1f\n",
             ((float)clock() - start) / CLOCKS_PER_SEC );
    _close( handle );
    exit( 0 );
}
```

// Example: IS.C

```
/* IS.C illustrates character classification functions including:
 *       isprint   __isascii   isalpha   isalnum
 *       isupper   islower     isdigit   isxdigit
 *       ispunct   isspace     iscntrl   isgraph
 *
 * Console output is also shown using:
 *      _cprintf
 *
 * See PRINTF.C for additional examples of formatting for _cprintf.
 */

#include <ctype.h>
#include <conio.h>

void main()
{
    int ch;

    /* Display each ASCII character with character type in a table. */
    for( ch = 0; ch < 256; ch++ )
    {
        if( ch % 22 == 0 )
        {
            if( ch )
                _getch();

            /* Note that _cprintf does not convert "\n" to a CR/LF sequence.
             * You can specify this sequence with "\n\r".
             */
            _cprintf( "\n\rNum Char ASCII Alpha AlNum Cap Low Digit " );
            _cprintf( "XDigit Punct White CTRL Print Graph \n\r" );
        }
        _cprintf( "%3d  ", ch );

        /* Console output functions (_cprintf, _cputs, and _putch) display
         * graphic characters for all values except 7 (bell), 8 (backspace),
         * 10 (line feed), 13 (carriage return), and 255. Characters 9 (tab)
         * and 27 (escape) may display differently in different operating
         * systems.
         */
        if( ch == 7 || ch == 8 || ch == 9 || ch == 10 ||
            ch == 13 || ch == 27 || ch == 255 )
            _cprintf("NV" );
        else
            _cprintf("%c ", ch );
        _cprintf( "%5s", __isascii( ch )  ? "Y" : "N" );
        _cprintf( "%6s", isalpha( ch )  ? "Y" : "N" );
        _cprintf( "%6s", isalnum( ch )  ? "Y" : "N" );
        _cprintf( "%5s", isupper( ch )  ? "Y" : "N" );
        _cprintf( "%4s", islower( ch )  ? "Y" : "N" );
        _cprintf( "%5s", isdigit( ch )  ? "Y" : "N" );
        _cprintf( "%7s", isxdigit( ch ) ? "Y" : "N" );
        _cprintf( "%6s", ispunct( ch )  ? "Y" : "N" );
        _cprintf( "%6s", isspace( ch )  ? "Y" : "N" );
        _cprintf( "%5s", iscntrl( ch )  ? "Y" : "N" );
        _cprintf( "%6s", isprint( ch )  ? "Y" : "N" );
        _cprintf( "%6s\n\r", isgraph( ch )  ? "Y" : "N" );
    }
}
```

// Example: KBHIT.C

```
/* KBHIT.C illustrates:
 *       _kbhit
 */

#include <conio.h>

void main()
{
    /* Display message until key is pressed. */
    while( !_kbhit() )
        _cputs( "Hit me!! " );

    /* Use _getch to throw key away. */
    _getch();
}
```

// Example: KEYBRD.C

```
/* KEYBRD.C illustrates:
 *      _bios_keybrd                        (MS-DOS-only)
 */

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

/* Macro to peek at a specified memory address */
#define peek( addr )        (*(unsigned char __far *)addr)

void main()
{
    unsigned key, shift, scan, ascii = 0;
    int kread = _KEYBRD_READ;
    int kready = _KEYBRD_READY;
    int kshiftstatus = _KEYBRD_SHIFTSTATUS;

    /* If bit 4 of the byte at 0x0040:0x0096 is set, the new keyboard
     * is present.
     */
    if( peek( 0x00400096 ) & 0x10 )
    {
        kread = _NKEYBRD_READ;
        kready = _NKEYBRD_READY;
        kshiftstatus = _NKEYBRD_SHIFTSTATUS;
    }
    printf( "New keyboard %s\n",
            (kread == _NKEYBRD_READ) ? "present" : "not present" );

    /* Read and display keys until ESC is pressed. */
    while( ascii != 27 )
    {
        /* Drain any keys in the keyboard type-ahead buffer, then get
         * the current key. If you want the last key typed rather than
         * the key currently being typed, omit the initial loop.
         */
        while( _bios_keybrd( kready ) )
            _bios_keybrd( kread );
        key = _bios_keybrd( kread );

        /* Get shift state. */
        shift = _bios_keybrd( kshiftstatus );

        /* Split key into scan and ascii parts. */
        scan = key >> 8;
        ascii = key & 0x00ff;

        /* Categorize key. */
        if( (ascii == 0) || (ascii == 0xE0 ) )
            printf( "ASCII: no\tChar: NA\t" );
        else if( ascii < 32 )
            printf( "ASCII: yes\tChar: ^%c\t", ascii + '@' );
        else
            printf( "ASCII: yes\tChar: %c \t", ascii );
        printf( "Code: %.2X\tScan: %.2X\t Shift: %.4X\n",
                ascii, scan, shift );
    }
}
```

// Example: LOCK.C

```
/* LOCK.C illustrates network file sharing functions:
 *      _sopen       _locking
 *
 * Also the global variable:
 *      _osmajor
 *
 * The program requires MS-DOS 3.0 or higher. The DOS SHARE command must
 * be loaded. The locking mechanism will only work if the program is
 * run from a network drive.
 */

#include <stdio.h>
#include <conio.h>
#include <io.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <sys\locking.h>
#include <share.h>
#include <stdlib.h>         /* For _osmajor and exit */

void error( char *msg );

void main( int argc, char *argv[] )
{
    int handle, i;
    char buf[1], msg[] = "Are any of these bytes locked?";

    /* Check for DOS version >= 3.0 */
    if( _osmajor < 3 )
        error( "Must be DOS 3.0 or higher" );

    /* If no argument, write file and lock some bytes in it. */
    if( argc == 1 )
    {
        /* Open file with deny none sharing. */
        handle = _sopen( "tmpfil", _O_BINARY | _O_RDWR | _O_CREAT,
                                   _SH_DENYNO, _S_IREAD | _S_IWRITE );
        if( handle == -1 )
            error( "Can't open file\n" );

        _write( handle, msg, sizeof( msg ) - 1 );

        /* Lock 10 bytes starting at byte 10. */
        _lseek( handle, 10L, SEEK_SET );
        if( _locking( handle, _LK_LOCK, 10L ) )
            error( "Locking failed\n" );
        printf( "Locked 10 bytes starting at byte 10\n" );

        /* Run a copy of ourself with argument to test. */
        system( "LOCK read" );
        _getch();

        /* Unlock. */
        _lseek( handle, 10L, SEEK_SET );
        _locking( handle, _LK_UNLCK, 10L );
        printf( "\nUnlocked 10 bytes starting at byte 10\n" );

        system( "LOCK read" );
        _getch();
```

```
    }

    /* Try to read some locked bytes. This branch taken during test. */
    else
    {
        /* Open file with deny none sharing. */
        handle = _sopen( "tmpfil", _O_BINARY | _O_RDWR,
                                   _SH_DENYNO, _S_IREAD | _S_IWRITE );
        if( handle == -1 )
            error( "Can't open file\n" );

        for( i = 0; i < sizeof( msg ) - 1; i++ )
        {
            /* Print characters until locked bytes are reached. */
            if( _read( handle, buf, 1 ) == -1 )
                break;
            else
                putchar( *buf );
        }
    }
    _close( handle );
    exit( 0 );
}

void error( char *errmsg )
{
    printf( errmsg );
    exit( 1 );
}
```

// Example: MATH.C

```
/* MATH.C illustrates floating-point math functions including:
 *    exp      pow       sqrt       frexp
 *    log      log10     ldexp      modf
 *    ceil     floor     fabs       fmod
 */

#include <math.h>
#include <float.h>
#include <stdio.h>
#include <stdlib.h>

void main()
{
    double x, rx, y;
    int n;

    printf( "\nEnter a real number: " );
    scanf( "%lf", &x );


    rx = frexp( x, &n );
    printf( "Mantissa: %2.2lf\tExponent: %d\n", rx, n );
    rx = modf( x, &y );
    printf( "Fraction: %2.2lf\tInteger: %lf\n", rx, y );

    printf("\nFunction\tResult for %2.2f\n\n", x );
    if( (rx = exp( x )) && (errno != ERANGE) )
        printf( "exp\t\t%2.2f\n", rx );
    else
        errno = 0;
    if( x > 0.0 )
        printf( "log\t\t%2.2f\n", log( x ) );
    if( x > 0.0 )
        printf( "log10\t\t%2.2f\n", log10( x ) );
    if( x >= 0.0 )
        printf( "sqrt\t\t%2.2f\n", sqrt( x ) );
    printf( "ceil\t\t%2.2f\n", ceil( x ) );
    printf( "floor\t\t%2.2f\n", floor( x ) );
    printf( "fabs\t\t%2.2f\n", fabs( x ) );

    printf( "\nEnter another real number: " );
    scanf( "%lf", &y );
    printf("\nFunction\tResult for %2.2f and %2.2f\n\n", x, y );
    printf( "fmod\t\t%2.2f\n", fmod( x, y ) );
    rx = pow( x, y );
    if( (errno != ERANGE) && (errno != EDOM) )
        printf( "pow\t\t%2.2f\n", rx );
    else
        errno = 0;
    rx = _hypot( x, y );
    if( errno != ERANGE )
        printf( "hypot\t\t%2.2f\n", _hypot( x, y ) );
    else
        errno = 0;

    printf( "\nEnter an integer exponent: " );
    scanf( "%d", &n );
    rx = ldexp( x, n );
    if( errno != ERANGE )
    {
```

```
        printf("\nFunction\tResult for %2.2f to power %d\n\n", x, n );
        printf( "ldexp\t\t%2.2f\n", rx );
    }
}
```

// Example: MATHERR.C

```
/* MATHERR.C illustrates writing an error routine for math functions.
 * The error function must be:
 *      _matherr
 *
 * To use _matherr, you must turn on the No Extended Dictionary option
 * within the development environment or use the /NOE linker option
 * outside the environment. For example:
 *      CL matherr.c /link /NOE
 */

#include <math.h>
#include <string.h>
#include <stdio.h>

void main()
{
    /* Do several math operations that cause errors. The _matherr
     * routine handles _DOMAIN errors, but lets the system handle
     * other errors normally.
     */
    printf( "log( -2.0 ) = %e\n", log( -2.0 ) );
    printf( "log10( -5.0 ) = %e\n", log10( -5.0 ) );
    printf( "log( 0.0 ) = %e\n", log( 0.0 ) );
}

/* Handle several math errors caused by passing a negative argument
 * to log or log10 (_DOMAIN errors). When this happens, _matherr
 * returns the natural or base-10 logarithm of the absolute value
 * of the argument and suppresses the usual error message.
 */
int _matherr( struct _exception *except )
{
    /* Handle _DOMAIN errors for log or log10. */
    if( except->type == _DOMAIN )
    {
        if( strcmp( except->name, "log" ) == 0 )
        {
            except->retval = log( -(except->arg1) );
            printf( "Special: using absolute value: %s: _DOMAIN error\n",
                    except->name );
            return 1;
        }
        else if( strcmp( except->name, "log10" ) == 0 )
        {
            except->retval = log10( -(except->arg1) );
            printf( "Special: using absolute value: %s: _DOMAIN error\n",
                    except->name );
            return 1;
        }
    }
    else
    {
        printf( "Normal: " );
        return 0;    /* Else use the default actions */
    }
}
```

// Example: MBLEN.CPP

```
/* MBLEN.CPP illustrates the behavior of the mblen function
 */

#include <stdlib.h>
#include <stdio.h>

void main( void )
{
    int      i;
    char    *pmbc = (char *)malloc( sizeof( char ) );
    wchar_t  wc   = L'a';

    printf( "Convert a wide character to multibyte character:\n" );
    i = wctomb( pmbc, wc );
    printf( "\tCharacters converted: %u\n", i );
    printf( "\tMultibyte character: %x\n\n", pmbc );

    printf( "Find length--in bytes--of multibyte character:\n" );
    i = mblen( pmbc, MB_CUR_MAX );
    printf( "\tLength--in bytes--of multibyte character: %u\n", i );
    printf( "\tWide character: %x\n\n", pmbc );

    printf( "Attempt to find length of a NULL pointer:\n" );
    pmbc = NULL;
    i = mblen( pmbc, MB_CUR_MAX );
    printf( "\tLength--in bytes--of multibyte character: %u\n", i );
    printf( "\tWide character: %x\n\n", pmbc );

    printf( "Attempt to find length of a wide-character NULL:\n" );
    wc = L'\0';
    wctomb( pmbc, wc );
    i  = mblen( pmbc, MB_CUR_MAX );
    printf( "\tLength--in bytes--of multibyte character: %u\n", i );
    printf( "\tWide character: %x\n", pmbc );
}
```

// Example: MBSTOWCS.CPP

```
/* MBSTOWCS.CPP illustrates the behavior of the mbstowcs function
 */

#include <stdlib.h>
#include <stdio.h>

void main( void )
{
    int i;
    char    *pmbnull  = NULL;
    char    *pmbhello = (char *)malloc( MB_CUR_MAX );
    wchar_t *pwchello = L"Hi";
    wchar_t *pwc      = (wchar_t *)malloc( sizeof( wchar_t ));

    printf( "Convert to multibyte string:\n" );
    i = wcstombs( pmbhello, pwchello, MB_CUR_MAX );
    printf( "\tCharacters converted: %u\n", i );
    printf( "\tHex value of first" );
    printf( " multibyte character: %#.4x\n\n", pmbhello );

    printf( "Convert back to wide-character string:\n" );
    i = mbstowcs( pwc, pmbhello, MB_CUR_MAX );
    printf( "\tCharacters converted: %u\n", i );
    printf( "\tHex value of first" );
    printf( " wide character: %#.4x\n\n", pwc );

    printf( "Attempt to convert a wide-character NULL pointer:\n" );
    i = mbstowcs( pwc, pmbnull, MB_CUR_MAX );
    printf( "\tCharacters converted: %u\n", i );
}
```

// Example: MBTOWC.CPP

```
/* MBTOWC.CPP illustrates the behavior of the mbtowc function
 */

#include <stdlib.h>
#include <stdio.h>

void main( void )
{
    int      i;
    char    *pmbc    = (char *)malloc( sizeof( char ) );
    wchar_t  wc      = L'a';
    wchar_t *pwcnull = NULL;
    wchar_t *pwc     = (wchar_t *)malloc( sizeof( wchar_t ) );

    printf( "Convert a wide character to multibyte character:\n" );
    i = wctomb( pmbc, wc );
    printf( "\tCharacters converted: %u\n", i );
    printf( "\tMultibyte character: %x\n\n", pmbc );

    printf( "Convert multibyte character back to a wide character:\n" );
    i = mbtowc( pwc, pmbc, MB_CUR_MAX );
    printf( "\tBytes converted: %u\n", i );
    printf( "\tWide character: %x\n\n", pwc );

    printf( "Attempt to convert when target is NULL\n" );
    printf( "  returns the length of the multibyte character:\n" );
    i = mbtowc( pwcnull, pmbc, MB_CUR_MAX );
    printf( "\tLength of multibyte character: %u\n\n", i );

    printf( "Attempt to convert a NULL pointer to a" );
    printf( " wide character:\n" );
    pmbc = NULL;
    i = mbtowc( pwc, pmbc, MB_CUR_MAX );
    printf( "\tBytes converted: %u\n", i );
}
```

// Example: MKFPSTR.C

```
/* MKFPSTR.C illustrates building and displaying floating-point strings
 * without printf. Functions illustrated include:
 *      strcat          strncat         _cscanf          _fcvt
 */

#include <stdlib.h>
#include <conio.h>
#include <string.h>

void main()
{
    int decimal, sign;
    int precision = 7;
    char *pnumstr, numbuf[50] = "", tmpbuf[50];
    double number1, number2;

    _cputs( "Enter two floating-point numbers: " );
    _cscanf( "%lf %lf", &number1, &number2 );
    _putch( '\n' );

    /* Use information from _fcvt to format number string. */
    pnumstr = _fcvt( number1 + number2, precision, &decimal, &sign );

    /* Start with sign if negative. */
    if( sign )
        strcat( numbuf, "-" );

    if( decimal <= 0 )
    {
        /* If decimal is to the left of first digit (decimal negative),
         * put in leading zeros, then add digits.
         */
        strcat( numbuf, "0." );
        memset( tmpbuf, '0', (size_t)abs( decimal ) );
        tmpbuf[abs( decimal )] = 0;
        strcat( numbuf, tmpbuf );
        strcat( numbuf, pnumstr );
    }
    else
    {
        /* If decimal is to the right of first digit, put in leading
         * digits. Then add decimal and trailing digits.
         */
        strncat( numbuf, pnumstr, (size_t)decimal );
        strcat( numbuf, "." );
        strcat( numbuf, pnumstr + decimal );
    }
    _cputs( strcat( "Total is: ", numbuf ) );
}
```

// Example: MODES.C

```
/* MODES.C illustrates configuration and text window functions including:
 *    _setvideomoderows   _setvideomode   _getvideoconfig
 *    _settextwindow      _outtext
 *
 * See TEXT.C for another use of _outtext. See SCROLL.C for another use
 * of _settextwindow.
 */

#include <conio.h>
#include <stdio.h>
#include <graph.h>

short modes[] = { _TEXTBW40,     _TEXTC40,      _TEXTBW80,
                  _TEXTC80,      _MRES4COLOR,   _MRESNOCOLOR,
                  _HRESBW,       _TEXTMONO,     _HERCMONO,
                  _MRES16COLOR,  _HRES16COLOR,  _ERESNOCOLOR,
                  _ERESCOLOR,    _VRES2COLOR,   _VRES16COLOR,
                  _MRES256COLOR, _ORESCOLOR
                };
char *names[] = { "TEXTBW40",    "TEXTC40",     "TEXTBW80",
                  "TEXTC80",     "MRES4COLOR",  "MRESNOCOLOR",
                  "HRESBW",      "TEXTMONO",    "HERCMONO",
                  "MRES16COLOR", "HRES16COLOR", "ERESNOCOLOR",
                  "ERESCOLOR",   "VRES2COLOR",  "VRES16COLOR",
                  "MRES256COLOR","ORESCOLOR"
                 };
short rows[] = { 60, 50, 43, 30, 25 };  /* Possible number of rows  */

void main()
{
    short c, i, j, x, y, row, num = sizeof(modes) / sizeof(modes[0]);
    struct _videoconfig vc;
    char b[500];                        /* Buffer for string        */

    _displaycursor( _GCURSOROFF );

    /* Try each mode. */
    for( i = 0; i <= num; i++ )
    {
        for (j = 0; j < 5; j++ )
        {
            /* Try each possible number of rows. */
            row = _setvideomoderows( modes[i], rows[j] );
            if( (!row) || (rows[j] != row) )
                continue;
            else
            {
                _getvideoconfig( &vc );
                y = (vc.numtextrows - 12) / 2;
                x = (vc.numtextcols - 25) / 2;

                /* Use text window to place output in middle of screen. */
                _settextwindow( y, x,
                                vc.numtextrows - y, vc.numtextcols - x );

                /* Write all information to a string, then output string. */
                c  = sprintf( b,     "Video mode: %s\n", names[i] );
                c += sprintf( b + c, "X pixels:   %d\n", vc.numxpixels );
                c += sprintf( b + c, "Y pixels:   %d\n", vc.numypixels );
                c += sprintf( b + c, "Columns:    %d\n", vc.numtextcols );
```

```
                c += sprintf( b + c, "Rows:       %d\n", vc.numtextrows );
                c += sprintf( b + c, "Colors:     %d\n", vc.numcolors );
                c += sprintf( b + c, "Bits/pixel: %d\n", vc.bitsperpixel );
                c += sprintf( b + c, "Pages:      %d\n", vc.numvideopages );
                c += sprintf( b + c, "Mode:       %d\n", vc.mode );
                c += sprintf( b + c, "Adapter:    %d\n", vc.adapter );
                c += sprintf( b + c, "Monitor:    %d\n", vc.monitor );
                c += sprintf( b + c, "Memory:     %d",   vc.memory );
                _outtext( b );
                _getch();
            }
        }
    }
    _displaycursor( _GCURSORON );
    _setvideomode( _DEFAULTMODE );
}
```

// Example: MORE.C

```
/* MORE.C illustrates line input and output using:
 *      gets            puts            _isatty          _fileno
 *
 * Like the DOS MORE command, it is a filter whose input and output can
 * be redirected.
 *
 * By default, this program is built for MS-DOS.
 */

#include <stdio.h>
#include <io.h>
#include <bios.h>

void main()
{
    long line = 0, page = 0;
    char tmp[128];

    /* Get each line from standard input and write to standard output.   */
    while( 1 )
    {
        /* If standard output is screen, wait for key after each screen. */
        if( _isatty( _fileno( stdout ) ) )
        {
            /* Wait for key every 23 lines.  */
            if( !(++line % 23 ) )
                _bios_keybrd( _KEYBRD_READ );
        }
        /* Must be redirected to file, so send a header every 58 lines.  */
        else
        {
            if( !(line++ % 58) )
                printf( "\f\nPage: %d\n\n", ++page );
        }

        /* Get and put a line of text. Note that NULL return indicates
         * error or end-of-file. For this program, we don't care which.
         */
        if( gets( tmp ) == NULL )
            break;
        puts( tmp );

    }
}
```

// Example: MOVEMEM.C

```
/* MOVEMEM.C illustrates direct memory access using functions:
 *      _movedata        _FP_SEG          _FP_OFF      (MS-DOS-only)
 *
 * Also illustrated:
 *      #pragma pack
 *
 * See COPY2.C, ALARM.C, and SYSCALL.C for
 * more examples of _FP_SEG and _FP_OFF.
 */

#include <memory.h>
#include <stdio.h>
#include <dos.h>

#pragma pack( 1 )    /* Use pragma to force packing on byte boundaries. */

struct LOWMEMVID
{
    char     vidmode;       /* 0x449 */
    unsigned scrwid;        /* 0x44A */
    unsigned scrlen;        /* 0x44C */
    unsigned scroff;        /* 0x44E */
    struct   LOCATE
    {
        unsigned char col;
        unsigned char row;
    } csrpos[8];            /* 0x450 */
    struct   CURSIZE
    {
        unsigned char end;
        unsigned char start;
    } csrsize;              /* 0x460 */
    char     page;          /* 0x462 */
} vid;
struct LOWMEMVID __far *pvid = &vid;

void main()
{
    int page;

    /* Move system information into uninitialized structure variable. */
    _movedata( 0, 0x449, _FP_SEG( pvid ), _FP_OFF( pvid ), sizeof( vid ) );

    printf( "Move data from low memory 0000:0449 to structure at %Fp\n\n",
             (void __far *)&vid );

    printf( "Mode:\t\t\t%u\n", vid.vidmode );
    printf( "Page:\t\t\t%u\n", vid.page );
    printf( "Screen width:\t\t%u\n", vid.scrwid );
    printf( "Screen length:\t\t%u\n", vid.scrlen );
    printf( "Cursor size:\t\tstart: %u\tend: %u\n",
             vid.csrsize.start, vid.csrsize.end );
    printf( "Cursor location:\t" );
    for( page = 0; page < 8; page++ )
        printf( "page:\t%u\tcolumn: %u\trow: %u\n\t\t\t",
                page, vid.csrpos[page].col, vid.csrpos[page].row );
}
```

// Example: MSB.C

```
/* MSB.C illustrates conversion between IEEE floating-point format
 * and MS Binary format. Functions illustrated include:
 *      _fieeetomsbin    _fmsbintoieee    _dieeetomsbin    _dmsbintoieee
 *
 * Only _fieeetomsbin is specifically used, but the others work the same.
 */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *binstr32( unsigned long num, char *buffer );  /* Prototype */

void main()
{
    union
    {
        float f;            /* As float */
        unsigned long ul;   /* As unsigned long (for binary conversion) */
    } msbin, ieee;
    char tmpmsbin[33], tmpieee[33];

    /* Convert an IEEE number to MS binary and show the result in binary. */
    ieee.f = 4.352f;
    if( _fieeetomsbin( &ieee.f, &msbin.f ) )
    {
        printf( "Can't convert\n" );
        exit( 1 );
    }

    /* Convert to binary string. */
    binstr32( msbin.ul, tmpmsbin );
    binstr32( ieee.ul, tmpieee );
    printf( "%f in MS Binary format:\t%32s\n", ieee.f, tmpmsbin );
    printf( "%f in IEEE format:\t%32s\n", ieee.f, tmpieee );
    exit( 0 );
}

/* Converts unsigned long to string of 32 binary characters. */
char *binstr32( unsigned long num, char *buffer )
{
    char tmp[33];
    unsigned long len;

    memset( buffer, '0', 32 );
    len = strlen( _ultoa( num, tmp, 2 ) );
    strcpy( buffer + 32 - len, tmp );
    return buffer;
}
```

// Example: MSERIES.C

```
/* MSERIES.C illustrates presentation graphics multiseries chart
 * routines using:                                  (MS-DOS-only)
 *    _pg_chartms
 */

#include <conio.h>
#include <graph.h>
#include <string.h>
#include <pgchart.h>
#include <stdlib.h>

/* Note that data are declared as a multidimension array. Since multiseries
 * chart functions expect single-series data, this array must be cast in
 * the function call. See the _pg_analyzechartms example (ANALYZE.C) for
 * an alternate method using a single-dimension array.
 */
#define TEAMS  4
#define MONTHS 3
float __far values[TEAMS][MONTHS] = { { .435F,   .522F,   .671F  },
                                      { .533F,   .431F,   .590F  },
                                      { .723F,   .624F,   .488F  },
                                      { .329F,   .446F,   .401F  } };
char __far *months[MONTHS] =          { "May",   "June",  "July"  };
char __far *teams[TEAMS] = { "Reds", "Sox",  "Cubs", "Mets" };

void main()
{
    _chartenv env;

    if( !_setvideomode( _MAXRESMODE ) ) /* Find a valid graphics mode */
        exit( 1 );
    _pg_initchart();                    /* Initialize chart system    */

    /* Multiseries bar chart */
    _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Little League Records" );
    _pg_chartms( &env, months, (float __far *)values,
                 TEAMS, MONTHS, MONTHS, teams );
    _getch();
    _clearscreen( _GCLEARSCREEN );

    /* Multiseries column chart */
    _pg_defaultchart( &env, _PG_COLUMNCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Little League Records" );
    _pg_chartms( &env, months, (float __far *)values,
                 TEAMS, MONTHS, MONTHS, teams );
    _getch();
    _clearscreen( _GCLEARSCREEN );

    /* Multiseries line chart */
    _pg_defaultchart( &env, _PG_LINECHART, _PG_POINTANDLINE );
    strcpy( env.maintitle.title, "Little League Records" );
    _pg_chartms( &env, months, (float __far *)values,
                 TEAMS, MONTHS, MONTHS, teams );
    _getch();
    _clearscreen( _GCLEARSCREEN );

    /* Multiseries line chart showing only two columns out of three
     * and three series out of four.
     */
```

```
    _pg_defaultchart( &env, _PG_LINECHART, _PG_POINTANDLINE );
    strcpy( env.maintitle.title, "Partial Little League Records" );
    _pg_chartms( &env, &months[1], &values[1][1],
                 TEAMS - 1, MONTHS - 1, MONTHS, &teams[1] );
    _getch();
    _setvideomode( _DEFAULTMODE );
}
```

// Example: NULLFILE.C

```
/* NULLFILE.C illustrates these functions:
 *      _chsize          _umask          _setmode
 *      _creat           _fstat
 */

#include <stdio.h>
#include <conio.h>
#include <io.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <stdlib.h>
#include <time.h>

void main()
{
    int fhandle;
    long fsize;
    struct _stat fstatus;
    char fname[80];

    /* Create a file of a specified length. */
    printf( "What dummy file do you want to create: " );
    gets( fname );
    if( !_access( fname, 0 ) )
    {
        printf( "File exists" );
        exit( 1 );
    }

    /* Mask out write permission. This means that a later call to open
     * will not be able to set write permission. This is not particularly
     * useful in MS-DOS, but _umask is provided primarily for compatibility
     * with systems (such as UNIX) that allow multiple permission levels.
     */
    _umask( _S_IWRITE );

    /* Despite write request, file is read-only because of mask. */
    if( (fhandle = _creat( fname, _S_IREAD | _S_IWRITE )) == -1 )
    {
        printf( "File can't be created" );
        exit( 1 );
    }

    /* Since _creat uses the default mode (usually text), you must
     * use _setmode to make sure the mode is binary.
     */
    _setmode( fhandle, _O_BINARY );

    printf( "How long do you want the file to be? " );
    scanf( "%ld", &fsize );
    _chsize( fhandle, fsize );

    /* Display statistics. */
    _fstat( fhandle, &fstatus );
    printf( "File: %s\n", fname );
    printf( "Size: %ld\n", fstatus.st_size );
    printf( "Drive %c:\n", fstatus.st_dev + 'A' );
    printf( "Permission: %s\n",
            (fstatus.st_mode & _S_IWRITE) ? "Read/Write" : "Read Only" );
```

```
    printf( "Created: %s", ctime( &fstatus.st_atime ) );

    _close( fhandle );
    exit( 0 );
}
```

// Example: NUMTOA.C

```
/* NUMTOA.C illustrates number-to-string conversion functions including:
 *      _itoa            _ltoa            _ultoa
 */

#include <stdlib.h>
#include <stdio.h>

void main()
{
    int  base, i;
    long l;
    unsigned long ul;
    char buffer[60];

    printf( "Enter an integer: " );
    scanf( "%d", &i );
    for( base =  2; base <= 16; base += 2 )
    {
        _itoa( i, buffer, base );
        if( base != 10 )
            printf( "%d in base %d is: %s\n", i, base, buffer );
    }

    printf( "Enter a long integer: " );
    scanf( "%ld", &l );
    for( base =  2; base <= 16; base += 2 )
    {
        _ltoa( l, buffer, base );
        if( base != 10 )
            printf( "%ld in base %d is: %s\n", l, base, buffer );
    }

    printf( "Enter an unsigned long integer: " );
    scanf( "%lu", &ul );
    for( base =  2; base <= 16; base += 2 )
    {
        _ultoa( ul, buffer, base );
        if( base != 10 )
            printf( "%lu in base %d is: %s\n", ul, base, buffer );
    }
}
```

// Example: PAGE.C

```
/* PAGE.C illustrates video page functions including:       (MS-DOS-only)
 *      _getactivepage  _getvisualpage  _setactivepage  _setvisualpage
 */

#include <conio.h>
#include <graph.h>
#include <time.h>
#include <stdlib.h>

void delay( clock_t wait );         /* Prototype */
char *jumper[4][3] = { { { "\\o/" }, { " O "  }, { "/ \\" } },
                       { {  "_o_" }, { " O "  }, { "( )"  } },
                       { {  " o " }, { "/O\\" }, { "/ \\" } },
                       { {  " o " }, { " O "  }, { "( )"  } } };
void main()
{
    short oldvpage, oldapage, page, row, col, line;
    struct _videoconfig vc;

    _getvideoconfig( &vc );
    if( vc.numvideopages < 4 )
        exit( 1 );              /* Fail for monochrome */
    oldapage  = _getactivepage();
    oldvpage  = _getvisualpage();
    if( !_setvideomoderows( _TEXTBW40, 25 ) )
        exit( 1 );              /* Fail if no 40-column mode   */
    _displaycursor( _GCURSOROFF );

    /* Draw image on each page. */
    for( page = 0; page < 4; page++ )
    {
        _setactivepage( page );
        for( row = 1; row < 23; row += 7 )
        {
            for( col = 1; col < 37; col += 7 )
            {
                for( line = 0; line < 3; line++ )
                {
                    _settextposition( row + line, col );
                    _outtext( jumper[page][line] );
                }
            }
        }
    }

    while( !_kbhit() )
        /* Cycle through pages 0 to 3. */
        for( page = 0; page < 4; page++ )
        {
            _setvisualpage( page );
            delay( 100L );
        }
    _getch();

    /* Restore original page. */
    _setvideomode( _DEFAULTMODE );
    _setactivepage( oldapage );
    _setvisualpage( oldvpage );
}
```

```
/* Pauses for a specified number of microseconds. */
void delay( clock_t wait )
{
    clock_t goal;

    goal = wait + clock();
    while( goal > clock() )
        ;
}
```

// Example: PAGER.C

```
/* PAGER.C illustrates line input and output on stream text files.
 * Functions illustrated include:
 *      fopen           fclose          _fcloseall      feof
 *      fgets           fputs           rename
 */

#include <stdio.h>
#include <io.h>
#include <dos.h>
#include <stdlib.h>
#include <string.h>

#define MAXSTRING 128
#define PAGESIZE 55

void quit( int error );
FILE *infile, *outfile;
char outpath[] = "tmXXXXXX";

void main( int argc, char *argv[] )
{
    char tmp[MAXSTRING];
    long line = 0, page = 1;

    /* Open file (from command line) and output (temporary) files. */
    if( (infile = fopen( argv[1], "rt" )) == NULL )
        quit( 1 );
    _mktemp( outpath );
    if( (outfile = fopen( outpath, "wt" )) == NULL )
        quit( 2 );

    /* Get each line from input and write to output. */
    while( 1 )
    {
        /* Insert form feed and page header at the top of each page. */
        if( !(line++ % PAGESIZE) )
            fprintf( outfile, "\f\nPage %d\n\n", page++ );

        if( fgets( tmp, MAXSTRING - 1, infile ) == NULL )
            if( feof( infile ) )
                break;
            else
                quit( 3 );
        if( fputs( tmp, outfile ) == EOF )
            quit( 3 );
    }

    /* Close files and move temporary file to original by deleting
     * original and renaming temporary.
     */
    _fcloseall();
    remove( argv[1] );
    rename( outpath, argv[1] );
    exit( 0 );
}

/* Handles errors */
void quit( int error )
{
    switch( error )
```

```
    {
        case 1:
            perror( "Can't open input file" );
            break;
        case 2:
            perror( "Can't open output file" );
            fclose( infile );
            break;
        case 3:
            perror( "Error processing file" );
            fclose( infile );
            fclose( outfile );
            remove( outpath );
            break;
    }
    exit( error );
}
```

// Example: PALETTE.C

```
/* PALETTE.C illustrates functions for assigning color values to
 * color indexes. Functions illustrated include:   (MS-DOS-only)
 *      _remappalette       _remapallpalette
 */

#include <graph.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>

/* Macro for mixing Red, Green, and Blue elements of color */
#define RGB(r,g,b) (((long) ((b) << 8 | (g)) << 8) | (r))

long tmp, pal[256];
void main()
{
    short red, blue, green;
    short inc, i, mode, cells, x, y, xinc, yinc;
    char buf[40];
    struct _videoconfig vc;

    /* Make sure all palette numbers are valid. */
    for( i = 0; i < 256; i++ )
        pal[i] = _BLACK;

    /* Loop through each graphics mode that supports palettes. */
    for( mode = _MRES4COLOR; mode <= _MRES256COLOR; mode++ )
    {
        if( mode == _ERESNOCOLOR )
            mode++;
        if( !_setvideomode( mode ) )
            continue;

        /* Set variables for each mode. */
        _getvideoconfig( &vc );
        switch( vc.numcolors )
        {
        case 256:           /* Active bits in this order:           */
            cells = 13;
            inc = 12;       /* ???????? ??bbbbbb ??gggggg ??rrrrrr  */
            break;
        case  16:
            cells = 4;
            if( (vc.mode == _ERESCOLOR) || (vc.mode == _VRES16COLOR) )
                inc = 16;   /* ???????? ??bb???? ??gg???? ??rr????  */
            else
                inc = 32;   /* ???????? ??Bb???? ??Gg???? ??Rr????  */
            break;
        case   4:
            cells = 2;
            inc = 32;       /* ???????? ??Bb???? ??Gg???? ??Rr????  */
            break;
        default:
            continue;
        }
        xinc = vc.numxpixels / cells;
        yinc = vc.numypixels / cells;

        /* Fill palette arrays in BGR order. */
        for( i = 0, blue = 0; blue < 64; blue += inc )
```

```
            for( green = 0; green < 64; green += inc )
                for( red = 0; red < 64; red += inc )
                {
                    pal[i] = RGB( red, green, blue );
                    /* Special case of using 6 bits to represent 16 colors.
                     * If both bits are on for any color, intensity is set.
                     * If one bit is set for a color, the color is on.
                     */
                    if( inc == 32 )
                        pal[i + 8] = pal[i] | (pal[i] >> 1);
                    i++;
                }

        /* If palettes available, remap all palettes at once. */
        if( !_remapallpalette( pal ) )
        {
            _setvideomode( _DEFAULTMODE );
            _outtext( "Palettes not available with this adapter" );
            exit( 1 );
        }

        /* Draw colored squares. */
        for( i = 0, x = 0; x < ( xinc * cells ); x += xinc )
            for( y = 0; y < ( yinc * cells); y += yinc )
            {
                _setcolor( i++ );
                _rectangle( _GFILLINTERIOR, x, y, x + xinc, y + yinc );
            }

        /* For 256-color mode, not all colors are shown. The number of
         * colors from mixing three base colors will never be the same
         * as the number that can be shown on a two-dimensional grid.
         */
        sprintf( buf, "Mode %d has %d colors", vc.mode, vc.numcolors );
        _setcolor( vc.numcolors / 2 );
        _outtext( buf );
        _getch();

        /* Change each palette entry separately in GRB order. */
        for( i = 0, green = 0; green < 64; green += inc )
            for( red = 0; red < 64; red += inc )
                for(blue = 0; blue < 64; blue += inc )
                {
                    tmp = RGB( red, green, blue );
                    _remappalette( i, tmp );
                    if( inc == 32 )
                        _remappalette( i + 8, tmp | (tmp >> 1) );
                    i++;
                }
        _getch();
    }
    _setvideomode( _DEFAULTMODE );
}
```

// Example: PGPAL.C

```
/* PGPAL.C illustrates presentation graphics palettes and the functions
 * that modify them, including:                          (MS-DOS-only)
 *      _pg_getpalette      _pg_setpalette      _pg_resetpalette
 *      _pg_getstyleset     _pg_setstyleset     _pg_resetstyleset
 *
 * It also illustrates functions for displaying text on charts:
 *      _pg_hlabelchart     _pg_vlabelchart
 */

#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include <graph.h>
#include <pgchart.h>

#define TEAMS  2
#define MONTHS 3
float __far values[TEAMS][MONTHS] = { { .435F,   .522F,   .671F  },
                                      { .533F,   .431F,   .401F  } };
char  __far *months[MONTHS] =         { "May",   "June",  "July" };
char  __far *teams[TEAMS] = { "Cubs", "Reds" };

_fillmap fill1 = { 0x99, 0x33, 0x66, 0xcc, 0x99, 0x33, 0x66, 0xcc };
_fillmap fill2 = { 0x99, 0xcc, 0x66, 0x33, 0x99, 0xcc, 0x66, 0x33 };
_styleset styles;
_palettetype pal;

void main( void )
{
    _chartenv env;

    if( !_setvideomode( _MAXRESMODE ) )     /* Find a valid graphics mode */
        exit( 1 );
    _pg_initchart();                        /* Initialize chart system    */

    /* Modify global set of line styles used for borders, grids, and
     * data connectors. Note that this change is used before
     * _pg_defaultchart, which will use the style set.
     */
    _pg_getstyleset( styles );              /* Get styles and modify      */
    styles[1] = 0x5555;                     /*   style 1 (used for        */
    _pg_setstyleset( styles );              /*   borders)--then set new   */

    _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );

    /* Modify palette for data lines, colors, fill patterns, and
     * characters. Note that the line styles are set in the palette, not
     * in the style set, so that only data connectors will be affected.
     */
    _pg_getpalette( pal );                  /* Get default palette        */
    pal[1].plotchar = 16;                   /* Set to ASCII 16 and 17     */
    pal[2].plotchar = 17;
    memcpy( pal[1].fill, fill1, 8 );        /* Copy fill masks to palette */
    memcpy( pal[2].fill, fill2, 8 );
    pal[1].color = 3;                       /* Change palette colors      */
    pal[2].color = 4;
    pal[1].style = 0xfcfc;                  /* Change palette line styles */
    pal[2].style = 0x0303;
    _pg_setpalette( pal );                  /* Put modified palette       */
```

```
    /* Multiseries bar chart */
    strcpy( env.maintitle.title, "Little League Records - Customized" );
    _pg_chartms( &env, months, (float __far *)values,
                 TEAMS, MONTHS, MONTHS, teams );
    _getch();
    _clearscreen( _GCLEARSCREEN );

    /* Multiseries line chart */
    _pg_defaultchart( &env, _PG_LINECHART, _PG_POINTANDLINE );
    strcpy( env.maintitle.title, "Little League Records - Customized" );
    _pg_chartms( &env, months, (float __far *)values,
                 TEAMS, MONTHS, MONTHS, teams );

    /* Print labels */
    _pg_hlabelchart( &env, (short)(env.chartwindow.x2 * .75F),
                           (short)(env.chartwindow.y2 * .10F),
                           12, "Up and up!" );
    _pg_vlabelchart( &env, (short)(env.chartwindow.x2 * .75F),
                           (short)(env.chartwindow.y2 * .45F),
                           13, "Sliding down!" );
    _getch();
    _clearscreen( _GCLEARSCREEN );

    _pg_resetpalette();                     /* Restore default palette    */
    _pg_resetstyleset();                    /*   and style set            */

    /* Multiseries bar chart */
    _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Little League Records - Default" );
    _pg_chartms( &env, months, (float __far *)values,
                 TEAMS, MONTHS, MONTHS, teams );
    _getch();
    _clearscreen( _GCLEARSCREEN );

    /* Multiseries line chart */
    _pg_defaultchart( &env, _PG_LINECHART, _PG_POINTANDLINE );
    strcpy( env.maintitle.title, "Little League Records - Default" );
    _pg_chartms( &env, months, (float __far *)values,
                 TEAMS, MONTHS, MONTHS, teams );
    _getch();

    _setvideomode( _DEFAULTMODE );
}
```

// Example: PRINTF.C

```
/* PRINTF.C illustrates output formatting with:
 *      printf
 *
 * The rules for formatting also apply to _cprintf, _snprintf, sprintf,
 * vfprintf, vprintf, _vsnprintf, and vsprintf. For other examples of
 * printf formatting, see EXTDIR.C (sprintf), WPRINTF.C (vprintf),
 * TABLE.C (fprintf), ROTATE.C (printf), and IS.C (_cprintf).
 */

#include <stdio.h>

void main()
{
    char ch = 'h', *string = "computer";
    int count = 234, *ptr, hex = 0x10, oct = 010, dec = 10;
    double fp = 251.7366;

    /* Display integers. */
    printf("%d    %+d    %06d    %X    %x    %o\n\n",
            count, count, count, count, count, count );

    /* Count characters printed. */
    printf( "            V\n" );
    printf( "1234567890123%n45678901234567890\n", &count );
    printf( "Number of characters printed: %d\n\n", count );

    /* Display characters. */
    printf( "%10c%5c\n\n", ch, ch );

    /* Display strings. */
    printf( "%25s\n%25.4s\n\n", string, string );

    /* Display real numbers. */
    printf( "%f    %.2f    %e    %E\n\n", fp, fp, fp, fp );

    /* Display in different radixes. */
    printf( "%i    %i    %i\n\n", hex, oct, dec );

    /* Display pointers. */
    ptr = &count;
    printf( "%Np    %p    %Fp\n", ptr, (int __far *)ptr, (int __far *)ptr );
}
```

// Example: QSORT.C

```
/* QSORT.C illustates randomizing, sorting, and searching. Functions
 * illustrated include:
 *  srand       rand        qsort
 *  _lfind      _lsearch    bsearch
 *
 * The _lsearch function is not specifically shown in the program, but
 * its use is the same as _lfind except that if it does not find the
 * element; it inserts it at the end of the array rather than failing.
 */

#include <search.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>

#define  ASIZE 1000
unsigned array[ASIZE];

/* Macro to get a random integer within a specified range */
#define getrandom( min, max ) ((rand() % (int)(((max)+1) - (min))) + (min))

/* Must be declared before call */
int __cdecl cmpgle( const void *elem1, const void *elem2 );
int __cdecl cmpe( const void *key, const void *tableentry );

void main()
{
    unsigned i, *result, elements = ASIZE, key = ASIZE / 2;

    /* Seed the random number generator with current time. */
    srand( (unsigned)time( NULL ) );

    /* Build a random array of integers. */
    printf( "Randomizing . . .\n" );
    for( i = 0; i < ASIZE; i++ )
        array[i] = getrandom( 1, ASIZE );

    /* Show every tenth element. */
    printf( "Printing every tenth element . . .\n" );
    for( i = 9; i < ASIZE; i += 10 )
    {
        printf( "%5u", array[i] );
        if( !(i % 15) )
            printf( "\n" );
    }

    /* Find element using linear search. */
    printf( "\nDoing linear search . . .\n" );
    result = (unsigned *)_lfind( &key, array, &elements, sizeof( unsigned ), cmpe );

    if( result == NULL )
        printf( "  Value %u not found\n", key );
    else
        printf( "  Value %u found in element %u\n", key, result - array + 1);

    /* Sort array using Quicksort algorithm. */
    printf( "Sorting . . .\n" );
    qsort( array, ASIZE, sizeof( int ), cmpgle );
```

```
    /* Show every tenth element. */
    printf( "Printing every tenth element . . .\n" );
    for( i = 9; i < ASIZE; i += 10 )
    {
        printf( "%5u", array[i] );
        if( !(i % 15) )
            printf( "\n" );
    }

    /* Find element using binary search. Note that the array must
     * be previously sorted before using binary search.
     */
    printf( "\nDoing binary search . . .\n" );
    result = (unsigned *)bsearch( &key, array, elements, sizeof( unsigned ),
cmpgle );
    if( result == NULL )
        printf( "  Value %u not found\n", key );
    else
        printf( "  Value %u found in element %u\n", key, result - array + 1 );
}

/* Compares and returns greater than (1), less than (-1), or equal to (0).
 * This function is called by qsort and bsearch. When used with qsort the
 * order of entries is unimportant. When used with bsearch, elem1 is the
 * key to be found, and elem2 is the current table entry.
 */
int __cdecl cmpgle( const void *elem1, const void *elem2 )
{
    if( *(unsigned *)elem1 > *(unsigned *)elem2 )
        return 1;
    else if( *(unsigned *)elem1 < *(unsigned *)elem2 )
        return -1;
    else
        return 0;
}

/* Compares and returns equal (1) or not equal (0). This function is
 * called by _lfind and _lsearch.
 */
int __cdecl cmpe( const void *key, const void *tableentry )
{
    return !( *(unsigned *)key == *(unsigned *)tableentry );
}
```

// Example: REALLOC.C

```
/* REALLOC.C illustrates heap functions:
 *      calloc      realloc         _expand
 */

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

void main()
{
    int  *bufint;
    char *bufchar;

    printf( "Allocate two 512 element buffers\n" );
    if( (bufint = (int *)calloc( 512, sizeof( int ) )) == NULL )
        exit( 1 );
    printf( "Allocated %d bytes at %Fp\n",
            _msize( bufint ), (void __far *)bufint );

    if( (bufchar = (char *)calloc( 512, sizeof( char ) )) == NULL )
        exit( 1 );
    printf( "Allocated %d bytes at %Fp\n",
            _msize( bufchar ), (void __far *)bufchar );

    /* Expand the second buffer, reallocate the first. Note that trying
     * to expand the first buffer would fail because the second buffer
     * would be in the way.
     */
    if( (bufchar = (char *)_expand( bufchar, 1024 )) == NULL )
        printf( "Can't expand" );
    else
        printf( "Expanded block to %d bytes at %Fp\n",
                _msize( bufchar ), (void __far *)bufchar );

    if( (bufint = (int *)realloc( bufint, 1024 * sizeof( int ) )) == NULL )
        printf( "Can't reallocate" );
    else
        printf( "Reallocated block to %d bytes at %Fp\n",
                _msize( bufint ), (void __far *)bufint );

    /* Free memory */
    free( bufint );
    free( bufchar );
    exit( 0 );
}
```

// Example: RECORDS1.C

```
/* RECORDS1.C illustrates reading and writing of file records using seek
 * functions including:
 *      fseek       rewind      ftell
 *
 * Other general functions illustrated include:
 *      tmpfile     _rmtmp      fread       fwrite
 *
 * Also illustrated:
 *      struct
 *
 * See RECORDS2.C for a version of this program using fgetpos and fsetpos.
 */

#include <stdio.h>
#include <io.h>
#include <string.h>

/* File record */
struct RECORD
{
    int     integer;
    long    doubleword;
    double  realnum;
    char    string[15];
} filerec = { 0, 1, 10000000.0, "eel sees tar" };

void main()
{
    int c, newrec;
    size_t recsize = sizeof( filerec );
    FILE *recstream;
    long recseek;

    /* Create and open temporary file. */
    recstream = tmpfile();

    /* Write 10 unique records to file. */
    for( c = 0; c < 10; c++ )
    {
        ++filerec.integer;
        filerec.doubleword *= 3;
        filerec.realnum /= (c + 1);
        _strrev( filerec.string );

        fwrite( &filerec, recsize, 1, recstream );
    }

    /* Find a specified record. */
    do
    {
        printf( "Enter record betweeen 1 and 10 (or 0 to quit): " );
        scanf( "%d", &newrec );

        /* Find and display valid records. */
        if( (newrec >= 1) && (newrec <= 10) )
        {
            recseek = (long)((newrec - 1) * recsize);
            fseek( recstream, recseek, SEEK_SET );

            fread( &filerec, recsize, 1, recstream );
```

```
            printf( "Integer:\t%d\n", filerec.integer );
            printf( "Doubleword:\t%ld\n", filerec.doubleword );
            printf( "Real number:\t%.2f\n", filerec.realnum );
            printf( "String:\t\t%s\n\n", filerec.string );
        }
    } while( newrec );

    /* Starting at first record, scan each for specific value. The following
     * line is equivalent to:
     *      fseek( recstream, 0L, SEEK_SET );
     */
    rewind( recstream );

    do
    {
        fread( &filerec, recsize, 1, recstream );
    } while( filerec.doubleword < 1000L );

    recseek = ftell( recstream );
    /* Equivalent to: recseek = fseek( recstream, 0L, SEEK_CUR ); */
    printf( "\nFirst doubleword above 1000 is %ld in record %d\n",
            filerec.doubleword, recseek / recsize );

    /* Close and delete temporary file. */
    _rmtmp();
}
```

// Example: RECORDS2.C

```
/* RECORDS2.C illustrates reading and writing of file records with the
 * following functions:
 *      fgetpos     fsetpos
 *
 * See RECORDS1.C for a version using fseek, rewind, and ftell.
 */

#include <stdio.h>
#include <io.h>

/* File record */
struct RECORD
{
    int     integer;
    long    doubleword;
    double  realnum;
} filerec = { 0, 1, 10000000.0 };

void main()
{
    int c, newrec;
    size_t recsize = sizeof( filerec );
    FILE *recstream;
    fpos_t recpos;

    /* Create and open temporary file. */
    recstream = tmpfile();

    /* Write 10 unique records to file. */
    for( c = 0; c < 10; c++ )
    {
        ++filerec.integer;
        filerec.doubleword *= 3;
        filerec.realnum /= (c + 1);

        fwrite( &filerec, recsize, 1, recstream );
    }

    /* Find a specified record. */
    do
    {
        printf( "Enter record betweeen 1 and 10 (or 0 to quit): " );
        scanf( "%d", &newrec );

        /* Find and display valid records. */
        if( (newrec >= 1) && (newrec <= 10) )
        {
            recpos = (fpos_t)((newrec - 1) * recsize);
            fsetpos( recstream, &recpos );
            fread( &filerec, recsize, 1, recstream );

            printf( "Integer:\t%d\n", filerec.integer );
            printf( "Doubleword:\t%ld\n", filerec.doubleword );
            printf( "Real number:\t%.2f\n", filerec.realnum );
        }
    } while( newrec );

    /* Starting at first record, scan each for specific value. */
    recpos = (fpos_t)0;
    fsetpos( recstream, &recpos );
```

```
    do
        fread( &filerec, recsize, 1, recstream );
    while( filerec.doubleword < 1000L );

    fgetpos( recstream, &recpos );
    printf( "\nFirst doubleword above 1000 is %ld in record %d\n",
            filerec.doubleword, recpos / recsize );

    /* Close and delete temporary file. */
    _rmtmp();
}
```

// Example: ROTATE.C

```
/* ROTATE.C illustrates bit rotation functions including:
 *      _rotl           _rotr           _lrotl          _lrotr
 *
 * The _lrotl and _lrotr functions are not illustrated, but they are the
 * same as _rotl and _rotr except that they work on long integers.
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char *binstr( int num, char *buffer );      /* Prototype */

void main()
{
    int  shift, i, ir, il;
    char tmpbuf[20];

    printf( "Enter integer: " );
    scanf( "%d", &i );
    printf( "\n\n" );

    /* Display table header for rotates. */
    printf( "%6s    %-7s    %16s    %-7s    %16s\n",
            " ", "Left", " ", "Right", " " );
    printf( "%6s    %7s    %16s    %7s    %16s\n\n",
            "Shift", "Decimal", "Binary", "Decimal", "Binary" );

    /* Display table of rotated values. */
    for( shift = 0; shift <= 16; shift++ )
    {
        il = _rotl( i, shift ); ;
        printf( "%5d    %7d    %16s    ", shift, il, binstr( il, tmpbuf ) );
        ir = _rotr( i, shift );
        printf( "%7d    %16s\n", ir, binstr( ir, tmpbuf ) );
    }
}

/* Converts integer to string of 16 binary characters. */
char *binstr( int num, char *buffer )
{
    char tmp[17];
    int  len;

    memset( buffer, '0', 16 );
    len = strlen( _itoa( num, tmp, 2 ) );
    strcpy( buffer + 16 - len, tmp );
    return buffer;
}
```

// Example: SCANF.C

```
/* SCANF.C illustrates formatted input. Functions illustrated include:
 *          scanf           fflush          _flushall
 *
 * For other examples of formatted input, see TABLE.C (fscanf) and
 * SETTIME.C (sscanf).
 */

#include <stdio.h>
#include <conio.h>

void main()
{
    int result, integer;
    float fp;
    char string[81];

    /* Get numbers. */
    printf( "Enter an integer and a floating-point number: " );
    scanf( "%d %f", &integer, &fp );
    printf( "%d + %f = %f\n\n", integer, fp, integer + fp );

    /* Read each word as a string. */
    printf( "Enter a sentence of four words with scanf: " );
    for( integer = 0; integer < 4; integer++ )
    {
        scanf( "%s", string );
        printf( "%s\n", string );
    }

    /* Clear the input buffer and get with gets. */
    fflush( stdin );
    printf( "Enter the same sentence with gets: " );
    gets( string );
    printf( "%s\n", string );

    /* Specify delimiters. The ^ inside the brackets says accept any
     * character except the following other characters in brackets (tab
     * and newline in the example).
     */
    printf( "Enter the same sentence with scanf and delimiters: " );
    scanf( "%[^\n\t]s", string );
    printf( "%s\n", string );

    /* Loop until input value is 0. */
    printf( "\n\nEnter numbers in C decimal (num), hex (0xnum), " );
    printf( "or octal (0num) format.\nEnter 0 to quit\n\n" );
    do
    {
        printf( "Enter number: " );
        result = scanf( "%i", &integer );
        if( result )
            /* Display valid integers. */
            printf( "Decimal: %i  Octal: 0%o  Hexadecimal: 0x%X\n\n",
                    integer, integer, integer );
        else
        {   /* Read invalid characters. Then flush and continue. */
            scanf( "%s", string );
            printf( "Invalid number: %s\n\n", string );
            _flushall();
            integer = 1;
```

```
        }
    } while( integer );
}
```

// Example: SCAT.C

```
/* SCAT.C illustrates presentation graphics scatter chart functions
 * including:                                              (MS-DOS-only)
 *    _pg_chartscatter   _pg_chartscatterms
 */

#include <conio.h>
#include <graph.h>
#include <string.h>
#include <stdlib.h>
#include <pgchart.h>

#define ITEMS  5
#define SERIES 2
float __far people[SERIES][ITEMS]  = { { 235.F, 423.F, 596.F, 729.F, 963.F },
                                       { 285.F, 392.F, 634.F, 801.F, 895.F }
                                     };
float __far profits[SERIES][ITEMS] = { { 0.9F,  2.3F,  5.4F,  8.0F,  9.3F  },
                                       { 4.2F,  3.4F,  3.6F,  2.9F,  2.7F  }
                                     };
char  __far *companies[SERIES] = { "Goodstuff, Inc.", "Badjunk & Co." };

void main()
{
    _chartenv env;

    if( !_setvideomode( _MAXRESMODE ) ) /* Find a valid graphics mode */
        exit( 1 );
    _pg_initchart();                    /* Initialize chart system    */

    /* Show single-series scatter chart. */
    _pg_defaultchart (&env, _PG_SCATTERCHART, _PG_POINTONLY );
    strcpy( env.maintitle.title, "Goodstuff, Inc." );
    strcpy( env.xaxis.axistitle.title, "Employees" );
    strcpy( env.yaxis.axistitle.title, "Profitability" );
    _pg_chartscatter( &env, people[0], profits[0], ITEMS );
    _getch();
    _clearscreen( _GCLEARSCREEN );

    /* Show multiseries scatter chart. */
    _pg_defaultchart (&env, _PG_SCATTERCHART, _PG_POINTONLY );
    strcpy( env.xaxis.axistitle.title, "Employees" );
    strcpy( env.yaxis.axistitle.title, "Profitability" );
    _pg_chartscatterms( &env, (float __far *)people, (float __far *)profits,
                        SERIES, ITEMS, ITEMS, companies );
    _getch();

    _setvideomode( _DEFAULTMODE );
}
```

// Example: SCROLL.C

```
/* SCROLL.C illustrates:
 *      _gettextwindow      _settextwindow      _scrolltextwindow
 *
 * See MODES.C for another use of _settextwindow.
 */

#include <stdio.h>
#include <conio.h>
#include <graph.h>

void deleteline( void );
void insertline( void );
void status( char *msg );

void main()
{
    short row;
    char  buf[40];

    /* Set up screen for scrolling; put text window around scroll area. */
    _settextrows( 25 );
    _clearscreen( _GCLEARSCREEN );
    for( row = 1; row <= 25; row++ )
    {
        _settextposition( row, 1 );
        sprintf( buf, "Line %c             %2d", row + 'A' - 1, row );
        _outtext( buf );
    }
    _getch();
    _settextwindow( 1, 1, 25, 10 );

    /* Delete some lines. */
    _settextposition( 11, 1 );
    for( row = 12; row < 20; row++ )
        deleteline();
    status( "Deleted 8 lines" );

    /* Insert some lines. */
    _settextposition( 5, 1 );
    for( row = 1; row < 6; row++ )
        insertline();
    status( "Inserted 5 lines" );

    /* Scroll up and down. */
    _scrolltextwindow( -7 );
    status( "Scrolled down 7 lines" );
    _scrolltextwindow( 5 );
    status( "Scrolled up 5 lines" );
    _setvideomode( _DEFAULTMODE );
}

/* Delete lines by scrolling them off the top of the current text window.
 * Save and restore original window.
 */
void deleteline()
{
    short left, top, right, bottom;
    struct _rccoord rc;

    _gettextwindow( &top, &left, &bottom, &right );
```

```
    rc = _gettextposition();
    _settextwindow( rc.row, left, bottom, right );
    _scrolltextwindow( _GSCROLLUP );
    _settextwindow( top, left, bottom, right );
    _settextposition( rc.row, rc.col );
}

/* Insert some lines by scrolling in blank lines from the top of the
 * current text window. Save and restore original window.
 */
void insertline()
{
    short left, top, right, bottom;
    struct _rccoord rc;

    _gettextwindow( &top, &left, &bottom, &right );
    rc = _gettextposition();
    _settextwindow( rc.row, left, bottom, right );
    _scrolltextwindow( _GSCROLLDOWN );
    _settextwindow( top, left, bottom, right );
    _settextposition( rc.row, rc.col );
}

/* Display and clear status in its own window. */
void status( char *msg )
{
    short left, top, right, bottom;

    _gettextwindow( &top, &left, &bottom, &right );
    _settextwindow( 1, 50, 2, 80 );
    _outtext( msg );
    _getch();
    _clearscreen( _GWINDOW );
    _settextwindow( top, left, bottom, right );
}
```

// Example: SEEK.C

```
/* SEEK.C illustrates low-level file I/O functions including:
 *      _filelength      _lseek           _tell
 */

#include <io.h>
#include <conio.h>
#include <stdio.h>
#include <fcntl.h>          /* _O_ constant definitions */
#include <process.h>

void error( char *errmsg );

void main()
{
    int handle, ch;
    unsigned count;
    long position, length;
    char buffer[2], fname[80];

    /* Get file name and open file. */
    do
    {
        printf( "Enter file name: " );
        gets( fname );
        handle = _open( fname, _O_BINARY | _O_RDONLY );
    } while( handle == -1 );

    /* Get and print length. */
    length = _filelength( handle );
    printf( "\nFile length of %s is: %ld\n\n", fname, length );

    /* Report the character at a specified position. */
    do
    {
        printf( "Enter integer position less than file length: " );
        scanf( "%ld", &position );
    } while( position > length );

    _lseek( handle, position, SEEK_SET );
    if( _read( handle, buffer, 1 ) == -1 )
        error( "Read error" );
    printf( "Character at byte %ld is ASCII %u ('%c')\n\n",
            position, *buffer, *buffer );

    /* Search for a specified character and report its position. */
    _lseek( handle, 0L, SEEK_SET);           /* Set to position 0 */
    printf( "Type character to search for: " );
    ch = _getche();

    /* Read until character is found. */
    do
    {
        if( (count = _read( handle, buffer, 1 )) == -1 )
            error( "Read error" );
    } while( (*buffer != (char)ch) && count );

    /* Report the current position. */
    position = _tell( handle );
    if( count )
        printf( "\nFirst ASCII %u ('%c') is at byte %ld\n",
```

```
                ch, ch, position );
    else
        printf( "\nASCII %u ('%c') not found\n", ch, ch );
    _close( handle );
    exit( 0 );
}

void error( char *errmsg )
{
    perror( errmsg );
    exit( 1 );
}
```

// Example: SETROWS.C

```
/* SETROWS.C illustrates
 *      _settextrows
 */

#include <graph.h>
#include <stdlib.h>

void main( int argc, char **argv )
{
    struct _videoconfig vc;
    short rows = atoi( argv[1] );

    _getvideoconfig( &vc );

    /* Make sure new rows are valid and the same as requested rows. */
    if( !rows || (_settextrows( rows ) != rows) )
        _outtext( "\nSyntax: SETROWS [ 25 | 43 | 50 ]\n" );

    /* Always return old rows for batch file testing. */
    exit( vc.numtextrows );
}
```

// Example: SETSTR.C

```
/* SETSTR.C illustrates string and memory set functions including:
 *     memset           _strnset         _strset
 */

#include <memory.h>
#include <string.h>
#include <stdio.h>

char string[60] = "The quick brown dog jumps over the lazy fox ";
/*                           1         2         3         4         5 *
 *                 12345678901234567890123456789012345678901234567890 */
void main()
{
    printf( "Function:\tmemset with fill character 'Û'\n" );
    printf( "Destination:\t%s\n", string );
    memset( string + 10, 'Û', 5 );
    printf( "Result:\t\t%s\n\n", string );

    printf( "Function:\t_strnset with fill character 'Û'\n" );
    printf( "Destination:\t%s\n", string );
    _strnset( string + 15, 'Û', 5 );
    printf( "Result:\t\t%s\n\n", string );

    printf( "Function:\t_strset with fill character 'Û'\n" );
    printf( "Destination:\t%s\n", string );
    _strset( string + 20, 'Û' );
    printf( "Result:\t\t%s\n\n", string );
}
```

// Example: SETTIME.C

```
/* SETTIME.C illustrates getting and setting the MS-DOS time and date using:
 *      _dos_gettime    _dos_settime    _dos_getdate    _dos_setdate
 *                                                              (MS-DOS-only)
 * Formatted input to a string is illustrated using:
 *      sscanf
 *
 * See TABLE.C (fscanf) and SCANF.C for other examples of formatted input.
 */

#include <dos.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>

struct _dosdate_t ddate;
struct _dostime_t dtime;

void main()
{
    unsigned tmpday, tmpmonth, tmpyear;
    unsigned tmphour, tmpminute, tmpsecond;
    char *days[] = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
    char tmpbuf[20];

    /* Get current date. */
    _dos_getdate( &ddate );
    printf( "Date: %u/%02u/%02u %s\n",
            ddate.month, ddate.day, ddate.year - 1900,
            days[ddate.dayofweek] );
    printf( "Enter new date: " );

    /* Get a date string and if it's not 0 (for CR), assign it to date. */
    gets( tmpbuf );
    if( strlen( tmpbuf ) )
    {
        /* NOTE: You must read the month and day into a temporary unsigned
         * variable. If you try to read directly into the unsigned char
         * values ddate.day or ddate.month, sscanf (or scanf) will read a
         * word, thus overriding adjacent bytes.
         */
        sscanf( tmpbuf, "%u/%u/%u", &tmpmonth, &tmpday, &tmpyear );
        if( (tmpyear - 80) <= 20 )
            ddate.year = tmpyear + 1900;
        else if( (tmpyear >= 1980) && (tmpyear <= 2099 ) )
            ddate.year = tmpyear;
        if( (tmpmonth + 1) <= 13 )
            ddate.month = (unsigned char)tmpmonth;
        if( (tmpday + 1) <= 32 )
            ddate.day = (unsigned char)tmpday;
        _dos_setdate( &ddate );
        _dos_getdate( &ddate );
        printf( "New date: %u/%02u/%02u %s\n",
                ddate.month, ddate.day, ddate.year - 1900,
                days[ddate.dayofweek] );
    }

    /* Get current time. */
    _dos_gettime( &dtime );
    printf( "Time: %u:%02u:%02u\n", dtime.hour, dtime.minute, dtime.second );
    printf( "Enter new time: " );
```

```
    /* Get a time string and if it's not 0 (for CR), assign it to time. */
    gets( tmpbuf );
    if( strlen( tmpbuf ) )
    {
        sscanf( tmpbuf, "%u:%u:%u", &tmphour, &tmpminute, &tmpsecond );
        if( tmphour < 24 )
            dtime.hour = (unsigned char)tmphour;
        if( tmpminute < 60 )
            dtime.minute = (unsigned char)tmpminute;
        if( tmpsecond < 60 )
            dtime.second = (unsigned char)tmpsecond;
        _dos_settime( &dtime );
        _dos_gettime( &dtime );
        printf( "New time: %u:%02u:%02u\n",
                dtime.hour, dtime.minute, dtime.second );
    }
}
```

// Example: SIEVE.C

```
/* SIEVE.C illustrates timing functions including:
 *    clock   difftime   _bios_timeofday
 *
 * In addition to the timing use shown here, these functions can be
 * used for delay loops as shown for the clock function in BEEP.C.
 */

#include <time.h>
#include <stdio.h>
#include <bios.h>
#define TICKPERSEC 18.2

int mark[10000];

void main()
{
    time_t   tstart, tend;    /* For difftime         */
    clock_t  cstart, cend;    /* For clock            */
#if defined( _DOS )
    long     bstart, bend;    /* For _bios_timeofday  */
#endif
    register int i, loop;
    int      n, num, step;

    /* Start timing. */
    printf( "Working...\n" );
    time( &tstart );     /* Use time and difftime for timing to seconds   */
    cstart = clock();    /* Use clock for timing to hundredths of seconds */
#if defined( _DOS )      /* Define _DOS to use _bios_timeofday            */
    _bios_timeofday( _TIME_GETCLOCK, &bstart );
#endif
    /* Do timed Sieve of Erotosthenes. */
    for( loop = 0; loop < 250; ++loop)
        for( num = 0, n = 3; n < 10000; n += 2 )
            if( !mark[n] )
            {
                step = 2 * n;
                for (i = 3 * n; i < 10000; i += step)
                    mark[i] = -1;
                ++num;
            }

    /* End timing and print results. Note that _bios_timeofday doesn't
     * handle midnight rollover.
     */
    time( &tend );
    printf( "\ndifftime:\t\t%4.2f seconds to find %d primes 50 times\n",
             difftime( tend, tstart ), num );
    cend = clock();
    printf( "\nclock:\t\t\t%4.2f seconds to find %d primes 50 times\n",
            ((float)cend - cstart) / CLOCKS_PER_SEC, num );
#if defined( _DOS )
    _bios_timeofday( _TIME_GETCLOCK, &bend );
    printf( "\n_bios_timeofday:\t%4.2f seconds to find %d primes 50 times\n",
            ((float)bend - bstart) / TICKPERSEC, num );
#endif
}
```

// Example: SIGFP.C

```
/* SIGFP.C illustrates setting up floating-point signal routines.
 * Functions illustrated include:
 *      signal          _fpreset        setjmp          longjmp
 *
 * For another example of setting up signal routines, see SIGNAL.C.
 */

#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>
#include <string.h>

jmp_buf mark;                       /* Address for long jump to jump to */
int  fperr;                         /* Global error number */

void __cdecl fphandler( int sig, int num ); /* Prototypes */
void fpcheck( void );

void main()
{
    double n1, n2, r;
    int jmpret;

    /* Set up floating-point error handler. */
    if( signal( SIGFPE, fphandler ) == SIG_ERR )
    {
        fprintf( stderr, "Couldn't set SIGFPE\n" );
        abort();
    }

    /* Save stack environment for return in case of error. First time
     * through, jmpret is 0, so true conditional is executed. If an
     * error occurs, jmpret will be set to -1 and false conditional
     * will be executed.
     */
    jmpret = setjmp( mark );
    if( jmpret == 0 )
    {
        printf( "Test for invalid operation - " );
        printf( "enter two numbers: " );
        scanf( "%lf %lf", &n1, &n2 );

        r = n1 / n2;
        /* This won't be reached if error occurs. */
        printf( "\n\n%4.3g / %4.3g = %4.3g\n", n1, n2, r );

        r = n1 * n2;
        /* This won't be reached if error occurs. */
        printf( "\n\n%4.3g * %4.3g = %4.3g\n", n1, n2, r );
    }
    else
        fpcheck();
}

/* Handles SIGFPE (floating-point error) interrupt. */
void __cdecl fphandler( int sig, int num )
{
```

```
    /* Set global for outside check, since we don't want to do I/O in the
     * handler.
     */
    fperr = num;

    /* Initialize floating-point package. */
    _fpreset();

    /* Restore calling environment and jump back to setjmp. Return -1
     * so that setjmp will return false for conditional test.
     */
    longjmp( mark, -1 );
}

void fpcheck()
{
    char fpstr[30];

    switch( fperr )
    {
    case _FPE_INVALID:
        strcpy( fpstr, "Invalid number" );
        break;

    case _FPE_OVERFLOW:
        strcpy( fpstr, "Overflow" );
        break;

    case _FPE_UNDERFLOW:
        strcpy( fpstr, "Underflow" );
        break;

    case _FPE_ZERODIVIDE:
        strcpy( fpstr, "Divide by zero" );
        break;

    default:
        strcpy( fpstr, "Other floating-point error" );
        break;
    }
    printf( "Error %d: %s\n", fperr, fpstr );
}
```

// Example: SIGNAL.C

```
/* SIGNAL.C illustrates setting up signal interrupt routines.
 * Functions illustrated include signal and raise.
 *
 * Because C I/O functions are not safe inside signal routines,
 * the code uses conditionals to use system-level MS-DOS
 * services. Another option is to set global flags and do any
 * I/O operations outside the signal handler.
 */
#include <stdio.h>
#include <conio.h>
#include <signal.h>
#include <process.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
void ctrlchandler( int sig );         /* Prototypes */
void safeout( char *str );
int  safein( void );
void main( void )
{
   int ch;
   /* Install signal handler to modify CTRL+C behavior. */
   if( signal( SIGINT, ctrlchandler ) == SIG_ERR )
   {
      fprintf( stderr, "Couldn't set SIGINT\n" );
      abort();
   }
   /* Loop prints message to screen asking user to
    * enter Cntl+C--at which point the ctrlchandler
    * signal handler takes control.
    */
   do
   {
      printf( "Press Ctrl+C to enter handler.\n" );
   }
   while( ch = _getch());   /* Discard keystokes */
}
/* A signal handler must take a single argument. The argument can be
 * tested within the handler and thus allows a single signal handler
 * to handle several different signals. In this case, the parameter
 * is included to keep the compiler from generating a warning but is
 * ignored because this signal handler only handles one interrupt:
 * SIGINT (Ctrl+C).
 */
void ctrlchandler( int sig )
{
   int c;
   char str[] = " ";
   /* Disallow CTRL+C during handler. */
   signal( SIGINT, SIG_IGN );
   safeout( "User break - abort processing (y|n)? " );
   c = safein();
   str[0] = (char)c;
   safeout( str );
   safeout( "\r\n" );
   if( (c == 'y') || (c == 'Y') )
      abort();
   else
   {
      /* The CTRL+C interrupt must be reset to our handler because
```

```
       * by default it is reset to the system handler.
       */
      signal( SIGINT, ctrlchandler );
      safeout( "Press Ctrl+C to enter handler.\r\n" );
   }
}
/* Outputs a string using system level calls. */
void safeout( char *str )
{
   union _REGS inregs, outregs;
   inregs.h.ah = 0x0e;
   while( *str )
   {
      inregs.h.al = *str++;
      _int86( 0x10, &inregs, &outregs );
   }
}
/* Inputs a character using system level calls. */
int safein()
{
   return _bios_keybrd( _KEYBRD_READ ) & 0xff;
}
```

// Example: SPAWN.C

```
/* SPAWN.C illustrates the different versions of spawn including:
 *      _spawnl         _spawnle        _spawnlp        _spawnlpe
 *      _spawnv         _spawnve        _spawnvp        _spawnvpe
 *
 * Although SPAWN.C can spawn any program, you can verify how different
 * versions handle arguments and environment by compiling and
 * specifying the sample program ARGS.C. See EXEC.C for examples
 * of the similar exec functions.
 */

#include <stdio.h>
#include <conio.h>
#include <process.h>

char *my_env[] =                        /* Environment for _spawn?e */
{
    "THIS=environment will be",
    "PASSED=to child by the",
    "SPAWN=functions",
    NULL
};

void main()
{
    char *args[4], prog[80];
    int ch, r;

    printf( "Enter name of program to spawn: " );
    gets( prog );
    printf( " 1. _spawnl   2. _spawnle   3. _spawnlp   4. _spawnlpe\n" );
    printf( " 5. _spawnv   6. _spawnve   7. _spawnvp   8. _spawnvpe\n" );
    printf( "Type a number from 1 to 8 (or 0 to quit): " );
    ch = _getche();
    if( (ch < '1') || (ch > '8') )
        exit( -1 );
    printf( "\n\n" );

    /* Arguments for _spawnv? */
    args[0] = prog;
    args[1] = "_spawn??";
    args[2] = "two";
    args[3] = NULL;

    switch( ch )
    {
    case '1':
        r = _spawnl( _P_WAIT, prog, prog, "_spawnl", "two", NULL );
        break;
    case '2':
        r = _spawnle( _P_WAIT, prog, prog, "_spawnle", "two",
                    NULL, my_env );
        break;
    case '3':
        r = _spawnlp( _P_WAIT, prog, prog, "_spawnlp", "two", NULL );
        break;
    case '4':
        r = _spawnlpe( _P_WAIT, prog, prog, "_spawnlpe", "two",
                     NULL, my_env );
        break;
    case '5':
```

```
        r = _spawnv( _P_WAIT, prog, args );
        break;
    case '6':
        r = _spawnve( _P_WAIT, prog, args, my_env );
        break;
    case '7':
        r = _spawnvp( _P_WAIT, prog, args );
        break;
    case '8':
        r = _spawnvpe( _P_WAIT, prog, args, my_env );
        break;
    default:
        break;
    }
    if( r == -1 )
        printf( "Couldn't spawn process" );
    else
        printf( "\nReturned from SPAWN!" );
    exit( r );
}
```

// Example: STAR.C

```
/* STAR.C illustrates functions:                (MS-DOS-only)
 *      _polygon        _getwritemode       _setwritemode
 */

#include <conio.h>
#include <stdlib.h>
#include <graph.h>
#include <math.h>
#include <stdlib.h>

short wmodes[5]  = { _GPSET,   _GPRESET, _GXOR,    _GOR,     _GAND    };
char *wmstr[5]   = { "PSET  ", "PRESET", "XOR   ", "OR    ", "AND   " };

void star( double centerx, double centery, double radius,
           int writemode, int fill );

void main()
{
    short i, tcolor;
    double x, y;

    if( !_setvideomode( _MAXCOLORMODE ) )    /* Find valid graphics mode */
        exit( 1 );

    _setwindow( 0, -50.0, -40.0, 50.0, 40.0 );

    x = y = -10.0;
    star( x, y, 25.0, _GPSET, _GFILLINTERIOR );
    _getch();
    tcolor = _getcolor();
    _setcolor( 2 );
    _floodfill_w( x, y, tcolor );

    for( i = 0; i < 5; i++ )
    {
        _settextposition( 1, 1 );
        _outtext( wmstr[i] );
        star( x += 2.0, y += 1.5, 25.0, wmodes[i], _GBORDER );
        _getch();
    }
    _setvideomode( _DEFAULTMODE );
}

#define PI 3.1415
void star( double centerx, double centery, double radius,
           int writemode, int fill )
{
    int    wm, side;
    struct _wxycoord polyside[5];
    double radians;

    /* Save write mode and set new. */
    wm = _getwritemode();
    _setwritemode( writemode );

    /* Calculate points of star every 144 degrees, then connect them. */
    for( side = 0; side < 5; side++ )
    {
        radians = 144 * PI / 180;
        polyside[side].wx = centerx + (cos( side * radians ) * radius);
```

```
        polyside[side].wy = centery + (sin( side * radians ) * radius);
    }
    _polygon_wxy( fill, polyside, 5 );

    /* Restore original write mode. */
    _setwritemode( wm );
}
```

// Example: STRTONUM.C

```
/* STRTONUM.C illustrates string-to-number conversion functions including:
 *      strtod              strtol              strtoul
 */

#include <stdlib.h>
#include <stdio.h>

void main()
{
    char *string, *stopstring;
    double x;
    long l;
    unsigned long ul;
    int base;

    /* Convert string to double. */
    string = "3.1415926INVALID";
    x = strtod( string, &stopstring );
    printf( "\nString: %s\n", string );
    printf( "\tDouble:\t\t\t%f\n", x );
    printf( "\tScan stopped at:\t%s\n", stopstring );

    /* Convert string to long using bases 2, 4, and 8. */
    string = "-10110134932";
    printf( "\nString: %s\n", string );
    for( base = 2; base <= 8; base *= 2 )
    {
        l = strtol( string, &stopstring, base );
        printf( "\tBase %d signed long:\t%ld\n", base, l );
        printf( "\tScan stopped at:\t%s\n", stopstring );
    }

    /* Convert string to unsigned long using bases 2, 4, and 8. */
    string = "10110134932";
    printf( "\nString: %s\n", string );
    for( base = 2; base <= 8; base *= 2 )
    {
        ul = strtoul( string, &stopstring, base);
        printf("\tBase %d unsigned long:\t%ld\n", base, ul );
        printf("\tScan stopped at:\t%s\n", stopstring );
    }
}
```

// Example: SWAB.C

```
/* SWAB.C illustrates:
 *      _swab
 */

#include <stdlib.h>
#include <stdio.h>

char from[] = "BADCFEHGJILKNMPORQTSVUXWZY";
char to[] =   ".........................";

void main()
{
    printf( "Before:\t%s\n\t%s\n\n", from, to );
    _swab( from, to, sizeof( from ) );
    printf( "After:\t%s\n\t%s\n\n", from, to );
}
```

// Example: SYSCALL.C

```
/* SYSCALL.C illustrates system calls to DOS and BIOS interrupts using
 * functions:
 *      _intdos          _intdosx         _bdos               (MS-DOS-only)
 *      _int86           _int86x          _segread
 *
 * The _int86x call is not specifically shown in the example, but
 * it is the same as _intdosx, except that an interrupt number must be
 * supplied.
 */

#include <dos.h>
#include <stdio.h>
#define LPT1 0

union _REGS inregs, outregs;
struct _SREGS segregs;

void main()
{
    char __far *buffer = "Dollar-sign terminated string\n\r\n\r$";
    char __far *p;

    /* Get MS-DOS version using MS-DOS function 0x30. */
    inregs.h.ah = 0x30;
    _intdos( &inregs, &outregs );
    printf( "\nMajor: %d\tMinor: %d\tOEM number: %d\n\n",
        outregs.h.al, outregs.h.ah, outregs.h.bh );

    /* Print a $-terminated string on the screen using MS-DOS function 0x9. */
    inregs.h.ah = 0x9;
    inregs.x.dx = _FP_OFF( buffer );
    segregs.ds = _FP_SEG( buffer );
    _intdosx( &inregs, &outregs, &segregs );

    _segread( &segregs );
    printf( "Segments:\n\tCS\t%.4x\n\tDS\t%.4x\n\tES\t%.4x\n\tSS\t%.4x\n\n",
            segregs.cs, segregs.ds, segregs.es, segregs.ss );

    /* Make sure printer is available. Fail if any error bit is on,
     * or if either operation bit is off.
     */
    inregs.h.ah = 0x2;
    inregs.x.dx = LPT1;
    _int86( 0x17, &inregs, &outregs );
    if(  (outregs.h.ah & 0x29) || !(outregs.h.ah & 0x80) ||
                                  !(outregs.h.ah & 0x10) )
        printf( "Printer not available." );
    else
    {
        /* Output a string to the printer using MS-DOS function 0x5. */
        for( p = buffer; *p != '$'; p++ )
            _bdos( 0x05, *p, 0 );

        /* Do print screen. */
        inregs.h.ah = 0x05;
        _int86( 0x05, &inregs, &outregs );
    }
}
```

// Example: SYSINFO.C

```
/* SYSINFO.C illustrates miscellaneous DOS and BIOS status functions
 * including:
 *      _dos_getdrive       _dos_setdrive       _dos_getdiskfree
 *      _bios_memsize       _bios_equiplist     _bios_printer
 *                                                          (MS-DOS-only)
 * See DISK.C for another example of _dos_getdiskfree.
 *
 * Also illustrated:
 *      union               bitfield struct
 */

#include <dos.h>
#include <bios.h>
#include <conio.h>
#include <stdio.h>
#define LPT1 0

void main()
{
    struct _diskfree_t drvinfo;
    unsigned drive, drivecount, memory, pstatus;
    union
    {                                   /* Access equiment either as:    */
        unsigned u;                     /*   unsigned or                 */
        struct                          /*   bit fields                  */
        {
            unsigned diskflag : 1;      /* Diskette drive installed?     */
            unsigned coprocessor : 1;   /* Coprocessor? (except on PC)   */
            unsigned sysram : 2;        /* RAM on system board           */
            unsigned video : 2;         /* Startup video mode            */
            unsigned disks : 2;         /* Drives 00=1, 01=2, 10=3, 11=4 */
            unsigned dma : 1;           /* 0=Yes, 1=No (1 for PC Jr.)    */
            unsigned comports : 3;      /* Serial ports                  */
            unsigned game : 1;          /* Game adapter installed?       */
            unsigned modem : 1;         /* Internal modem?               */
            unsigned printers : 2;      /* Number of printers            */
        } bits;
    } equip;
    char y[] = "YES", n[] = "NO";

    /* Get current drive. */
    _dos_getdrive( &drive );
    printf( "Current drive:\t\t\t%c:\n", 'A' + drive - 1 );

    /* Set drive to current drive in order to get number of drives. */
    _dos_setdrive( drive, &drivecount );

    _dos_getdiskfree( drive, &drvinfo );
    printf( "Disk space free:\t\t%ld\n",
            (long)drvinfo.avail_clusters *
            drvinfo.sectors_per_cluster *
            drvinfo.bytes_per_sector );

    /* Get new drive and number of logical drives in system. */
    _dos_getdrive( &drive );
    printf( "Number of logical drives:\t%d\n", drivecount );

    memory = _bios_memsize();
    printf( "Memory:\t\t\t\t%dK\n", memory );
```

```
    equip.u = _bios_equiplist();

    printf( "Disk drive:\t\t\t%s\n", equip.bits.diskflag ? y : n );
    printf( "Coprocessor:\t\t\t%s\n", equip.bits.coprocessor ? y : n );
    printf( "Game adapter:\t\t\t%s\n", equip.bits.game ? y : n );
    printf( "Serial ports:\t\t\t%d\n", equip.bits.comports );
    printf( "Number of printers:\t\t%d\n", equip.bits.printers );

    /* Fail if any error bit is on, or if either operation bit is off. */
    pstatus = _bios_printer( _PRINTER_STATUS, LPT1, 0 );
    if( (pstatus & 0x29) || !(pstatus & 0x80) || !(pstatus & 0x10) )
        pstatus = 0;
    else
        pstatus = 1;
    printf( "Printer available:\t\t%s\n", pstatus ? y : n );
}
```

// Example: TABLE.C

```
/* TABLE.C illustrates reading and writing formatted file data using
 * functions:
 *    fprintf   fscanf
 *
 * For more examples of formatted input and output, see SCANF.C and
 * PRINTF.C.
 */

#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <stdlib.h>

void main()
{
    char buf[128];
    FILE *ftable;
    long l, tl;
    float fp, tfp;
    int  i, c = 'A';

    /* Open an existing file for reading. Fail if file doesn't exist. */
    if( (ftable = fopen( "table.smp", "r")) != NULL )
    {
        printf( "Reading table file\n" );
        /* Read data from file and total it. */
        for( i = 0, tl = 0L, tfp = 0.0f; i < 10; i++ )
        {
            fscanf( ftable, "\t%s %c: %ld %f\n", buf, &c, &l, &fp );
            tl += l;
            tfp += fp;
            printf( "\t%s %c: %7ld %9.2f\n", buf, c, l, fp );
        }
        printf( "\n\tTotal:  %7ld %9.2f\n", tl, tfp );
        remove( "table.smp" );
    }
    else
    {
        /* File did not exist. Create it for writing. */
        if( (ftable = fopen( "table.smp", "w" )) == NULL )
            exit( 1 );

        /* Write table to file. */
        for( i = 0, l = 99999L, fp = 3.14f; i < 10; i++ )
            fprintf( ftable, "\tLine %c: %7ld %9.2f\n",
                     c++, l /= 2, fp *= 2 );

        printf( "Created table file. Run again to read it.\n" );
    }
    fclose( ftable );
    exit( 0 );
}
```

// Example: TEMPNAME.C

```
/* TEMPNAME.C illustrates:
 *      tmpnam          _tempnam
 */

#include <stdio.h>

void main()
{
    char *name1, name2[L_tmpnam], *name3;
    int c;

    /* Create several temporary file names using internal buffer. */
    for( c = 0; c < 5; c++ )
        if( (name1 = tmpnam( NULL )) != NULL )
            printf( "%s is a safe temporary file name.\n", name1 );

    /* Create a temporary file name using external buffer. */
    if( tmpnam( name2 ) != NULL )
        printf( "%s is a safe temporary file name.\n", name2 );

    /* Create a temporary file name with prefix TEMP and place it in
     * the first of these directories that exists:
     *   1. TMP environment directory
     *   2. C:\TEMPFILE
     *   3. P_tmpdir directory (defined in stdio.h)
     */
    if( (name3 = _tempnam( "C:\\TEMPFILE", "TEMP" )) != NULL )
        printf( "%s is a safe temporary file name.\n", name3 );
}
```

// Example: TEXT.C

```
/* TEXT.C illustrates text output functions including:
 *    _gettextcolor   _getbkcolor   _gettextposition   _outtext
 *    _settextcolor   _setbkcolor   _settextposition   _clearscreen
 *
 * See MODES.C for another use of _outtext and WINDOW.C for another
 * use of _clearscreen.
 */

#include <conio.h>
#include <stdio.h>
#include <graph.h>

char buffer [80];

void main()
{
    short blink, fgd, oldfgd;
    long bgd, oldbgd;
    struct _rccoord oldpos;

    /* Save original foreground, background, and text position. */
    oldfgd = _gettextcolor();
    oldbgd = _getbkcolor();
    oldpos = _gettextposition();
    _clearscreen( _GCLEARSCREEN );

    /* First time no blink, second time blinking. */
    for( blink = 0; blink <= 16; blink += 16 )
    {

        /* Loop through 8 background colors. */
        for( bgd = 0; bgd < 8; bgd++ )
        {
            _setbkcolor( bgd );
            _settextposition( (short)(bgd + ((blink/16)*9)+3), (short)1 );
            _settextcolor( 15 );
            sprintf(buffer, "Back: %d Fore:", bgd );
            _outtext( buffer );

            /* Loop through 16 foreground colors. */
            for( fgd = 0; fgd < 16; fgd++ )
            {
                _settextcolor( fgd + blink );
                sprintf( buffer, " %2d ", fgd + blink );
                _outtext( buffer );
            }
        }
    }
    _getch();

    /* Restore original foreground and background. */
    _settextcolor( oldfgd );
    _setbkcolor( oldbgd );
    _clearscreen( _GCLEARSCREEN );
    _settextposition( oldpos.row, oldpos.col );
}
```

// Example:TIMES.C

```
/* TIMES.C illustrates various time and date functions including:
 *      time            _ftime          ctime       asctime
 *      localtime       gmtime          mktime      _tzset
 *      _strtime        _strdate        strftime
 *
 * Also the global variable:
 *      _tzname
 */

#include <time.h>
#include <stdio.h>
#include <sys\types.h>
#include <sys\timeb.h>
#include <string.h>

void main()
{
    char tmpbuf[128], ampm[] = "AM";
    time_t ltime;
    struct _timeb tstruct;
    struct tm *today, *gmt, xmas = { 0, 0, 12, 25, 11, 91 };

    /* Set time zone from TZ environment variable. If TZ is not set,
     * PST8PDT is used (Pacific standard time, daylight savings).
     */
    _tzset();

    /* Display MS-DOS-style date and time. */
    _strtime( tmpbuf );
    printf( "MS-DOS time:\t\t\t\t%s\n", tmpbuf );
    _strdate( tmpbuf );
    printf( "MS-DOS date:\t\t\t\t%s\n", tmpbuf );

    /* Get UNIX-style time and display as number and string. */
    time( &ltime );
    printf( "Time in seconds since UCT 1/1/70:\t%ld\n", ltime );
    printf( "UNIX time and date:\t\t\t%s", ctime( &ltime ) );

    /* Display UCT. */
    gmt = gmtime( &ltime );
    printf( "Universal coordinated time:\t\t%s", asctime( gmt ) );

    /* Convert to time structure and adjust for PM if necessary. */
    today = localtime( &ltime );
    if( today->tm_hour > 12 )
    {
   strcpy( ampm, "PM" );
   today->tm_hour -= 12;
    }
    if( today->tm_hour == 0 )  /* Adjust if midnight hour. */
   today->tm_hour = 12;

    /* Note how pointer addition is used to skip the first 11 characters
     * and printf is used to trim off terminating characters.
     */
    printf( "12-hour time:\t\t\t\t%.8s %s\n",
       asctime( today ) + 11, ampm );

    /* Print additional time information. */
    _ftime( &tstruct );
```

```
    printf( "Plus milliseconds:\t\t\t%u\n", tstruct.millitm );
    printf( "Zone difference in seconds from UCT:\t%u\n", tstruct.timezone );
    printf( "Time zone name:\t\t\t\t%s\n", _tzname[0] );
    printf( "Daylight savings:\t\t\t%s\n", tstruct.dstflag ? "YES" : "NO" );

    /* Make time for noon on Christmas, 1992. */
    if( mktime( &xmas ) != (time_t)-1 )
   printf( "Christmas\t\t\t\t%s\n", asctime( &xmas ) );

    /* Use time structure to build a customized time string. */
    today = localtime( &ltime );

    /* Use strftime to build a customized time string. */
    strftime( tmpbuf, 128,
         "Today is %A, day %d of %B in the year %Y.\n", today );
    printf( tmpbuf );
}
```

// Example: TOKEN.C

```
/* TOKEN.C illustrates tokenizing and searching for any of several
 * characters. Functions illustrated include:
 *      strcspn         strspn          strpbrk         strtok
 */

#include <string.h>
#include <stdio.h>

void main()
{
    char string[100], vowels[] = "aeiouAEIOU", seps[] = " \t\n,";
    char *p;
    int  count;

    printf( "Enter a string: " );
    gets( string );

    /* Delete one word at a time. */
    p = string;
    while( *p )
    {
        printf( "String remaining: %s\n", p );
        p += strcspn( p, seps );    /* Find next separator     */
        p += strspn( p, seps );     /* Find next nonseparator  */
    }

    /* Count vowels. */
    p = string;
    count = 0;
    do
    {
        p = strpbrk( p, vowels );   /* Find next vowel         */
        count++;
    } while( *(++p) );
    printf( "\nVowels in string: %d\n\n", count - 1 );

    /* Break into tokens. */
    count = 0;
    p = strtok( string, seps );     /* Find first token        */
    while( p != NULL )
    {
        printf( "Token %d: %s\n", ++count, p );
        p = strtok( NULL, seps );   /* Find next token         */
    }
}
```

// Example: TRIG.C

```
/* TRIG.C illustrates trigonometric functions including:
 *    cos          cosh          acos
 *    sin          sinh          asin
 *    tan          tanh          atan          atan2
 */

#include <math.h>
#include <float.h>
#include <stdio.h>
#include <stdlib.h>

void main()
{
    double x, rx, y;

    do
    {
        printf( "\nEnter a real number between 1 and -1: " );
        scanf( "%lf", &x );
    } while( (x > 1.0) || (x < -1.0) );

    printf("\nFunction\tResult for %2.2f\n\n", x );
    if( (x <= 1.0) && (x >= -1.0) )
    {
        printf( "acos\t\t%2.2f\n", acos( x ) );
        printf( "asin\t\t%2.2f\n", asin( x ) );
    }
    if( (rx = cos( x )) && (errno != ERANGE) )
        printf( "cos\t\t%2.2f\n", rx );
    else
        errno = 0;
    if( (rx = sin( x )) && (errno != ERANGE) )
        printf( "sin\t\t%2.2f\n", rx );
    else
        errno = 0;
    if( (rx = cosh( x )) && (errno != ERANGE) )
        printf( "cosh\t\t%2.2f\n", rx );
    else
        errno = 0;
    if( (rx = sinh( x )) && (errno != ERANGE) )
        printf( "sinh\t\t%2.2f\n", rx );
    else
        errno = 0;

    printf( "\nEnter a real number of any size: " );
    scanf( "%lf", &x );
    printf("\nFunction\tResult for %2.2f\n\n", x );
    printf( "atan\t\t%2.2f\n", atan( x ) );
    if( (rx = tan( x )) && (errno != ERANGE) )
        printf( "tan\t\t%2.2f\n", rx );
    else
        errno = 0;
    printf( "tanh\t\t%2.2f\n", tanh( x ) );

    printf( "\nEnter another real number of any size: " );
    scanf( "%lf", &y );
    printf("\nFunction\tResult for %2.2f and %2.2f\n\n", x, y );
    if( (rx = atan2( x, y )) != 0 )
        printf( "atan2\t\t%2.2f\n", rx );
    else
```

```
        errno = 0;
}
```

// Example: TYPEIT.C

```
/* TYPEIT.C illustrates reassigning handles and streams using functions:
 *      freopen         _dup            _dup2
 *
 * The example also illustrates:
 *      _setargv
 *
 * To make the program handle wildcards, link it with the SETARGV.OBJ
 * file. You can do this in the development environment by creating a
 * project containing TYPEIT.C and SETARGV.OBJ (include the path or put
 * in current directory). You must also turn off the Extended Dictionary
 * option within the environment or use the /NOE linker option outside
 * the environment. For example:
 *    CL typeit.c setargv /link /NOE
 */

#include <stdio.h>
#include <conio.h>
#include <io.h>
#include <process.h>

void main( int argc, char **argv )
{
    FILE *ftmp;
    int htmp;

    /* Duplicate handle of stdin. Save the original to restore later. */
    htmp = _dup( _fileno( stdin ) );

    /* Process each command line argument. */
    while( *(++argv) != NULL )
    {
        /* Original stdin used for _getch. */
        printf( "Press any key to display file: %s\n", *argv );
        _getch();

        /* Reassign stdin to input file. */
        ftmp = freopen( *argv, "rb", stdin );

        if( (ftmp == NULL) || (htmp == -1 ) )
        {
            _dup2( htmp, _fileno( stdin ) );
            perror( "Can't reassign standard input" );
            exit( 1 );
        }

        /* Spawn MORE, which will receive open input file as its standard
         * input. MORE can be the MS-DOS MORE or the sample program MORE.C.
         */
        _spawnlp( _P_WAIT, "MORE", "MORE", NULL );

        /* Reassign stdin back to original so that we can use the
         * original stdin to get a key.
         */
        _dup2( htmp, _fileno( stdin ) );
    }
    exit( 0 );
}
```

// Example: UNGET.C

```
/* UNGET.C illustrates getting and ungetting characters from the console.
 * Functions illustrated include:
 *      _getch          _ungetch        _putch
 */

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

void skiptodigit( void );
int getnum( void );

void main()
{
    int c, array[5];

    /* Get five numbers from console. Then display them. */
    printf( "Enter five numbers:\n" );
    for( c = 0; c < 5; c++ )
    {
        skiptodigit();
        array[c] = getnum();
        printf( "\t" );
    }
    for( c  = 0; c < 5; c++ )
        printf( "\n\r%d", array[c] );
    printf( "\n" );
}

/* Converts digit characters into a number until a nondigit is received */
int getnum()
{
    int ch, num =  0;

    while( isdigit( ch = _getch() ) )
    {
        _putch( ch );                       /* Display digit     */
        num = (num * 10) + ch  - '0';       /* Convert to number */
    }
    _ungetch( ch );                         /* Put nondigit back */
    return num;                             /* Return result     */
}

/* Throws away nondigit characters */
void skiptodigit()
{
    int ch;

    while( !isdigit( ch = _getch() ) )
        ;
    _ungetch( ch );
}
```

// Example: VARARG.C

```
/* VARARG.C illustrates passing a variable number of arguments using the
 * following macros:
 *      va_start        va_arg      va_end
 *
 * Also the ANSI and UNIX type:
 *      va_list
 * and the UNIX types:
 *      va_alist        va_dcl
 */

/* Define symbol UNIX for UNIX version. */
#include <stdio.h>
#if !defined( UNIX )     /* ANSI-compatible version          */
#include <stdarg.h>
int average( int first, ... );
#else                    /* UNIX-compatible version          */
#include <varargs.h>
int average();           /* To avoid compiler warnings,      */
#endif                   /* don't check argument types.      */

void main()
{
    /* Call with 3 integers (-1 is used as terminator). */
    printf( "Average is: %d\n", average( 2, 3, 4, -1 ) );

    /* Call with 4 integers. */
    printf( "Average is: %d\n", average( 5, 7, 9, 11, -1 ) );

    /* Call with just -1 terminator. */
    printf( "Average is: %d\n", average( -1 ) );
}

/* Returns the average of a variable list of integers. */
#if !defined( UNIX )                /* ANSI compatible version        */
int average( int first, ... )
{
    int count = 0, sum = 0, i = first;
    va_list marker;

    va_start( marker, first );      /* Initialize variable arguments  */
    while( i != -1 )
    {
        sum += i;
        count++;
        i = va_arg( marker, int);
    }
    va_end( marker );               /* Reset variable arguments       */
    return( sum ? (sum / count) : 0 );
}
#else       /* UNIX-compatible version must use old-style definition. */
int average( va_alist )
va_dcl
{
    int i, count, sum;
    va_list marker;

    va_start( marker );             /* Initialize variable arguments  */
    for( sum = count = 0; (i = va_arg( marker, int)) != -1;  count++ )
        sum += i;
    va_end( marker );               /* Reset variable arguments       */
```

```
    return( sum ? (sum / count) : 0 );
}
#endif
```

// Example: VCNT.C

```
/* VCNT.C: This program locks a block of virtual memory five times with
 * _vlock, and then unlocks it five times with _vunlock, calling
 * _vlockcnt after each operation to report the number of locks held.
 */

#include <stdio.h>
#include <stdlib.h>
#include <vmemory.h>

void main()
{
    int i, count;
    _vmhnd_t handle;
    int __far *buffer;

    if ( !_vheapinit( 0, _VM_ALLDOS, _VM_XMS | _VM_EMS ) )
    {
        printf( "Could not initialize virtual memory manager. \n" );
        exit( -1 );
    }

    if ( (handle = _vmalloc( 100 * sizeof(int) )) == _VM_NULL )
    {
        _vheapterm();
        exit( -1 );
    }

    printf( "Block of virtual memory allocated.\n" );

    printf( "Locking...\n" );
    for ( i = 0; i < 5; i++ )
    {
        if ( (buffer = (int __far *)_vlock( handle )) == NULL )
        {
            _vheapterm();
            exit( -1 );
        }

        count = _vlockcnt( handle );
        printf( "%d locks held.\n", count );
    }

    printf("Unlocking...\n" );
    for ( i = 0; i < 5; i++ )
    {
        _vunlock( handle, _VM_CLEAN );

        count = _vlockcnt( handle );
        printf( "%d locks held.\n", count );
    }

    _vfree( handle );
    _vheapterm();
}
```

// Example: VLOAD.C

```
/* VLOAD.C: This program loads a block of virtual memory with _vload,
 * writes to it, and loads in a new block. It then reloads the first
 * block and verifies that its contents haven't changed.
 */

#include <stdio.h>
#include <stdlib.h>
#include <vmemory.h>

void main()
{
    int i, flag;
    _vmhnd_t handle1,
             handle2;
    int __far *buffer1;
    int __far *buffer2;

    if ( !_vheapinit( 0, _VM_ALLDOS, _VM_XMS | _VM_EMS ) )
    {
        printf( "Could not initialize virtual memory manager. \n" );
        exit( -1 );
    }

    if ( ( (handle1 = _vmalloc( 100 * sizeof(int) )) == _VM_NULL ) ||
         ( (handle2 = _vmalloc( 100 * sizeof(int) )) == _VM_NULL )   )
    {
        _vheapterm();
        exit( -1 );
    }

    printf( "Two blocks of virtual memory allocated.\n" );

    if ( (buffer1 = (int __far *)_vload( handle1, _VM_DIRTY )) == NULL )
    {
        _vheapterm();
        exit( -1 );
    }

    printf( "buffer1 loaded: valid until next call to VM manager.\n" );
    for ( i = 0; i < 100; i++ )      /* write to buffer1 */
        buffer1[i] = i;

    if ( (buffer2 = (int __far *)_vload( handle2, _VM_DIRTY )) == NULL )
    {
        _vheapterm();
        exit( -1 );
    }

    printf( "buffer2 loaded. buffer 1 no longer valid.\n" );

    if ( (buffer1 = (int __far *)_vload( handle1, _VM_CLEAN )) == NULL )
    {
        _vheapterm();
        exit( -1 );
    }

    printf( "buffer1 reloaded.\n" );

    flag = 0;
    for ( i = 0; i < 100; i++ )
```

```
        if ( buffer1[i] != i )
            flag = 1;

    if ( !flag )
        printf( "contents of buffer1 verified.\n" );

    _vfree( handle1 );
    _vfree( handle2 );
    _vheapterm();
}
```

// Example: VLOCK.C

```
/* VLOCK.C: This program locks a block of virtual memory using _vlock,
 * writes to it, loads in a new block with _vload, and then verifies
 * that the contents of the locked block are still accessible. It then
 * unlocks the block with _vunlock.
 */

#include <stdio.h>
#include <stdlib.h>
#include <vmemory.h>

void main()
{
    int i, flag;
    _vmhnd_t handle1,
             handle2;
    int __far *buffer1;
    int __far *buffer2;

    if ( !_vheapinit( 0, _VM_ALLDOS, _VM_XMS | _VM_EMS ) )
    {
        printf( "Could not initialize virtual memory manager. \n" );
        exit( -1 );
    }

    if ( ( (handle1 = _vmalloc( 100 * sizeof(int) )) == _VM_NULL ) ||
         ( (handle2 = _vmalloc( 100 * sizeof(int) )) == _VM_NULL )   )
    {
        _vheapterm();
        exit( -1 );
    }

    printf( "Two blocks of virtual memory allocated.\n" );

    if ( (buffer1 = (int __far *)_vlock( handle1 )) == NULL )
    {
        _vheapterm();
        exit( -1 );
    }

    printf( "buffer1 locked: valid until unlocked.\n" );
    for ( i = 0; i < 100; i++ )      /* write to buffer1 */
        buffer1[i] = i;

    if ( (buffer2 = (int __far *)_vload( handle2, _VM_DIRTY )) == NULL )
    {
        _vheapterm();
        exit( -1 );
    }

    printf( "buffer2 loaded. buffer 1 still valid.\n" );

    flag = 0;
    for ( i = 0; i < 100; i++ )
        if ( buffer1[i] != i )
            flag = 1;

    if ( !flag )
        printf( "contents of buffer1 verified.\n" );

    _vunlock( handle1, _VM_DIRTY );
```

```
    _vfree( handle1 );
    _vfree( handle2 );
    _vheapterm();
}
```

// Example: VRSIZE.C

```
/* VRSIZE.C: This program allocates a block of virtual memory with
 * _vmalloc and uses _vmsize to display the size of that block. Next,
 * it uses _vrealloc to expand the amount of virtual memory and calls
 * _vmsize again to display the new amount of memory allocated.
 */

#include <stdio.h>
#include <stdlib.h>
#include <vmemory.h>

void main()
{
    _vmhnd_t handle;
    unsigned long block_size;

    if ( !_vheapinit( 0, _VM_ALLDOS, _VM_XMS | _VM_EMS ) )
    {
        printf( "Could not initialize virtual memory manager.\n" );
        exit( -1 );
    }

    printf( "Requesting 100 bytes of virtual memory.\n" );
    if ( (handle = _vmalloc( 100 )) == _VM_NULL )
    {
        _vheapterm();
        exit( -1 );
    }

    block_size = _vmsize( handle );
    printf( "Received %d bytes of virtual memory.\n", block_size );

    printf( "Resizing block to 200 bytes. \n" );
    if ( (handle = _vrealloc( handle, 200 )) == _VM_NULL )
    {
        _vheapterm();
        exit( -1 );
    }

    block_size = _vmsize( handle );
    printf( "Block resized to %d bytes.\n", block_size );

    _vfree( handle );
    _vheapterm();
}
```

// Example: WABOUT.C

```
/* WABOUT.C - Demonstrate setting the
 * About dialog box string with _wabout
 */
#include <stdio.h>
#include <io.h>
char string[512];
void main( void )
{
   int nRes;
   for ( ; ; )
   {
      printf( "\nEnter the About string: " );
      gets(string);
      printf( "\nAbout string = %s\n", string );
      printf( "Setting about string..." );
      nRes = _wabout( string );
      printf( "_wabout result = %i", nRes );
      printf( "\nTry 'About' in the Help menu\n" );
   }
}
```

// Example: WCLOSE.C

```
/* WCLOSE.C - Demonstrate closing QuickWin windows */
#include <fcntl.h>
#include <stdio.h>
#include <io.h>
#include <stdlib.h>
#define PERSISTFLAG _WINNOPERSIST
#define OPENFLAGS   _O_RDWR
void main( void )
{
   int wfh;   /* File handle for window */
   int nRes;  /* Window write results */
   int wc;    /* Window closure results */
   struct _wopeninfo wininfo; /* Open information */
   /* Set up window open information */
   wininfo._version = _QWINVER;
   wininfo._title = "Window Closing";
   wininfo._wbufsize = _WINBUFDEF;
   /* Open a window with _wopen */
   wfh = _wopen( &wininfo, NULL, OPENFLAGS );
   if( wfh == -1 )
   {
      printf( "***ERROR: On _wopen\n" );
      exit( -1 );
   }
   /* Write in the window */
   nRes = write( wfh, "Windows Everywhere!\n", 20 );
   /* Close the window with _wclose */
   wc = _wclose( wfh, PERSISTFLAG );
   exit( 0 );
}
```

// Example: WCSTOMBS.CPP

```
/* WCSTOMBS.CPP illustrates the behavior of the wcstombs function */
#include <stdio.h>
#include <stdlib.h>
void main( void )
{
   int      i;
   char    *pmbbuf   = (char *)malloc( MB_CUR_MAX );
   wchar_t *pwcEOL   = L'\0';
   wchar_t *pwchello = L"Hello, world.";
   printf( "Convert entire wide-character string:\n" );
   i = wcstombs( pmbbuf, pwchello, MB_CUR_MAX );
   printf( "\tCharacters converted: %u\n", i );
   printf( "\tMultibyte character: %s\n\n", pmbbuf );
   printf( "Attempt to convert null character:\n" );
   i = wcstombs( pmbbuf, pwcEOL, MB_CUR_MAX );
   printf( "\tCharacters converted: %u\n", i );
   printf( "\tMultibyte character: %s\n\n", pmbbuf );
}
```

// Example: WCTOMB.CPP

```
/* WCTOMB.CPP illustrates the behavior of the wctomb function */
#include <stdio.h>
#include <stdlib.h>
void main( void )
{
   int i;
   wchar_t wc = L'a';
   char *pmbnull = NULL;
   char *pmb = (char *)malloc( sizeof( char ) );

   printf( "Convert a wide character:\n" );
   i = wctomb( pmb, wc );
   printf( "\tCharacters converted: %u\n", i );
   printf( "\tMultibyte character: %.1s\n\n", pmb );
   printf( "Attempt to convert when target is NULL:\n" );
   i = wctomb( pmbnull, wc );
   printf( "\tCharacters converted: %u\n", i );
   printf( "\tMultibyte character: %.1s\n", pmbnull );
}
```

// Example: WGETFOC.C

```
/* WGETFOC.C - Demonstrate testing which QuickWin
 * window is the active window with _wgetfocus
 */
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMWINS      4    /* Number of windows */
#define OPENFLAGS   "w"   /* Access permission */
void main( void )
{
   int i, nRes;
   int sf, gf;          /* Set/Get focus results */
   FILE *wins[NUMWINS]; /* Array of file pointers */
   /* Open NUMWINS windows */
   /* NULL arguments accept default characteristics */
   for( i = 0; i < NUMWINS; i++ )
   {
      wins[i] = _fwopen( NULL, NULL, OPENFLAGS );
      if( wins[i] == NULL )
      {
         printf( "***ERROR: On _fwopen #%i\n", i );
         exit( -1 );
      }
      /* Write in each window */
      nRes = fprintf( wins[i], "Windows!\n" );
   }
   /* Tile child windows with _wmenuclick */
   nRes = _wmenuclick( _WINTILE );
   if( nRes == -1 )
   {
      printf( "***ERROR: _wmenuclick\n" );
      exit( -1 );
   }
   /* Pass the focus from window to window */
   for( i = 0; i < NUMWINS; i++ )
   {
      sf = _wsetfocus( _fileno( wins[i] ) );
      gf = _wgetfocus();
      if(( sf == -1 ) || ( gf == -1 )  || ( gf != _fileno( wins[i] ) ) )
      {
         printf( "***ERROR: _wsetfocus/_wgetfocus\n" );
         exit( -1 );
      }
   }
   nRes = _fcloseall();
   exit( 0 );
}
```

// Example: WGETSIZE.C

```
/* WGETSIZE.C - Demonstrate getting the
 * size of a QuickWin window on the screen
 */
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#define OPENFLAGS    "w"         /* Access permission */
#define PERSISTFLAG _WINPERSIST  /* Keep on screen */
void main( void )
{
   int nRes;               /* Result */
   FILE *wp;               /* File pointer */
   struct _wsizeinfo ws;   /* Size information */
   /* Open a window */
   /* NULL arguments accept default characteristics */
   wp = _fwopen( NULL, NULL, OPENFLAGS );
   if( wp == NULL )
   {
      printf( "***ERROR:_fwopen\n" );
      exit( -1 );
   }
   /* Get the window's size and screen position */
   ws._version = _QWINVER;
   nRes = _wgetsize( _fileno( wp ), _WINCURRREQ, &ws );
   if( nRes == -1 )
   {
      printf( "***ERROR: _wgetsize\n" );
      exit( -1 );
   }
   nRes = fprintf( wp, "Size:\n" );
   nRes = fprintf( wp, "  Upper Left: x = %d\n", ws._x );
   nRes = fprintf( wp, "              y = %d\n", ws._y );
   nRes = fprintf( wp, "  Width:      w = %d\n", ws._w );
   nRes = fprintf( wp, "  Height:     h = %d\n", ws._h );
   nRes = _wclose( _fileno( wp ), PERSISTFLAG );
   exit( 0 );
}
```

// Example: WGSCRBUF.C

```
/* WGSCRBUF.C - Demonstrate examining the current
 * size of a QuickWin window's screen buffer
 */
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMWINS      4    /* Number of windows */
#define OPENFLAGS   "w"   /* Access permission */
void main( void )
{
   long nSize;           /* Size of screen buffer */
   int nRes;             /* Write result */
   FILE *wp;             /* File pointer */
   /* Open a window */
   /* NULL arguments accept default characteristics */
   wp = _fwopen( NULL, NULL, OPENFLAGS );
   if( wp == NULL )
   {
      printf( "***ERROR:_fwopen\n" );
      exit( -1 );
   }
   /* Get the size of its screen buffer */
   nSize = _wgetscreenbuf( _fileno( wp ) );
   nRes = fprintf( wp, "Screen buffer holds %li chars\n", nSize );
   nRes = _wclose( _fileno( wp ), _WINPERSIST );
   exit( 0 );
}
```

// Example: WINDOW.C

```
/* WINDOW.C illustrates windows and coordinate systems using the
 * following functions:                                  (MS-DOS-only)
 *    _setviewport   _setvieworg    _setcliprgn     _setwindow
 *    _rectangle     _rectangle_w   _rectangle_wxy  _clearscreen
 *    _ellipse       _ellipse_w     _ellipse_wxy
 *
 * Although not all illustrated here, functions ending in _w
 * are similar to _rectangle_w and _ellipse_w; functions ending
 * in _wxy are similar to _rectangle_wxy and _ellipse_wxy.
 */

#include <conio.h>
#include <graph.h>
#include <stdlib.h>

void main()
{
    short xhalf, yhalf, xquar, yquar;
    struct _wxycoord upleft, botright;
    struct _videoconfig vc;

    if( !_setvideomode( _MAXRESMODE ) ) /* Find a valid graphics mode */
        exit( 1 );
    _getvideoconfig( &vc );

    xhalf = vc.numxpixels / 2;
    yhalf = vc.numypixels / 2;
    xquar = xhalf / 2;
    yquar = yhalf / 2;

    /* First window - integer physical coordinates */
    _setviewport( 0, 0, xhalf - 1, yhalf - 1 );
    _rectangle( _GBORDER, 0,  0, xhalf - 1, yhalf - 1 );
    _ellipse( _GFILLINTERIOR, xquar / 4, yquar / 4,
                              xhalf - (xquar / 4), yhalf - (yquar / 4) );
    _getch();
    _clearscreen( _GVIEWPORT );
    _rectangle( _GBORDER, 0,  0, xhalf - 1, yhalf - 1 );

    /* Second window - integer world coordinates with clip region */
    _setcliprgn( xhalf, 0, vc.numxpixels, yhalf );
    _setvieworg( xhalf + xquar - 1, yquar - 1 );
    _rectangle( _GBORDER, -xquar + 1, -yquar + 1, xquar, yquar );
    _ellipse( _GFILLINTERIOR, (-xquar * 3) / 4, (-yquar * 3) / 4,
                              (xquar * 3) / 4, (yquar * 3) / 4 );
    _getch();
    _clearscreen( _GVIEWPORT );
    _rectangle( _GBORDER, -xquar + 1, -yquar + 1, xquar, yquar );

    /* Third window */
    _setviewport( xhalf, yhalf, vc.numxpixels - 1, vc.numypixels - 1 );
    _setwindow( 0, -4.0, -5.0, 4.0, 5.0 );
    _rectangle_w( _GBORDER, -4.0, -5.0, 4.0, 5.0 );
    _ellipse_w( _GFILLINTERIOR, -3.0, -3.5, 3.0, 3.5 );
    _getch();
    _clearscreen( _GVIEWPORT );
    _rectangle_w( _GBORDER, -4.0, -5.0, 4.0, 5.0 );

    /* Fourth window */
    _setviewport( 0, yhalf, xhalf - 1, vc.numypixels - 1 );
```

```
    _setwindow( 0, -4.0, -5.0, 4.0, 5.0 );
    upleft.wx = -4.0;
    upleft.wy = -5.0;
    botright.wx = 4.0;
    botright.wy = 5.0;
    _rectangle_wxy( _GBORDER, &upleft, &botright );
    upleft.wx = -3.0;
    upleft.wy = -3.5;
    botright.wx = 3.0;
    botright.wy = 3.5;
    _ellipse_wxy( _GFILLINTERIOR, &upleft, &botright );

    _getch();
    _setvideomode( _DEFAULTMODE );
}
```

// Example: WMENUCLK.C

```
/* WMENUCLK.C - Demonstrate choosing a menu
 * command with the QuickWin _wmenuclick function
 */
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMWINS      4    /* Number of windows */
#define OPENFLAGS   "w"   /* Access permission */
void main( void )
{
   int i, nRes;
   int wm;              /* Menu click result */
   int sf, gf;          /* Set/Get focus results */
   FILE *wins[NUMWINS]; /* Array of file pointers */
   /* Open NUMWINS windows */
   /* NULL arguments accept default characteristics */
   for( i = 0; i < NUMWINS; i++ )
   {
      wins[i] = _fwopen( NULL, NULL, OPENFLAGS );
      if( wins[i] == NULL )
      {
         printf( "***ERROR: On _fwopen #%i\n", i );
         exit( -1 );
      }
     /* Write in each window */
     nRes = fprintf( wins[i], "Windows!" );
   }
   /* Tile child windows with _wmenuclick */
   wm = _wmenuclick( _WINTILE );
   if( wm == -1 )
   {
      printf( "***ERROR: _wmenuclick\n" );
      exit( -1 );
   }
   /* Pass the focus from window to window */
   for( i = 0; i < NUMWINS; i++ )
   {
      sf = _wsetfocus( _fileno( wins[i] ) );
      gf = _wgetfocus();
      if(( sf == -1 ) || ( gf == -1 )  || ( gf != _fileno( wins[i] ) ) )
      {
          printf( "***ERROR: _wsetfocus/_wgetfocus\n" );
          exit( -1 );
      }
   }
   nRes = _fcloseall();
   exit( 0 );
}
```

// Example: WOPEN.C

```
/* WOPEN.C - Demonstrate opening
 * a QuickWin window with _wopen
 */
#include <fcntl.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#define PERSISTFLAG   _WINNOPERSIST
#define OPENFLAGS     _O_RDWR
void main( void )
{
   int wfh;                  /* File handle for window */
   int nRes;                 /* Window write results */
   struct _wopeninfo wininfo;/* Open information */
   /* Set up window open information */
   wininfo._version = _QWINVER;
   wininfo._title = "Window Closing";
   wininfo._wbufsize = _WINBUFDEF;
   /* Open a window with _wopen */
   /* NULL second argument accepts default size */
   wfh = _wopen( &wininfo, NULL, OPENFLAGS );
   if( wfh == -1 )
   {
      printf( "***ERROR: On _wopen\n" );
      exit( -1 );
   }
   /* Write in the window */
   nRes = write ( wfh, "Windows Everywhere!\n", 20 );
   /* Close the window with _wclose */
   nRes = _wclose( wfh, PERSISTFLAG );
   exit( 0 );
}
```

// Example: WPRINTF.C

```
/* WPRINTF.C shows how to use vprintf functions to write new
 * versions of printf. Functions in this catagory include:
 *      vsprintf     vprintf     vfprintf     _vsnprintf
 *
 * The vsprintf function is used in the example. The other
 * variations can be used similarly. For other examples of
 * formatted output, see EXTDIR.C (sprintf), TABLE.C
 * (fprintf), and PRINTF.C.
 */

#include <stdio.h>
#include <graph.h>
#include <string.h>
#include <stdarg.h>
#include <malloc.h>

int wprintf( short row, short col, short clr, long bclr, char *fmt, ... );

void main()
{
    short fgd = 0;
    long  bgd = 0L;

    _clearscreen( _GCLEARSCREEN );
    _outtext( "Color text example:\n\n" );

    /* Loop through 8 background colors. */
    for( bgd = 0L; bgd < 8; bgd++ )
    {
        wprintf( (short)(bgd + 3), 1, 15, bgd, "Back: %d Fore:", bgd );

        /* Loop through 16 foreground colors. */
        for( fgd = 0; fgd < 16; fgd++ )
            wprintf( -1, -1, fgd, -1L, " %2d ", fgd );
    }
}

/* Full-screen window version of printf that takes row, column, textcolor,
 * and background color as its first arguments, followed by normal printf
 * format strings (except that \t is not handled). You can specify -1 for
 * any of the first arguments to use the current value. The function returns
 * the number of characters printed, or a negative number for errors.
 */
int wprintf( short row, short col, short clr, long bclr, char *fmt, ... )
{
    struct  _rccoord tmppos;
    va_list marker;
    char    *buffer;
    int     increment = 256, size = increment, written = -1;

    /* Set text position. */
    tmppos = _gettextposition();
    if( row < 1 )
        row = tmppos.row;
    if( col < 1 )
        col = tmppos.col;
    _settextposition( row, col );

    /* Set foreground and background colors. */
    if( clr >= 0 )
```

```
        _settextcolor( clr );
    if( bclr >= 0 )
        _setbkcolor( bclr );

    /* Write text to a string and output the string. */
    va_start( marker, fmt );
    while ( written == -1 )
    {
        if(( buffer = (char *)malloc( size )) == NULL )
            return -1;
        written = _vsnprintf( buffer, size, fmt, marker );
        if ( written == -1 )
        {
             size += increment;
        }
     }
    va_end( marker );
    _outtext( buffer );
    free( buffer );
    return written;
}
```

// Example: WRAP.C

```
/* WRAP.C illustrates:
 *      _wrapon
 */

#include <conio.h>
#include <graph.h>

void main()
{
    short i;

    _clearscreen( _GCLEARSCREEN );
    _settextwindow( 1, 1, 25, 38 );
    _wrapon( _GWRAPON );
    for( i = 1; i < 80; i++ )
        _outtext( "Wrap on! " );

    _settextwindow( 1, 41, 25, 80 );
    _wrapon( _GWRAPOFF );
    for( i = 1; i < 80; i++ )
        _outtext( "Wrap off!  " );
    _getch();
    _clearscreen( _GCLEARSCREEN );
}
```

// Example: WSETFOC.C

```
/* WSETFOC.C - Demonstrate making a new QuickWin
 * window the active window with _wsetfocus
 */
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMWINS      4     /* Number of windows */
#define OPENFLAGS   "w"    /* Access permission */
void main( void )
{
   int i, nRes, wm;
   int sf, gf;          /* Set/Get focus results */
   FILE *wins[NUMWINS]; /* Array of file pointers */
   /* Open NUMWINS windows */
   /* NULL arguments accept default characteristics */
   for( i = 0; i < NUMWINS; i++ )
   {
      wins[i] = _fwopen( NULL, NULL, OPENFLAGS );
      if( wins[i] == NULL )
      {
         printf( "***ERROR: On _fwopen #%i\n", i );
         exit( -1 );
      }
       /* Write in each window */
       nRes = fprintf( wins[i], "Windows!\n" );
   }
   /* Tile child windows with _wmenuclick */
   wm = _wmenuclick( _WINTILE );
   if( wm == -1 )
   {
      printf( "***ERROR: _wmenuclick\n" );
      exit( -1 );
   }
   /* Pass the focus from window to window */
   for( i = 0; i < NUMWINS; i++ )
   {
      sf = _wsetfocus( _fileno( wins[i] ) );
      gf = _wgetfocus();
      if(( sf == -1 ) || ( gf == -1 )  || ( gf != _fileno( wins[i] ) ) )
      {
         printf( "***ERROR: _wsetfocus/_wgetfocus\n" );
         exit( -1 );
      }
   }
   nRes = _fcloseall();
   exit( 0 );
}
```

// Example: WSSCRBUF.C

```
/* WSSCRBUF.C - Demonstrate setting the size of a
 * QuickWin window's screen buffer
 * Note: The size is set here to an amount smaller than
 * the default size, but you can set it larger as well
 */
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMWINS      4        /* Number of windows */
#define OPENFLAGS   "w"       /* Access permission */
#define NUMLINES    100       /* Lines of text to write */
void main( void )
{
   int i;                   /* Loop variable */
   long nSize;              /* Old size of screen buffer */
   int nWinBufSize = 1500L; /* New size */
   int nRes;                /* Result */
   FILE *wp;                /* File pointer */
   /* Open a window */
   /* NULL arguments accept default characteristics */
   wp = _fwopen( NULL, NULL, OPENFLAGS );
   if( wp == NULL )
   {
      printf( "***ERROR:_fwopen\n" );
      exit( -1 );
   }
   /* Get the size of its screen buffer */
   nSize = _wgetscreenbuf( _fileno( wp ) );
   nRes = fprintf( wp, "Screen buffer holds %li chars\n", nSize );
   /* Reset the screen buffer size */
   nRes = _wsetscreenbuf( _fileno( wp ), nWinBufSize );
   /* Write many lines in the window */
   for( i = 0; i < NUMLINES; i++ )
   {
      nRes = fprintf( wp, "%i Lines (Windows!)\n", i );
   }
   nRes = fprintf( wp, "\nWhen the program ends, try using the scroll bars\n" );
   nRes = _wclose( _fileno( wp ), _WINPERSIST );
   exit( 0 );
}
```

// Example: WSETSIZE.C

```
/* WSETSIZE.C - Demonstrate setting the
 * size of a QuickWin window on the screen
 */
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#define OPENFLAGS    "w"         /* Access permission */
#define PERSISTFLAG _WINPERSIST  /* Keep on screen */
void main( void )
{
   int nRes;               /* Result */
   FILE *wp;               /* File pointer */
   struct _wsizeinfo ws;   /* Size information */
   /* Open a window */
   /* NULL arguments accept default characteristics */
   wp = _fwopen( NULL, NULL, OPENFLAGS );
   if( wp == NULL )
   {
      printf( "***ERROR:_fwopen\n" );
      exit( -1 );
   }
   /* Minimize the window to an icon */
   ws._version = _QWINVER;
   ws._type = _WINSIZEMIN;
   nRes = _wsetsize( _fileno( wp ), &ws );
   if( nRes == -1 )
   {
      printf( "***ERROR: _wsetsize\n" );
      exit( -1 );
   }
   nRes = _wclose( _fileno( wp ), PERSISTFLAG );
   exit( 0 );
}
```

// Example: WYIELD.C

```
/* WYIELD.C - Demonstrate yielding processor time from a
 * QuickWin program so that other Windows programs can
 * process their message queues; uses _wyield
 */
#include <io.h>
void compute(void);        /* Function prototype */
void main( void )
{
   int l;
   for( l = 0; l <= 10000; l++ )
   {
      compute();       /* Time-consuming function you supply */
      if( l % 1000 )
         _wyield();        /* Yield once every 1000 loops */
   }
}
void compute(void)
{
   /* Intensive computations */
}
```

feof

#include <stdio.h>

Syntax int feof(FILE **stream*);

Parameter	**Description**
stream	Pointer to FILE structure

The feof routine (implemented both as a function and as a macro) determines whether the end of *stream* has been reached. Once the end of the file is reached, read operations return an end-of-file indicator until the stream is closed or until rewind, fsetpos, fseek, or clearerr is called against it.

Return Value

The feof function returns a nonzero value after the first read operation that attempts to read past the end of the file. It returns 0 if the current position is not end-of-file. There is no error return.

clearerr
_eof
ferror
perror

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FEOF.C: This program uses feof to indicate when
 * it reaches the end of the file FEOF.C. It also
 * checks for errors with ferror.
 */
#include <stdio.h>
#include <stdlib.h>
void main( void )
{
   int  count, total = 0;
   char buffer[100];
   FILE *stream;
   if( (stream = fopen( "feof.c", "r" )) == NULL )
      exit( 1 );
   /* Cycle until end of file reached: */
   while( !feof( stream ) )
   {
      /* Attempt to read in 10 bytes: */
      count = fread( buffer, sizeof( char ), 100, stream );
      if( ferror( stream ) )       {
         perror( "Read error" );
         break;
      }
      /* Total up actual bytes read */
      total += count;
   }
   printf( "Number of bytes read = %d\n", total );
   fclose( stream );
}
```

ferror

#include <stdio.h>

Syntax int ferror(FILE **stream*);

Parameter	**Description**
stream	Pointer to FILE structure

The ferror routine (implemented both as a function and as a macro) tests for a reading or writing error on the file associated with *stream*. If an error has occurred, the error indicator for the stream remains set until the stream is closed or rewound, or until clearerr is called against it.

Return Value

If no error has occurred on *stream*, ferror returns 0. Otherwise, it returns a nonzero value.

clearerr
_eof
feof
perror
fopen

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

fflush

#include <stdio.h>

Syntax int fflush(FILE **stream*);

Parameter	**Description**
stream	Pointer to FILE structure

If the file associated with *stream* is open for output, fflush writes to that file the contents of the buffer associated with the stream. If the stream is open for input, fflush clears the contents of the buffer. The fflush function negates the effect of any prior call to ungetc against *stream*.

Buffers are automatically flushed when they are full, when the stream is closed, or when a program terminates normally without closing the stream. Also, fflush(NULL) flushes all streams opened for output.

The stream remains open after the call. The fflush function has no effect on an unbuffered stream.

Return Value

The fflush function returns the value 0 if the buffer was successfully flushed. The value 0 is also returned in cases in which the specified stream has no buffer or is open for reading only. A return value of EOF indicates an error.

Note If fflush returns EOF, data may have been lost because of a failed write. When setting up a critical error handler, it is safest to turn buffering off with the setvbuf function or to use low-level I/O routines such as _open, _close, and _write instead of the stream I/O functions.

fclose
_flushall
setbuf

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FFLUSH.C */
#include <stdio.h>
#include <conio.h>
void main( void )
{
   int integer;
   char string[81];
   /* Read each word as a string. */
   printf( "Enter a sentence of four words with scanf: " );
   for( integer = 0; integer < 4; integer++ )
   {
      scanf( "%s", string );
      printf( "%s\n", string );
   }
   /* You must flush the input buffer before using gets. */
   fflush( stdin );
   printf( "Enter the same sentence with gets: " );
   gets( string );
   printf( "%s\n", string );
}
```

fgetc, _fgetchar

#include <stdio.h>

Syntax int fgetc(FILE **stream*);

int _fgetchar(void);

Parameter	**Description**
stream	Pointer to FILE structure

The fgetc function reads a single character from the current position of the file associated with *stream*. The character is converted and returned as an int. The function then increments the associated file pointer (if any) to point to the next character. The _fgetchar function is equivalent to fgetc(stdin).

The fgetc and _fgetchar routines are identical to getc and getchar, but they are functions rather than macros.

Return Value

The fgetc and _fgetchar functions return the character read. They return EOF to indicate an error or end-of-file. Use feof or ferror to distinguish between an error and an end-of-file condition.

fgetc

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fgetchar

Standards: None
16-Bit: MS-DOS, QWIN

fputc
_fputchar
getc
getchar

```
/* FGETC.C: This program uses getc to read the first
 * 80 input  characters (or until the end of input)
 * and place them into a string named  buffer.
 */
#include <stdio.h>
#include <stdlib.h>
void main( void )
{
   FILE *stream;
   char buffer[81];
   int  i, ch;
   /* Open file to read line from: */
   if( (stream = fopen( "fgetc.c", "r" )) == NULL )
      exit( 0 );
   /* Read in first 80 characters and place them in "buffer": */
   ch = fgetc( stream );
   for( i=0; (i < 80 ) && ( feof( stream ) == 0 ); i++ )
   {
      buffer[i] = (char)ch;
      ch = fgetc( stream );
   }
   /* Add null to end string */
   buffer[i] = '\0';
   printf( "%s\n", buffer );
   fclose( stream );
}
```

fgetpos

#include <stdio.h>

Syntax int fgetpos(FILE **stream*, fpos_t **pos*);

Parameter	Description
stream	Target stream
pos	Position-indicator storage

The fgetpos function gets the current value of the *stream* argument's file-position indicator and stores it in the object pointed to by *pos*. The fsetpos function can later use information stored in *pos* to reset the *stream* argument's pointer to its position at the time fgetpos was called.

The *pos* value is stored in an internal format and is intended for use only by the fgetpos and fsetpos functions.

Return Value

If successful, the fgetpos function returns 0. On failure, it returns a nonzero value and sets errno to one of the following manifest constants (defined in STDIO.H):

Constant	Meaning
EBADF	The specified stream is not a valid file handle or is not accessible.
EINVAL	The *stream* value is invalid.

fsetpos

Standards: ANSI

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FGETPOS.C: This program opens a file and reads
 * bytes at several different locations.
 */
#include <stdio.h>
void main( void )
{
   FILE   *stream;
   fpos_t pos;
   char   buffer[20];
   if( (stream = fopen( "fgetpos.c", "rb" )) == NULL )
      printf( "Trouble opening file\n" );
   else
   {
      /* Read some data and then check the position. */
      fread( buffer, sizeof( char ), 10, stream );
      if( fgetpos( stream, &pos ) != 0 )
         perror( "fgetpos error" );
      else
      {
         fread( buffer, sizeof( char ), 10, stream );
         printf( "10 bytes at byte %ld: %.10s\n", pos, buffer );
      }
   /* Set a new position and read more data */
   pos = 140;
   if( fsetpos( stream, &pos ) != 0 )
      perror( "fsetpos error" );
   fread( buffer, sizeof( char ), 10, stream );
   printf( "10 bytes at byte %ld: %.10s\n", pos, buffer );
   fclose( stream );
   }
}
```

fgets

#include <stdio.h>

Syntax char *fgets(char **string*, int *n*, FILE **stream*);

Parameter	Description
string	Storage location for data
n	Maximum number of characters to read
stream	Pointer to FILE structure

The fgets function reads a string from the input *stream* argument and stores it in *string*. Characters are read from the current stream position up to and including the first newline character ('\n'), up to the end of the stream, or until the number of characters read is equal to *n*-1, whichever comes first. The result is stored in *string*, and a null character ('\0') is appended. The newline character, if read, is included in the string. The fgets function is similar to the gets function; however, gets replaces the newline character with NULL.

Return Value

If successful, the fgets function returns *string*. It returns NULL to indicate either an error or end-of-file condition. Use feof or ferror to determine whether an error occurred.

fputs
gets
puts

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FGETS.C: This program uses fgets to display
 * a line from a file on the screen.
 */
#include <stdio.h>
void main( void )
{
   FILE *stream;
   char line[100];
   if( (stream = fopen( "fgets.c", "r" )) != NULL )
   {
      if( fgets( line, 100, stream ) == NULL)
         printf( "fgets error\n" );
      else
         printf( "%s", line);
      fclose( stream );
   }
}
```

_fieeetomsbin, _fmsbintoieee

#include <math.h>

Syntax int _fieeetomsbin(float **src4*, float **dst4*);

int _fmsbintoieee(float **src4*, float **dst4*);

Parameter	**Description**
scr4	Value to be converted
dst4	Converted value

The _fieeetomsbin routine converts a single-precision floating-point number in IEEE (Institute of Electrical and Electronic Engineers) format to Microsoft (MS) binary format.

The _fmsbintoieee routine converts a floating-point number in Microsoft binary format to IEEE format.

These routines allow C programs (which store floating-point numbers in the IEEE format) to use numeric data in random-access data files created with Microsoft Basic (which stores floating-point numbers in the Microsoft binary format), and vice versa.

The argument *src4* points to the float value to be converted. The result is stored at the location given by *dst4*.

These routines do not handle IEEE NANs ("not a number") and infinities. IEEE denormals are treated as 0 in the conversions.

Return Value

These functions return 0 if the conversion is successful and 1 if the conversion causes an overflow.

_dieeetomsbin
_dmsbintoieee

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_filelength

#include <io.h> Required only for function declarations

Syntax long _filelength*(* int *handle*);

Parameter	**Description**
handle	Target file handle

The _filelength function returns the length, in bytes, of the target file associated with *handle.*

Return Value

The _filelength function returns the file length in bytes. A return value of -1L indicates an error, and an invalid handle sets errno to EBADF.

_chsize
_fileno
_fstat
_stat

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fileno

#include <stdio.h>

Syntax int _fileno(FILE **stream*);

Parameter	**Description**
stream	Pointer to FILE structure

The _fileno routine returns the file handle currently associated with *stream*. This routine is implemented both as a function and as a macro.

Return Value

The _fileno routine returns the file handle. There is no error return. The result is undefined if *stream* does not specify an open file.

Use _fileno for compatibility with ANSI naming conventions of non-ANSI functions. Use fileno and link with OLDNAMES.LIB for UNIX compatibility.

_fdopen
_filelength
fopen
freopen

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FILENO.C: This program uses _fileno to obtain
 * the file handle for some standard C streams.
 */
#include <stdio.h>
void main( void )
{
   printf( "The file handle for stdin is %d\n", _fileno( stdin ) );
   printf( "The file handle for stdout is %d\n", _fileno( stdout ) );
   printf( "The file handle for stderr is %d\n", _fileno( stderr ) );
}
```

_floodfill, _floodfill_w

#include <graph.h>

Syntax short __far _floodfill(short *x*, short *y*, short *boundary*);

short __far _floodfill_w(double *wx*, double *wy*, short *boundary*);

Parameter	Description
x, *y*	Start point
wx, *wy*	Start point
boundary	Boundary color of area to be filled

The functions in the _floodfill family fill an area of the display, using the current color and fill mask. The _floodfill routine begins filling at the view-coordinate point (*x*, *y*). The _floodfill_w routine begins filling at the window-coordinate point (*wx*, *wy*).

If this point lies inside the figure, the interior is filled; if it lies outside the figure, the background is filled. The point must be inside or outside the figure to be filled, not on the figure boundary itself. Filling occurs in all directions, stopping at the color of *boundary*.

Return Value

The _floodfill functions return a nonzero value if the fill is successful. They return 0 if the fill could not be completed, the starting point lies on the *boundary* color, or the start point lies outside the clipping region.

_ellipse functions
_getcolor
_getfillmask
_grstatus
_pie functions
_setfillmask
_setcliprgn
_setcolor

Standards: None
16-Bit: MS-DOS, QWIN

```
/* FLOODFIL.C: This program draws a series of
 * nested rectangles in different colors,
 * constantly changing the background color.
 */
#include <conio.h>
#include <stdlib.h>
#include <graph.h>
void main( void )
{
   int loop;
   short xvar, yvar;
   /* find a valid graphics mode */
   if( !_setvideomode( _MAXCOLORMODE ) ) exit( 1 );
   for( xvar = 163, loop = 0; xvar < 320; loop++, xvar += 3 )
   {
      _setcolor( (short)(loop % 16) );
      yvar = (short)(xvar * 5 / 8);
      _rectangle( _GBORDER, 320-xvar, 200-yvar, xvar, yvar );
      _setcolor( (short)(rand() % 16) );
      _floodfill( 0, 0, (short)(loop % 16) );
   }
   _getch();
   _setvideomode( _DEFAULTMODE );
}
```

floor, floorl

#include <math.h>

Syntax double floor(double *x*);

long double floorl(long double *x*);

Parameter	Description
x	Floating-point value

The floor and floorl functions return a floating-point value representing the largest integer that is less than or equal to *x*.

The floorl function is the 80-bit counterpart, and it uses the 80-bit, 10-byte coprocessor form of arguments and return values. See the long double functions for more details on this data type.

Return Value

These functions return the floating-point result. There is no error return.

floor

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

floorl

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

ceil
fmod

```
/* FLOOR.C: This example displays the largest integers
 * less than or equal to the floating-point values 2.8
 * and -2.8. It then shows the smallest integers greater
 * than or equal to 2.8 and -2.8.
 */
#include <math.h>
#include <stdio.h>
void main( void )
{
   double y;
   y = floor( 2.8 );
   printf( "The floor of 2.8 is %f\n", y );
   y = floor( -2.8 );
   printf( "The floor of -2.8 is %f\n", y );
   y = ceil( 2.8 );
   printf( "The ceil of 2.8 is %f\n", y );
   y = ceil( -2.8 );
   printf( "The ceil of -2.8 is %f\n", y );
}
```

_flushall

#include <stdio.h>

Syntax int _flushall(void);

The _flushall function writes to its associated files the contents of all buffers associated with open output streams. All buffers associated with open input streams are cleared of their current contents. The next read operation (if there is one) then reads new data from the input files into the buffers.

Buffers are automatically flushed when they are full, when streams are closed, or when a program terminates normally without closing streams.

All streams remain open after the call to _flushall.

Return Value

The _flushall function returns the number of open streams (input and output). There is no error return.

fflush

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FLUSHALL.C: This program uses _flushall
 * to flush all open buffers.
 */
#include <stdio.h>
void main( void )
{
   int numflushed;
   numflushed = _flushall();
   printf( "There were %d streams flushed\n", numflushed );
}
```

fmod, fmodl

#include <math.h>

Syntax double fmod(double *x*, double *y*);

long double fmodl(long double *x*, long double *y*);

Parameter	**Description**
x, *y*	Floating-point values

The fmod and fmodl functions calculate the floating-point remainder *f* of *x* / *y* such that *x* = *i* * *y* + *f*, where *i* is an integer, *f* has the same sign as *x*, and the absolute value of *f* is less than the absolute value of *y*.

The fmodl function is the 80-bit counterpart; it uses the 80-bit, 10-byte coprocessor form of arguments and return values. See the long double functions for more details on this data type.

Return Value

These functions return the floating-point remainder. If *y* is 0, the function returns 0.

fmod

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

fmodl

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

ceil
fabs
floor

```c
/* FMOD.C: This program displays a
 * floating-point remainder.
 */
#include <math.h>
#include <stdio.h>
void main( void )
{
   double x = -10.0, y = 3.0, z;
   z = fmod( x, y );
   printf( "The remainder of %.2f / %.2f is %f\n", x, y, z );
}
```

fopen

#include <stdio.h>

Syntax FILE *fopen(const char **filename*, const char **mode*);

Parameter	**Description**
filename	Filename
mode	Type of access permitted

The fopen function opens the file specified by *filename*. The character string *mode* specifies the type of access requested for the file, as follows:

"r"
Opens for reading. If the file does not exist or cannot be found, the fopen call will fail.

"w"
Opens an empty file for writing. If the given file exists, its contents are destroyed.

"a"
Opens for writing at the end of the file (appending); creates the file first if it doesn't exist.

"r+"
Opens for both reading and writing. (The file must exist.)

"w+"
Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.

"a+"
Opens for reading and appending; creates the file first if it doesn't exist.

When a file is opened with the "a" or "a+" access type, all write operations occur at the end of the file. Although the file pointer can be repositioned using fseek or rewind, the file pointer is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

When the "r+", "w+", or "a+" access type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when you switch between reading and writing, there must be an intervening fflush, fsetpos, fseek, or rewind operation. The current position can be specified for the fsetpos or fseek operation, if desired.

In addition to the values listed above, the following characters can be included in *mode* to specify the translation mode for newline characters:

t
Open in text (translated) mode. In this mode, carriage-return-line-feed (CR-LF) combinations are translated into single line feeds (LF) on input and LF characters are translated to CR-LF combinations on output. Also , CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading or for reading/writing, fopen checks for a CTRL+Z at the end of the file and removes it, if possible. This is done because using the fseek and ftell functions to move within a file that ends with a CTRL+Z may cause fseek to behave improperly near the end of the file.

b
Open in binary (untranslated) mode; the above translations are suppressed.

c
Enable the commit flag for the associated *filename* so that the contents of the file buffer are written directly to disk if either fflush or _flushall is called.

n
Reset the commit flag for the associated *filename* to "no-commit". This is the default. It will also override the global commit flag if you have linked your program with COMMODE.OBJ. The global

commit flag default is "no-commit" unless you explicitly link your program with COMMODE.OBJ.

If t or b is not given in *mode*, the translation mode is defined by the default-mode variable _fmode. If t or b is prefixed to the argument, the function will fail and return NULL.

Return Value

The fopen function returns a pointer to the open file. A null pointer value indicates an error.

Note that the c, n, and t options are not part of the ANSI standard for fopen; they are Microsoft extensions and should not be used where ANSI portability is desired.

fclose
_fcloseall
_fdopen
ferror
_fileno
freopen
_open
_setmode

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FOPEN.C: This program opens files named "data"
 * and "data2".It  uses fclose to close "data" and
 * _fcloseall to close all remaining files.
 */
#include <stdio.h>
FILE *stream, *stream2;
void main( void )
{
   int numclosed;
   /* Open for read (will fail if 'data does not exist) */
   if( (stream  = fopen( "data", "r" )) == NULL )
      printf( "The file 'data' was not opened\n" );
   else
      printf( "The file 'data' was opened\n" );
   /* Open for write */
   if( (stream2 = fopen( "data2", "w+" )) == NULL )
      printf( "The file 'data2' was not opened\n" );
   else
      printf( "The file 'data2' was opened\n" );
   /* Close stream */
   if( fclose( stream ) )
      printf( "The file 'data' was not closed\n" );
   /* All other files are closed: */
   numclosed = _fcloseall( );
   printf( "Number of files closed by _fcloseall: %u\n", numclosed );
}
```

_FP_OFF, _FP_SEG

#include <dos.h>

Syntax unsigned _FP_OFF(void __far **address*);

unsigned _FP_SEG(void __far **address*);

Parameter	Description
address	Far pointer to memory address

The _FP_OFF and _FP_SEG macros can be used to set or get the offset and segment, respectively, of the far pointer at *address*.

The argument to _FP_OFF and _FP_SEG must be a far pointer variable. It cannot be an array name, or a near pointer cast to a far pointer.

Return Value

The _FP_OFF macro returns an offset. The _FP_SEG macro returns a segment address.

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* _FP_SEG.C: This program uses _FP_SEG and _FP_OFF to
 * obtain the segment and offset of the long pointer p.
 */
#include <dos.h>
#include <malloc.h>
#include <stdio.h>
void main( void )
{
   void __far *p;
   unsigned int seg_val;
   unsigned int off_val;
   p = _fmalloc( 100 );      /* Points pointer at something */
   seg_val = _FP_SEG( p );      /* Gets address pointed to */
   off_val = _FP_OFF( p );
   printf( "Segment is %.4X; Offset is %.4X\n", seg_val, off_val );
}
```

_fpreset

#include <float.h>

Syntax void _fpreset(void);

The _fpreset function reinitializes the floating-point-math package. This function is usually used in conjunction with signal, system, or the _exec or _spawn functions.

If a program traps floating-point error signals (SIGFPE) with signal, it can safely recover from floating-point errors by invoking _fpreset and using longjmp.

In MS-DOS versions prior to 3.0, a child process executed by _exec, _spawn, or system may affect the floating-point state of the parent process if an 8087, 80287, or 80387 coprocessor is used. If you are using either coprocessor, the following precautions are recommended:

- The _exec, _spawn, and system functions should not be called during the evaluation of a floating-point expression.
- The _fpreset function should be called after these routines if there is a possibility of the child process performing any floating-point operations.

Return Value

None.

_exec functions
signal
_spawn functions

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FPRESET.C: This program uses signal to set up a
 * routine for  handling floating-point errors.
 */
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#include <stdlib.h>
#include <float.h>
#include <math.h>
#include <string.h>
jmp_buf mark;              /* Address for long jump to jump to */
int     fperr;             /* Global error number */
void __cdecl fphandler( int sig, int num );   /* Prototypes */
void fpcheck( void );
void main( void )
{
   double n1, n2, r;
   int jmpret;
   /* Set up floating-point error handler. The compiler
    * will generate a warning because it expects
    * signal-handling functions to take only one argument.
    */
   if( signal( SIGFPE, fphandler ) == SIG_ERR )
   {
      fprintf( stderr, "Couldn't set SIGFPE\n" );
      abort();
   }
   /* Save stack environment for return in case of error. First time
    * through, jmpret is 0, so true conditional is executed. If an
    * error occurs, jmpret will be set to -1 and false conditional
    * will be executed.
    */
   jmpret = setjmp( mark );
   if( jmpret == 0 )
   {
      printf( "Test for invalid operation - " );
      printf( "enter two numbers: " );
      scanf( "%lf %lf", &n1, &n2 );
      r = n1 / n2;
      /* This won't be reached if error occurs. */
      printf( "\n\n%4.3g / %4.3g = %4.3g\n", n1, n2, r );
      r = n1 * n2;
      /* This won't be reached if error occurs. */
      printf( "\n\n%4.3g * %4.3g = %4.3g\n", n1, n2, r );
   }
   else
      fpcheck();
}
/* fphandler handles SIGFPE (floating-point error) interrupt. Note
 * that this prototype accepts two arguments and that the prototype
 * for signal in the run-time library expects a signal handler to
 * have only one argument.
 *
 * The second argument in this signal handler allows processing of
 * _FPE_INVALID, _FPE_OVERFLOW, _FPE_UNDERFLOW, and _FPE_ZERODIVIDE
 * all of which are Microsoft-specific symbols that augment the
 * information provided by SIGFPE. The compiler will generate a
 * warning, which is harmless and expected.
 */
void fphandler( int sig, int num )
```

```
{
   /* Set global for outside check since we don't want
    * to do I/O in the handler.
    */
   fperr = num;
   /* Initialize floating-point package. */
   _fpreset();
   /* Restore calling environment and jump back to setjmp. Return -1
    * so that setjmp will return false for conditional test.
    */
   longjmp( mark, -1 );
}
void fpcheck( void )
{
   char fpstr[30];
   switch( fperr )
   {
   case _FPE_INVALID:
       strcpy( fpstr, "Invalid number" );
       break;
   case _FPE_OVERFLOW:
       strcpy( fpstr, "Overflow" );
       break;
   case _FPE_UNDERFLOW:
       strcpy( fpstr, "Underflow" );
       break;
   case _FPE_ZERODIVIDE:
       strcpy( fpstr, "Divide by zero" );
       break;
   default:
       strcpy( fpstr, "Other floating point error" );
       break;
   }
   printf( "Error %d: %s\n", fperr, fpstr );
}
```

fprintf

#include <stdio.h>

Syntax int fprintf(FILE **stream*, const char **format* [, *argument*]...);

Parameter	Description
stream	Pointer to FILE structure
format	Format-control string
argument	Optional arguments

The fprintf function formats and prints a series of characters and values to the output *stream*. Each *argument* (if any) is converted and output according to the corresponding format specification in *format*.

The *format* argument has the same form and function that it does for the printf function; see the printf function for more information on *format* and *argument*.

Return Value

The fprintf function returns the number of characters printed, or a negative value in the case of an output error.

_cprintf
fscanf
printf
sprintf

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN

```
/* FPRINTF.C: This program uses fprintf to format various
 * data and print them to the file named FPRINTF.OUT. It
 * then displays FPRINTF.OUT on the screen using the system
 * function to invoke the MS-DOS TYPE command.
 */
#include <stdio.h>
#include <process.h>
FILE *stream;
void main( void )
{
   int    i = 10;
   double fp = 1.5;
   char   s[] = "this is a string";
   char   c = '\n';
   stream = fopen( "fprintf.out", "w" );
   fprintf( stream, "%s%c", s, c );
   fprintf( stream, "%d\n", i );
   fprintf( stream, "%f\n", fp );
   fclose( stream );
   system( "type fprintf.out" );
}
```

fputc, _fputchar

#include <stdio.h>

Syntax int fputc(int *c, FILE *stream*);

int _fputchar(int *c*);

Parameter	Description
c	Character to be written
stream	Pointer to FILE structure

The fputc function writes the single character *c* to the output *stream* at the current position. The _fputchar function is equivalent to fputc(*c,* stdout*)*.

The fputc and _fputchar routines are similar to putc and putchar but are functions rather than macros.

Return Value

The fputc and _fputchar functions return the character written. A return value of EOF indicates an error.

fgetc
_fgetchar
putc
putchar

fputc

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fputchar

Standards: None
16-Bit: MS-DOS, QWIN

```
/* FPUTC.C: This program uses fputc and _fputchar
 * to send a character array to stdout.
 */
#include <stdio.h>
void main( void )
{
   char strptr1[] = "This is a test of fputc!!\n";
   char strptr2[] = "This is a test of _fputchar!!\n";
   char *p;
   /* Print line to stream using fputc. */
   p = strptr1;
   while( (*p != '\0') && fputc( *(p++), stdout ) != EOF ) ;
   /* Print line to stream using _fputchar. */
   p = strptr2;
   while( (*p != '\0') && _fputchar( *(p++) ) != EOF )
      ;
}
```

fputs

#include <stdio.h>

Syntax int fputs(const char **string*, FILE **stream*);

Parameter	Description
string	String to be output
stream	Pointer to FILE structure

The fputs function copies *string* to the output *stream* at the current position. The terminating null character ('\0') is not copied.

Return Value

The fputs function returns a nonnegative value if it is successful. If an error occurs, it returns EOF.

fgets
gets
puts

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FPUTS.C: This program uses fputs to write
 * a single line to the stdout stream.
 */
#include <stdio.h>
void main( void )
{
   fputs( "Hello world from fputs.\n", stdout );
}
```

fread

#include <stdio.h>

Syntax size_t fread(void *_buffer_, size_t _size_, size_t _count_, FILE *_stream_);

Parameter	Description
buffer	Storage location for data
size	Item size in bytes
count	Maximum number of items to be read
stream	Pointer to FILE structure

The fread function reads up to *count* items of *size* bytes from the input *stream* and stores them in *buffer*. The file pointer associated with *stream* (if there is one) is increased by the number of bytes actually read.

If the given stream is opened in text mode, carriage-return-line-feed pairs are replaced with single line-feed characters. The replacement has no effect on the file pointer or the return value.

The file-pointer position is indeterminate if an error occurs. The value of a partially read item cannot be determined.

Return Value

The fread function returns the number of full items actually read, which may be less than *count* if an error occurs or if the file end is encountered before reaching *count*.

The feof or ferror function should be used to distinguish a read error from an end-of-file condition. If *size* or *count* is 0, fread returns 0 and the buffer contents are unchanged.

fwrite
_read

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FREAD.C: This program opens a file named FREAD.OUT and
 * writes 25 characters to the file. It then tries to open
 * FREAD.OUT and read in 25 characters. If the attempt succeeds,
 * the program displays the number of actual items read.
 */
#include <stdio.h>
void main( void )
{
   FILE *stream;
   char list[30];
   int  i, numread, numwritten;
   /* Open file in text mode: */
   if( (stream = fopen( "fread.out", "w+t" )) != NULL )
   {
      for ( i = 0; i < 25; i++ )
         list[i] = (char)('z' - i);
      /* Write 25 characters to stream */
      numwritten = fwrite( list, sizeof( char ), 25, stream );
      printf( "Wrote %d items\n", numwritten );
      fclose( stream );
   }
   else
      printf( "Problem opening the file\n" );
   if( (stream = fopen( "fread.out", "r+t" )) != NULL )
   {
      /* Attempt to read in 25 characters */
      numread = fread( list, sizeof( char ), 25, stream );
      printf( "Number of items read = %d\n", numread );
      printf( "Contents of buffer = %.25s\n", list );
      fclose( stream );
   }
   else
      printf( "Was not able to open the file\n" );
}
```

free Functions

#include <stdlib.h> For ANSI compatibility (free only)

#include <malloc.h> Required only for function declarations

Syntax void free(void *_memblock_);

void _bfree(__segment _seg_, void __based(void) *_memblock_);

void _ffree(void __far *_memblock_);

void _nfree(void __near *_memblock_);

Parameter	Description
memblock	Allocated memory block
seg	Based-heap segment selector

The free family of functions deallocates a memory block. The argument *memblock* points to a memory block previously allocated through a call to calloc, malloc, or realloc. The number of bytes freed is the number of bytes specified when the block was allocated (or reallocated, in the case of realloc). After the call, the freed block is available for allocation.

The *seg* argument specifies the based heap containing the memory block to be freed by the _bfree function.

Attempting to free an invalid pointer may affect subsequent allocation and cause errors. An invalid pointer is one not allocated with the appropriate call.

The following restrictions apply to use of the free, _bfree, _ffree, and _nfree functions:

Blocks allocated with	Should be freed with
calloc, malloc, realloc	free
_bcalloc _bmalloc, _brealloc	_bfree
_fcalloc, _fmalloc, _frealloc	_ffree
_ncalloc, _nmalloc, _nrealloc	_nfree

A NULL pointer argument is ignored.

In large data models (compact-, large-, and huge-model programs), free maps to _ffree. In small data models (tiny-, small-, and medium-model programs), free maps to _nfree.

The various free functions deallocate a memory block in the segments shown in the list below:

Function	Data Segment
free	Depends on data model of program
_bfree	Based heap specified by *seg* value
_ffree	Far heap (outside default data segment)
_nfree	Near heap (inside default data segment)

Return Value

None.

calloc functions
malloc functions
realloc functions

free

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_bfree, _ffree, _nfree

Standards: None
16-Bit: MS-DOS, WIN, WIN DLL

```
/* MALLOC.C: This program allocates memory with
 * malloc, then frees the memory with free.
 */
#include <stdlib.h>         /* Definition of _MAX_PATH */
#include <stdio.h>
#include <malloc.h>
void main( void )
{
   char *string;
   /* Allocate space for a path name */
   string = malloc( _MAX_PATH );
   if( string == NULL )
      printf( "Insufficient memory available\n" );
   else
      printf( "Memory space allocated for path name\n" );
   free( string );
   printf( "Memory freed\n" );
}
```

_freect

#include <malloc.h> Required only for function declarations

Syntax unsigned int _freect(size_t *size*);

Parameter	**Description**
size	Item size in bytes

The _freect function tells you how much memory is available for dynamic memory allocation in the near heap. It does so by returning the approximate number of times your program can call _nmalloc (or malloc in small data models) to allocate an item *size* bytes long in the near heap (default data segment).

Return Value

The _freect function returns the number of calls as an unsigned integer.

calloc functions
_expand functions
malloc functions
_memavl
_msize functions
realloc functions

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FREECT.C: This program determines how much free
 * space is available for integers in the default
 * data segment. Then it allocates space for 1000
 * integers and checks the space again, using _freect.
 */
#include <malloc.h>
#include <stdio.h>
void main( void )
{
   int i;
   /* First report on the free space: */
   printf( "Integers (approximate) available on heap: %u\n\n",
      _freect( sizeof(int) ) );
   /* Allocate space for 1000 integers:\n */
   for( i = 0;   i < 1000;   ++i )
      malloc( sizeof( int ) );
   /* Report again on the free space: */
   printf( "After allocating space for 1000 integers:\n" );
   printf( "Integers (approximate) available on heap: %u\n\n",
      _freect( sizeof(int) ) );
}
```

freopen

#include <stdio.h>

Syntax FILE *freopen(const char **filename*, const char **mode*, FILE **stream*);

Parameter	Description
filename	Path of new file
mode	Type of access permitted
stream	Pointer to FILE structure

The freopen function closes the file currently associated with *stream* and reassigns *stream* to the file specified by *filename*. The freopen function is typically used to redirect the pre-opened files stdin, stdout, and stderr to files specified by the user. The new file associated with *stream* is opened with *mode*, which is a character string specifying the type of access requested for the file, as follows:

"r"
Opens for reading. If the file does not exist or cannot be found, the freopen call fails.

"w"
Opens an empty file for writing. If the given file exists, its contents are destroyed.

"a"
Opens for writing at the end of the file (appending); creates the file first if it does not exist.

"r+"
Opens for both reading and writing. (The file must exist.)

"w+"
Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.

"a+"
Opens for reading and appending; creates the file first if it does not exist.

Use the "w" and "w+" types with care, as they can destroy existing files.

When a file is opened with the "a" or "a+" access type, all write operations take place at the end of the file. Although the file pointer can be repositioned using fseek or rewind, the file pointer is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

When the "r+", "w+", or "a+" access type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when you switch between reading and writing, there must be an intervening fsetpos, fseek, or rewind operation. The current position can be specified for the fsetpos or fseek operation, if desired.

In addition to the values listed above, one of the following characters may be included in the *mode* string to specify the translation mode for new lines.

t
Open in text (translated) mode; carriage-return-line-feed (CR-LF) combinations are translated into single line-feed (LF) characters on input; LF characters are translated to CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading, or writing and reading, the run-time library checks for a CTRL+Z at the end of the file and removes it, if possible. This is done because using the fseek and ftell functions to move within a file may cause fseek to behave improperly near the end of the file.

b
Open in binary (untranslated) mode; the above translations are suppressed.

If t or b is not given in the *mode* string, the translation mode is defined by the default mode variable _fmode.

Return Value

The freopen function returns a pointer to the newly opened file. If an error occurs, the original file is closed and the function returns a NULL pointer value.

The t option is not part of the ANSI standard for freopen; it is a Microsoft extension that should not be used where ANSI portability is desired.

fclose
_fcloseall
_fdopen
_fileno
fopen
_open
_setmode

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FREOPEN.C: This program reassigns _stdaux to the file
 * named FREOPEN.OUT and writes a line to that file.
 */
#include <stdio.h>
#include <stdlib.h>
FILE *stream;
void main( void )
{
   /* Reassign "_stdaux" to "freopen.out": */
   stream = freopen( "freopen.out", "w", _stdaux );
   if( stream == NULL )
      fprintf( stdout, "error on freopen\n" );
   else
   {
      fprintf( stream, "This will go to the file 'freopen.out'\n" );
      fprintf( stdout, "successfully reassigned\n" );
      fclose( stream );
   }
   system( "type freopen.out" );
}
```

frexp, frexpl

#include <math.h>

Syntax double frexp(double *x*, int **expptr*);

long double frexpl(long double *x*, int **expptr*);

Parameter	Description
x	Floating-point value
expptr	Pointer to stored integer exponent

The frexp and frexpl functions break down the floating-point value (*x*) into a mantissa (*m*) and an exponent (*n*), such that the absolute value of *m* is greater than or equal to 0.5 and less than 1.0, and *x* equals *m* times (2 raised to the power of n). The integer exponent *n* is stored at the location pointed to by *expptr*.

The frexpl function is the 80-bit counterpart and uses an 80-bit, 10-byte coprocessor form of arguments and return values. See the long double functions for more details on this data type.

Return Value

These functions return the mantissa. If *x* is 0, the function returns 0 for both the mantissa and the exponent. There is no error return.

ldexp functions
modf

frexp

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

frexpl

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FREXP.C: This program calculates frexp( 16.4, &n )
 * then displays y and n.
 */
#include <math.h>
#include <stdio.h>
void main( void )
{
   double x, y;
   int n;
   x = 16.4;
   y = frexp( x, &n );
   printf( "frexp( %f, &n ) = %f, n = %d\n", x, y, n );
}
```

fscanf

#include <stdio.h>

Syntax int fscanf(FILE **stream*, const char **format* [, *argument*]...);

Parameter	Description
stream	Pointer to FILE structure
format	Format-control string
argument	Optional arguments

The fscanf function reads data from the current position of *stream* into the locations given by *argument* (if any). Each argument must be a pointer to a variable with a type that corresponds to a type specifier in *format*. The format controls the interpretation of the input fields and has the same form and function as the *format* argument for the scanf function; see scanf for a description of *format*.

Return Value

The fscanf function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF for an error or end-of-file on *stream* before the first conversion. A return value of 0 means that no fields were assigned.

_cscanf
fprintf
scanf
sscanf

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN

```
/* FSCANF.C: This program writes formatted
 * data to a file. It then uses fscanf to
 * read the various data back from the file.
 */
#include <stdio.h>
FILE *stream;
void main( void )
{
   long l;
   float fp;
   char s[81];
   char c;
   stream = fopen( "fscanf.out", "w+" );
   if( stream == NULL )
      printf( "The file fscanf.out was not opened\n" );
   else
   {
      fprintf( stream, "%s %ld %f%c", "a-string", 65000, 3.14159, 'x' );
      /* Set pointer to beginning of file: */
      fseek( stream, 0L, SEEK_SET );
      /* Read data back from file: */
      fscanf( stream, "%s", s );
      fscanf( stream, "%ld", &l );
      fscanf( stream, "%f", &fp );
      fscanf( stream, "%c", &c );
      /* Output data read: */
      printf( "%s\n", s );
      printf( "%ld\n", l );
      printf( "%f\n", fp );
      printf( "%c\n", c );
      fclose( stream );
   }
}
```

fseek

#include <stdio.h>

Syntax int fseek(FILE **stream*, long *offset*, int *origin*);

Parameter	Description
stream	Pointer to FILE structure
offset	Number of bytes from *origin*
origin	Initial position

The fseek function moves the file pointer (if any) associated with *stream* to a new location that is *offset* bytes from *origin*. The next operation on the stream takes place at the new location. On a stream open for update, the next operation can be either a read or a write.

The argument *origin* must be one of the following constants defined in STDIO.H:

Origin	Definition
SEEK_CUR	Current position of file pointer
SEEK_END	End of file
SEEK_SET	Beginning of file

The fseek function can be used to reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file. An attempt to position the pointer before the beginning of the file causes an error only if you explicitly link with LSEEKCHK.OBJ.

The fseek function clears the end-of-file indicator and negates the effect of any prior ungetc calls against *stream*.

When a file is opened for appending data, the current file position is determined by the last I/O operation, not by where the next write would occur. If no I/O operation has yet occurred on a file opened for appending, the file position is the start of the file.

For streams opened in text mode, fseek has limited use because carriage-return-line-feed translations can cause fseek to produce unexpected results. The only fseek operations guaranteed to work on streams opened in text mode are:

- Seeking with an offset of 0 relative to any of the *origin* values
- Seeking from the beginning of the file with an offset value returned from a call to ftell

Return Value

If successful, fseek returns 0. Otherwise, it returns a nonzero value. On devices incapable of seeking, the return value is undefined.

ftell
_lseek
rewind

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FSEEK.C: This program opens the file FSEEK.OUT and
 * moves the pointer to the file's beginning.
 */
#include <stdio.h>
void main( void )
{
   FILE *stream;
   char line[81];
   int  result;
   stream = fopen( "fseek.out", "w+" );
   if( stream == NULL )
      printf( "The file fseek.out was not opened\n" );
   else
   {
      fprintf( stream, "The fseek begins here: "
                       "This is the file 'fseek.out'.\n" );
      result = fseek( stream, 23L, SEEK_SET);
      if( result )
         perror( "Fseek failed" );
      else
      {
         printf( "File pointer is set to middle of first line.\n" );
         fgets( line, 80, stream );
         printf( "%s", line );
      }
      fclose( stream );
  }
}
```

fsetpos

#include <stdio.h>

Syntax int fsetpos(FILE **stream*, const fpos_t **pos*);

Parameter	**Description**
stream	Target stream
pos	Position-indicator storage

The fsetpos function sets the file-position indicator for *stream* to the value of *pos*, which is obtained in a prior call to fgetpos against *stream*.

The function clears the end-of-file indicator and undoes any effects of the ungetc function on *stream*. After calling fsetpos, the next operation on *stream* may be either input or output.

Return Value

If successful, the fsetpos function returns 0. On failure, the function returns a nonzero value and sets errno to one of the following manifest constants (defined in ERRNO.H):

Constant	**Meaning**
EBADF	The object that *stream* points to is not a valid file handle, or the file is not accessible.
EINVAL	An invalid *stream* value was passed.

fgetpos

Standards: ANSI

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fsopen

#include <stdio.h>

#include <share.h> *shflag* constants

Syntax FILE *_fsopen(const char **filename*, const char **mode*, int *shflag*);

Parameter	**Description**
filename	Filename to open
mode	Type of access permitted
shflag	Type of sharing allowed

The _fsopen function opens the file specified by *filename* as a stream and prepares the file for subsequent shared reading or writing, as defined by the *mode* and *shflag* arguments.

The character string *mode* specifies the type of access requested for the file, as follows:

"r"
Opens for reading. If the file does not exist or cannot be found, the _fsopen call will fail.

"w"
Opens an empty file for writing. If the given file exists, its contents are destroyed.

"a"
Opens for writing at the end of the file (appending); creates the file first if it does not exist.

"r+"
Opens for both reading and writing. (The file must exist.)

"w+"
Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.

"a+"
Opens for reading and appending; creates the file first if it does not exist.

Use the "w" and "w+" types with care, as they can destroy existing files.

When a file is opened with the "a" or "a+" access type, all write operations occur at the end of the file. Although the file pointer can be repositioned using fseek or rewind, the file pointer is always moved back to the end of the file before any write operation is carried out. Thus, existing data cannot be overwritten.

When the "r+", "w+", or "a+" access type is specified, both reading and writing are allowed (the file is said to be open for "update"). However, when switching between reading and writing, there must be an intervening fsetpos, fseek, or rewind operation. The current position can be specified for the fsetpos or fseek operation, if desired.

In addition to the values listed above, one of the following characters can be included in *mode* to specify the translation mode for new lines:

t
Open in text (translated) mode. In this mode, carriage-return-line-feed (CR-LF) combinations are translated into single line feeds (LF) on input and LF characters are translated to CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files

opened for reading or reading/writing, _fsopen checks for a CTRL+Z at the end of the file and removes it, if possible. This is done because using the fseek and ftell functions to move within a file that ends with a CTRL+Z may cause fseek to behave improperly near the end of the file.

b

Open in binary (untranslated) mode; the above translations are suppressed.

If t or b is not given in *mode*, the translation mode is defined by the default-mode variable _fmode. If t or b is prefixed to the argument, the function will fail and will return NULL.

The argument *shflag* is a constant expression consisting of one of the following manifest constants, defined in SHARE.H. If SHARE.COM (or SHARE.EXE for some versions of MS-DOS) is not installed, MS-DOS ignores the sharing mode. (See your system documentation for detailed information about sharing modes.)

Constant	Meaning
_SH_COMPAT	Sets compatibility mode
_SH_DENYNO	Permits read and write access
_SH_DENYRD	Denies read access to file
_SH_DENYRW	Denies read and write access to file
_SH_DENYWR	Denies write access to file

The _fsopen function should be used only under MS-DOS versions 3.0 and later. Under earlier versions of MS-DOS, the *shflag* argument is ignored.

Return Value

The _fsopen function returns a pointer to the stream. A NULL pointer value indicates an error.

fclose
_fcloseall
_fdopen
ferror
_fileno
fopen
freopen
_open
_setmode
_sopen

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FSOPEN.C:
 */
#include <stdio.h>
#include <stdlib.h>
#include <share.h>
void main( void )
{
   FILE *stream;
   /* Open output file for writing. Using _fsopen allows us to
    * ensure that no one else writes to the file while we are
    * writing to it.
    */
   if( (stream = _fsopen( "outfile", "wt", _SH_DENYWR )) != NULL )
   {
      fprintf( stream, "No one else in the network can write "
                       "to this file until we are done.\n" );
      fclose( stream );
   }
   /* Now others can write to the file while we read it. */
   system( "type outfile" );
}
```

_fstat

#include <sys\types.h>, <sys\stat.h>

Syntax int _fstat(int *handle*, struct _stat **buffer*);

Parameter	Description
handle	Handle of open file
buffer	Pointer to structure to store results

The _fstat function obtains information about the open file associated with *handle* and stores it in the structure pointed to by *buffer*. The structure, whose type _stat is defined in SYS\STAT.H, contains the following fields:

Field	Value
st_atime	Time of last access of file.
st_ctime	Time of creation of file.
st_dev	Either *handle* in the case of a device, or the drive number of the disk containing the file if the drive number is available (MS-DOS only); otherwise 0.
st_mode	Bit mask for file-mode information. The _S_IFCHR bit is set if *handle* refers to a device. The _S_IFREG bit is set if *handle* refers to an ordinary file. The read/write bits are set according to the file's permission mode. (_S_IFCHR and other constants are defined in SYS\STAT.H.)
st_mtime	Time of last modification of file.
st_nlink	Always 1.
st_rdev	Either *handle* in the case of a device, or the drive number of the disk containing the file if the drive number is available (MS-DOS only); otherwise 0.
st_size	Size of the file in bytes.

In the File Allocation Table (FAT) file system supported by MS-DOS, the creation and last access times of a file are not kept separately. Therefore, st_atime, st_ctime, and st_mtime are always the same. These fields have different values under UNIX.

Note that if *handle* refers to a device, the size and time fields in the _stat structure are not meaningful. Also, because STAT.H uses the _dev_t type, which is defined in TYPES.H, you must include TYPES.H before STAT.H in your code.

Return Value

The _fstat function returns the value 0 if the file-status information is obtained. A return value of -1

indicates an error; in this case, errno is set to EBADF, indicating an invalid file handle.

Use _fstat for compatibility with ANSI naming conventions of non-ANSI functions. Use fstat and link with OLDNAMES.LIB for UNIX compatibility.

_access
_chmod
_filelength
_stat

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FSTAT.C: This program uses _fstat to report
 * the size of a file named FSTAT.OUT.
 */
#include <io.h>
#include <fcntl.h>
#include <time.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main( void )
{
   struct _stat buf;
   int fh, result;
   char buffer[] = "A line to output";
   if( (fh = _open( "f_stat.out", _O_CREAT | _O_WRONLY | _O_TRUNC )) ==  -1 )
      exit( 1 );
   _write( fh, buffer, strlen( buffer ) );
   /* Get data associated with "fh": */
   result = _fstat( fh, &buf );
   /* Check if statistics are valid: */
   if( result != 0 )
      printf( "Bad file handle\n" );
   else
   {
      printf( "File size     : %ld\n", buf.st_size );
      printf( "Drive number  : %d\n", buf.st_dev );
      printf( "Time modified : %s", ctime( &buf.st_atime ) );
   }
   _close( fh );
}
```

ftell

#include <stdio.h>

Syntax long ftell(FILE **stream*);

Parameter	**Description**
stream	Target FILE structure

The ftell function gets the current position of the file pointer (if any) associated with *stream*. The position is expressed as an offset relative to the beginning of the stream.

Note that when a file is opened for appending data, the current file position is determined by the last I/O operation, not by where the next write would occur. For example, if a file is opened for an append and the last operation was a read, the file position is the point where the next read operation would start, not where the next write would start. (When a file is opened for appending, the file position is moved to end-of-file before any write operation.) If no I/O operation has yet occurred on a file opened for appending, the file position is the beginning of the file.

Return Value

The ftell function returns the current file position. The value returned by ftell may not reflect the physical byte offset for streams opened in text mode, because text mode causes carriage-return-line-feed translation. Use ftell in conjunction with the fseek function to return to file locations correctly. On error, the function returns -1L and errno is set to one of the following constants, defined in ERRNO.H:

Constant	**Description**
EBADF	Bad file number. The *stream* argument is not a valid file-handle value or does not refer to an open file.
EINVAL	Invalid argument. An invalid *stream* argument was passed to the function.

On devices incapable of seeking (such as terminals and printers), or when *stream* does not refer to an open file, the return value is undefined.

fgetpos
fseek
_lseek
_tell

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FTELL.C: This program opens a file named FTELL.C
 * for reading and tries to read 100 characters. It
 * then uses ftell to determine the position of the
 * file pointer and displays this position.
 */
#include <stdio.h>
FILE *stream;
void main( void )
{
   long position;
   char list[100];
   if( (stream = fopen( "ftell.c", "rb" )) != NULL )
   {
      /* Move the pointer by reading data: */
      fread( list, sizeof( char ), 100, stream );
      /* Get position after read: */
      position = ftell( stream );
      printf( "Position after trying to read 100 bytes: %ld\n", position );
      fclose( stream );
   }
}
```

_ftime

#include <sys\types.h>, <sys\timeb.h>

Syntax void _ftime(struct _timeb **timeptr*);

Parameter	Description
timeptr	Pointer to structure defined in SYS\TIMEB.H

The _ftime function gets the current time and stores it in the structure pointed to by *timeptr*. The _timeb structure is defined in SYS\TIMEB.H. It contains four fields (dstflag, millitm, time, and timezone), which have the following values:

dstflag
: Nonzero if daylight saving time is currently in effect for the local time zone. (See _tzset for an explanation of how daylight saving time is determined.)

millitm
: Fraction of a second in milliseconds. On MS-DOS systems, the last digit is always 0 because millitm is incremented to the nearest one-hundredth of a second.

time
: Time in seconds since midnight (00:00:00), January 1, 1970, Universal coordinated time (UCT).

timezone
: Difference in minutes, moving westward, between Universal Coordinated Time and local time. The value of timezone is set from the value of the global variable _timezone (see _tzset).

Return Value

The _ftime function gives values to the fields in the structure pointed to by *timeptr*. It does not return a value.

asctime
ctime
gmtime
localtime
time
_tzset

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FTIME.C: This program uses _ftime to obtain the current
 * time and then stores this time in timebuffer.
 */
#include <stdio.h>
#include <sys\timeb.h>
#include <time.h>
void main( void )
{
   struct _timeb timebuffer;
   char *timeline;
   _ftime( &timebuffer );
   timeline = ctime( & ( timebuffer.time ) );
   printf( "The time is %.19s.%hu %s", timeline, timebuffer.millitm,
&timeline[20] );
}
```

_fullpath

#include <stdlib.h>

Syntax char *_fullpath(char **buffer*, const char **pathname*, size_t *maxlen*);

Parameter	Description
buffer	Full path buffer
path	Relative path name
maxlen	Length of the buffer pointed to by *buffer*

The _fullpath routine converts the partial path stored in *path* to a fully qualified path that is stored in *buffer*. Unlike _makepath, the _fullpath routine can be used with .\ and ..\ in the path.

If the length of the fully qualified path is greater than the value of *maxlen*, then NULL is returned; otherwise, the address of *buffer* is returned.

If the *buffer* is NULL, _fullpath will allocate a buffer of _MAX_PATH size using malloc and the *maxlen* argument is ignored. It is the caller's responsibility to deallocate this buffer (using free) as appropriate.

If the *path* argument specifies a disk drive, the current directory of this drive is combined with the path. If the drive is not valid, _fullpath returns NULL.

Return Value

The _fullpath function returns a pointer to the buffer containing the absolute path (*buffer*). If there is an error, _fullpath returns NULL.

_getcwd
_getdcwd
_makepath
_splitpath

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* FULLPATH.C: This program demonstrates how _fullpath
 * creates a full path from a partial path.
 */
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <direct.h>
char full[_MAX_PATH], part[_MAX_PATH];
void main( void )
{
   while( 1 )
   {
      printf( "Enter partial path or ENTER to quit: " );
      gets( part );
      if( part[0] == 0 )
         break;
      if( _fullpath( full, part, _MAX_PATH ) != NULL )
         printf( "Full path is: %s\n", full );
      else
         printf( "Invalid path\n" );
   }
}
```

_fwopen

#include <stdio.h>

Syntax FILE * _fwopen(struct _wopeninfo **wopeninfo*, struct _wsizeinfo **wsizeinfo*, const char * *mode*);

Parameter	Description
wopeninfo	Pointer to a _wopeninfo structure
wsizeinfo	Pointer to a _wsizeinfo structure
mode	Type of access permitted

The _fwopen function is a high-level call that opens a new QuickWin window, returning a file-stream pointer. This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see Chapter 8 of *Programming Techniques*.

The _wopeninfo and _wsizeinfo structures, declared in STDIO.H, are used to pass window initialization information, including the window's initial size and position on the screen. You can pass NULL for these arguments to accept QuickWin defaults or declare variables of these two structure types and fill in their fields.

If you declare _wopeninfo and _wsizeinfo variables, assign the _QWINVER macro to the _version field. _QWINVER is the current QuickWin version, defined in STDIO.H.

For the _wopeninfo variable, assign a null-terminated string to the _title field containing the desired window title. You can also optionally set the size of the window's screen buffer in the _wbufsize field. The default is 2,048 bytes, but you can pass some other number or the value _WINBUFINF. This causes the buffer to be reallocated continually so that all window output is retained for scrolling.

For the _wsizeinfo variable, assign one of the following values to the _type field:

Value	Meaning
_WINSIZEMIN	Minimize the window
_WINSIZEMAX	Maximize the window
_WINSIZECHAR	Use character coordinates for the window size

If the type is _WINSIZECHAR, you must supply the _x, _y, _h, and _w values in the remainder of the structure. They specify the upper-left corner and the height and width of the window (in characters).

The *mode* argument is a pointer to the stream I/O mode. The _fwopen function accepts the same mode values as the STDIO.H fopen function:

Type	Description
"r"	Opens for reading
"w"	Opens for writing
"r+"	Opens for both reading and writing
"w+"	Opens for both reading and writing

In addition to the values listed above, one of the following characters can be included in *mode* to specify the translation mode for newline characters:

Mode	Meaning
t	Open in text (translated) mode
b	Open in binary (untranslated) mode

If t or b is not given in *mode*, the translation mode is defined by the default-mode variable _fmode. If t or b is prefixed to the argument, the function fails and returns NULL.

If _fwopen is successful, the returned stream can be passed to standard STDIO.H functions such as fread, fwrite, and fprintf. If you write to a stream and then read from it, or if you read from a stream and then write to it, call the STDIO.H rewind function between the I/O calls. To close an open window stream, call the STDIO.H function fclose. If you have opened a window with _fwopen, you can use the _fileno macro to obtain a file handle, which you can then pass to other QuickWin calls, such as _wsetscreenbuf or _wsetsize.

Return Value

If successful, the _fwopen function returns a stream pointer (FILE *) to the new window. A return value of NULL indicates an error.

fclose
_fileno
_wabout
_wclose
_wgetfocus
_wgetscreenbuf
_wgetsize
_wmenuclick
_wopen
_wsetfocus
_wsetscreenbuf
_wsetsize
_wyield

Standards: None
16-Bit: QWIN

```
/* FWOPEN.C - Demonstrate opening QuickWin windows
 * with _fwopen
 */

#include <io.h>
#include <stdio.h>
#include <stdlib.h>

#define OPENFLAGS "w"              /* Access permission */

void main()
{
    struct _wopeninfo wininfo;     /* Open information */
    char wintitle[32]="QuickWin "; /* Title for window */
    FILE *wp;                      /* FILE ptr to window */
    int nRes;                      /* I/O result */

    /* Set up window info structure for _fwopen */
    wininfo._version = _QWINVER;
    wininfo._title = wintitle;
    wininfo._wbufsize = _WINBUFDEF;

    /* Check current 'exit behavior' setting */
    /* Test should be true, since default is _WINEXITPERSIST */
    /* So set new behavior to prompt user */
    if( _wgetexit() == _WINEXITPERSIST )
       _wsetexit( _WINEXITPROMPT );

    /* Create a new window */
    /* NULL second argument accepts default size/position */
    wp = _fwopen( &wininfo, NULL, OPENFLAGS );
    if( wp == NULL )
    {
        printf( "***ERROR: _fwopen\n" );
        exit( -1 );
    }

    /* Write in the window */
    nRes = fprintf( wp, "Hello, QuickWin!\n" );

    /* Close the window */
    nRes = fclose( wp );

    /* On exiting anywhere, user is prompted
     * to keep window on screen or not
     */
    exit( 0 );

}
```

fwrite

#include <stdio.h>

Syntax size_t fwrite(const void *_buffer_, size_t _size_, size_t _count_, FILE * _stream_);

Parameter	Description
buffer	Pointer to data to be written
size	Item size in bytes
count	Maximum number of items to be written
stream	Pointer to FILE structure

The fwrite function writes up to *count* items, of length *size* each, from *buffer* to the output *stream*. The file pointer associated with *stream* (if there is one) is incremented by the number of bytes actually written.

If *stream* is opened in text mode, each carriage return is replaced with a carriage-return-line-feed pair. The replacement has no effect on the return value.

Return Value

The fwrite function returns the number of full items actually written, which may be less than *count* if an error occurs. Also, if an error occurs, the file-position indicator cannot be determined.

fread
_write

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_gcvt

#include <stdlib.h> Required only for function declarations

Syntax char *_gcvt(double *value*, int *digits*, char **buffer*);

Parameter	Description
value	Value to be converted
digits	Number of significant digits stored
buffer	Storage location for result

The _gcvt function converts a floating-point *value* to a character string (which includes a decimal point and a possible sign byte) and stores the string in *buffer*. The *buffer* should be large enough to accommodate the converted value plus a terminating null character ('\0'), which is appended automatically. If a buffer size of significant digits + 1 is used, the function will overwrite the end of the buffer. This is because the converted string includes a decimal point and can contain sign and exponent information. There is no provision for overflow.

The _gcvt function attempts to produce *digits* significant digits in decimal format. If this is not possible, it produces *digits* significant digits in exponential format. Trailing zeros may be suppressed in the conversion.

Return Value

The _gcvt function returns a pointer to the string of digits. There is no error return.

Use _gcvt for compatibility with ANSI naming conventions of non-ANSI functions. Use gcvt and link with OLDNAMES.LIB for UNIX compatibility.

atof
atoi
atol
_ecvt
_fcvt

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN

```
/* _GCVT.C: This program converts -3.1415e5
 * to its string representation.
 */
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   char buffer[50];
   double source = -3.1415e5;
   _gcvt( source, 7, buffer );
   printf( "source: %f  buffer: '%s'\n", source, buffer );
   _gcvt( source, 7, buffer );
   printf( "source: %e  buffer: '%s'\n", source, buffer );
}
```

_getactivepage

#include <graph.h>

Syntax short __far _getactivepage(void);

The _getactivepage function returns the number of the current active page.

This function works differently in QuickWin applications. For specific information, see *Programming Techniques.*

Return Value

The function returns the number of the current active video page. All hardware combinations support at least one page (page number 0).

_getvideoconfig
_getvisualpage
_grstatus
_setactivepage
_setvideomode
_setvisualpage

Standards: None
16-Bit: MS-DOS, QWIN

```
/* PAGE.C illustrates video page functions including:
 *  _getactivepage  _getvisualpage
 *  _setactivepage  _setvisualpage
*/
#include <conio.h>
#include <graph.h>
#include <stdlib.h>
void main( void )
{
   short  oldvpage, oldapage, page;
   struct _videoconfig vc;
   _getvideoconfig( &vc );
   if( vc.numvideopages < 4 )
      exit( 1 );              /* Fail for or monochrome. */
   oldapage  = _getactivepage();
   oldvpage  = _getvisualpage();
   _displaycursor( _GCURSOROFF );
   /* Draw arrows in different place on each page. */
   for( page = 0; page < vc.numvideopages; page++ )
   {
      _setactivepage( page );
      _setvisualpage( page );
      _clearscreen( _GCLEARSCREEN );
      _settextposition( 12, 10 * page );
      _outtext( ">>>>>>>>>>>" );
   }
   while( !_kbhit() )
      /* Cycle through pages 1 to 3 to show moving image. */
      for( page = 0; page < vc.numvideopages; page++ )
         _setvisualpage( page );
   _getch();
   /* Restore original page (normally 0) to restore screen. */
   _setactivepage( oldapage );
   _setvisualpage( oldvpage );
   _displaycursor( _GCURSORON );
}
```

_getarcinfo

#include <graph.h>

Syntax short __far _getarcinfo(struct _xycoord __far **start*, struct _xycoord __far **end*, struct _xycoord __far **fillpoint*);

Parameter	Description
start	Starting point of arc
end	Ending point of arc
fillpoint	Point at which pie fill will begin

The _getarcinfo function determines the endpoints in viewport coordinates of the most recently drawn arc or pie.

If successful, the _getarcinfo function updates the *start* and *end* _xycoord structures to contain the endpoints (in viewport coordinates) of the arc drawn by the most recent call to one of the _arc or _pie functions.

In addition, *fillpoint* specifies a point from which a pie can be filled. This is useful for filling a pie in a color different from the border color. After a call to _getarcinfo, change colors using the _setcolor function. Use the color, along with the coordinates in *fillpoint*, as arguments for the _floodfill function.

Return Value

The _getarcinfo function returns a nonzero value if successful. If neither the _arc nor the _pie function has been successfully called since the last time the screen was cleared or a new graphics mode or viewport was selected, the _getarcinfo function returns 0.

_arc functions
_floodfill
_getvideoconfig
_grstatus
_pie functions

Standards:	None
16-Bit:	MS-DOS, QWIN

_amblksiz Variable

Syntax unsigned _amblksiz;

The _amblksiz variable controls memory heap granularity.

It is declared in the MALLOC.H include file as follows:

extern unsigned int _amblksiz;

The value of _amblksiz is used to control how memory is obtained from the operating system for the heap. The initial requested size for a segment of memory for the heap manager is based on the amount of current allocation request plus overhead for the heap manager's bookkeeping chores, that is, just enough to satisfy the allocation request at hand (for example, a malloc or calloc). However, when the heap manager grows a segment, it does so in multiples of _amblksiz. The value of _amblksiz represents a trade-off between the number of times the operating system must be called to grow a segment to its maximum size (no more than 640K for MS-DOS) and the amount of memory potentially wasted (available but not used) at the end of the heap.

The default value of _amblksiz is 8K. The value can be changed by direct assignment in your program. For example:

```
_amblksiz = 2048;
```

The actual value used internally by the heap manager will be the given value, rounded up to the nearest whole power of 2 (so an _amblksiz value of 4K-1 is the same as a value of 4K).

Note that adjusting the value of _amblksiz affects allocation in the near, far, and based heaps. The value of _amblksiz has no effect on huge memory blocks (those allocated with _halloc and similar functions).

_daylight, _timezone, and _tzname Variables

Syntax int _daylight

long _timezone

char *_tzname[2]

The _daylight, _timezone, and _tzname variables are global time-zone variables used in time functions.

They are declared in the TIME.H include file as follows:

extern int _daylight;

extern long _timezone;

extern char *_tzname[2];

Some time and date routines use the _daylight, _timezone, and _tzname variables to make local-time adjustments. Whenever a program calls the _ftime, localtime, or _tzset function, the values of _daylight, _timezone, and _tzname are determined from the value of the TZ environment variable. If you do not explicitly set the value of TZ, the default value of "PST8PDT" is used. The following list shows each variable and its value:

Variable	Value
_daylight	Nonzero if a daylight-saving-time zone (DST) is specified in TZ; otherwise, 0. Default value is 1.
_timezone	Difference in seconds between Universal Coordinated Time and the local time. Default value is 28 800.
_tzname[0]	Three-letter time-zone name derived from the TZ environment variable. Default value is "PST" (Pacific standard time).
_tzname[1]	Three-letter daylight-saving-time-zone name derived from the TZ environment variable. Default value is "PDT" (Pacific daylight time). If the DST zone is omitted from TZ, _tzname[1] is an empty string.

_doserrno, errno, _sys_errlist, and _sys_nerr Variables

Syntax int _doserrno;

int errno;

char *_sys_errlist[ ];

int _sys_nerr;

The _doserrno, errno, _sys_errlist, and_sys_nerr variables contain error codes and are used by the perror and strerror routines to print error information.

These variables are declared in the STDLIB.H include file. Manifest constants for the errno variables

are declared in the ERRNO.H include file. The declarations are as follows:

```
extern int _doserrno;

extern int errno;

extern char _sys_errlist[ ];

extern int_sys_nerr;
```

The errno variable is set to an integer value to reflect the type of error that has occurred in a system-level call. Each errno value is associated with an error message, which can be printed with the perror routine or stored in a string with the strerror routine.

Note that only some routines set the errno variable. If a routine sets errno, the description of the routine in the reference section says so explicitly.

The value of errno reflects the error value for the last call that set errno. However, this value is not necessarily reset by later successful calls. To avoid confusion, test for errors immediately after a call.

The include file ERRNO.H contains the definitions of the errno values. However, not all of the definitions given in ERRNO.H are used in MS-DOS. Some of the values in ERRNO.H are present to maintain compatibility with the UNIX (and XENIX) operating system.

The errno values in MS-DOS are a subset of the values for errno in XENIX systems. Thus, the errno value is not necessarily the same as the actual error code returned by a MS-DOS system call. To access the actual MS-DOS error code, use the _doserrno variable, which contains this value.

In general, you should use _doserrno only for error detection in operations involving input and output, since the errno values for input and output errors have MS-DOS error-code equivalents. In other cases, the value of _doserrno is undefined.

The_sys_errlist variable is an array; the perror and strerror routines use it to process error information. The_sys_nerr variable tells how many elements the_sys_errlist array contains.

The following table gives the errno values for MS-DOS, the system error message for each value, and the value of each constant. Note that only the ERANGE and EDOM constants are specified in the ANSI standard.

Constant	**Meaning**	**Value**
E2BIG	Argument list too long	7
EACCES	Permission denied	13
EBADF	Bad file number	9
EDEADLOCK	Resource deadlock would occur	36
EDOM	Math argument	33
EEXIST	File exists	17
EINVAL	Invalid argument	22
EMFILE	Too many open files	24
ENOENT	No such file or directory	2
ENOEXEC	Exec format error	8
ENOMEM	Not enough memory	12

ENOSPC	No space left on device	28
ERANGE	Result too large	34
EXDEV	Cross-device link	18

_fileinfo Variable

Syntax int _fileinfo

The _fileinfo variable determines whether or not a process's open file information, in the form of the _C_FILE_INFO entry in the environment, will be passed to child processes. If _fileinfo is 0, the _C_FILE_INFO information is not passed to the child processes. If _fileinfo is not 0, the _C_FILE_INFO information is passed to child processes.

By default, _fileinfo is 0 and thus the _C_FILE_INFO information is not passed to child processes. There are two ways to modify the default value of _fileinfo:

- Link the supplied object file FILEINFO.OBJ into your program. Use the /NOE LINK option to avoid multiple symbol definitions.
- Set the _fileinfo variable to a nonzero value directly within your program.

_fmode Variable

Syntax int _fmode

The _fmode variable controls the default file-translation mode.

It is declared in the STDLIB.H include file as follows:

extern int _fmode;

By default, the value of _fmode is _O_TEXT, causing files to be translated in text mode (unless specifically opened or set to binary mode). When _fmode is set to _O_BINARY, the default mode is binary. You can set _fmode to the flag _O_BINARY by linking with BINMODE.OBJ or by assigning _fmode the _O_BINARY value.

Locale Macros

The two ANSI macros, MB_LEN_MAX and MB_CUR_MAX, are useful when writing portable programs for international markets. The following list describes them and gives the include file where each is defined.

Macro	Description
MB_CUR_MAX	The MB_CUR_MAX macro, defined in STDLIB.H, expands to the maximum number of bytes in a multibyte character of the current locale.
MB_LEN_MAX	The MB_LEN_MAX macro, defined in LIMITS.H, gives the maximum number of bytes in a multibyte character.

_osmajor, _osminor, _osver, _winmajor, _winminor, _winver, _osversion, _osmode, and _cpumode Variables

#include <stdlib.h>
<dos.h> (_osversion only)

Syntax

```
extern unsigned char _osmajor;
extern unsigned char _osminor;
extern unsigned int _osver;
extern unsigned char _winmajor;
extern unsigned char _winminor;
extern unsigned int _winver;
extern unsigned int _osversion;
extern unsigned char _osmode;
extern unsigned char _cpumode;
```

The _osmajor, _osminor, and _osversion variables specify the version number of MS-DOS in use. The _osmajor variable holds the " major" version number, and the _osminor variable stores the "minor" version number. Thus, under MS-DOS version 5.0, _osmajor is 5 and _osminor is 0. The _osver variable holds both values: its high byte contains the major version number and its low byte contains the minor version number. The _osversion variable also holds both values, but in reverse order. The _osver variable is generally more useful for testing the MS-DOS version because it has the two bytes in the correct order for comparison.

The _winmajor, _winminor, and _winver variables specify the version of Windows in use. These variables are not available in the MS-DOS libraries. The _winmajor variable holds the major version number, and the _winminor variable holds the minor version number. Thus, under Windows 3.10 (also called Windows 3.1), _winmajor is 3 and _winminor is 10.

Note that the _osver and _winver variables contain the version numbers in the order that makes sense for comparison, not in the order returned by the respective get version system calls.

Note MS-DOS and Windows minor version numbers are sometimes specified without a trailing 0. For example, DOS 3.3 means version 3.30, and Windows 3.1 means 3.10.

These variables are useful for creating programs that run in different versions of MS-DOS and Windows. For example, you can test the _osmajor variable before making a call to _sopen; if the major version number is earlier (less) than 3, _open should be used instead of _sopen.

The _osmode variable indicates the currently running operating system: _DOS_MODE or _WIN_MODE.

The _cpumode variable indicates the mode of the currently running operating system: _REAL_MODE or _PROT_MODE.

environ Variable

Syntax char **_environ;

The environ variable is a pointer to the strings in the process environment.

It is declared in the STDLIB.H include file as follows:

extern char **_environ;

The _environ variable provides access to memory areas containing process-specific information.

The _environ variable is a pointer to an array of pointers to the strings that constitute the process environment. The environment consists of one or more entries of the form

`NAME=`*`string`*

where NAME is the name of an environment variable and *string* is the value of that variable. The string can be empty. The initial environment settings are taken from the operating-system environment at the time of program execution.

The getenv and _putenv routines use the _environ variable to access and modify the environment table. When _putenv is called to add or delete environment settings, the environment table changes size; its location in memory may also change, depending on the program's memory requirements. The _environ variable is adjusted in these cases and always points to the correct table location.

_psp Variable

Syntax unsigned _psp;

The _psp variable contains the segment address of the program segment prefix (PSP) for the process. It is declared in the STDLIB.H include file as follows:

extern unsigned int _psp;

The PSP contains execution information about the process, such as a copy of the command line that invoked the process and the return address on process termination or interrupt. The _psp variable can be used to form a long pointer to the PSP, where _psp is the segment value and 0 is the offset value.

Note that the _psp variable is supported only in MS-DOS.

_pgmptr Variable

Syntax extern char __far * _pgmptr;

The **_**pgmptr variable is automatically initialized at startup and points to a string that contains a copy of the path of the just-started program; _pgmptr will contain whatever the program that spawns it passes as argv[0]. The command interpreter (COMMAND.COM) will pass a full path; other programs may pass a filename, a relative path, or a full path. It is declared in the STDLIB.H include file as follows:

extern char __far *_pgmptr;

It is defined as a global variable in the run-time library and declared in CRT0DAT.ASM, which is part of the startup code. This code is linked to any module that contains a main function.

The following program demonstrates the use of **_**pgmptr:

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

void main( void )
{
   printf("The full path of the executing program is : %Fs\n", _pgmptr);
}
```

In MS-DOS versions 3.0 and later, *argv*[0] also contains a pointer to the full path of the executing program.

Global Variables

_amblksiz	_fileinfo	_pgmptr
_cpumode	_fmode	_psp
_daylight	_osmajor	_sys_errlist
_doserrno	_osminor	_sys_nerr
environ	_osmode	_timezone
errno	_osversion	_tzname

_getbkcolor

#include <graph.h>

Syntax long __far _getbkcolor(void);

The _getbkcolor function returns the current background color. The default is 0.

In a color text mode such as _TEXTC80, _setbkcolor accepts, and _getbkcolor returns, a color index. For example, _setbkcolor(2L) sets the background color to color index 2. The actual color displayed depends on the palette mapping for color index 2. The default for color index 2 is green in a color text mode.

In a color graphics mode such as _ERESCOLOR, _setbkcolor accepts, and _getbkcolor returns, a color value (as used in _remappalette). The value for the simplest background colors is given by the manifest constants defined in the GRAPH.H include file. For example, _setbkcolor(_GREEN) sets the background color in a graphics mode to green. These manifest constants are provided as a convenience in defining and manipulating the most common colors. In general, the actual range of colors is much greater.

In most cases, whenever a color argument is long, it refers to a color value, and whenever it is short, it refers to a color index. The two exceptions are _setbkcolor and _getbkcolor, described above. For a more complete discussion of colors, see _remappalette.

Return Value

The function returns the current background color. There is no error return.

_remappalette
_setbkcolor

Standards: None

16-Bit: MS-DOS, QWIN

getc, getchar

#include <stdio.h>

Syntax int getc(FILE **stream*);

int getchar(void);

Parameter	**Description**
stream	Current stream

The getc routine reads a single character from the *stream* position and increments the associated file pointer (if there is one) to point to the next character. The getchar routine is identical to getc(stdin).

The getc and getchar routines are similar to fgetc and _fgetchar, respectively, but are implemented both as macros and functions.

Return Value

Both getc and getchar return the character read. A return value of EOF indicates an error or end-of-file condition. Use ferror or feof to determine whether an error or end-of-file occurred.

fgetc
_fgetchar
_getch
_getche
putc
putchar
ungetc

getc

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

getchar

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN

```
/* GETC.C: This program uses getchar to read a single line
 * of input from stdin, places this input in buffer, then
 * terminates the string before printing it to the screen.
 */
#include <stdio.h>
void main( void )
{
   char buffer[81];
   int i, ch;
   printf( "Enter a line: " );
   /* Read in single line from "stdin": */
   for( i = 0; (i < 80) &&  ((ch = getchar()) != EOF) && (ch != '\n'); i++ )
      buffer[i] = (char)ch;
   /* Terminate string with null character: */
   buffer[i] = '\0';
   printf( "%s\n", buffer );
}
```

_getch, _getche

#include <conio.h> Required only for function declarations

Syntax int _getch(void);

int _getche(void);

The _getch function reads a single character from the console without echoing. The _getche function reads a single character from the console and echoes the character read. Neither function can be used to read CTRL+C. Neither function can be used with QuickWin programs.

When reading a function key or cursor-moving key, the _getch and _getche functions must be called twice; the first call returns 0 or 0xE0, and the second call returns the actual key code.

Return Value

Both the _getch and _getche functions return the character read. There is no error return.

Standards:	None
16-Bit:	MS-DOS

_cgets
getchar
_ungetch

```
/* GETCH.C: This program reads characters from
 * the keyboard until it receives a 'Y' or 'y'.
 */
#include <conio.h>
#include <ctype.h>
void main( void )
{
   int ch;
   _cputs( "Type 'Y' when finished typing keys: " );
   do
   {
      ch = _getch();
      ch = toupper( ch );
   } while( ch != 'Y' );
   _putch( ch );
   _putch( '\r' );    /* Carriage return */
   _putch( '\n' );    /* Line feed       */
}
```

_getcolor

#include <graph.h>

Syntax short __far _getcolor(void);

The _getcolor function returns the current graphics color index. The default is the highest legal index in the current palette.

Return Value

The _getcolor function returns the current color index.

_setcolor

Standards: None

16-Bit: MS-DOS, QWIN

_getcurrentposition Functions

#include <graph.h>

Syntax struct _xycoord __far _getcurrentposition(void);

struct _wxycoord __far _getcurrentposition_w(void);

The _getcurrentposition functions return the coordinates of the current graphics output position. The _getcurrentposition function returns the position as an _xycoord structure, defined in GRAPH.H.

The _xycoord structure contains the following elements:

Element	Description
short xcoord	*x* coordinate
short ycoord	*y* coordinate

The _getcurrentposition_w function returns the position as a _wxycoord structure, defined in GRAPH.H.

The _wxycoord structure contains the following elements:

Element	Description
double wx	window *x* coordinate
double wy	window *y* coordinate

The current position can be changed by the _lineto, _moveto, and _outgtext functions.

The default position, set by _setvideomode, _setvideomoderows, or _setviewport, is the center of the viewport.

Only graphics output starts at the current position; these functions do not affect text output, which begins at the current text position. (See _settextposition for more information.)

Return Value

The _getcurrentposition functions return the coordinates of the current graphics output position. There is no error return.

_grstatus
_lineto functions
_moveto functions
_outgtext

Standards: None

16-Bit: MS-DOS, QWIN

```
/* GCURPOS.C: This program sets a random current location,
 * then gets that location with _getcurrentposition.
 */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <graph.h>
char   buffer[255];
void main( void )
{
   struct _videoconfig vc;
   struct _xycoord position;
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   _getvideoconfig( &vc );
   /* Move to random location and report that location. */
   _moveto( rand() % vc.numxpixels, rand() % vc.numypixels );
   position = _getcurrentposition();
   sprintf( buffer, "x = %d, y = %d", position.xcoord, position.ycoord );
   _settextposition( 1, 1 );
   _outtext( buffer );
   _getch();
   _setvideomode( _DEFAULTMODE );
}
```

_getcwd

#include <direct.h> Required only for function declarations

Syntax char *_getcwd(char **buffer*, int *maxlen*);

Parameter	**Description**
buffer	Storage location for path
maxlen	Maximum length of path

The _getcwd function gets the full path of the current working directory for the default drive and stores it at *buffer*. The integer argument *maxlen* specifies the maximum length for the path. An error occurs if the length of the path (including the terminating null character) exceeds *maxlen.*

The *buffer* argument can be NULL; a buffer of at least size *maxlen* (more only if necessary) will automatically be allocated, using malloc, to store the path. This buffer can later be freed by calling free and passing it the _getcwd return value (a pointer to the allocated buffer).

Note that _getcwd returns a string that represents the path of the current working directory. If the current working directory is set to the root, the string will end with a backslash (\ ). If the current working directory is set to a directory other than the root, the string will end with the name of the directory and not with a backslash.

Return Value

The _getcwd function returns a pointer to *buffer*. A NULL return value indicates an error, and errno is set to one of the following values:

Value	**Meaning**
ENOMEM	Insufficient memory to allocate *maxlen* bytes (when a NULL argument is given as *buffer*)
ERANGE	Path longer than *maxlen* characters

Use _getcwd for compatibility with ANSI naming conventions of non-ANSI functions. Use getcwd and link with OLDNAMES.LIB for UNIX compatibility.

_chdir
_mkdir
_rmdir

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* This program places the name of the current directory in
 * the buffer array, then displays the name of the current
 * directory on the screen. Specifying a length of _MAX_DIR
 * leaves room for the longest legal directory name.
 */
#include <direct.h>
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   char buffer[_MAX_DIR];
   /* Get the current working directory: */
   if( _getcwd( buffer, _MAX_DIR ) == NULL )
      perror( "_getcwd error" );
   else
      printf( "%s\n", buffer );
}
```

_getdcwd

#include <direct.h> Required only for function declarations

Syntax char *_getdcwd(int *drive*, char **buffer*, int *maxlen*);

Parameter	Description
drive	Disk drive
buffer	Storage location for path
maxlen	Maximum length of path

The _getdcwd function gets the full path of the current working directory on the specified drive and stores it at *buffer*. The argument *maxlen* specifies the maximum length for the path. An error occurs if the length of the path (including the terminating null character) exceeds *maxlen*.

The *drive* argument specifies the drive (0 = default drive, 1=A, 2=B, etc.). The *buffer* argument can be NULL; a buffer of at least size *maxlen* (more only if necessary) will automatically be allocated, using malloc, to store the path. This buffer can later be freed by calling free and passing it the _getdcwd return value (a pointer to the allocated buffer).

Note that _getdcwd returns a string that represents the path of the current working directory. If the current working directory is set to the root, the string will end with a backslash (\). If the current working directory is set to a directory other than the root, the string will end with the name of the directory and not with a backslash.

Return Value

The _getdcwd function returns *buffer*. A NULL return value indicates an error, and errno is set to one of the following values:

Value	Meaning
ENOMEM	Insufficient memory to allocate *maxlen* bytes (when a NULL argument is given as *buffer*)
ERANGE	Path longer than *maxlen* characters

_chdir
_getcwd
_getdrive
_mkdir
_rmdir

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* GETDRIVE.C illustrates drive functions including:
 *      _getdrive       _chdrive        _getdcwd
 */
#include <stdio.h>
#include <conio.h>
#include <direct.h>
#include <stdlib.h>
#include <ctype.h>
void main( void )
{
   int ch, drive, curdrive;
   static char path[_MAX_PATH];
   /* Save current drive. */
   curdrive = _getdrive();
   printf( "Available drives are: \n" );
   /* If we can switch to the drive, it exists. */
   for( drive = 1; drive <= 26; drive++ )
      if( !_chdrive( drive ) )
         printf( "%c: ", drive + 'A' - 1 );
   while( 1 )
   {
      printf( "\nType drive letter to check or ESC to quit: " );
      ch = _getch();
      if( ch == 27 )
         break;
      if( isalpha( ch ) )
         _putch( ch );
      if( _getdcwd( toupper( ch ) - 'A' + 1, path, _MAX_PATH ) != NULL )
         printf( "\nCurrent directory on that drive is %s\n", path );
   }
   /* Restore original drive.*/
   _chdrive( curdrive );
   printf( "\n" );
}
```

_getdrive

#include <direct.h>

Syntax int _getdrive(void);

The _getdrive function returns the current (default) drive (1=A, 2=B, etc.).

Return Value

The return value is stated above. There is no error return.

_chdrive
_dos_getdrive
_dos_setdrive
_getcwd
_getdcwd

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

getenv

#include <stdlib.h> Required only for function declarations

Syntax char *getenv(const char **varname*);

Parameter	Description
varname	Name of environment variable

The getenv function searches the list of environment variables for an entry corresponding to *varname*. Environment variables define the environment in which a process executes. (For example, the LIB environment variable defines the default search path for libraries to be linked with a program.) Because the getenv function is case sensitive, the *varname* variable should match the case of the environment variable.

The getenv function returns a pointer to an entry in the environment table. It is, however, only safe to retrieve the value of the environment variable using the returned pointer. To modify the value of an environmental variable, use the _putenv function.

The getenv and _putenv functions use the copy of the environment contained in the global variable _environ to access the environment. Programs that use the *envp* argument to main and the _putenv function may retrieve invalid information. The safest programming practice is to use getenv and _putenv.

The getenv function operates only on the data structures accessible to the run-time library and not on the environment "segment" created for the process by the operating system.

Return Value

The getenv function returns a pointer to the environment table entry containing the current string value of *varname*. The return value is NULL if the given variable is not currently defined.

_putenv

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* GETENV.C: This program uses getenv to retrieve
 * the LIB environment variable and then uses
 * _putenv to change it to a new value.
 */
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   char *libvar;
   /* Get the value of the LIB environment variable. */
   libvar = getenv( "LIB" );
   if( libvar != NULL )
      printf( "Original LIB variable is: %s\n", libvar );
   /* Attempt to change path. Note that this only affects the environment
    * variable of the current process. The command processor's environment
    * is not changed.
    */
   _putenv( "LIB=c:\\mylib;c:\\yourlib" );
   /* Get new value. */
   libvar = getenv( "LIB" );
   if( libvar != NULL )
      printf( "New LIB variable is: %s\n", libvar );
}
```

_getfillmask

#include <graph.h>

Syntax unsigned char __far * __far _getfillmask(unsigned char __far **mask*);

Parameter	Description
mask	Mask array

Some graphics routines (_ellipse, _floodfill, _pie, _polygon, and _rectangle) can fill part or all of the screen with the current color. The fill mask controls the pattern used for filling.

The _getfillmask function returns the current fill mask. The mask is an 8-by-8-bit array, in which each bit represents a pixel. If the bit is 1, the corresponding pixel is set to the current color; if the bit is 0, the pixel is left unchanged. The mask is repeated over the entire fill area. If no fill mask is set, or if *mask* is NULL, a solid (unpatterned) fill is performed using the current color.

Return Value

If no mask is set, the function returns NULL. Otherwise, it returns the current fill mask.

_ellipse functions
_floodfill
_pie functions
_polygon functions
_rectangle functions
_setfillmask

Standards: None
16-Bit: MS-DOS, QWIN

```
/* GFILLMSK.C: This program illustrates
 * _getfillmask and _setfillmask
 */
#include <conio.h>
#include <stdlib.h>
#include <graph.h>
void ellipsemask( short x1, short y1, short x2, short y2,
   const unsigned char __far *newmask );
unsigned char mask1[8] =
   { 0x43, 0x23, 0x7c, 0xf7, 0x8a, 0x4d, 0x78,  0x39 };
unsigned char mask2[8] =
   { 0x18, 0xad, 0xc0, 0x79, 0xf6, 0xc4, 0xa8,  0x23 };
char oldmask[8];
void main( void )
{
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   /* Set first fill mask and draw rectangle. */
   _setfillmask( mask1 );
   _rectangle( _GFILLINTERIOR, 20, 20, 100, 100 );
   _getch();
   /* Call routine that saves and restores mask. */
   ellipsemask( 60, 60, 150, 150, mask2 );
   _getch();
   /* Back to original mask. */
   _rectangle( _GFILLINTERIOR, 120, 120, 190, 190 );
   _getch();
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}

/* Draw an ellipse with a specified fill mask. */
void ellipsemask( short x1, short y1, short x2, short y2,
   const unsigned char __far *newmask )
{
   unsigned char savemask[8];
   _getfillmask( savemask );                    /* Save mask        */
   _setfillmask( newmask );                     /* Set new mask     */
   _ellipse( _GFILLINTERIOR, x1, y1, x2, y2 );  /* Use new mask     */
   _setfillmask( savemask );                    /* Restore original */
}
```

_getfontinfo

#include <graph.h>

Syntax short __far _getfontinfo(struct _fontinfo __far **fontbuffer*);

Parameter	Description
fontbuffer	Buffer to hold font information

The _getfontinfo function gets the current font characteristics and stores them in a _fontinfo structure, defined in GRAPH.H.

The _fontinfo structure contains the following elements:

Element	Contents
int type	Specifies vector (1) or bitmapped (0) font
int ascent	Specifies pixel distance from top to baseline
int pixwidth	Specifies the character width in pixels; 0 indicates a proportional font
int pixheight	Specifies the character height in pixels
int avgwidth	Specifies the average character width in pixels
char filename [81]	Specifies the filename, including the path
char facename [32]	Specifies the font name

If you have used the _setfont function to load and scale a vector font, the _getfontinfo function may return unexpected values for the ascent, pixheight, and avgwidth elements. This is because _getfontinfo reads from the current font's .FON file, which _setfont does not update.

The _getfontinfo function works differently in QuickWin applications. For specific information, see *Programming Techniques.*

Return Value

The _getfontinfo function returns a negative number if a font has not been registered or loaded.

_getgtextextent
_outgtext
_registerfonts
_setfont
_setgtextvector
_unregisterfonts

Standards: None
16-Bit: MS-DOS, QWIN

_getgtextextent

#include <graph.h>

Syntax short __far _getgtextextent(const char __far **text*);

Parameter	**Description**
text	Text to be analyzed

The _getgtextextent function returns the width in pixels that would be required to print the *text* string using _outgtext with the current font.

This function is particularly useful for determining the size of text that uses proportionally spaced fonts.

Return Value

The _getgtextextent function returns the width in pixels. It returns -1 if a font has not been registered.

_getfontinfo
_outgtext
_registerfonts
_setfont
_unregisterfonts

Standards: None

16-Bit: MS-DOS, QWIN

_getgtextvector

#include <graph.h>

Syntax struct _xycoord __far _getgtextvector(void);

The _getgtextvector function gets the current orientation for font text output. The current orientation is used in calls to the _outgtext function.

The text-orientation vector, which determines the direction of font-text rotation on the screen, is returned in a structure of type _xycoord. The *xcoord* and *ycoord* members of the structure describe the vector. The text-rotation options are shown below:

(x, y)	**Text Orientation**
(1,0)	Horizontal text (default)
(0,1)	Rotated 90 degrees counterclockwise
(-1,0)	Rotated 180 degrees
(0,-1)	Rotated 270 degrees counterclockwise

Return Value

The _getgtextvector function returns the current text-orientation vector in a structure of type _xycoord.

_getgtextextent
_grstatus
_outgtext
_setfont
_setgtextvector

Standards: None

16-Bit: MS-DOS, QWIN

_getimage Functions

#include <graph.h>

Syntax void __far _getimage(short *x1*, short *y1*, short *x2*, short *y2*, char __huge **image*);

void __far _getimage_w(double *wx1*, double *wy1*, double *wx2*, double *wy2*, char __huge **image*);

void __far _getimage_wxy(struct_wxycoord __far **pwxy1*, struct_wxycoord __far **pwxy2*, char __huge **image*);

Parameter	Description
x1, *y1*	Upper-left corner of bounding rectangle
x2, *y2*	Lower-right corner of bounding rectangle
wx1, *wy1*	Upper-left corner of bounding rectangle
wx2, *wy2*	Lower-right corner of bounding rectangle
pwxy1	Upper-left corner of bounding rectangle
pwxy2	Lower-right corner of bounding rectangle
image	Storage buffer for screen image

The _getimage functions store the screen image defined by a specified bounding rectangle into the buffer pointed to by *image*.

The _getimage function defines the bounding rectangle with the view coordinates (*x1*, *y1*) and (*x2*, *y2*).

The _getimage_w function defines the bounding rectangle with the window coordinates (*wx1*, *wy1*) and (*wx2*, *wy2*).

The _getimage_wxy function defines the bounding rectangle with the window-coordinate pairs *pwxy1* and *pwxy2*.

The buffer must be large enough to hold the image.

Return Value

None. Use _grstatus to check success.

_grstatus
_imagesize functions
_putimage functions

Standards: None
16-Bit: MS-DOS, QWIN

```
/* GIMAGE.C: This example illustrates animation routines,
 *  including:   _imagesize     _getimage     _putimage
 */
#include <conio.h>
#include <stddef.h>
#include <stdlib.h>
#include <malloc.h>
#include <graph.h>
short action[5]  = { _GPSET,   _GPRESET, _GXOR,     _GOR,      _GAND      };
char *descrip[5] = {  "PSET  ", "PRESET", "XOR   ", "OR    ", "AND   "  };
void exitfree( char __huge *buffer );
void main( void )
{
   char  __huge *buffer;   /* Far pointer (with _fmalloc) could be used. */
   long  imsize;
   short i, x, y = 30;
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   /* Measure the image to be drawn and allocate memory for it. */
   imsize = (size_t)_imagesize( -16, -16, +16, +16 );
   buffer = _halloc( imsize, sizeof( char ) );
   if ( buffer == (char __far *)NULL )
      exit( 1 );
   _setcolor( 3 );
   for ( i = 0; i < 5; i++ )
   {
      /* Draw ellipse at new position and get a copy of it. */
      x = 50; y += 40;
      _ellipse( _GFILLINTERIOR, x - 15, y - 15, x + 15, y + 15 );
      _getimage( x - 16, y - 16, x + 16, y + 16, buffer );
      if( _grstatus() ) exitfree( buffer );     /* Quit on error */
      /* Display action type and copy a row of ellipses with that type. */
      _settextposition( 1, 1 );
      _outtext( descrip[i] );
      while( x < 260 )
      {
         x += 5;
         _putimage( x - 16, y - 16, buffer, action[i] );
         if( _grstatus() < 0 )      /* Ignore warnings, quit on errors. */
            exitfree( buffer );
      }
      _getch();
   }
   exitfree( buffer );
}
void exitfree( char __huge *buffer )
{
   _hfree( buffer );
   exit( !_setvideomode( _DEFAULTMODE ) );
}
```

_getlinestyle

#include <graph.h>

Syntax unsigned short __far _getlinestyle(void);

Some graphics routines (_lineto, _polygon, and _rectangle) output straight lines to the screen. The type of line can be controlled with the current line-style mask.

The _getlinestyle function returns the current line-style mask. The mask is a 16-bit array in which each bit represents a pixel in the line being drawn. If the bit is 1, the corresponding pixel is set to the color of the line (the current color). If the bit is 0, the corresponding pixel is left unchanged. The mask is repeated over the length of the line. The default mask is 0xFFFF (a solid line).

Return Value

If no mask has been set, _getlinestyle returns the default mask.

_lineto functions
_polygon functions
_rectangle functions
_setlinestyle
_setwritemode

Standards: None
16-Bit: MS-DOS, QWIN

```
/* GLINESTY.C: This program illustrates
 * _setlinestyle and _getlinestyle.
 */
#include <conio.h>
#include <stdlib.h>
#include <graph.h>
void zigzag( short x1, short y1, short size );
void main( void )
{
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXCOLORMODE ) )
      exit( 1 );
   /* Set line style and draw rectangle. */
   _setlinestyle( 0x4d );
   _rectangle( _GBORDER, 10, 10, 60, 60 );
   _getch();
   /* Draw figure with function that changes and restores line style. */
   zigzag( 100, 100, 90 );
   _getch();
   /* Original style reused. */
   _rectangle( _GBORDER, 190, 190, 130, 130 );
   _getch();
   _setvideomode( _DEFAULTMODE );
}
/* Draw box with changing line styles. Restore original style. */
void zigzag( short x1, short y1, short size )
{
   short x, y, oldcolor;
   unsigned short oldstyle;
   unsigned short style[16] = { 0x0001, 0x0003, 0x0007, 0x000f,
                                0x001f, 0x003f, 0x007f, 0x00ff,
                                0x01ff, 0x03ff, 0x07ff, 0x0fff,
                                0x1fff, 0x3fff, 0x7fff, 0xffff };
   oldcolor = _getcolor();
   oldstyle = _getlinestyle();            /* Save old line style.         */
   for( x = 3, y = 3; x < size; x += 3, y += 3 )
   {
      _setcolor( x % 16 );
      _setlinestyle( style[x % 16] );     /* Set and use new line styles */
      _rectangle( _GBORDER, x1 - x, y1 - y, x1 + x, y1 + y );
   }
   _setlinestyle( oldstyle );             /* Restore old line style.      */
   _setcolor( oldcolor );
}
```

_getphyscoord

#include <graph.h>

Syntax struct _xycoord __far _getphyscoord(short *x*, short *y*);

Parameter	Description
x, *y*	View coordinates to translate

The _getphyscoord function translates the view coordinates (*x*, *y*) to physical coordinates and returns them in an _xycoord structure, defined in GRAPH.H.

The _xycoord structure contains the following elements:

Element	Description
short xcoord	*x* coordinate
short ycoord	*y* coordinate

Return Value

None.

_getviewcoord functions
_grstatus
_setvieworg
_setviewport

Standards: None

16-Bit: MS-DOS, QWIN

_getpid

#include <process.h> Required only for function declarations

Syntax int _getpid(void);

The _getpid function returns the process ID, an integer that uniquely identifies the calling process.

In the MS-DOS environment, the process ID is usually considered to be the address of the program segment prefix, or PSP. However, in environments with multiple MS-DOS sessions, such as Windows, this value is often not unique. Therefore, the value returned by _getpid( ) in the MS-DOS libraries is a value based on a combination of the program segment prefix and the system time at the moment when _getpid is called for the first time.

Return Value

The _getpid function returns the process ID. There is no error return.

Use _getpid for compatibility with ANSI naming conventions of non-ANSI functions. Use getpid and link with OLDNAMES.LIB for UNIX compatibility.

_mktemp

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* GETPID.C: This program uses _getpid to obtain
 * the process ID and then prints the ID.
 */
#include <stdio.h>
#include <process.h>
void main( void )
{
   /* If run from MS-DOS, shows different ID for MS-DOS than for MS-DOS
    * shell.
    */
   printf( "\nProcess id: %d\n", _getpid() );
}
```

_getpixel Functions

#include <graph.h>

Syntax short __far _getpixel(short *x*, short *y*);

short __far _getpixel_w(double *wx*, double *wy*);

Parameter	Description
x, *y*	Pixel position
wx, *wy*	Pixel position

The functions in the _getpixel family return the pixel value (a color index) at a specified location. The _getpixel function uses the view coordinate (*x*, *y*). The _getpixel_w function uses the window coordinate (*wx*, *wy*). The range of possible pixel values is determined by the current video mode. The color translation of pixel values is determined by the current palette.

Return Value

If successful, the function returns the color index. If the function fails (for example, the point lies outside the clipping region, or the program is in a text mode), it returns -1.

_getvideoconfig
_grstatus
_remapallpalette
_remappalette
_selectpalette
_setpixel functions
_setvideomode

Standards: None
16-Bit: MS-DOS, QWIN

```
/* GPIXEL.C: This program assigns different
 * colors to randomly selected pixels.
 */
#include <conio.h>
#include <stdlib.h>
#include <graph.h>
void main( void )
{
   short xvar, yvar;
   struct _videoconfig vc;
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXCOLORMODE ) )
      exit( 1 );
   _getvideoconfig( &vc );
   /* Draw filled ellipse to turn on certain pixels. */
   _ellipse( _GFILLINTERIOR, vc.numxpixels / 6, vc.numypixels / 6,
                             vc.numxpixels / 6 * 5, vc.numypixels / 6 * 5 );
   /* Draw random pixels in random colors... */
   while( !_kbhit() )
   {
      /* ...but only if they are already on (inside the ellipse). */
      xvar = rand() % vc.numxpixels;
      yvar = rand() % vc.numypixels;
      if( _getpixel( xvar, yvar ) != 0 )
      {
         _setcolor( rand() % 16 );
         _setpixel( xvar, yvar );
      }
   }
   _getch();          /* Throw away the keystroke. */
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}
```

gets

#include <stdio.h>

Syntax char *gets(char **buffer*);

Parameter	Description
buffer	Storage location for input string

The gets function reads a line from the standard input stream stdin and stores it in *buffer*. The line consists of all characters up to and including the first newline character ('\n'). The gets function then replaces the newline character with a null character ('\0') before returning the line. In contrast, the fgets function retains the newline character.

Return Value

If successful, the gets function returns its argument. A NULL pointer indicates an error or end-of-file condition. Use ferror or feof to determine which one has occurred.

fgets
fputs
puts

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN

```
/* GETS.C */
#include <stdio.h>
void main( void )
{
   char line[81];
   printf( "Input a string: " );
   gets( line );
   printf( "The line entered was: %s\n", line );
}
```

_gettextcolor

#include <graph.h>

Syntax short __far _gettextcolor(void);

The _gettextcolor function returns the color index of the current text color. The text color is set by the _settextcolor function and affects text output with the _outtext and _outmem functions only. The _setcolor function sets the color for font text output using the _outgtext function.

The default is 7 in text modes; it is the highest legal color index of the current palette in graphics modes.

Return Value

The _gettextcolor function returns the color index of the current text color.

_getvideoconfig
_outmem
_outtext
_remappalette
_selectpalette
_setcolor
_settextcolor

Standards: None

16-Bit: MS-DOS, QWIN

_gettextcursor

#include <graph.h>

Syntax short __far _gettextcursor(void);

The _gettextcursor function returns the current cursor attribute (i.e., the shape). This function works only in text video modes.

Return Value

The function returns the current cursor attribute, or -1 if an error occurs (such as a call to the function in a graphics mode).

_displaycursor
_grstatus
_settextcursor

Standards: None

16-Bit: MS-DOS, QWIN

_gettextposition

#include <graph.h>

Syntax struct _rccoord __far _gettextposition(void);

The _gettextposition function returns the current text position as an _rccoord structure, defined in GRAPH.H.

The _rccoord structure contains the following elements:

Element	Description
short row	Row coordinate
short col	Column coordinate

The text position given by the coordinates (1,1) is defined as the upper-left corner of the text window.

Text output from the _outtext and _outmem functions begins at the current text position. Font text is not affected by the current text position. Font text output begins at the current graphics output position, which is a separate position. Use the _moveto function to set the graphics output position.

Standard console routines such as printf do not use or maintain information about the display, such as current text position, which is contained in the graphics library. Therefore, you cannot use standard console routines to output graphics text.

Return Value

The function returns the current text position as an _rccoord structure, defined in GRAPH.H.

_getcurrentposition functions
_moveto functions
_outmem
_outtext
_settextposition
_settextwindow
_wrapon

Standards: None

16-Bit: MS-DOS, QWIN

_gettextwindow

#include <graph.h>

Syntax void __far _gettextwindow(short __far *_r1_, short __far *_c1_, short __far *_r2_, short __far *_c2_);

Parameter	Description
r1	Top row of current text window
c1	Leftmost column of current text window
r2	Bottom row of current text window
c2	Rightmost column of current text window

The _gettextwindow function finds the boundaries of the current text window. The text window is the region of the screen to which output from the _outtext and _outmem functions is limited. By default, this is the entire screen, unless it has been redefined by the _settextwindow function.

The window defined by _settextwindow has no effect on output from the _outgtext function. Text displayed with _outgtext is limited to the current viewport.

Return Value

None.

_gettextposition
_outmem
_outtext
_scrolltextwindow
_settextposition
_settextwindow
_wrapon

Standards:	None
16-Bit:	MS-DOS, QWIN

_getvideoconfig

#include <graph.h>

Syntax struct _videoconfig __far * __far _getvideoconfig(struct _videoconfig __far *_config_);

Parameter	Description
config	Configuration information

The _getvideoconfig function returns the current graphics environment configuration in a _videoconfig structure, defined in GRAPH.H.

The values returned reflect the currently active video adapter and monitor, as well as the current video mode.

The _videoconfig structure contains the following members, each of which is of type short:

Member	Contents
numxpixels	Number of pixels on the *x* axis
numypixels	Number of pixels on the *y* axis
numtextcols	Number of text columns available
numtextrows	Number of text rows available
numcolors	Number of color indices
bitsperpixel	Number of bits per pixel
numvideopages	Number of available video pages
adapter	Active display adapter
mode	Current video mode
monitor	Active display monitor
memory	Adapter video memory in kilobytes

The values for the adapter member of the _videoconfig structure are given by the manifest constants shown in the list below. For any applicable adapter (_CGA, _EGA, or _VGA), the corresponding Olivetti adapter (_OCGA, _OEGA, or _OVGA) represents a superset of graphics capabilities.

Adapter Constant	Meaning
_CGA	Color Graphics Adapter
_EGA	Enhanced Graphics Adapter
_HGC	Hercules Graphics Card
_MCGA	Multicolor Graphics Array
_MDPA	Monochrome Display Printer Adapter
_OCGA	Olivetti (AT&T) Color Graphics

	Adapter
_OEGA	Olivetti (AT&T) Enhanced Graphics Adapter
_OVGA	Olivetti (AT&T) Video Graphics Array
_VGA	Video Graphics Array
_SVGA	Super Video Graphics Array (VESA)

The values for the monitor member of the _videoconfig structure are given by the manifest constants listed below:

Monitor Constant	Meaning
_ANALOG	Analog monochrome and color
_ANALOGCOLOR	Analog color only
_ANALOGMONO	Analog monochrome only
_COLOR	Color (or enhanced monitor emulating a color monitor)
_ENHCOLOR	Enhanced color
_MONO	Monochrome monitor

In every text mode, including monochrome, the _getvideoconfig function returns the value 32 for the number of available colors. The value 32 indicates the range of values (0-31) accepted by the _settextcolor function. This includes 16 normal colors (0-15) and 16 blinking colors (16-31). Blinking is selected by adding 16 to the normal color index. Because monochrome text mode has fewer unique display attributes, some color indices are redundant. However, because blinking is selected in the same manner, monochrome text mode has the same range (0-31) as other text modes.

The _getvideoconfig function works differently in QuickWin applications. For specific information, see *Programming Techniques.*

Return Value

The _getvideoconfig function returns the video configuration information in a structure, as noted above. There is no error return.

_setvideomode
_setvideomoderows

Standards: None

16-Bit: MS-DOS, QWIN

```
/* GVIDCFG.C: This program displays information
 * about the current video configuration.
 */
#include <stdio.h>
#include <graph.h>
void main( void )
{
   struct _videoconfig vc;
   short  c;
   char   b[500];               /* Buffer for string */
   _getvideoconfig( &vc );
   /* Write all information to a string, then output string. */
   c  = sprintf( b,     "X pixels:     %d\n", vc.numxpixels );
   c += sprintf( b + c, "Y pixels:     %d\n", vc.numypixels );
   c += sprintf( b + c, "Text columns: %d\n", vc.numtextcols );
   c += sprintf( b + c, "Text rows:    %d\n", vc.numtextrows );
   c += sprintf( b + c, "Colors:       %d\n", vc.numcolors );
   c += sprintf( b + c, "Bits/pixel:   %d\n", vc.bitsperpixel );
   c += sprintf( b + c, "Video pages:  %d\n", vc.numvideopages );
   c += sprintf( b + c, "Mode:         %d\n", vc.mode );
   c += sprintf( b + c, "Adapter:      %d\n", vc.adapter );
   c += sprintf( b + c, "Monitor:      %d\n", vc.monitor );
   c += sprintf( b + c, "Memory:       %d\n", vc.memory );
   _outtext( b );
}
```

_getviewcoord Functions

#include <graph.h>

Syntax struct _xycoord __far _getviewcoord(short *x*, short *y*);

struct _xycoord __far _getviewcoord_w(double *wx*, double *wy*);

struct _xycoord __far _getviewcoord_wxy(struct _wxycoord __far **pwxy1*);

Parameter	Description
x, *y*	Physical point to translate
wx, *wy*	Window point to translate
pwxy1	Window point to translate

The _getviewcoord routines translate the specified coordinates (*x*, *y*) from one coordinate system to view coordinates and then return them in an _xycoord structure, defined in GRAPH.H. The _xycoord structure contains the following elements:

Element	Description
short xcoord	*x* coordinate
short ycoord	*y* coordinate

The various _getviewcoord routines translate in the following manner:

Routine	Translation
_getviewcoord	Physical coordinates (*x*, *y*) to view coordinates
_getviewcoord_w	Window coordinates (*wx*, *wy*) to view coordinates
_getviewcoord_wxy	Window coordinates structure (*pwxy1*) to view coordinates

In Microsoft C version 5.1, the function _getviewcoord was called _getlogcoord.

Return Value

The _getviewcoord function returns the coordinates as noted above. There is no error return.

_getphyscoord
_getwindowcoord
_grstatus

Standards: None

16-Bit: MS-DOS, QWIN

_getvisualpage

#include <graph.h>

Syntax short __far _getvisualpage(void);

The _getvisualpage function returns the current visual page number.

This function works differently in QuickWin applications. For specific information, see *Programming Techniques.*

Return Value

The function returns the number of the current visual page. All hardware combinations support at least one page (page number 0).

_getactivepage
_gettextcolor
_gettextposition
_outtext
_setactivepage
_settextcolor
_settextposition
_settextwindow
_setvideomode
_setvisualpage
_wrapon

Standards: None

16-Bit: MS-DOS, QWIN

_getw

#include <stdio.h>

Syntax int _getw(FILE **stream*);

Parameter	**Description**
stream	Pointer to FILE structure

The _getw function reads the next binary value of type int from the file associated with *stream* and increments the associated file pointer (if there is one) to point to the next unread character. The _getw function does not assume any special alignment of items in the stream.

Return Value

The _getw function returns the integer value read. A return value of EOF may indicate an error or end-of-file. However, because the EOF value is also a legitimate integer value, feof or ferror should be used to verify an end-of-file or error condition.

Use _getw for compatibility with ANSI naming conventions of non-ANSI functions. Use getw and link with OLDNAMES.LIB for UNIX compatibility.

The _getw function is provided primarily for compatibility with previous libraries. Note that portability problems may occur with _getw, because the size of the int type and the ordering of bytes within the int type differ across systems.

__putw

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* GETW.C: This program uses _getw to read a word
 * from a stream, then performs an error check.
 */
#include <stdio.h>
#include <stdlib.h>
void main( void )
{
   FILE *stream;
   int i;
   if( (stream = fopen( "getw.c", "rb" )) == NULL )
      printf( "Couldn't open file\n" );
   else
   {
      /* Read a word from the stream: */
      i = _getw( stream );
      /* If there is an error... */
      if( ferror( stream ) )
      {
         printf( "_getw failed\n" );
         clearerr( stream );
      }
      else
         printf( "First data word in file: 0x%.4x\n", i );
      fclose( stream );
   }
}
```

_getwindowcoord

#include <graph.h>

Syntax struct _wxycoord __far _getwindowcoord(short *x*, short *y*);

Parameter	Description
x, *y*	Viewport coordinate to translate

The _getwindowcoord function translates the view coordinates (*x*, *y*) to window coordinates and returns them in the _wxycoord structure, defined in GRAPH.H.

The _wxycoord structure contains the following elements:

Element	Description
double wx	*x* coordinate
double wy	*y* coordinate

Return Value

The function returns the coordinates in the _wxycoord structure. There is no error return.

_getphyscoord
_getviewcoord functions
_moveto functions
_setwindow

Standards: None

16-Bit: MS-DOS, QWIN

_getwritemode

#include <graph.h>

Syntax short __far _getwritemode(void);

The _getwritemode function returns the current logical write mode, which is used when drawing lines with the _lineto, _polygon, and _rectangle functions.

The default value is _GPSET, which causes lines to be drawn in the current graphics color. The other possible return values are _GXOR, _GAND, _GOR, and _GPRESET. See _putimage for more details on these manifest constants.

Return Value

The _getwritemode function returns the current logical write mode, or -1 if not in graphics mode.

_grstatus
_lineto functions
_putimage functions
_rectangle functions
_setcolor
_setlinestyle
_setwritemode

Standards: None

16-Bit: MS-DOS, QWIN

```
/* GWRMODE.C: This program illustrates
 * getwritemode _and _setwritemode.
 */
#include <conio.h>
#include <stdlib.h>
#include <graph.h>
short wmodes[5]  = { _GPSET,   _GPRESET, _GXOR,     _GOR,      _GAND     };
char *wmstr[5]   = { "PSET  ", "PRESET", "XOR   ", "OR    ", "AND   "  };
void box( short x, short y, short size, short writemode, short  fillmode );
void main( void )
{
   short i, x, y;
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXCOLORMODE ) )
      exit( 1 );
   x = y = 70;
   box( x, y, 50, _GPSET, _GFILLINTERIOR );
   _setcolor( 2 );
   box( x, y, 40, _GPSET, _GFILLINTERIOR );
   for( i = 0; i < 5; i++ )
   {
      _settextposition( 1, 1 );
      _outtext( wmstr[i] );
      box( x += 12, y += 12, 50, wmodes[i], _GBORDER );
      _getch();
    }
    _setvideomode( _DEFAULTMODE );
    exit( 0 );
}
void box( short x, short y, short size, short writemode, short  fillmode )
{
   short wm;
   wm = _getwritemode();        /* Save write mode and set new. */
   _setwritemode( writemode );
   _rectangle( fillmode, x - size, y - size, x + size, y + size );
   _setwritemode( wm );         /* Restore original write mode. */
}
```

gmtime

#include <time.h>

Syntax struct tm *gmtime(const time_t **timer*);

Parameter	**Description**
timer	Pointer to stored time

The gmtime function converts the *timer* value to a structure. The *timer* argument represents the seconds elapsed since midnight (00:00:00), January 1, 1970, Universal Coordinated Time. This value is usually obtained from a call to the time function.

The gmtime function breaks down the *timer* value and stores it in a structure of type tm, defined in TIME.H. The structure result reflects Universal Coordinated Time, not local time.

The fields of the structure type tm store the following values, each of which is an int:

Field	**Value Stored**
tm_sec	Seconds after the minute (0-59)
tm_min	Minutes after the hour (0-59)
tm_hour	Hours since midnight (0-23)
tm_mday	Day of month (1-31)
tm_mon	Month (0-11; January = 0)
tm_year	Year (current year minus 1900)
tm_wday	Day of week (0-6; Sunday = 0)
tm_yday	Day of year (0-365; January 1 = 0)
tm_isdst	Always 0 for gmtime

The gmtime, mktime, and localtime functions use a single statically allocated structure to hold the result. Each call to one of these routines destroys the result of any previous call.

If *timer* represents a date before midnight, January 1, 1970, gmtime returns NULL.

Note The target environment should attempt to determine whether daylight saving time is in effect.

Return Value

The gmtime function returns a pointer to the structure result. There is no error return.

asctime
ctime
_ftime
localtime
time

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* GMTIME.C: This program uses gmtime to convert a long-
 * integer representation of Universal Coordinated Time
 * to a structure named  newtime, then uses asctime to
 * convert this structure to an output string.
 */
#include <time.h>
#include <stdio.h>
void main( void )
{
   struct tm *newtime;
   long ltime;
   time( &ltime );
   /* Obtain Universal Coordinated Time: */
   newtime = gmtime( &ltime );
   printf( "Universal Coordinated Time is %s\n", asctime( newtime ) );
}
```

_grstatus

#include <graph.h>

Syntax short __far _grstatus(void);

The _grstatus function returns the status of the most recently used graphics function. The _grstatus function can be used immediately following a call to a graphics routine to determine if errors or warnings were generated. Return values less than 0 are errors, and values greater than 0 are warnings.

The following manifest constants are defined in the GRAPH.H header file for use with the _grstatus function:

Value	Constant	Meaning
0	_GROK	Success.
-1	_GRERROR	Graphics error.
-2	_GRMODENOTSUPPORTED	Requested video mode not supported.
-3	_GRNOTINPROPERMODE	Requested routine only works in certain video modes.
-4	_GRINVALIDPARAMETER	One or more arguments invalid.
-5	_GRFONTFILENOTFOUND	No matching font file found.
-6	_GRINVALIDFONTFILE	One or more font files invalid.
-7	_GRCORRUPTEDFONTFILE	One or more font files inconsistent.
-8	_GRINSUFFICIENTMEMORY	Not enough memory to allocate buffer or to complete a _floodfill operation.
-9	_GRINVALIDIMAGEBUFFER	Image buffer data inconsistent.
1	_GRNOOUTPUT	Nothing drawn.
2	_GRCLIPPED	Output was clipped to viewport.

3	_GRPARAMETERALTERED	One or more input arguments was altered to be within range, or pairs of arguments were interchanged to be in the proper order.

After a graphics call, use an if statement to compare the return value of _grstatus to _GROK. For example:

```
if( _grstatus < _GROK )
   /*handle graphics error*/ ;
```

The functions listed below cannot cause errors, and they all set _grstatus to _GROK:

_displaycursor	_getvideoconfig
_getactivepage	_getvisualpage
_getbkcolor	_outmem
_getgtextvector	_outtext
_gettextcolor	_unregisterfonts
_gettextposition	_wrapon
_gettextwindow	

See the list below for the graphics functions that affect _grstatus. The list shows error or warning messages that can be set by the graphics function. In addition to the error codes listed, any of these functions can produce the _GRERROR error code.

Function	**Possible _grstatus Error Codes**	**Possible _grstatus Warning Codes**
_arc functions	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER,	_GRNOOUTPUT, _GRCLIPPED
_clearscreen	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER	
_ellipse functions	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER, _GRINSUFFICIENTMEMORY	_GRNOOUTPUT, _GRCLIPPED
_floodfill functions	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER, _GRINSUFFICIENTMEMORY	_GRNOOUTPUT
_getarcinfo	_GRNOTINPROPERMODE	
_getcurrentposition functions	_GRNOTINPROPERMODE	
_getfontinfo	(_GRERROR only)	
_getgtextextent	(_GRERROR only)	
_getgtextvector	_GRPARAMETERALTERED	
_getimage functions	_GRNOTINPROPERMODE	_GRPARAMETERALTERED

_getphyscoord	_GRNOTINPROPERMODE	
_getpixel functions	_GRNOTINPROPERMODE	
_gettextcursor	_GRNOTINPROPERMODE	
_getviewcoord functions	_GRNOTINPROPERMODE	
_getwindowcoord	_GRNOTINPROPERMODE	
_getwritemode	_GRNOTINPROPERMODE	
_imagesize functions	_GRNOTINPROPERMODE	
_lineto functions	_GRNOTINPROPERMODE	_GRNOOUTPUT, _GRCLIPPED
_moveto functions	_GRNOTINPROPERMODE	
_outgtext	_GRNOTINPROPERMODE	_GRCLIPPED, _GRNOOUTPUT
_pie functions	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER, _GRINSUFFICIENTMEMORY	_GRNOOUTPUT, _GRCLIPPED
_polygon functions	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER, _GRINSUFFICIENTMEMORY	_GRNOOUTPUT, _GRCLIPPED
_putimage functions	_GRERROR, _GRNOTINPROPERMODE, _GRINVALIDPARAMETER, _GRINVALIDIMAGEBUFFER	_GRPARAMETERALTERED, _GRNOOUTPUT
_rectangle functions	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER, _GRINSUFFICIENTMEMORY	_GRNOOUTPUT, _GRCLIPPED
_registerfonts	_GRCORRUPTEDFONTFILE, _GRFONTFILENOTFOUND, _GRINSUFFICIENTMEMORY, _GRINVALIDFONTFILE	
_remappalette	_GRERROR, _GRINVALIDPARAMETER	
_remapallpalette	_GRERROR, _GRINVALIDPARAMETER	
_scrolltextwindow	_GRNOOUTPUT	
_selectpalette	_GRNOTINPROPERMODE	_GRINVALIDPARAMETER
_setactivepage	_GRINVALIDPARAMETER	
_setbkcolor	_GRINVALIDPARAMETER	_GRPARAMETERALTERED
_setcliprgn	_GRNOTINPROPERMODE	_GRPARAMETERALTERED
_setcolor	_GRNOTINPROPERMODE	_GRPARAMETERALTERED
_setfont	_GRERROR, _GRFONTFILENOTFOUND, _GRINSUFFICIENTMEMORY, _GRPARAMETERALTERED	
_setgtextvector	_GRPARAMETERALTERED	
_setpixel	_GRNOTINPROPERMODE	_GRNOOUTPUT
_settextcolor		_GRPARAMETERALTERED

_settextcursor	_GRNOTINPROPERMODE	
_settextposition		_GRPARAMETERALTERED
_settextrows	_GRINVALIDPARAMETER	_GRPARAMETERALTERED
_settextwindow		_GRPARAMETERALTERED
_setvideomode	_GRERROR, _GRMODENOTSUPPORTED, _GRINVALIDPARAMETER	
_setvideomoderows	_GRERROR, _GRMODENOTSUPPORTED, _GRINVALIDPARAMETER	
_setvieworg	_GRNOTINPROPERMODE	
_setviewport	_GRNOTINPROPERMODE	_GRPARAMETERALTERED
_setvisualpage	_GRINVALIDPARAMETER	
_setwindow	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER	_GRPARAMETERALTERED
_setwritemode	_GRNOTINPROPERMODE, _GRINVALIDPARAMETER	

The _grstatus function works differently in QuickWin applications. For specific information, see *Programming Techniques.*

Return Value

The _grstatus function returns the status of the most recently used graphics function.

_arc functions
_ellipse functions
_floodfill functions
_lineto functions
_pie functions
_remapallpalette
_setactivepage
_setbkcolor
_setcolor
_setpixel functions
_settextcolor
_settextcursor
_setvisualpage
_setwindow
_setwritemode

Standards: None

16-Bit: MS-DOS

_halloc

#include <malloc.h> Required only for function declarations

Syntax void __huge *_halloc(long *num,* size_t *size*);

Parameter	Description
num	Number of elements
size	Length in bytes of each element

The _halloc function allocates a huge array from the operating system consisting of *num* elements, each of which is *size* bytes long. Each element is initialized to 0. If the size of the array is greater than 128K (131,072 bytes), the size of an array element must then be a power of 2.

Use the _hfree function to deallocate a block of memory returned by halloc.

Return Value

The _halloc function returns a void huge pointer to the allocated space, which is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void huge, use a type cast on the return value. If the request cannot be satisfied, the return value is NULL.

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* HALLOC.C: This program uses _halloc to allocate space for
 * 30,000 long integers, then uses _hfree to deallocate the memory.
 */
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
void main( void )
{
   long __huge *hbuf;
   /* Allocate huge buffer */
   hbuf = (long __huge *)_halloc( 30000L, sizeof( long ) );
   if ( hbuf == NULL )
      printf( "Insufficient memory available\n" );
   else
      printf( "Memory successfully allocated\n" );
   /* Free huge buffer */
   _hfree( hbuf );
}
```

_hard Functions

#include <dos.h>

Syntax void _harderr(void(__far **handler*)( ));

void _hardresume(int *result*);

void _hardretn(int *error*);

Parameter	Description
handler ( )	New INT 0x24 handler
result	Handler return argument
error	Error to return from

These three functions are used to handle critical error conditions that use MS-DOS interrupt 0x24. The _harderr function installs a new critical-error handler for interrupt 0x24.

When a critical error occurs, control is passed to the function specified in the _harderr call. The _hardresume and _hardretn functions control how the program will return from the critical error handler.

The _hardresume function returns to MS-DOS the code that encountered the critical error.

The _hardretn function returns directly to the application program that issued the INT 0x21 MS-DOS system call, which, in turn, encountered the critical error.

The _harderr function does not directly install the handler pointed to by *handler*; instead, _harderr installs a handler that calls the function referenced by *handler*. The handler calls the function with the following arguments:

handler(unsigned *deverror*, unsigned *errcode*, unsigned __far **devhdr*);

The *deverror* argument is the device error code. It contains the AX register value passed by MS-DOS to the INT 0x24 handler. The *errcode* argument is the DI register value that MS-DOS passes to the handler. The low-order byte of *errcode* can be one of the following values:

Code	Meaning
0	Attempt to write to a write-protected disk
1	Unknown unit
2	Drive not ready
3	Unknown command
4	Cyclic-redundancy-check error in data
5	Bad drive-request structure length
6	Seek error
7	Unknown media type
8	Sector not found
9	Printer out of paper
10	Write fault
11	Read fault
12	General failure

The *devhdr* argument is a far pointer to a device header that contains descriptive information about the device on which the error occurred. The user-defined handler must not change the information in the device-header control block.

Errors on Disk Devices

If the error occurred on a disk device, the high-order bit (bit 15) of the *deverror* argument will be set to 0, and the *deverror* argument will indicate the following:

Bit	Meaning
15	Disk error if false (0).
14	Not used.
13	"Ignore" response not allowed if false (0).
12	"Retry" response not allowed if false (0).
11	"Fail" response not allowed if false (0). Note that MS-DOS changes "fail" to "abort".
10, 9	**Code** **Location** 00 DOS 01 File allocation table 10 Directory 11 Data area
8	Read error if false; write error if true.

The low-order byte of *deverror* indicates the drive in which the error occurred (0 = drive A, 1 = drive B, etc.).

Errors on Other Devices

If the error occurs on a device other than a disk drive, the high-order bit (bit 15) of the *deverror* argument is 1. The attribute word located at offset 4 in the device-header block indicates the type of device that had the error. If bit 15 of the attribute word is 0, the error is a bad memory image of the file allocation table. If the bit is 1, the error occurred on a character device and bits 0-3 of the attribute word indicate the type of device, as shown in the following list:

Bit	Meaning
0	Current standard input
1	Current standard output
2	Current null device
3	Current clock device

Restrictions on Handler Functions

The user-defined handler function can issue only system calls 0x01 through 0x0C, or 0x59. Thus, many of the standard C run-time functions (such as the I/O and _heap functions) cannot be used in a hardware error handler. System call 0x59 can be used to obtain further information about the error that occurred.

Using _hardresume and _harderr

If the handler returns, it can do so in several different ways:

- Via the return statement
- By calling the _hardresume function
- By calling the _hardretn function

If the handler returns from _hardresume or from a return statement, control returns to MS-DOS.

The _hardresume function should be called only from within the user-defined hardware error-handler function. The result supplied to _hardresume must be one of the following constants:

Constant	Action
_HARDERR_ABORT	Aborts the program by issuing INT 0x24
_HARDERR_FAIL	Fails the system call that is in progress (this is not supported on MS-DOS 2.x)
_HARDERR_IGNORE	Ignores the error
_HARDERR_RETRY	Retries the operation

The _hardretn function allows the user-defined hardware error handler to return directly to the application program rather than returning to MS-DOS. The application resumes at the point just after the failing I/O function request. The _hardretn function should be called only from within a user-defined hardware error-handler function.

The error argument of _hardretn should be an MS-DOS error code, as opposed to the XENIX-style error code that is available in errno. Refer to *MS-DOS Encyclopedia* (Duncan, ed.; Redmond, Wa.: Microsoft Press, 1988) or *Programmer's PC Sourcebook* 2nd ed. (Hogan; Redmond, Wa.: Microsoft Press, 1991) for information about the MS-DOS error codes that may be returned by a given MS-DOS function call.

If the failing I/O function request is an INT 0x21 function greater than or equal to function 0x38, _hardretn will then return to the application with the carry flag set and the AX register set to the _hardretn *error* argument. If the failing INT 0x21 function request is less than function 0x38 and the function can return an error, the AL register will be set to 0xFF on return to the application. If the failing INT 0x21 does not have a way of returning an error condition (which is true of certain INT 0x21 functions below 0x38), the error argument of _hardretn is not used, and no error code is returned to the application.

Return Value

None.

_chain_intr
_dos_getvect
_dos_setvect

Standards: None
16-Bit: MS-DOS

_heapadd Functions

#include <malloc.h> Required only for function declarations

Syntax int _heapadd(void __far **memblock*, size_t *size*);

int _bheapadd(__segment *seg*, void __based (void) **memblock*, size_t *size*);

Parameter	Description
seg	Based-heap segment selector
memblock	Pointer to heap memory
size	Size in bytes of memory to add

The _heapadd and _bheapadd functions add an unused piece of memory to the heap. The _bheapadd function adds the memory to the based heap specified by *seg*. The _heapadd function looks at the segment value and, if it is DGROUP, adds the memory to the near heap. Otherwise, _heapadd adds the memory to the far heap.

Return Value

These functions return 0 if successful, or -1 if an error occurred.

free functions
_halloc
_hfree
malloc functions
realloc functions

_heapadd

Standards: None
16-Bit: MS-DOS

_bheapadd

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* HEAPMIN.C: This program illustrates heap
 * management using _heapadd and _heapmin.
 */
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <malloc.h>
void heapdump( char *msg );     /* Prototype */
char s1[] = { "Here are some strings that we use at first, then don't\n"  };
char s2[] = { "need any more. We'll give their space to the heap.\n" };
void main( void )
{
   int *p[3], i;
   printf( "%s%s", s1, s2 );
   heapdump( "Initial heap" );
   /* Give space of used strings to heap. */
   if ( _heapadd( s1, sizeof( s1 )) == -1 )
      printf("Error.\n");
   if ( _heapadd( s2, sizeof( s2 )) == -1 )
      printf("Error.\n");
   heapdump( "After adding used strings" );
   /* Allocate some blocks. Some may use string blocks from _heapadd. */
   for( i = 0; i < 2; i++ )
      if( (p[i] = (int *)calloc( 10 * (i + 1), sizeof( int ) )) == NULL )
      {
         --i;
         break;
      }
   heapdump( "After allocating memory" );
   /* Free some of the blocks. */
   free( p[1] );
   free( p[2] );
   heapdump( "After freeing memory" );
   /* Minimize heap. */
   _heapmin();
   heapdump( "After compacting heap" );
}
/* Walk through heap entries, displaying information about each block. */
void heapdump( char *msg )
{
   _HEAPINFO hi;
   printf( "%s\n", msg );
   hi._pentry = NULL;
   while( _heapwalk( &hi ) == _HEAPOK )
      printf( "\t%s block at %Fp of size %u\t\n",
              hi._useflag == _USEDENTRY ? "USED" : "FREE",
              hi._pentry, hi._size );
   printf("Press any key.\n");
   _getch();
}
```

_heapchk Functions

#include <malloc.h>

Syntax int _heapchk(void);

int _bheapchk(__segment *seg*);

int _fheapchk(void);

int _nheapchk(void);

Parameter	Description
seg	Specified base heap

The _heapchk routines help to debug heap-related problems by checking for minimal consistency of the heap. Each function checks a particular heap, as listed below:

Function	Heap Checked
_heapchk	Depends on data model of program
_bheapchk	Based heap specified by *seg* value
_fheapchk	Far heap (outside the default data segment)
_nheapchk	Near heap (inside the default data segment)

In large data models (that is, compact-, large-, and huge-model programs), _heapchk maps to _fheapchk. In small data models (tiny-, small-, and medium-model programs), _heapchk maps to _nheapchk.

For _heapchk, if the *seg* value is _NULLSEG, all based heap segments are checked; otherwise, only the specified one is checked.

Return Value

All four routines return an integer value that is one of the following manifest constants (defined in MALLOC.H):

Constant	Meaning
_HEAPBADBEGIN	Initial header information cannot be found, or it is bad.
_HEAPBADNODE	Bad node has been found, or the heap is damaged.
_HEAPEMPTY	Heap has not been initialized.
_HEAPOK	Heap appears to be consistent.

_heapset functions
_heapwalk functions

_heapchk

Standards: None
16-Bit: MS-DOS

_bheapchk, _fheapchk

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_nheapchk

Standards: None
16-Bit: MS-DOS

```
/* HEAPCHK.C: This program checks the heap for
 * consistency and prints an appropriate message.
 */
#include <malloc.h>
#include <stdio.h>
void main( void )
{
   int  heapstatus;
   char *buffer;
   /* Allocate and deallocate some memory */
   if( (buffer = (char *)malloc( 100 )) != NULL )
      free( buffer );
   /* Check heap status */
   heapstatus = _heapchk();
   switch( heapstatus )
   {
   case _HEAPOK:
      printf(" OK - heap is fine\n" );
      break;
   case _HEAPEMPTY:
      printf(" OK - heap is empty\n" );
      break;
   case _HEAPBADBEGIN:
      printf( "ERROR - bad start of heap\n" );
      break;
   case _HEAPBADNODE:
      printf( "ERROR - bad node in heap\n" );
      break;
   }
}
```

_heapmin Functions

#include <malloc.h>

Syntax int _heapmin(void);

int _bheapmin(__segment *seg*)

int _fheapmin(void);

int _nheapmin(void);

Parameter	Description
seg	Specified based-heap selector

The _heapmin functions minimize the heap by releasing unused heap memory to the operating system.

The various _heapmin functions release unused memory in these heaps:

Function	Heap Minimized
_heapmin	Depends on data model of program.
_bheapmin	Based heap specified by *seg* value; _NULLSEG specifies all based heaps.
_fheapmin	Far heap (outside default data segment).
_nheapmin	Near heap (inside default data segment).

In large data models (that is, compact-, large-, and huge-model programs), _heapmin maps to _fheapmin. In small data models (tiny-, small-, and medium-model programs), _heapmin maps to _nheapmin.

For _heapmin, if the supplied *seg* value is _NULLSEG, all based heap segments are minimized; otherwise, only the specified one is minimized.

Based-heap segments are never freed (i.e., unlinked from the based heap list and released back to the operating system) by the _bheapmin function. The _bfreeseg function is used for that purpose.

Return Value

The _heapmin functions return 0 if the function completed successfully, or -1 in the case of an error.

_bfreeseg
free functions
malloc functions

_heapmin

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_bheapmin, **_fheapmin**, **_nheapmin**

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_heapset Functions

#include <malloc.h>

Syntax int _heapset(unsigned int *fill*);

int _bheapset(__segment *seg*, unsigned int *fill*);

int _fheapset(unsigned int *fill*);

int _nheapset(unsigned int *fill*);

Parameter	Description
fill	Fill character
seg	Specified based-heap segment selector

The _heapset family of routines helps debug heap-related problems in programs by showing free memory locations or nodes unintentionally overwritten.

The _heapset routines first check for minimal consistency on the heap in a manner identical to that of the _heapchk functions. In addition, the _heapset functions set each byte of the heap's free entries to the *fill* value. This known value shows which memory locations of the heap contain free nodes and which locations contain data that were unintentionally written to freed memory.

The various _heapset functions check and fill these heaps:

Function	Heap Filled
_heapset	Depends on data model of program.
_bheapset	Based heap specified by *seg* value; _NULLSEG specifies all based heaps.
_fheapset	Far heap (outside default data segment).
_nheapset	Near heap (inside default data segment).

In large data models (that is, compact-, large-, and huge-model programs), _heapset maps to _fheapset. In small data models (tiny-, small-, and medium-model programs), _heapset maps to _nheapset.

For _heapset, if the *seg* value is _NULLSEG, all based heap segments are checked; otherwise, only

the specified one is checked.

Return Value

All four routines return an int whose value is one of the following manifest constants (defined in MALLOC.H):

Constant	Meaning
_HEAPBADBEGIN	Initial header information cannot be found, or it is invalid.
_HEAPBADNODE	Bad node has been found, or the heap is damaged.
_HEAPEMPTY	Heap has not been initialized.
_HEAPOK	Heap appears to be consistent.

_heapset

Standards: None
16-Bit: MS-DOS

_bheapset, _fheapset

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_nheapset

Standards: None
16-Bit: MS-DOS

_heapchk functions
_heapwalk functions

```
/* HEAPSET.C: This program checks the heap and
 * fills in free entries with the character 'Z'.
 */
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
void main( void )
{
   int heapstatus;
   char *buffer;
   if( (buffer = malloc( 1 )) == NULL )  /* Make sure heap is initialized */
      exit( 0 );
   heapstatus = _heapset( 'Z' );         /* Fill in free entries */
   switch( heapstatus )
   {
   case _HEAPOK:
      printf( "OK - heap is fine\n" );
      break;
   case _HEAPEMPTY:
      printf( "OK - heap is empty\n" );
      break;
   case _HEAPBADBEGIN:
      printf( "ERROR - bad start of heap\n" );
      break;
   case _HEAPBADNODE:
      printf( "ERROR - bad node in heap\n" );
      break;
   }
   free( buffer );
}
```

_heapwalk Functions

#include <malloc.h>

Syntax int _heapwalk(_HEAPINFO *_entryinfo_);

int _bheapwalk(__segment _seg_, _HEAPINFO *_entryinfo_);

int _fheapwalk(_HEAPINFO *_entryinfo_);

int _nheapwalk(_HEAPINFO *_entryinfo_);

Parameter	Description
entryinfo	Buffer to contain heap information
seg	Based-heap segment selector

The _heapwalk family of routines helps debug heap-related problems in programs.

The _heapwalk routines walk through the heap, traversing one entry per call, and return a pointer to a structure of type _HEAPINFO that contains information about the next heap entry. The _HEAPINFO type, defined in MALLOC.H, contains the following elements:

Element	Description
int far *_pentry	Heap entry pointer
size_t _size	Size of heap entry
int _useflag	Entry "in use" flag

A call to _heapwalk that returns _HEAPOK stores the size of the entry in the _size field and sets the _useflag field to either _FREEENTRY or _USEDENTRY (both are constants defined in MALLOC.H). To obtain this information about the first entry in the heap, pass the _heapwalk routine a pointer to a _HEAPINFO structure whose _pentry member is NULL.

The various _heapwalk functions walk through and gather information on these heaps:

Function	Heap Walked
_heapwalk	Depends on data model of program.
_bheapwalk	Based heap specified by *seg* value; _NULLSEG specifies all based heaps.
_fheapwalk	Far heap (outside default data segment).
_nheapwalk	Near heap (inside default data segment).

In large data models (that is, compact-, large-, and huge-model programs), _heapwalk maps to _fheapwalk. In small data models (tiny-, small-, and medium-model programs), _heapwalk maps to _nheapwalk.

For _heapwalk, if the *seg* value is _NULLSEG, all based heap segments will be traversed; otherwise, only the specified based heap is walked.

Return Value

All three routines return one of the following manifest constants (defined in MALLOC.H):

Constant	Meaning
_HEAPBADBEGIN	The initial header information cannot be found, or it is invalid.
_HEAPBADNODE	A bad node has been found, or the heap is damaged.
_HEAPBADPTR	The _pentry field of the _HEAPINFO structure does not contain a valid pointer into the heap.
_HEAPEND	The end of the heap has been reached successfully.
_HEAPEMPTY	The heap has not been initialized.
_HEAPOK	No errors so far; the _HEAPINFO structure contains information about the next entry.

_heapwalk

Standards: None
16-Bit: MS-DOS

_bheapwalk, _fheapwalk

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_nheapwalk

Standards: None
16-Bit: MS-DOS

_heapchk functions
_heapset functions

```
/* HEAPWALK.C: This program "walks" the heap, starting
 * at the beginning (_pentry = NULL). It prints out each
 * heap entry's use, location, and size. It also prints
 * out information about the overall state of the heap as
 * soon as _heapwalk returns a value other than _HEAPOK.
*/
#include <stdio.h>
#include <malloc.h>
void heapdump( void );
void main( void )
{
   char *buffer;
   heapdump();
   if( (buffer = malloc( 59 )) != NULL )
   {
      heapdump();
      free( buffer );
   }
   heapdump();
}
void heapdump( void )
{
   _HEAPINFO hinfo;
   int heapstatus;
   hinfo._pentry = NULL;
   while( ( heapstatus = _heapwalk( &hinfo ) ) == _HEAPOK )
   { printf( "%6s block at %Fp of size %4.4X\n",
        ( hinfo._useflag == _USEDENTRY ? "USED" : "FREE" ),
          hinfo._pentry, hinfo._size );
   }
   switch( heapstatus )
   {
   case _HEAPEMPTY:
      printf( "OK - empty heap\n" );
      break;
   case _HEAPEND:
      printf( "OK - end of heap\n" );
      break;
   case _HEAPBADPTR:
      printf( "ERROR - bad pointer to heap\n" );
      break;
   case _HEAPBADBEGIN:
      printf( "ERROR - bad start of heap\n" );
      break;
   case _HEAPBADNODE:
      printf( "ERROR - bad node in heap\n" );
      break;
   }
}
```

_hfree

#include <malloc.h> Required only for function declarations

Syntax void _hfree(void __huge **memblock*);

Parameter	Description
memblock	Pointer to allocated memory block

The _hfree function deallocates a memory block; the freed memory is returned to the operating system. The *memblock* argument points to a memory block previously allocated through a call to _halloc. The number of bytes freed is the number of bytes specified when the block was allocated.

Note that attempting to free an invalid *memblock* argument (one not allocated with _halloc) may affect subsequent allocation and cause errors.

Return Value

None.

_halloc

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* HALLOC.C: This program uses _halloc to allocate space for
 * 30,000 long integers, then uses _hfree to deallocate the memory.
 */
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
void main( void )
{
   long __huge *hbuf;
   /* Allocate huge buffer */
   hbuf = (long __huge *)_halloc( 30000L, sizeof( long ) );
   if ( hbuf == NULL )
      printf( "Insufficient memory available\n" );
   else
      printf( "Memory successfully allocated\n" );
   /* Free huge buffer */
   _hfree( hbuf );
}
```

_hypot, _hypotl

#include <math.h>

Syntax double _hypot(double *x*, double *y*);

long double _hypotl(long double *x*, long double *y*);

Parameter	Description
x, *y*	Floating-point values

The _hypot and _hypotl functions calculate the length of the hypotenuse of a right triangle, given the length of the two sides *x* and *y* (or *xl* and *yl*). A call to _hypot is equivalent to the square root of (*x* squared + *y* squared).

The _hypotl function uses the 80-bit, 10-byte coprocessor form of arguments and return values. See the long double functions for more details on this data type.

Return Value

These functions return the length of the hypotenuse if successful or HUGE_VAL (_LHUGE_VAL for _hypotl) on overflow. The errno variable is set to ERANGE on overflow.

__cabs

_hypot

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_hypotl

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* HYPOT.C: This program prints the
 * hypotenuse of a right  triangle.
*/

#include <math.h>
#include <stdio.h>

void main( void )
{
   double x = 3.0, y = 4.0;

   printf( "If a right triangle has sides %2.1f and %2.1f, "
           "its hypotenuse is %2.1f\n", x, y, _hypot( x, y ) );
}
```

_imagesize Functions

#include <graph.h>

Syntax long __far _imagesize(short *x1*, short *y1*, short *x2*, short *y2*);

long __far _imagesize_w(double *wx1*, double *wy1*, double *wx2*, double *wy2*);

long __far _imagesize_wxy(struct _wxycoord __far **pwxy1*, struct _wxycoord __far **pwxy2*);

Parameter	Description
x1, *y1*	Upper-left corner of bounding rectangle
x2, *y2*	Lower-right corner of bounding rectangle
wx1, *wy1*	Upper-left corner of bounding rectangle
wx2, *wy2*	Lower-right corner of bounding rectangle
pwxy1	Upper-left corner of bounding rectangle
pwxy2	Lower-right corner of bounding rectangle

The functions in the _imagesize family return the number of bytes needed to store the image defined by the bounding rectangle and specified by the coordinates given in the function call.

The _imagesize function defines the bounding rectangle in terms of view-coordinate points (*x1*, *y1*) and (*x2*, *y2*).

The _imagesize_w function defines the bounding rectangle in terms of window-coordinate points (*x1*, *y1*) and (*x2*, *y2*).

The _imagesize_wxy function defines the bounding rectangle in terms of the window-coordinate pairs *pwxy1* and *pwxy2*.

Return Value

The function returns the storage size of the image in bytes. There is no error return.

_getimage functions
_getvideoconfig
_putimage functions

Standards: None
16-Bit: MS-DOS, QWIN

_inp, _inpw

#include <conio.h> Required only for function declarations

Syntax int _inp(unsigned *port*);
unsigned _inpw(unsigned *port*);

Parameter	**Description**
port	Port number

The _inp and _inpw functions read a byte and a word, respectively, from the specified input port. The input value can be any unsigned integer in the range 0 - 65,535.

Return Value

The functions return the byte or word read from *port*. There is no error return.

_outp
_outpw

Standards:	None
16-Bit:	MS-DOS

_inchar

#include <stdio.h>, <graph.h>

Syntax short __far char __cdecl _inchar(void);

The _inchar routine reads a single character from the keyboard and returns the ASCII value of that character without any buffering. A graphics child window must have focus when this routine is called for the key to be accepted. The _inchar routine must be called before a character can be read. This function does not echo its input.

Return Value

This function returns an ASCII key code.

```
/* INCHAR.C - Illustrates the QuickWin graphics
 * routine for character input: _inchar
 */
#include <stdio.h>
#include <graph.h>
void main()
{
   int status = 0, key = 0;
   char buffer[80];
   /* Open a graphics child window and set font to Times Roman*/
   status = _setvideomode( _MAXRESMODE );
   status = _registerfonts( "dummy string" );
   status = _setfont( "t'roman'" );
   status = _setcolor( 11 );
   status = _settextcolor( 14 );
   _clearscreen( _GCLEARSCREEN );
   /* Execute this loop until user enters a 'q' to quit*/
   while( key != 'q' )
   {
      /* Prompt user and read a key*/
      _settextposition( 1, 1 );
      _outtext( "this window must have focus for _inchar to work!" );
      _settextposition( 6, 1 );
      _outtext( "Enter a key ('q' to quit):" );
      key = _inchar();
      _clearscreen( _GCLEARSCREEN );
      _settextposition( 8, 1 );
      sprintf( buffer, "ASCII code = %d\n", key );
      _outtext( buffer );
      _moveto( 30, 30 );
      sprintf( buffer, "%d", key);
      _outgtext( buffer );
   }
   _settextposition( 10, 1 );
   _outtext( "Program finished." );
   _unregisterfonts();
}
```

_int86

#include <dos.h>

Syntax int _int86(int *intnum*, union _REGS **inregs*, union _REGS **outregs*);

Parameter	Description
intnum	Interrupt number
inregs	Register values on call
outregs	Register values on return

The _int86 function executes the 8086-processor-family interrupt specified by the interrupt number *intnum*. Before executing the interrupt, _int86 copies the contents of *inregs* to the corresponding registers. After the interrupt returns, the function copies the current register values to *outregs*. It also copies the status of the system carry flag to the cflag field in the *outregs* argument. The *inregs* and *outregs* arguments are unions of type _REGS. The union type is defined in the include file DOS.H.

Do not use the _int86 function to call interrupts that modify the DS register. Instead, use the _int86x function. (The _int86x function loads the DS and ES registers from the *segregs* argument and also stores the DS and ES registers into *segregs* after the function call.)

The _REGS type is defined in the include file DOS.H.

Return Value

The return value is the value in the AX register after the interrupt returns. If the cflag field in *outregs* is nonzero, an error has occurred; in such cases, the _doserrno variable is also set to the corresponding error code.

_bdos
_int86x
_intdos
_intdosx

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* This program places the name of the current directory in
 * the buffer array, then displays the name of the current
 * directory on the screen. Specifying a length of _MAX_DIR
 * leaves room for the longest legal directory name.
 */
#include <direct.h>
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   char buffer[_MAX_DIR];
   /* Get the current working directory: */
   if( _getcwd( buffer, _MAX_DIR ) == NULL )
      perror( "_getcwd error" );
   else
      printf( "%s\n", buffer );
}
```

_int86x

#include <dos.h>

Syntax int _int86x(int *intnum*, union _REGS **inregs*, union _REGS **outregs*, struct _SREGS **segregs*);

Parameter	Description
intnum	Interrupt number
inregs	Register values on call
outregs	Register values on return
segregs	Segment-register values on call

The _int86x function executes the 8086-processor-family interrupt specified by the interrupt number *intnum*. Unlike the _int86 function, _int86x accepts segment-register values in *segregs*, enabling programs that use large-model data segments or far pointers to specify which segment or pointer should be used during the system call.

Before executing the specified interrupt, _int86x copies the contents of *inregs* and *segregs* to the corresponding registers. Only the DS and ES register values in *segregs* are used. After the interrupt returns, the function copies the current register values to *outregs*, copies the current ES and DS values to *segregs*, and restores DS. It also copies the status of the system carry flag to the cflag field in *outregs*.

The _REGS and _SREGS types are defined in the include file DOS.H.

Segment values for the *segregs* argument can be obtained by using either the _segread function or the _FP_SEG macro.

Return Value

The return value is the value in the AX register after the interrupt returns. If the cflag field in *outregs* is nonzero, an error has occurred; in such cases, the _doserrno variable is also set to the corresponding error code.

_bdos
_FP_SEG
_int86
_intdos
_intdosx
_segread

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* This program places the name of the current directory in
 * the buffer array, then displays the name of the current
 * directory on the screen. Specifying a length of _MAX_DIR
 * leaves room for the longest legal directory name.
 */
#include <direct.h>
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   char buffer[_MAX_DIR];
   /* Get the current working directory: */
   if( _getcwd( buffer, _MAX_DIR ) == NULL )
      perror( "_getcwd error" );
   else
      printf( "%s\n", buffer );
}
```

_intdos

#include <dos.h>

Syntax int _intdos(union _REGS **inregs*, union _REGS **outregs*);

Parameter	**Description**
inregs	Register values on call
outregs	Register values on return

The _intdos function invokes the MS-DOS system call specified by register values defined in *inregs* and returns the effect of the system call in *outregs*. The *inregs* and *outregs* arguments are unions of type _REGS. The _REGS type is defined in the include file DOS.H.

To invoke a system call, _intdos executes an INT 21H instruction. Before executing the instruction, the function copies the contents of *inregs* to the corresponding registers. After the INT instruction returns, _intdos copies the current register values to *outregs*. It also copies the status of the system carry flag to the cflag field in *outregs*. A nonzero cflag field indicates the flag was set by the system call and also indicates an error condition.

The _intdos function is used to invoke MS-DOS system calls that take arguments for input or output in registers other than DX (DH/DL) and AL. The _intdos function is also used to invoke system calls that indicate errors by setting the carry flag. Under any other conditions, the _bdos function can be used.

Do not use the _intdos function to call interrupts that modify the DS register. Instead, use the _intdosx or _int86x function.

Return Value

The _intdos function returns the value of the AX register after the system call is completed. If the cflag field in *outregs* is nonzero, an error has occurred and _doserrno is also set to the corresponding error code.

_bdos
_intdosx

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* INTDOS.C: This program uses _intdos to invoke
 * MS-DOS system call 2AH (gets the current date).
 */
#include <dos.h>
#include <stdio.h>
void main( void )
{
   union _REGS inregs, outregs;
   inregs.h.ah = 0x2a;           /* DOS Get Date function: */
   _intdos( &inregs, &outregs );
   printf( "Date: %d/%d/%d\n", outregs.h.dh, outregs.h.dl, outregs.x.cx );
}
```

_intdosx

#include <dos.h>

Syntax int _intdosx(union _REGS **inregs*, union _REGS **outregs*, struct _SREGS **segregs*);

Parameter	Description
inregs	Register values on call
outregs	Register values on return
segregs	Segment-register values on call

The _intdosx function invokes the MS-DOS system call specified by register values defined in *inregs* and returns the results of the system call in *outregs*. Unlike the _intdos function, _intdosx accepts segment-register values in *segregs*, enabling programs that use large-model data segments or far pointers to specify which segment or pointer should be used during the system call. The _REGS and _SREGS types are defined in the include file DOS.H.

To invoke a system call, _intdosx executes an INT 21H instruction. Before executing the instruction, the function copies the contents of *inregs* and *segregs* to the corresponding registers. Only the DS and ES register values in *segregs* are used. After the INT instruction returns, _intdosx copies the current register values to *outregs* and restores DS. It also copies the status of the system carry flag to the cflag field in *outregs*. A nonzero cflag field indicates the flag was set by the system call and also indicates an error condition.

The _intdosx function is used to invoke MS-DOS system calls that take an argument in the ES register or that take a DS register value different from the default data segment.

Segment values for the *segregs* argument can be obtained by using either the _segread function or the _FP_SEG macro.

Return Value

The _intdosx function returns the value of the AX register after the system call is completed. If the cflag field in *outregs* is nonzero, an error has occurred; in such cases, _doserrno is also set to the corresponding error code.

_bdos
_FP_SEG
_intdos
_segread

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* INTDOSX.C Sends a $-terminated string
 * to the standard output device
 */
#include <dos.h>
#include <stdio.h>
char __far *buffer = "Dollar-sign terminated string\n\r\n\r$";
void main( void )
{
   union  _REGS inregs, outregs;
   struct _SREGS segregs;
   /* Print a $-terminated string on the screen
   /* using MS-DOS function 0x09.
    */
   inregs.h.ah = 0x9;
   inregs.x.dx = _FP_OFF( buffer );
   segregs.ds  = _FP_SEG( buffer );
   _intdosx( &inregs, &outregs, &segregs );
}
```

is Functions

#include <ctype.h>

Syntax

```
int isalnum( int c );
int isalpha( int c );
int __isascii( int c );
int iscntrl( int c );
int __iscsym( int c );
int __iscsymf( int c );
int isdigit( int c );
int isgraph( int c );
int islower( int c );
int isprint( int c );
int ispunct( int c );
int isspace( int c );
int isupper( int c );
int isxdigit( int c );
```

Parameter	Description
c	Integer to be tested

Each function in the is family tests a given integer value, returning a nonzero value if the integer satisfies the test condition and 0 if it does not. The ASCII character set is assumed.

The is functions and their test conditions are listed below:

Function	Test Condition
isalnum	Alphanumeric ('A'-'Z', 'a'-'z', or '0'-'9')
isalpha	Letter ('A'-'Z' or 'a'-'z')
__isascii	ASCII character (0x00 - 0x7F)
iscntrl	Control character (0x00 - 0x1F or 0x7F)
__iscsym	Letter, underscore, or digit
__iscsymf	Letter or underscore
isdigit	Digit ('0'-'9')
isgraph	Printable character except space (' ')
islower	Lowercase letter ('a'-'z')
isprint	Printable character (0x20 - 0x7E)
ispunct	Punctuation character
isspace	White-space character (0x09 - 0x0D or 0x20)
isupper	Uppercase letter ('A'-'Z')

isxdigit	Hexadecimal digit ('A'-'F','a'-'f', or '0'-'9')

The __isascii routine produces meaningful results for all integer values. However, the remaining routines produce a defined result only for integer values corresponding to the ASCII character set (that is, only where __isascii holds true) or for the non-ASCII value EOF (defined in STDIO.H).

These routines are implemented both as functions and as macros.

Use __isascii for compatibility with ANSI naming conventions of non-ANSI functions. Use isascii and link with OLDNAMES.LIB for UNIX compatibility.

Return Value

These routines return a nonzero value if the integer satisfies the test condition and 0 if it does not.

isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit:

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

__isascii

Standards: UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

__iscsym, __iscsymf

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

__toascii
tolower
toupper functions

```
/* ISFAM.C: This program tests all characters between 0x0
 * and 0x7F, then displays each character with abbreviations
 * for the character-type codes that apply.
 */
#include <stdio.h>
#include <ctype.h>
void main( void )
{
   int ch;
   for( ch = 0; ch <= 0x7F; ch++ )
   {
      printf( "%.2x  ", ch );
      printf( " %c", isprint( ch )  ? ch   : '\0' );
      printf( "%4s", isalnum( ch )  ? "AN" : "" );
      printf( "%3s", isalpha( ch )  ? "A"  : "" );
      printf( "%3s", __isascii( ch )  ? "AS" : "" );
      printf( "%3s", iscntrl( ch )  ? "C"  : "" );
      printf( "%3s", __iscsym( ch )  ? "CS "  : "" );
      printf( "%3s", __iscsymf( ch )  ? "CSF"  : "" );
      printf( "%3s", isdigit( ch )  ? "D"  : "" );
      printf( "%3s", isgraph( ch )  ? "G"  : "" );
      printf( "%3s", islower( ch )  ? "L"  : "" );
      printf( "%3s", ispunct( ch )  ? "PU" : "" );
      printf( "%3s", isspace( ch )  ? "S"  : "" );
      printf( "%3s", isprint( ch )  ? "PR" : "" );
      printf( "%3s", isupper( ch )  ? "U"  : "" );
      printf( "%3s", isxdigit( ch ) ? "X"  : "" );
      printf( "\n" );
   }
}
```

_isatty

#include <io.h> Required only for function declarations

Syntax int _isatty(int *handle*);

Parameter	Description
handle	Handle referring to device to be tested

The _isatty function determines whether *handle* is associated with a character device (a terminal, console, printer, or serial port).

Return Value

The _isatty function returns a nonzero value if the device is a character device. Otherwise, the return value is 0.

Use _isatty for compatibility with ANSI naming conventions of non-ANSI functions. Use isatty and link with OLDNAMES.LIB for UNIX compatibility.

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* ISATTY.C: This program checks to see whether
 * stdout has been redirected to a file.
 */
#include <stdio.h>
#include <io.h>
void main( void )
{
   if( _isatty( _fileno( stdout ) ) )
      printf( "stdout has not been redirected to a file\n" );
   else
      printf( "stdout has been redirected to a file\n");
}
```

_itoa

#include <stdlib.h> Required only for function declarations

Syntax char *_itoa(int *value*, char **string*, int *radix*);

Parameter	Description
value	Number to be converted
string	String result
radix	Base of *value*

The _itoa function converts the digits of the given *value* argument to a null-terminated character string and stores the result (up to 17 bytes) in *string*. The *radix* argument specifies the base of *value*; it must be in the range 2-36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (-).

Return Value

The _itoa function returns a pointer to *string*. There is no error return.

_ltoa
_ultoa

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* ITOA.C: This program converts integers of various
 * sizes to strings in various radixes.
 */
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   char buffer[20];
   int  i = 3445;
   long l = -344115L;
   unsigned long ul = 1234567890UL;
   _itoa( i, buffer, 10 );
   printf( "String of integer %d (radix 10): %s\n", i, buffer );
   _itoa( i, buffer, 16 );
   printf( "String of integer %d (radix 16): 0x%s\n", i, buffer );
   _itoa( i, buffer, 2  );
   printf( "String of integer %d (radix 2): %s\n", i, buffer );
   _ltoa( l, buffer, 16 );
   printf( "String of long int %ld (radix 16): 0x%s\n", l, buffer );
   _ultoa( ul, buffer, 16 );
   printf( "String of unsigned long %lu (radix 16): 0x%s\n", ul, buffer );
}
```

_kbhit

#include <conio.h> Required only for function declarations

Syntax int _kbhit(void);

The _kbhit function checks the console for a recent keystroke. If the function returns a nonzero value, a keystroke is waiting in the buffer. The program can then call _getch or _getche to get the keystroke.

Return Value

The _kbhit function returns a nonzero value if a key has been pressed. Otherwise, it returns 0.

Standards: None

16-Bit: MS-DOS

```
/* KBHIT.C: This program loops until the user
 * presses a key. If _kbhit returns nonzero, a
 * keystroke is waiting in the buffer. The program
 * can call _getch or _getche to get the keystroke.
 */
#include <conio.h>
#include <stdio.h>
void main( void )
{
   /* Display message until key is pressed. */
   while( !_kbhit() )
      _cputs( "Hit me!! " );
   /* Use _getch to throw key away. */
   printf( "\nKey struck was '%c'\n", _getch() );
   _getch();
}
```

labs

#include <stdlib.h> Required only for function declarations

#include <math.h>

Syntax long labs(long *n*);

Parameter	**Description**
n	Long-integer value

The labs function produces the absolute value of its long-integer argument *n*.

Return Value

The labs function returns the absolute value of its argument. There is no error return.

abs
_cabs
fabs

Standards: ANSI

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* ABS.C: This program computes and displays
 * the absolute values of several numbers.
 */
#include  <stdio.h>
#include  <math.h>
#include  <stdlib.h>
void main( void )
{
   int    ix = -4, iy;
   long   lx = -41567L, ly;
   double dx = -3.141593, dy;
   iy = abs( ix );
   printf( "The absolute value of %d is %d\n", ix, iy);
   ly = labs( lx );
   printf( "The absolute value of %ld is %ld\n", lx, ly);
   dy = fabs( dx );
   printf( "The absolute value of %f is %f\n", dx, dy );
}
```

ldexp, ldexpl

#include <math.h>

Syntax double ldexp(double *x*, int *exp*);

long double ldexpl(long double *x*, int *exp*);

Parameter	Description
x	Floating-point value
exp	Integer exponent

The ldexp and ldexpl functions calculate the value of *x* * (2 raised to the power of *exp*).

Return Value

These functions return an exponential value if successful. On overflow (depending on the sign of *x*), _ldexp returns +/-HUGE_VAL, and ldexpl returns +/-_LHUGE_VAL; the errno variable is set to ERANGE.

The ldexpl function uses the 80-bit, 10-byte coprocessor form of arguments and return values. See the long double functions for more details on this data type.

frexp
modf

ldexp

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

ldexpl

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* LDEXP.C */
#include <math.h>
#include <stdio.h>
void main( void )
{
   double x = 4.0, y;
   int p = 3;
   y = ldexp( x, p );
   printf( "%2.1f times two to the power of %d is %2.1f\n", x, p, y );
}
```

ldiv

#include <stdlib.h>

Syntax ldiv_t ldiv (long int *numer*, long int *denom*);

Parameter	Description
numer	Numerator
denom	Denominator

The ldiv function divides *numer* by *denom*, computing the quotient and the remainder. The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer that is less than the absolute value of the mathematical quotient. If the denominator is 0, the program will terminate with an error message.

The ldiv function is similar to the div function, with the difference being that the arguments and the members of the returned structure are all of type long int.

The ldiv_t structure, defined in STDLIB.H, contains the following elements:

Element	Description
long int quot	Quotient
long int rem	Remainder

Return Value

The ldiv function returns a structure of type ldiv_t, comprising both the quotient and the remainder.

div

Standards: ANSI

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* LDIV.C: This program takes two long integers
 * as command-line arguments and displays the
 * results of the integer division.
 */
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
void main( void )
{
   long x = 5149627, y = 234879;
   ldiv_t div_result;
   div_result = ldiv( x, y );
   printf( "For %ld / %ld, the quotient is ", x, y );
   printf( "%ld, and the remainder is %ld\n", div_result.quot, div_result.rem );
}
```

_lfind

#include <search.h> Required only for function declarations

Syntax void *_lfind(const void **key*, const void **base*, unsigned int **num*, unsigned int *width*, int (__cdecl **compare*)(const void **elem1*, const void **elem2*));

Parameter	Description
key	Object to search for
base	Pointer to base of search data
num	Number of array elements
width	Width of array elements
compare( )	Pointer to comparison routine
elem1	Pointer to the key for the search
elem2	Pointer to the array element to be compared with the key

The _lfind function performs a linear search for the value *key* in an array of *num* elements; each element is *width* bytes in size. (Unlike bsearch, _lfind does not require the array to be sorted.) The *base* argument is a pointer to the base of the array to be searched.

The *compare* argument is a pointer to a user-supplied routine that compares two array elements and then returns a value specifying their relationship. The _lfind function calls the *compare* routine one or more times during the search, passing pointers to two array elements on each call. This routine must compare the elements, then return one of the following values:

Value	Meaning
Nonzero	Elements are different
0	Elements are identical

Return Value

If the key is found, _lfind returns a pointer to the element of the array at *base* that matches *key*. If the key is not found, _lfind returns NULL.

Use _lfind for compatibility with ANSI naming conventions of non-ANSI functions. Use lfind and link with OLDNAMES.LIB for UNIX compatibility.

bsearch
_lsearch
qsort

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* LFIND.C: This program uses _lfind to search for
 * the word "hello" in the command-line arguments.
 */
#include <search.h>
#include <string.h>
#include <stdio.h>
int compare( const void *arg1, const void *arg2 );
void main( unsigned int argc, char **argv )
{
   char **result;
   char *key = "hello";
   result = (char **)_lfind( &key, argv, &argc, sizeof(char *), compare );
   if( result )
      printf( "%s found\n", *result );
   else
      printf( "hello not found!\n" );
}
int compare(const void *arg1, const void *arg2 )
{
   return( _stricmp( * (char**)arg1, * (char**)arg2 ) );
}
```

_lineto Functions

#include <graph.h>

Syntax short __far _lineto(short *x*, short *y*);

short __far _lineto_w(double *wx*, double *wy*);

Parameter	Description
x, *y*	End point
wx, *wy*	End point

The functions in the _lineto family draw a line from the current graphics position up to and including the destination point. The destination point for the _lineto function is given by the view-coordinate point (*x*, *y*). The destination point for the _lineto_w function is given by the window-coordinate point (*wx*, *wy*).

The line is drawn using the current color, logical write mode, and line style. If no error occurs, _lineto sets the current graphics position to the view-coordinate point (*x*, *y*); _lineto_w sets the current position to the window-coordinate point (*wx*, *wy*).

If you use _floodfill to fill in a closed figure drawn with _lineto calls, the figure must be drawn with a solid line-style pattern.

Return Value

The _lineto and _lineto_w routines return a nonzero value if anything is drawn; otherwise, they return 0.

_getcurrentposition functions
_moveto functions
_setlinestyle

Standards: None
16-Bit: MS-DOS, QWIN

```
/* MOVETO.C: This program draws line
 * segments of different colors.
 */
#include <graph.h>
#include <stdlib.h>
#include <conio.h>
void main( void )
{
   short x, y, xinc, yinc, color = 1;
   struct _videoconfig v;
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXCOLORMODE ) )
      exit( 1 );
   _getvideoconfig( &v );
   xinc = v.numxpixels / 50;
   yinc = v.numypixels / 50;
   for( x = 0, y = v.numypixels - 1; x < v.numxpixels; x += xinc, y -=  yinc )
   {
      _setcolor( color++ % 16 );
      _moveto( x, 0 );
      _lineto( 0, y );
   }
   _getch();
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}
```

localeconv

#include <locale.h>

Syntax struct lconv *localeconv(void);

The localeconv function gets detailed information on the locale-specific settings for numeric formatting of the program's current locale. This information is stored in a structure of type lconv.

The lconv structure, defined in LOCALE.H, contains the following members:

char *decimal_point
Decimal-point character for nonmonetary quantities.

char *thousands_sep
Character used to separate groups of digits to the left of the decimal point for nonmonetary quantities.

char *grouping
Size of each group of digits in nonmonetary quantities.

char *int_curr_symbol
International currency symbol for the current locale. The first three characters specify the alphabetic international currency symbol as defined in the *ISO 4217 Codes for the Representation of Currency and Funds* standard. The fourth character (immediately preceding the null character) is used to separate the international currency symbol from the monetary quantity.

char *currency_symbol
Local currency symbol for the current locale.

char *mon_decimal_point
Decimal-point character for monetary quantities.

char *mon_thousands_sep
Separator for groups of digits to the left of the decimal place in monetary quantities.

char *mon_grouping
Size of each group of digits in monetary quantities.

char *positive_sign
String denoting sign for nonnegative monetary quantities.

char *negative_sign
String denoting sign for negative monetary quantities.

char int_frac_digits
Number of digits to the right of the decimal point in internationally formatted monetary quantities.

char frac_digits
Number of digits to the right of the decimal point in formatted monetary quantities.

char p_cs_precedes
Set to 1 if the currency symbol precedes the value for a nonnegative formatted monetary quantity. Set to 0 if the symbol follows the value.

char p_sep_by_space
Set to 1 if the currency symbol is separated by a space from the value for a nonnegative formatted monetary quantity. Set to 0 if there is no space separation.

char n_cs_precedes
Set to 1 if the currency symbol precedes the value for a negative formatted monetary quantity. Set to 0 if the symbol succeeds the value.

char n_sep_by_space
Set to 1 if the currency symbol is separated by a space from the value for a negative formatted monetary quantity. Set to 0 if there is no space separation.

char p_sign_posn

Position of positive sign in nonnegative formatted monetary quantities.

char n_sign_posn
Position of positive sign in negative formatted monetary quantities.

The char * members of the struct are pointers to strings. Any of these (other than char *decimal_point) that equals "" is either of zero length or is not supported in the current locale. The char members of the struct are nonnegative numbers. Any of these that equals CHAR_MAX is not supported in the current locale.

The elements of grouping and mon_grouping are interpreted according to the following rules:

CHAR_MAX
No further grouping is to be performed.

0
The previous element is to be repeatedly used for the remainder of the digits.

n
The integer value *n* is the number of digits that make up the current group. The next element is examined to determine the size of the next group of digits before the current group.

The values for p_sign_posn and n_sign_posn are interpreted according to the following rules:

0
Parentheses surround the quantity and currency symbol

1
Sign string precedes the quantity and currency symbol

2
Sign string follows the quantity and currency symbol

3
Sign string immediately precedes the currency symbol

4
Sign string immediately follows the currency symbol

Return Value

The localeconv function returns a pointer to a filled-in object of type struct lconv. The values contained in the object can be overwritten by subsequent calls to localeconv and do not directly modify the object. Calls to the setlocale function with *category* values of LC_ALL, LC_MONETARY, or LC_NUMERIC will overwrite the contents of the structure.

setlocale
strcoll
strftime
strxfrm

Standards: ANSI

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

localtime

#include <time.h>

Syntax struct tm *localtime(const time_t **timer*);

Parameter	**Description**
timer	Pointer to stored time

The localtime function converts a time stored as a time_t value and stores the result in a structure of type tm. The long value *timer* represents the seconds elapsed since midnight (00:00:00), January 1, 1970, Universal Coordinated Time. This value is usually obtained from the time function.

The fields of the structure type tm store the following values, each of which is an int:

Field	**Value Stored**
tm_sec	Seconds after the minute (0-59)
tm_min	Minutes after the hour (0-59)
tm_hour	Hours since midnight (0-23)
tm_mday	Day of month (1-31)
tm_mon	Month (0-11; January = 0)
tm_year	Year (current year minus 1900)
tm_wday	Day of week (0-6; Sunday = 0)
tm_yday	Day of year (0-365; January 1 = 0)
tm_isdst	Positive value if daylight saving time is in effect; 0 if daylight saving time is not in effect; negative value if status of daylight saving time is unknown

Note that the gmtime, mktime, and localtime functions use a single statically allocated tm structure for the conversion. Each call to one of these routines destroys the result of the previous call.

The localtime function makes corrections for the local time zone if the user first sets the environment variable TZ. When TZ is set, three other environment variables (_timezone, _daylight, and _tzname) are automatically set as well. See _tzset for a description of these variables.

The TZ variable is not part of the ANSI standard definition of localtime but is a Microsoft extension.

Note The target environment should attempt to determine whether daylight saving time is in effect.

Return Value

The localtime function returns a pointer to the structure result. If the value in *timer* represents a date before midnight, January 1, 1970, the function returns NULL.

asctime
ctime
_ftime
gmtime
time
_tzset

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* LOCALTIME.C: This program uses time to get the current time and
 * then uses localtime to convert this time to a structure representing
 * the local time. The program converts the result from a 24-hour clock
 * to a 12-hour clock and determines the proper extension (AM or PM).
 */

#include <stdio.h>
#include <string.h>
#include <time.h>

void main( void )
{
        struct tm *newtime;
        char am_pm[] = "AM";
        time_t long_time;

        time( &long_time );                     /* Get time as long integer. */
        newtime = localtime( &long_time );      /* Convert to local time. */

        if( newtime->tm_hour > 12 )             /* Set up extension. */
                strcpy( am_pm, "PM" );
        if( newtime->tm_hour > 12 )             /* Convert from 24-hour */
                newtime->tm_hour -= 12;         /*   to 12-hour clock.  */
        if( newtime->tm_hour == 0 )             /*Set hour to 12 if midnight. */
                newtime->tm_hour = 12;

        printf( "%.19s %s\n", asctime( newtime ), am_pm );
}
```

_locking

#include <sys\locking.h>

#include <io.h> Required only for function declarations

Syntax int _locking(int *handle*, int *mode*, long *nbytes*);

Parameter	**Description**
handle	File handle
mode	File-locking mode
nbytes	Number of bytes to lock

The _locking function locks or unlocks *nbytes* bytes of the file specified by *handle*. Locking bytes in a file prevents access to those bytes by other processes. All locking or unlocking begins at the current position of the file pointer and proceeds for the next *nbytes* bytes. It is possible to lock bytes past the end of the file.

The *mode* argument specifies the locking action to be performed. It must be one of the following manifest constants:

Constant	**Action**
_LK_LOCK	Locks the specified bytes. If the bytes cannot be locked, immediately tries again after 1 second. If, after 10 attempts, the bytes cannot be locked, returns an error.
_LK_NBLCK	Locks the specified bytes. If bytes cannot be locked, returns an error.
_LK_NBRLCK	Same as _LK_NBLCK.
_LK_RLCK	Same as _LK_LOCK.
_LK_UNLCK	Unlocks the specified bytes. (The bytes must have been previously locked.)

More than one region of a file can be locked, but no overlapping regions are allowed.

When a region of a file is being unlocked, it must correspond to a region that was previously locked. The _locking function does not merge adjacent regions; if two locked regions are adjacent, each region must be unlocked separately.

Regions should be locked only briefly and should be unlocked before closing a file or exiting the program.

The _locking function should be used only with MS-DOS versions 3.0 and later; it has no effect under earlier versions of MS-DOS. Also, file sharing must be loaded to use the _locking function. Note that with MS-DOS versions 3.0 and 3.1, the files locked by parent processes may become unlocked when child processes exit.

Return Value

The _locking function returns 0 if successful. A return value of -1 indicates failure, and errno is set to one of the following values:

Value	Meaning
EACCES	Locking violation (file already locked or unlocked).
EBADF	Invalid file handle.
EDEADLOCK	Locking violation. This is returned when the _LK_LOCK or _LK_RLCK flag is specified and the file cannot be locked after 10 attempts.
EINVAL	An invalid argument was given to the function.

Use _locking for compatibility with ANSI naming conventions of non-ANSI functions. Use locking and link with OLDNAMES.LIB for UNIX compatibility.

__creat
__open

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* LOCKING.C: This program opens a file with sharing. It locks
 * some bytes before reading them, then unlocks them. Note that the
 * program works correctly only if the following conditions are met:
 *     - The file exists
 *     - The program is run with MS-DOS version 3.0 or later
 *       with file sharing installed (SHARE.COM or SHARE.EXE), or
 *       if a Microsoft Networks compatible network is running
 */
#include <io.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <sys\locking.h>
#include <share.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
void main( void )
{
   int  fh, numread;
   char buffer[40];
   /* Quit if can't open file or system doesn't support sharing. */
   fh = _sopen( "locking.c", _O_RDWR, _SH_DENYNO, _S_IREAD | _S_IWRITE );
   if( (fh == -1) || (_osmajor < 3) )
      exit( 1 );
   /* Lock some bytes and read them. Then unlock. */
   if( _locking( fh, LK_NBLCK, 30L ) != -1 )
   {
      printf( "No one can change these bytes while I'm reading them\n" );
      numread = _read( fh, buffer, 30 );
      printf( "%d bytes read: %.30s\n", numread, buffer );
      lseek( fh, 0L, SEEK_SET );
     _locking( fh, LK_UNLCK, 30L );
      printf( "Now I'm done. Do what you will with them\n" );
   }
   else
      perror( "Locking failed\n" );
   _close( fh );
}
```

log Functions

#include <math.h>

Syntax double log(double *x*);

double log10(double *x*);

long double logl(long double *x*);

long double log10l(long double *x*);

Parameter	Description
x	Value whose logarithm is to be found

The log and log10 functions calculate the natural logarithm and the base-10 logarithm, respectively, of *x*. The logl and log10l functions are the 80-bit counterparts and use the 80-bit, 10-byte coprocessor form of arguments and return values. See the long double functions for more details on this data type.

Return Value

The log functions return the logarithm of *x* if successful. If *x* is negative, the functions print a _DOMAIN error message to stderr, return the value -HUGE_VAL (-_LHUGE_VAL for the long double functions), and set errno to EDOM. If *x* is 0, they print a _SING error message to stderr, and set errno to ERANGE.

Error handling can be modified by using the _matherr or _matherrl routine.

exp
_matherr
pow functions

log, log10

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

logl, log10l

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* LOG.C: This program uses log and log10
 * to calculate the natural logarithm and
 * the base-10 logarithm of 9,000.
 */
#include <math.h>
#include <stdio.h>
void main( void )
{
   double x = 9000.0;
   double y;
   y = log( x );
   printf( "log( %.2f ) = %f\n", x, y );
   y = log10( x );
   printf( "log10( %.2f ) = %f\n", x, y );
}
```

long double Functions

The 8087 family of numeric coprocessor chips supports the 80-bit precision floating-point data type. Beginning with Microsoft C version 6.0, the long double functions, whose names end with l, map the C long double type into this 80-bit, 10-byte form. Unlike the regular floating-point functions (such as acos), which return values of type double, these long double functions (such as _acosl) return values of type long double. The long double functions also return their values on the coprocessor stack for all calling conventions.

The long double type is also supported by the addition of the "L" prefix for a floating-point format specification in the printf and scanf family of functions.

The long double versions are described with their regular counterparts. These are the regular run-time math functions with corresponding long double equivalents:

Function	Long Double Form	Function	Long Double Form
acos	acosl	frexp	frexpl
asin	asinl	_hypot	_hypotl
atan	atanl	ldexp	ldexpl
atan2	atan2l	log	logl
atof	_atold	log10	log10l
_cabs	_cabsl	_matherr	_matherrl
ceil	ceill	modf	modfl
cos	cosl	pow	powl
cosh	coshl	sin	sinl
exp	expl	sinh	sinhl
fabs	fabsl	sqrt	sqrtl
floor	floorl	tan	tanl
fmod	fmodl	tanh	tanhl

longjmp

#include <setjmp.h>

Syntax void longjmp(jmp_buf *env*, int *value*);

Parameter	Description
env	Variable in which environment is stored
value	Value to be returned to setjmp call

The longjmp function restores a stack environment and execution locale previously saved in *env* by setjmp. The setjmp and longjmp functions provide a way to execute a nonlocal goto; they are typically used to pass execution control to error handling or recovery code in a previously called routine without using the normal call and return conventions.

A call to setjmp causes the current stack environment to be saved in *env*. A subsequent call to longjmp restores the saved environment and returns control to the point immediately following the

corresponding setjmp call. Execution resumes as if *value* had just been returned by the setjmp call. The values of all variables (except register variables) that are accessible to the routine receiving control contain the values they had when longjmp was called. The values of register variables are unpredictable.

The longjmp function must be called before the function that called setjmp returns. If longjmp is called after the function calling setjmp returns, unpredictable program behavior results.

The value returned by setjmp must be nonzero. If *value* is passed as 0, the value 1 is substituted in the actual return.

Observe the following four restrictions when using longjmp:

- Do not assume that the values of the register variables will remain the same. The values of register variables in the routine calling setjmp may not be restored to the proper values after longjmp is executed.
- Do not use longjmp to transfer control from within one overlay to within another. The overlay manager keeps the overlay in memory after a call to longjmp.
- Do not use longjmp to transfer control out of an interrupt-handling routine unless the interrupt is caused by a floating-point exception. In this case, a program may return from an interrupt handler via longjmp if it first reinitializes the floating-point math package by calling _fpreset.
- Do not use longjmp or setjmp from a C++ program.

Return Value

None.

setjmp

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_lrotl, _lrotr

#include <stdlib.h>

Syntax unsigned long _lrotl(unsigned long *value*, int *shift*);
unsigned long _lrotr(unsigned long *value*, int *shift*);

Parameter	Description
value	Value to be rotated
shift	Number of bits to shift

The _lrotl and _lrotr functions rotate *value* by *shift* bits. The _lrotl function rotates the value left. The _lrotr function rotates the value right. Both functions "wrap" bits rotated off one end of *value* to the other end.

Return Value

Both functions return the rotated value. There is no error return.

__rotl
__rotr

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* LROT.C */
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   unsigned long val = 0x0fac35791;
   printf( "0x%8.8lx rotated left eight times is 0x%8.8lx\n", val, _lrotl( val,
8 ) );
   printf( "0x%8.8lx rotated right four times is 0x%8.8lx\n", val, _lrotr( val,
4 ) );
}
```

_lsearch

#include <search.h> Required only for function declarations

Syntax void *_lsearch(const void **key*, const void **base*, unsigned int **num*, unsigned int *width*, int (__cdecl **compare*)(const void **elem1*, const void **elem2*));

Parameter	Description
key	Object to search for
base	Pointer to base of search data
num	Number of elements
width	Width of elements
compare	Pointer to comparison routine
elem1	Pointer to the key for the search
elem2	Pointer to the array element to be compared with the key

The _lsearch function performs a linear search for the value *key* in an array of *num* elements, each of *width* bytes in size. (Unlike bsearch, _lsearch does not require the array to be sorted.) The *base* argument is a pointer to the base of the array to be searched.

If *key* is not found, _lsearch adds it to the end of the array.

The *compare* argument is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. The _lsearch function calls the *compare* routine one or more times during the search, passing pointers to two array elements on each call. This routine must compare the elements, then return one of the following values:

Value	Meaning
Nonzero	Elements are different
0	Elements are identical

Return Value

If the key is found, _lsearch returns a pointer to the element of the array at *base* that matches *key*. If the key is not found, _lsearch returns a pointer to the newly added item at the end of the array.

Use _lsearch for compatibility with ANSI naming conventions of non-ANSI functions. Use lsearch and link with OLDNAMES.LIB for UNIX compatibility.

bsearch
_lfind

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_lseek

#include <io.h> Required only for function declarations

#include <stdio.h>

Syntax long _lseek(int *handle*, long *offset*, int *origin*);

Parameter	**Description**
handle	Handle referring to open file
offset	Number of bytes from *origin*
origin	Initial position

The _lseek function moves the file pointer associated with *handle* to a new location that is *offset* bytes from *origin*. The next operation on the file occurs at the new location. The *origin* argument must be one of the following constants, which are defined in STDIO.H:

Origin	**Definition**
SEEK_SET	Beginning of file
SEEK_CUR	Current position of file pointer
SEEK_END	End of file

The _lseek function can be used to reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file. An attempt to position the pointer before the beginning of the file causes an error only if you explicitly link with LSEEKCHK.OBJ.

Return Value

The _lseek function returns the offset, in bytes, of the new position from the beginning of the file. The function returns -1L to indicate an error and sets errno to one of the following values:

Value	**Meaning**
EBADF	Invalid file handle
EINVAL	Invalid value for *origin*, or position specified by *offset* is before the beginning of the file

On devices incapable of seeking (such as terminals and printers), the return value is undefined.

Use _lseek for compatibility with ANSI naming conventions of non-ANSI functions. Use lseek and link with OLDNAMES.LIB for UNIX compatibility.

fseek
_tell

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* LSEEK.C: This program first opens a file named LSEEK.C.
 * It then uses _lseek to find the beginning of the file,
 * to find the current position in the file, and to find
 * the end of the file.
 */
#include <io.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   int fh;
   long pos;               /* Position of file pointer */
   char buffer[10];
   fh = _open( "lseek.c", _O_RDONLY );
   /* Seek the beginning of the file: */
   pos = _lseek( fh, 0L, SEEK_SET );
   if( pos == -1L )
      perror( "_lseek to beginning failed" );
   else
      printf( "Position for beginning of file seek = %ld\n", pos );
   /* Move file pointer a little */
    _read( fh, buffer, 10 );
   /* Find current position: */
   pos = _lseek( fh, 0L, SEEK_CUR );
   if( pos == -1L )
      perror( "_lseek to current position failed" );
   else
      printf( "Position for current position seek = %ld\n", pos );
   /* Set the end of the file: */
   pos = _lseek( fh, 0L, SEEK_END );
   if( pos == -1L )
      perror( "_lseek to end failed" );
   else
      printf( "Position for end of file seek = %ld\n", pos );
   _close( fh );
}
```

_ltoa

#include <stdlib.h> Required only for function declarations

Syntax char *_ltoa(long *value*, char **string*, int *radix*);

Parameter	Description
value	Number to be converted
string	String result
radix	Base of *value*

The _ltoa function converts the digits of *value* to a null-terminated character string and stores the result (up to 33 bytes) in *string*. The *radix* argument specifies the base of *value*, which must be in the range 2-36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (-).

Return Value

The _ltoa function returns a pointer to *string*. There is no error return.

_itoa
_ultoa

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_makepath

#include <stdlib.h>

Syntax void _makepath(char **path*, const char **drive*, const char **dir*, const char **fname*, const char **ext*);

Parameter	**Description**
path	Full path buffer
drive	Drive letter
dir	Directory path
fname	Filename
ext	File extension

The _makepath routine creates a single path, composed of a drive letter, directory path, filename, and filename extension. The *path* argument should point to an empty buffer large enough to hold the complete path. The constant _MAX_PATH, defined in STDLIB.H, specifies the maximum size *path* that the _makepath function can handle. The other arguments point to buffers containing the path elements:

drive

The *drive* argument contains a letter (A, B, etc.) corresponding to the desired drive and an optional trailing colon. The _makepath routine will insert the colon automatically in the composite path if it is missing. If *drive* is a null character or an empty string, no drive letter and colon will appear in the composite *path* string.

dir

The *dir* argument contains the path of directories, not including the drive designator or the actual filename. The trailing slash is optional, and either forward slashes (\) or backslashes (/) or both may be used in a single *dir* argument. If a trailing slash (/ or \ ) is not specified, it will be inserted automatically. If *dir* is a null character or an empty string, no slash is inserted in the composite *path* string.

fname

The *fname* argument contains the base filename without any extensions. If *fname* is NULL or points to an empty string, no filename is inserted in the composite *path* string.

ext

The *ext* argument contains the actual filename extension, with or without a leading period (.). The _makepath routine will insert the period automatically if it does not appear in *ext*. If *ext* is a null character or an empty string, no period is inserted in the composite *path* string.

There are no size limits on any of the above four fields. However, the composite path must be no larger than the _MAX_PATH constant. The _MAX_PATH limit permits a path much larger than current operating-system versions will handle.

Return Value

None.

_fullpath
_splitpath

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* MAKEPATH.C */
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   char path_buffer[_MAX_PATH];
   char drive[_MAX_DRIVE];
   char dir[_MAX_DIR];
   char fname[_MAX_FNAME];
   char ext[_MAX_EXT];
   _makepath( path_buffer, "c", "\\sample\\crt\\", "makepath", "c" );
   printf( "Path created with _makepath: %s\n\n", path_buffer );
   _splitpath( path_buffer, drive, dir, fname, ext );
   printf( "Path extracted with _splitpath:\n" );
   printf( "  Drive: %s\n", drive );
   printf( "  Dir: %s\n", dir );
   printf( "  Filename: %s\n", fname );
   printf( "  Ext: %s\n", ext );
}
```

malloc Functions

#include <stdlib.h> For ANSI compatibility (malloc only)

#include <malloc.h> Required only for function declarations

Syntax void *malloc(size_t *size*);

void __based(void) *_bmalloc(__segment *seg*, size_t *size*);

void __far *_fmalloc(size_t *size*);

void __near *_nmalloc(size_t *size*);

Parameter	Description
size	Bytes to allocate
seg	Based heap segment selector

Functions in the malloc family allocate a memory block of at least *size* bytes. The block may be larger than *size* bytes because of space required for alignment and maintenance information. If *size* is 0, each of these functions allocates a zero-length item in the heap and returns a valid pointer to that item.

The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

In large data models (compact-, large-, and huge-model programs), malloc maps to _fmalloc. In small data models (tiny-, small-, and medium-model programs), malloc maps to _nmalloc. The _fmalloc function allocates a memory block of at least *size* bytes in the far heap, which is outside the default data segment.

The bmalloc function allocates a memory block of at least *size* bytes in the based heap segment specified by the segment selector *seg*.

The malloc functions allocate memory in the heap segment specified below:

Function	Heap Segment
malloc	Depends on data model of program
_bmalloc	Based heap segment specified by *seg* value
_fmalloc	Far heap (outside default data segment)
_nmalloc	Near heap (within default data segment)

The functions listed below call the malloc family of routines. In addition, the startup code uses malloc to allocate storage for the environ/envp and argv strings and arrays.

The following routines call malloc:

_brealloc	fscanf	_putw
calloc	fseek	scanf
_execl	fsetpos	_searchenv
_execle	_fullpath	setvbuf
_execlp	fwrite	_spawnl
_execlpe	getc	_spawnle

_execv	getchar	_spawnlp
_execve	_getcwd	_spawnlpe
_execvp	_getdcwd	_spawnv
_execvpe	gets	_spawnve
fgetc	_getw	_spawnvp
_fgetchar	_nrealloc	_spawnvpe
fgets	_popen	_strdup
fprintf	printf	system
fputc	putc	_tempnam
_fputchar	putchar	ungetc
fputs	_putenv	vfprintf
fread	puts	vprintf
_frealloc		

The following routines call _nmalloc:

_nrealloc
_ncalloc
_nstrdup
realloc (in small data models)

The following routines call _fmalloc:

_frealloc
_fcalloc
_fstrdup
realloc (in large data models)

In Microsoft C version 5.1, the _fmalloc function retried allocating within the default data segment (that is, in the near heap) if sufficient memory was not available outside the default data segment. Since version 6.0, _fmalloc returns NULL under these conditions.

The _freect, _memavl, and _memmax functions called malloc in Microsoft C version 5.1 but do not do so in versions 6.0, 7.0, and this product.

Return Value

The malloc function returns a void pointer to the allocated space. The _nmalloc function returns a (void __near *) and _fmalloc returns a (void __far *). The _bmalloc function returns a (void __based(void) *).

The _malloc, _fmalloc, and _nmalloc functions return NULL if there is insufficient memory available. The _bmalloc function returns _NULLOFF if there is insufficient memory available.

Always check the return from the malloc function, even if the amount of memory requested is small.

malloc

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_bmalloc, _fmalloc, _nmalloc

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

calloc functions
free functions
realloc functions

```
/* MALLOC.C: This program allocates memory with
 * malloc, then frees the memory with free.
 */
#include <stdlib.h>         /* Definition of _MAX_PATH */
#include <stdio.h>
#include <malloc.h>
void main( void )
{
   char *string;
   /* Allocate space for a path name */
   string = malloc( _MAX_PATH );
   if( string == NULL )
      printf( "Insufficient memory available\n" );
   else
      printf( "Memory space allocated for path name\n" );
   free( string );
   printf( "Memory freed\n" );
}
```

_matherr, _matherrl

#include <math.h>

Syntax int _matherr(struct _exception *_except_);

int _matherrl(struct _exceptionl *_except_);

Parameter	Description
except	Pointer to structure containing error information

The _matherr functions process errors generated by the functions of the math library. The math functions call the appropriate _matherr routine whenever an error is detected. The _matherrl function uses the 80-bit, 10-byte coprocessor form of arguments and return values. See the long double functions for more details on this data type.

The user can provide a different definition of the _matherr or _matherrl function to carry out special error handling.

When an error occurs in a math routine, _matherr is called with a pointer to an _exception type structure (defined in MATH.H) as an argument.

The _exception structure contains the following elements:

Element	Description
int type	Exception type
char *name	Name of function where error occurred
double arg1, arg2	First and second (if any) argument to function
double retval	Value to be returned by function

The type specifies the type of math error. It is one of the following values, defined in MATH.H:

Value	Meaning
_DOMAIN	Argument domain error
_SING	Argument singularity
_OVERFLOW	Overflow range error
_PLOSS	Partial loss of significance
_TLOSS	Total loss of significance
_UNDERFLOW	The result is too small to be represented. (This condition is not currently supported.)

The structure member name is a pointer to a null-terminated string containing the name of the function that caused the error. The structure members arg1 and arg2 specify the values that caused the error. (If only one argument is given, it is stored in arg1.)

The default return value for the given error is retval. If you change the return value, remember that the return value must specify whether an error actually occurred. If the _matherr function returns 0, an error message can be displayed and errno is set to an appropriate error value. If _matherr returns a

nonzero value, no error message is displayed, and errno remains unchanged.

Return Value

The _matherr functions should return 0 to indicate an error, and a nonzero value to indicate successful corrective action.

Use _matherr for compatibility with ANSI naming conventions of non-ANSI functions. Use matherr and link with OLDNAMES.LIB for UNIX compatibility.

_matherr

Standards: UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_matherrl

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

acos functions
asin functions
atan functions
bessel functions
_cabs
cos functions
exp
_hypot
log functions
pow
sin functions
sqrt
tan functions

```
/* MATHERR.C: To use _matherr, you must turn off the
 * Extended Dictionary flag within the Microsoft
 * Programmer's WorkBench environment, or use the /NOE
 * linker option outside the environment. For example:
 *     CL _matherr.c /link /NOE
 */
#include <math.h>
#include <string.h>
#include <stdio.h>
void main( void )
{
   /* Do several math operations that cause errors. The _matherr
    * routine handles _DOMAIN errors, but lets the system handle
    * other errors normally.
    */
   printf( "log( -2.0 ) = %e\n", log( -2.0 ) );
   printf( "log10( -5.0 ) = %e\n", log10( -5.0 ) );
   printf( "log( 0.0 ) = %e\n", log( 0.0 ) );
}
/* Handle several math errors caused by passing a negative argument
    * to log or log10 (_DOMAIN errors). When this happens, _matherr returns
 * the natural or base-10 logarithm of the absolute value of the
 * argument and suppresses the usual error message.
 */
int _matherr( struct _exception *except )
{
   /* Handle _DOMAIN errors for log or log10. */
   if( except->type == _DOMAIN )
   {
      if( strcmp( except->name, "log" ) == 0 )
      {
         except->retval = log( -(except->arg1) );
         printf( "Special: using absolute value: %s: _DOMAIN error\n", except-
>name );
         return 1;
      }
      else if( strcmp( except->name, "log10" ) == 0 )
      {
         except->retval = log10( -(except->arg1) );
         printf( "Special: using absolute value: %s: _DOMAIN error\n", except-
>name );
         return 1;
      }
   }
   else
   {
      printf( "Normal: " );
      return 0;    /* Else use the default actions */
   }
}
```

__max

#include <stdlib.h>

Syntax type __max(*type a*, *type b*);

Parameter	Description
type	Any numeric data type
a, *b*	Values of any numeric type to be compared

The __max macro compares two values and returns the value of the larger one. The arguments can be of any numeric data type, signed or unsigned. Both arguments and the return value must be of the same data type.

Return Value

The macro returns the larger of the two arguments.

___min

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* MINMAX.C */
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   int a = 10;
   int b = 21;
   printf( "The larger of %d and %d is %d\n",  a, b, __max( a, b ) );
   printf( "The smaller of %d and %d is %d\n", a, b, __min( a, b ) );
}
```

mblen, _fmblen

#include <stdlib.h>

Syntax int mblen(const char *_mbstr_, size_t _count_);

int __far _fmblen(const char __far *_mbstr_, size_t _count_);

Parameter	Description
mbstr	The address of a sequence of bytes (a multibyte character)
count	The number of bytes to check

The mblen function returns the length in bytes of a valid multibyte character. It examines *count* or fewer bytes contained in *mbstr*. It will not examine more than MB_CUR_MAX bytes.

The _fmblen function is a model-independent (large-model) form of the mblen function.

Return Value

If *mbstr* is not NULL, both mblen and _fmblen return the length, in bytes, of the multibyte character. If *mbstr* is NULL, or the object that it points to is the wide-character null character ('\0'), both functions return 0. If the object that *mbstr* points to does not form a valid multibyte character within the first *count* characters, both functions return -1.

mblen

Standards: ANSI
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fmblen

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

mbstowcs
mbtowc
wcstombs
wctomb
MB_CUR_MAX
MB_LEN_MAX

```
/* MBLEN.CPP illustrates the behavior of the mblen function. */
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   int      i;
   char    *pmbc = (char *)malloc( sizeof( char ) );
   wchar_t  wc   = L'a';
   printf( "Convert a wide character to multibyte character:\n" );
   i = wctomb( pmbc, wc );
   printf( "\tCharacters converted: %u\n", i );
   printf( "\tMultibyte character: %x\n\n", pmbc );
   printf( "Find length--in bytes--of multibyte character:\n" );
   i = mblen( pmbc, MB_CUR_MAX );
   printf( "\tLength--in bytes--of multibyte character: %u\n", i );
   printf( "\tWide character: %x\n\n", pmbc );
   printf( "Attempt to find length of a NULL pointer:\n" );
   pmbc = NULL;
   i = mblen( pmbc, MB_CUR_MAX );
   printf( "\tLength--in bytes--of multibyte character: %u\n", i );
   printf( "\tWide character: %x\n", pmbc );
   printf( "Attempt to find length of a wide-character NULL:\n" );
   wc = L'\0';
   wctomb( pmbc, wc );
   i = mblen( pmbc, MB_CUR_MAX );
   printf( "\tLength--in bytes--of multibyte character: %u\n", i );
   printf( "\tWide character: %x\n", pmbc );
}
```

mbstowcs, _fmbstowcs

#include <stdlib.h>

Syntax size_t mbstowcs(wchar_t *_wcstr_, const char *_mbstr_, size_t _count_);

size_t __far _fmbstowcs(wchar_t __far *_wcstr_, const char __far *_mbstr_, size_t _count_);

Parameter	Description
wcstr	The address of a sequence of wide characters
mbstr	The address of a sequence of multibyte characters
count	The number of multibyte characters to convert

The mbstowcs function converts *count* or fewer multibyte characters pointed to by *mbstr* to a string of corresponding wide characters that are determined by the current locale. It stores the resulting wide-character string at the address represented by *wcstr*. The result is similiar to a series of calls to the mbtowc function.

If mbstowcs encounters the null character ('\0') either before or when *count* occurs, it converts the null character to a wide-character null character ('\0') and stops. Thus, the wide-character string at *wcstr* is null-terminated only if a null character is encountered during conversion. If the sequences pointed to by *wcstr* and *mbstr* overlap, the behavior is undefined.

The _fmbstowcs function is a model-independent (large-model) form of the mbstowcs function. It can be called from any point in any program.

Return Value

If mbstowcs or (_fmbstowcs) successfully converts the *source* string, it returns the number of converted multibyte characters. If either function encounters an invalid multibyte character, it returns -1. If the return value is *count*, the wide-character string is not null-terminated.

mbstowcs

Standards: ANSI
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fmbstowcs

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

mblen
mbtowc
wcstombs
wctomb
MB_CUR_MAX
MB_LEN_MAX

```
/* MBSTOWCS.CPP illustrates the behavior
 * of the mbstowcs function. */
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   int i;
   char    *pmbhello = (char *)malloc( MB_CUR_MAX );
   wchar_t *pwchello = L"Hi";
   wchar_t *pwc      = (wchar_t *)malloc( sizeof( wchar_t ));
   printf( "Convert to multibyte string:\n" );
   i = wcstombs( pmbhello, pwchello, MB_CUR_MAX );
   printf( "\tCharacters converted: %u\n", i );
   printf( "\tHex value of first" );
   printf( " multibyte character: %#.4x\n\n", pmbhello );
   printf( "Convert back to wide-character string:\n" );
   i = mbstowcs( pwc, pmbhello, MB_CUR_MAX );
   printf( "\tCharacters converted: %u\n", i );
   printf( "\tHex value of first" );
   printf( " wide character: %#.4x\n\n", pwc );
}
```

mbtowc, _fmbtowc

#include <stdlib.h>

Syntax int mbtowc(wchar_t *_wchar_, const char *_mbchar_, size_t _count_);

int __far _fmbtowc(wchar_t __far *_wchar_, const char __far *_mbchar_, size_t _count_);

Parameter	Description
wchar	The address of a wide character (type wchar_t)
mbchar	The address of a sequence of bytes (a multibyte character)
count	The number of bytes to check

The mbtowc function converts *count* or fewer bytes pointed to by mbchar, if *mbchar* is not NULL, to a corresponding wide character that is determined by the current locale. It stores the resulting wide character at *wchar*, if *wchar* is not NULL. It will not examine more than MB_CUR_MAX bytes.

The _fmbtowc function is a model-independent (large-model) form of the mbtowc function.

Return Value

If *mbchar* is not NULL and if the object that *mbchar* points to forms a valid multibyte character, both mbtowc and _fmbtowc return the length in bytes of the multibyte character.

If *mbchar* is NULL or the object that it points to is a wide-character null character ('\0'), both functions return 0. If the object that mbchar points to does not form a valid multibyte character within the first *count* characters, they return -1.

mbtowc

Standards: ANSI
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fmbtowc

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

mblen
mbtowc
wcstombs
wctomb
MB_CUR_MAX
MB_LEN_MAX

```
/* MBTOWC.CPP illustrates the behavior of the mbtowc function. */
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   int      i;
   char    *pmbc    = (char *)malloc( sizeof( char ) );
   wchar_t  wc      = L'a';
   wchar_t *pwcnull = NULL;
   wchar_t *pwc     = (wchar_t *)malloc( sizeof( wchar_t ) );
   printf( "Convert a wide character to multibyte character:\n" );
   i = wctomb( pmbc, wc );
   printf( "\tCharacters converted: %u\n", i );
   printf( "\tMultibyte character: %x\n\n", pmbc );
   printf( "Convert multibyte character back to a wide character:\n" );
   i = mbtowc( pwc, pmbc, MB_CUR_MAX );
   printf( "\tCharacters converted: %u\n", i );
   printf( "\tWide character: %x\n\n", pwc );
   printf( "Attempt to convert when target is NULL\n" );
   printf( "  returns the length of the multibyte character:\n" );
   i = mbtowc( pwcnull, pmbc, MB_CUR_MAX );
   printf( "\tlength of multibyte character: %u\n\n", i );
   printf( "Attempt to convert a NULL pointer to a" );
   printf( " wide character:\n" );
   pmbc = NULL;
   i = mbtowc( pwc, pmbc, MB_CUR_MAX );
   printf( "\tBytes converted: %u", i );
}
```

_memavl

#include <malloc.h> Required only for function declarations

Syntax size_t _memavl(void);

The _memavl function returns the approximate size, in bytes, of the memory available for dynamic memory allocation in the near heap (default data segment). The _memavl function can be used with calloc, malloc, or realloc in tiny, small, and medium memory models and with _ncalloc, _nmalloc or _nrealloc in any memory model.

The number returned by the _memavl function may not be the number of contiguous bytes. Consequently, a call to malloc requesting allocation of the size returned by _memavl may not succeed. Use the _memmax function to find the size of the largest available contiguous block of memory.

Return Value

The _memavl function returns the size in bytes as an unsigned integer.

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* MEMAVL.C: This program uses _memavl to determine the amount
 * of memory available for dynamic allocation. It then uses
 * malloc to allocate space for 5,000 long integers and uses
 * _memavl again to determine the new amount of available memory.
 */
#include <malloc.h>
#include <stdio.h>
void main( void )
{
   long __near *longptr;
   printf( "Memory available before _nmalloc = %u\n", _memavl() );
   if( (longptr = _nmalloc( 5000 * sizeof( long ) )) != NULL )
   {
      printf( "Memory available after _nmalloc = %u\n", _memavl() );
      _nfree( longptr );
   }
}
```

_memccpy, _fmemccpy

#include <memory.h> Required only for function declarations

#include <string.h> Use either STRING.H or MEMORY.H.

Syntax void *_memccpy(void **dest*, void **src*, int *c*, unsigned int *count*);

void __far * __far _fmemccpy(void __far **dest*, void __far **src*, int *c*, unsigned int *count*);

Parameter	Description
dest	Pointer to destination
src	Pointer to source
c	Last character to copy
count	Number of characters

The _memccpy and _fmemccpy functions copy 0 or more bytes of *src* to *dest*, halting when the character *c* has been copied or when *count* bytes have been copied, whichever comes first.

The _fmemccpy function is a model-independent (large-model) form of the _memccpy function. It can be called from any point in any program.

Return Value

If the character *c* is copied, _memccpy or _fmemccpy returns a pointer (or far pointer) to the byte in *dest* that immediately follows the character. If *c* is not copied, both return NULL.

Use _memccpy for compatibility with ANSI naming conventions of non-ANSI functions. Use memccpy and link with OLDNAMES.LIB for UNIX compatibility.

_memccpy

Standards: UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fmemccpy

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

memchr
memcmp
memcpy
memset

```
/* MEMCCPY.C */
#include <memory.h>
#include <stdio.h>
#include <string.h>
char string1[60] = "The quick brown dog jumps over the lazy fox";
void main( void )
{
   char buffer[61];
   char *pdest;
   printf( "Function:\t_memccpy 60 characters or to character 's'\n" );
   printf( "Source:\t\t%s\n", string1 );
   pdest = _memccpy( buffer, string1, 's', 60 );
   *pdest = '\0';
   printf( "Result:\t\t%s\n", buffer );
   printf( "Length:\t\t%d characters\n\n", strlen( buffer ) );
}
```

memchr, _fmemchr

#include <memory.h> Required only for function declarations

#include <string.h> Use either STRING.H (for ANSI compatibility) or MEMORY.H

Syntax void *memchr(const void **buf*, int *c*, size_t *count*);

void __far * __far _fmemchr(const void __far **buf*, int *c*, size_t *count*);

Parameter	Description
buf	Pointer to buffer
c	Character to look for
count	Number of characters

The memchr and _fmemchr functions look for the first occurrence of *c* in the first *count* bytes of *buf*. They stop when they find *c* or when they have checked the first *count* bytes.

The _fmemchr function is a model-independent (large-model) form of the memchr function. It can be called from any point in any program.

Return Value

If successful, memchr or _fmemchr returns a pointer (or a far pointer) to the first location of *c* in *buf*. Otherwise, they return NULL.

memchr

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fmemchr

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_memccpy
memcmp
memcpy
memset
strchr

```
/* MEMCHR.C */
#include <memory.h>
#include <stdio.h>
int  ch = 'r';
char str[] =    "lazy";
char string[] = "The quick brown dog jumps over the lazy fox";
char fmt1[] =   "         1         2         3         4         5";
char fmt2[] =   "12345678901234567890123456789012345678901234567890";
void main( void )
{
   char *pdest;
   int result;
   printf( "String to be searched:\n\t\t%s\n", string );
   printf( "\t\t%s\n\t\t%s\n\n", fmt1, fmt2 );
   printf( "Search char:\t%c\n", ch );
   pdest = memchr( string, ch, sizeof( string ) );
   result = pdest - string + 1;
   if( pdest != NULL )
      printf( "Result:\t\t%c found at position %d\n\n", ch, result );
   else
      printf( "Result:\t\t%c not found\n" );
}
```

memcmp, _fmemcmp

#include <memory.h> Required only for function declarations

#include <string.h> Use either STRING.H (for ANSI compatibility) or MEMORY.H

Syntax int memcmp(const void *_buf1_, const void *_buf2_, size_t _count_);

int __far _fmemcmp(const void __far *_buf1_, const void __far *_buf2_, size_t _count_);

Parameter	Description
buf1	First buffer
buf2	Second buffer
count	Number of characters

The memcmp and _fmemcmp functions compare the first *count* bytes of *buf1* and *buf2* and return a value indicating their relationship, as follows:

Value	Meaning
< 0	*buf1* less than *buf2*
= 0	*buf1* identical to *buf2*
> 0	*buf1* greater than *buf2*

The _fmemcmp function is a model-independent (large-model) form of the memcmp function. It can be called from any point in a program.

There is a semantic difference between the function version of memcmp and its intrinsic version. The function version supports huge pointers in compact-, large-, and huge-model programs, but the intrinsic version does not.

Return Value

The memcmp and _fmemcmp functions return an integer value, as described above.

memcmp

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fmemcmp

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_memccpy
memchr
memcpy
memset
strcmp
strncmp

```
/* MEMCMP.C: This program uses memcmp to compare
 * the strings named first and second. If the first
 * 19 bytes of the strings are equal, the program
 * considers the strings to be equal.
 */
#include <string.h>
#include <stdio.h>
void main( void )
{
   char first[]  = "12345678901234567890";
   char second[] = "12345678901234567891";
   int result;
   printf( "Compare '%.19s' to '%.19s':\n", first, second );
   result = memcmp( first, second, 19 );
   if( result < 0 )
      printf( "First is less than second.\n" );
   else if( result == 0 )
      printf( "First is equal to second.\n" );
   else if( result > 0 )
      printf( "First is greater than second.\n" );
   printf( "Compare '%.20s' to '%.20s':\n", first, second );
   result = memcmp( first, second, 20 );
   if( result < 0 )
      printf( "First is less than second.\n" );
   else if( result == 0 )
      printf( "First is equal to second.\n" );
   else if( result > 0 )
      printf( "First is greater than second.\n" );
}
```

memcpy, _fmemcpy

#include <memory.h> Required only for function declarations

#include <string.h> Use either STRING.H (for ANSI compatibility) or MEMORY.H

Syntax void *memcpy(void **dest*, const void **src*, size_t *count*);

void __far * __far _fmemcpy(void __far **dest*, const void __far **src*, size_t *count*);

Parameter	Description
dest	New buffer
src	Buffer to copy from
count	Number of characters to copy

The memcpy and _fmemcpy functions copy *count* bytes of *src* to *dest*. If the source and destination overlap, these functions do not ensure that the original source bytes in the overlapping region are copied before being overwritten. Use memmove to handle overlapping regions.

The _fmemcpy function is a model-independent (large-model) form of the memcpy function. It can be called from any point in any program.

There is a semantic difference between the function version of memcpy and its intrinsic version. The function version supports huge pointers in compact-, large-, and huge-model programs, but the intrinsic version does not.

Return Value

The memcpy and _fmemcpy functions return the value of *dest*.

memcpy

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fmemcpy

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_memccpy
memchr
memcmp
memmove
memset
strcpy
strncpy

```
/* MEMCPY.C. Illustrate overlapping copy: memmove
 * handles it correctly; memcpy does not.
 */
#include <memory.h>
#include <string.h>
#include <stdio.h>
char string1[60] = "The quick brown dog jumps over the lazy fox";
char string2[60] = "The quick brown fox jumps over the lazy dog";
/*                           1         2         3         4         5
 *                 123456789012345678901234567890123456789012345678901234567890
 */
void main( void )
{
   printf( "Function:\tmemcpy without overlap" );
   printf( "Source:\t\t%s\n", string1 + 40 );
   printf( "Destination:\t%s\n", string1 + 16 );
   memcpy( string1 + 16, string1 + 40, 3 );
   printf( "Result:\t\t%s\n", string1 );
   printf( "Length:\t\t%d characters\n\n", strlen( string1 ) );
   /* Restore string1 to original contents */
   memcpy( string1 + 16, string2 + 40, 3 );
   printf( "Function:\tmemmove with overlap\n" );
   printf( "Source:\t\t%s\n", string2 + 4 );
   printf( "Destination:\t%s\n", string2 + 10 );
   memmove( string2 + 10, string2 + 4, 40 );
   printf( "Result:\t\t%s\n", string2 );
   printf( "Length:\t\t%d characters\n", strlen( string2 ) );
   printf( "Function:\tmemcpy with overlap\n" );
   printf( "Source:\t\t%s\n", string1 + 4 );
   printf( "Destination:\t%s\n", string1 + 10 );
   memcpy( string1 + 10, string1 + 4, 40 );
   printf( "Result:\t\t%s\n", string1 );
   printf( "Length:\t\t%d characters\n\n", strlen( string1 ) );
}
```

_memicmp, _fmemicmp

#include <memory.h> Required only for function declarations

#include <string.h> Use either STRING.H (for ANSI compatibility) or MEMORY.H

Syntax int _memicmp(void *_buf1_, void *_buf2_, unsigned int _count_);

int __far _fmemicmp(void __far *_buf1_, void __far *_buf2_, unsigned int _count_);

Parameter	Description
buf1	First buffer
buf2	Second buffer
count	Number of characters

The _memicmp and _fmemicmp functions compare the first *count* characters of the two buffers *buf1* and *buf2* byte by byte. The comparison is made without regard to the case of letters in the two buffers; that is, uppercase and lowercase letters are considered equivalent. The _memicmp and _fmemicmp functions return a value indicating the relationship of the two buffers, as follows:

Value	Meaning
< 0	*buf1* less than *buf2*
= 0	*buf1* identical to *buf2*
> 0	*buf1* greater than *buf2*

The _fmemicmp function is a model-independent (large-model) form of the _memicmp function. It can be called from any point in any program.

Return Value

The _memicmp and _fmemicmp functions return an integer value, as described above.

Use _memicmp for compatibility with ANSI naming conventions of non- ANSI functions. Use memicmp and link with OLDNAMES.LIB for UNIX compatibility.

_memicmp

Standards: UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fmemicmp

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_memccpy
memchr
memcmp
memcpy
memset
_stricmp
_strnicmp

```
/* MEMICMP.C: This program uses _memicmp to compare
 * the first 29 letters of the strings named first and
 * second without regard to the case of the letters.
 */
#include <memory.h>
#include <stdio.h>
#include <string.h>
void main( void )
{
   int result;
   char first[] = "Those Who Will Not Learn from History";
   char second[] = "THOSE WHO WILL NOT LEARN FROM their mistakes";
   /* Note that the 29th character is right here ^ */
   printf( "Compare '%.29s' to '%.29s'\n", first, second );
   result = _memicmp( first, second, 29 );
   if( result < 0 )
      printf( "First is less than second.\n" );
   else if( result == 0 )
      printf( "First is equal to second.\n" );
   else if( result > 0 )
      printf( "First is greater than second.\n" );
}
```

_memmax

#include <malloc.h>

Syntax size_t _memmax(void);

The _memmax function returns the size (in bytes) of the largest contiguous block of memory that can be allocated from the near heap (i.e., the default data segment). Calling _nmalloc with the value returned by the _memmax function will succeed as long as _memmax returns a nonzero value.

Return Value

The function returns the block size, if successful. Otherwise, it returns 0, indicating that nothing more can be allocated from the near heap.

malloc functions
_msize functions

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* MEMMAX.C: This program uses _memmax and _nmalloc to allocate
 * the largest block of memory available in the near heap.
 */
#include <stddef.h>
#include <malloc.h>
#include <stdio.h>
void main( void )
{
   size_t contig;
   char *p;
   /* Determine contiguous memory size */
   contig = _memmax();
   printf( "Largest block of available memory is %u bytes long\n", contig  );
   if( contig )
   {
      p = _nmalloc( contig * sizeof( int ) );
      if( p == NULL )
         printf( "Error with malloc (should never occur)\n" );
      else
      {
         printf( "Maximum allocation succeeded\n" );
         free( p );
      }
   }
   else
      printf( "Near heap is already full\n" );
}
```

memmove, _fmemmove

#include <string.h>

Syntax void *memmove(void *_dest_, const void *_src_, size_t _count_);

void __far * __far _fmemmove(void __far *_dest_, const void __far *_src_, size_t _count_);

Parameter	Description
dest	Destination object
src	Source object
count	Number of characters to copy

The memmove and _fmemmove functions copy *count* characters from the source (*src*) to the destination (*dest*). If some regions of the source area and the destination overlap, the memmove and _fmemmove functions ensure that the original source bytes in the overlapping region are copied before being overwritten.

The _fmemmove function is a model-independent (large-model) form of the memmove function. It can be called from any point in any program.

Return Value

The memmove and _fmemmove functions return the value of *dest*.

memmove

Standards: ANSI
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fmemmove

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_memccpy
memcpy
strcpy
strncpy

memset, _fmemset

#include <memory.h> Required only for function declarations

#include <string.h> Use either STRING.H (for ANSI compatibility) or MEMORY.H

Syntax void *memset(void **dest*, int *c*, size_t *count*);

void __far * __far _fmemset(void __far **dest*, int *c*, size_t *count*);

Parameter	Description
dest	Pointer to destination
c	Character to set
count	Number of characters

The memset and _fmemset functions set the first *count* bytes of dest to the character *c*.

The _fmemset function is a model-independent (large-model) form of the memset function. It can be called from any point in any program.

There is a semantic difference between the function version of memset and its intrinsic version. The function version supports huge pointers in compact-, large-, and huge-model programs, but the intrinsic version does not.

Return Value

The memset and _fmemset functions return the value of *dest*.

memset

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fmemset

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_memccpy
memchr
memcmp
memcpy
_strnset

```
/* MEMSET.C: This program uses memset to
 * set the first four bytes of buffer to "*".
 */
#include <memory.h>
#include <stdio.h>
void main( void )
{
   char buffer[] = "This is a test of the memset function";
   printf( "Before: %s\n", buffer );
   memset( buffer, '*', 4 );
   printf( "After:  %s\n", buffer );
}
```

__min

#include <stdlib.h>

Syntax *type* __min(*type a*, *type b*);

Parameter	Description
type	Any numeric data type
a, *b*	Values of any numeric type to be compared

The __min macro compares two values and returns the value of the smaller one. The arguments can be of any numeric data type, signed or unsigned. Both arguments and the return value must be of the same data type.

Return Value

The macro returns the smaller of the two arguments.

$\underline{\underline{\max}}$

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_mkdir

#include <direct.h> Required only for function declarations

Syntax int _mkdir(const char **dirname*);

Parameter	**Description**
dirname	Path for new directory

The _mkdir function creates a new directory with the specified *dirname*. Only one directory can be created at a time, so only the last component of *dirname* can name a new directory.

The _mkdir function does not do any translation of path delimiters. All operating systems accept either " or "/ " internally as valid delimiters within paths.

Return Value

The _mkdir function returns the value 0 if the new directory was created. A return value of -1 indicates an error, and errno is set to one of the following values:

Value	**Meaning**
EACCES	Directory not created. The given name is the name of an existing file, directory, or device.
ENOENT	Path not found.

_chdir
_rmdir

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* MAKEDIR.C */
#include <direct.h>
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   if( _mkdir( "\\testtmp" ) == 0 )
   {
      printf( "Directory '\\testtmp' was successfully created\n" );
      system( "dir \\testtmp" );
      if( _rmdir( "\\testtmp" ) == 0 )
        printf( "Directory '\\testtmp' was successfully removed\n"  );
      else
         printf( "Problem removing directory '\\testtmp'\n" );
   }
   else
      printf( "Problem creating directory '\\testtmp'\n" );
}
```

_MK_FP

#include <dos.h>

Syntax void __far *_MK_FP(unsigned *seg*, unsigned *offset*);

Parameter	Description
seg	Segment value for pointer
offset	Offset value for pointer

The _MK_FP macro makes a far pointer from the segment value, *seg*, and the offset value, *offset*.

Return Value

The _MK_FP macro returns the resulting far pointer.

__FP_OFF
__FP_SEG

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_mktemp

#include <io.h> Required only for function declarations

Syntax char *_mktemp(char **template*);

Parameter	**Description**
template	Filename pattern

The _mktemp function creates a unique filename by modifying the given *template* argument. The *template* argument has the form:

*base*XXXXXX

where *base* is the part of the new filename that you supply, and the X 's are placeholders for the part supplied by _mktemp; _mktemp preserves *base* and replaces the six trailing X's with an alphanumeric character followed by a five-digit value. The five-digit value is a unique number identifying the calling process. The alphanumeric character is 0 ('0') the first time _mktemp is called with a given template.

In subsequent calls from the same process with copies of the same template, _mktemp checks to see if previously returned names have been used to create files. If no file exists for a given name, _mktemp returns that name. If files exist for all previously returned names, _mktemp creates a new name by replacing the alphanumeric character in the name with the next available lowercase letter. For example, if the first name returned is t012345 and this name is used to create a file, the next name returned will be ta12345. When creating new names, _mktemp uses, in order, '0' and then the lowercase letters 'a' through 'z'.

Note that the original template is modified by the first call to _mktemp. If you then call the _mktemp function again with the same template (i.e., the original one), you will get an error.

The _mktemp function generates unique filenames but does not create or open files.

Return Value

The _mktemp function returns a pointer to the modified template. The return value is NULL if the *template* argument is badly formed or no more unique names can be created from the given template.

Use _mktemp for compatibility with ANSI naming conventions of non-ANSI functions. Use mktemp and link with OLDNAMES.LIB for UNIX compatibility.

fopen
_getpid
_open
_tempnam
tmpfile

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* MKTEMP.C: The program uses _mktemp to create
 * five unique filenames. It opens each filename
 * to ensure that the next name is unique.
 */
#include <io.h>
#include <string.h>
#include <stdio.h>
char *template = "fnXXXXXX";
char *result;
char names[5][9];
void main( void )
{
   int i;
   FILE *fp;
   for( i = 0; i < 5; i++ )
   {
      strcpy( names[i], template );
      /* Attempt to find a unique filename: */
      result = _mktemp( names[i] );
      if( result == NULL )
         printf( "Problem creating the template" );
      else
      {
         if( (fp = fopen( result, "w" )) != NULL )
            printf( "Unique filename is %s\n", result );
         else
            printf( "Cannot open %s\n", result );
         fclose( fp );
      }
   }
}
```

mktime

#include <time.h>

Syntax time_t mktime(struct tm *_timeptr_);

Parameter	**Description**
timeptr	Pointer to time structure

The mktime function converts the supplied time structure (possibly incomplete) pointed to by *timeptr* into a fully defined structure with "normalized" values and then converts it to a time_t calendar time value. The structure for the tm is described in asctime.

To avoid obtaining unreliable results, set the tm_isdst field before calling mktime. If the input time is standard time, set tm_isdst to 0; if the input time is daylight saving time, set tm_isdst to a value greater than 0; if you don't know whether the input time is daylight saving time or standard time and you want mktime to make a good guess, set tm_isdst to a value less than 0.

The converted time has the same encoding as the values returned by the time function. The original values of the tm_wday and tm_yday components of the *timeptr* structure are ignored, and the original values of the other components are not restricted to their normal ranges.

If successful, mktime sets the values of tm_wday and tm_yday appropriately, and sets the other components to represent the specified calendar time, but with their values forced to the normal ranges; the final value of tm_mday is not set until tm_mon and tm_year are determined.

The mktime function handles dates in any time zone from midnight, January 1, 1970, to midnight, February 5, 2036. If timeptr references a date before midnight, January 1, 1970, mktime returns –1 cast to type time_t.

Note that the gmtime and localtime functions use a single statically allocated buffer for the conversion. If you supply this buffer to mktime, the previous contents will be destroyed.

Return Value

The mktime function returns the specified calendar time encoded as a value of type time_t. If the calendar time cannot be represented, the function returns the value -1 cast to type time_t.

asctime
gmtime
localtime
time

Standards: ANSI

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* MKTIME.C: The example takes a number of days
 * as input and returns the time, the current
 * date, and the specified number of days.
 */
#include <time.h>
#include <stdio.h>
void main( void )
{
   struct tm when;
   time_t now, result;
   int    days;
   time( &now );
   when = *localtime( &now );
   printf( "Current time is %s\n", asctime( &when ) );
   printf( "How many days to look ahead: " );
   scanf( "%d", &days );
   when.tm_mday = when.tm_mday + days;
   if( (result = mktime( &when )) != (time_t)-1 )
      printf( "In %d days the time will be %s\n", days, asctime( &when ) );
   else
      perror( "mktime failed" );
}
```

modf, modfl

#include <math.h>

Syntax double modf(double *x*, double **intptr*);

long double modfl(long double *x*, long double **intptr*);

Parameter	Description
x	Floating-point value
intptr	Pointer to stored integer portion

The modf functions break down the floating-point value *x* into fractional and integer parts, each of which has the same sign as *x*. The signed fractional portion of *x* is returned. The integer portion is stored as a floating-point value at *intptr*.

The modfl function uses the 80-bit, 10-byte coprocessor form of arguments and return values. See the long double functions for more details on this data type.

Return Value

The modf and modfl functions return the signed fractional portion of *x*. There is no error return.

modf

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

modfl

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

frexp
ldexp

```
/* MODF.C */
#include <math.h>
#include <stdio.h>
void main( void )
{
   double x, y, n;
   x = -14.87654321;      /* Divide x into its fractional */
   y = modf( x, &n );     /* and integer parts            */
   printf( "For %f, the fraction is %f and the integer is %.f\n", x, y, n  );
}
```

_movedata

#include <memory.h> Required only for function declarations

#include <string.h> Use either STRING.H or MEMORY.H

Syntax void _movedata(unsigned int *srcseg*, unsigned int *srcoff*, unsigned int *destseg*, unsigned int *destoff*, unsigned int *count*);

Parameter	Description
srcseg	Segment address of source
srcoff	Segment offset of source
destseg	Segment address of destination
destoff	Segment offset of destination
count	Number of bytes

The _movedata function copies *count* bytes from the source address specified by *srcseg*:*srcoff* to the destination address specified by *destseg*:*destoff*.

The _movedata function was intended to move far data in small-model programs. The newer model-independent _fmemcpy and _fmemmove functions should be used instead of the _movedata function. In large-model programs, the memcpy and memmove functions can also be used.

Segment values for the *srcseg* and *destseg* arguments can be obtained by using either the _segread function or the _FP_SEG macro.

The _movedata function does not handle all cases of overlapping moves correctly. These occur when part of the destination is the same memory area as part of the source. The memmove function correctly handles overlapping moves.

Return Value

None.

_FP_OFF
_FP_SEG
memcpy
memmove
_segread

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* MOVEDATA.C */
#include <memory.h>
#include <stdio.h>
#include <string.h>
#include <dos.h>
#include <malloc.h>
char __far *src = "This is a test.";
void main( void )
{
   char __far *dest;
   if( (dest = _fmalloc( 80 )) != NULL )
   {
      _movedata( _FP_SEG( src ),  _FP_OFF( src ),
                 _FP_SEG( dest ), _FP_OFF( dest ), _fstrlen( src ) + 1 );
      printf( "The source data at %Fp is '%Fs'\n", src, src );
      printf( "The destination data at %Fp is '%Fs'\n", dest, dest );
      _ffree( dest );
   }
}
```

_moveto Functions

#include <graph.h>

Syntax struct _xycoord __far _moveto(short *x*, short *y*);

struct _wxycoord __far _moveto_w(double *wx*, double *wy*);

Parameter	Description
x, *y*	View-coordinate point
wx, *wy*	Window-coordinate point

The _moveto functions move the current position to the specified point. The _moveto function uses the view-coordinate point (*x*, *y*) as the current position. The _moveto_w function uses the window-coordinate point (*wx*, *wy*) as the current position. No drawing takes place.

The _moveto function operates only in graphics video modes (e.g., _MRES4COLOR). Because it is a graphics function, the color of text is set by the _setcolor function, not by the _settextposition function.

Return Value

The function returns the coordinates of the previous position. The _moveto function returns the coordinates in an _xycoord structure. The _xycoord structure, defined in GRAPH.H, contains the following elements:

Element	Description
short xcoord	*x* coordinate
short ycoord	*y* coordinate

The _moveto_w function returns the coordinates in an _wxycoord structure, defined in GRAPH.H. The _wxycoord structure contains the following elements:

Element	Description
double wx	*x* window coordinate
double wy	*y* window coordinate

_lineto functions
_outgtext

Standards: None
16-Bit: MS-DOS, QWIN

```
/* MOVETO.C: This program draws line
 * segments of different colors.
 */
#include <graph.h>
#include <stdlib.h>
#include <conio.h>
void main( void )
{
   short x, y, xinc, yinc, color = 1;
   struct _videoconfig v;
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXCOLORMODE ) )
      exit( 1 );
   _getvideoconfig( &v );
   xinc = v.numxpixels / 50;
   yinc = v.numypixels / 50;
   for( x = 0, y = v.numypixels - 1; x < v.numxpixels; x += xinc, y -=  yinc )
   {
      _setcolor( color++ % 16 );
      _moveto( x, 0 );
      _lineto( 0, y );
   }
   _getch();
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}
```

_msize Functions

#include <malloc.h> Required only for function declarations

Syntax size_t _msize(void **memblock*);

size_t _bmsize(__segment *seg*, void __based(void) **memblock*);

size_t _fmsize(void __far **memblock*);

size_t _nmsize(void __near **memblock*);

Parameter	Description
memblock	Pointer to memory block
seg	Based-heap segment selector

The _msize family of functions returns the size, in bytes, of the memory block allocated by a call to the appropriate version of the calloc, malloc, or realloc functions.

In large data models (compact-, large-, and huge-model programs), _msize maps to _fmsize. In small data models (tiny-, small-, and medium-model programs), _msize maps to _nmsize.

The _nmsize function returns the size (in bytes) of the memory block allocated by a call to _nmalloc, and the _fmsize function returns the size (in bytes) of the memory block allocated by a call to _fmalloc or _frealloc. The _bmsize function returns the size of a block allocated in segment *seg* by a call to _bmalloc, _bcalloc, or _brealloc.

The location of the memory block is indicated below:

Function	Data Segment
_msize	Depends on data model of program
_bmsize	Based heap segment specified by *seg* value
_fmsize	Far heap segment (outside default data segment)
_nmsize	Default data segment (inside near heap)

Return Value

All four functions return the size (in bytes) as an unsigned integer.

_msize

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_bmsize, _fmsize, _nmsize

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

calloc functions
_expand functions
malloc functions
realloc functions

```
/* REALLOC.C: This program allocates a block of memory for
 * buffer and then uses _msize to display the size of that
 * block. Next, it uses realloc to expand the amount of
 * memory used by buffer and then calls _msize again to
 * display the new amount of memory allocated to buffer.
 */
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
void main( void )
{
   long *buffer;
   size_t size;
   if( (buffer = (long *)malloc( 1000 * sizeof( long ) )) == NULL )
      exit( 1 );
   size = _msize( buffer );
   printf( "Size of block after malloc of 1000 longs: %u\n", size );
   /* Reallocate and show new size: */
   if( (buffer = realloc( buffer, size + (1000 * sizeof( long )) )) ==  NULL )
      exit( 1 );
   size = _msize( buffer );
   printf( "Size of block after realloc of 1000 more longs: %u\n", size );
   free( buffer );
   exit( 0 );
}
```

_onexit, _fonexit

#include <stdlib.h>

Syntax _onexit_t _onexit(_onexit_t *func*);

_fonexit_t __far _fonexit(_fonexit_t *func*);

Parameter	Description
func	Pointer to function to be called at exit

The _onexit function is passed the address of a function (*func*) to be called when the program terminates normally. Successive calls to _onexit create a register of functions that is executed in LIFO (last-in-first-out) order. No more than 32 functions can be registered with _onexit; _onexit returns the value NULL if the number of functions exceeds 32. The functions passed to _onexit cannot take parameters.

The _fonexit function is a far version of _onexit; it can be used with any memory model.

Neither _onexit nor _fonexit is part of the ANSI definition; instead, both are Microsoft extensions. The ANSI-standard atexit function does the same thing as _onexit and should be used instead of _onexit when ANSI portability is desired.

Return Value

Both _onexit and _fonexit return a pointer to the function if successful and return NULL if there is no space left to store the function pointer.

Use _onexit for compatibility with ANSI naming conventions of non-ANSI functions. Use onexit and link with OLDNAMES.LIB for UNIX compatibility.

_onexit

Standards: UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fonexit

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

exit

```
/* ONEXIT.C */
#include <stdlib.h>
#include <stdio.h>
/* Prototypes */
int fn1(), fn2(), fn3(), fn4();
void main( void )
{
   _onexit( fn1 );
   _onexit( fn2 ); _onexit( fn3 );
   _onexit( fn4 );
   printf( "This is executed first.\n" );
}
int fn1()
{
   printf( "next.\n" );
   return 0;
}
int fn2()
{
   printf( "executed " );
   return 0;
}
int fn3()
{
   printf( "is " );
   return 0;
}
int fn4()
{
   printf( "This " );
   return 0;
}
```

_open

#include <fcntl.h>, <sys\types.h>, <sys\stat.h>, <io.h>

Syntax int _open(const char **filename*, int *oflag* [, int *pmode*]);

Parameter	**Description**
filename	Filename
oflag	Type of operations allowed
pmode	Permission mode

The _open function opens the file specified by *filename* and prepares the file for subsequent reading or writing, as defined by *oflag*. The *oflag* argument is an integer expression formed from one or more of the manifest constants defined in FCNTL.H (listed below). When two or more manifest constants are used to form the *oflag* argument, the constants are combined with the bitwise-OR operator (|). .

The FCNTL.H file defines the following manifest constants:

_O_APPEND
Repositions the file pointer to the end of the file before every write operation.

_O_BINARY
Opens file in binary (untranslated) mode.

_O_CREAT
Creates and opens a new file for writing; this has no effect if the file specified by *filename* exists.

_O_EXCL
Returns an error value if the file specified by *filename* exists. Only applies when used with _O_CREAT.

_O_RDONLY
Opens file for reading only; if this flag is given, neither _O_RDWR nor _O_WRONLY can be given.

_O_RDWR
Opens file for both reading and writing; if this flag is given, neither _O_RDONLY nor _O_WRONLY can be given.

_O_TEXT
Opens file in text (translated) mode.

_O_TRUNC
Opens and truncates an existing file to zero length; the file must have write permission. The contents of the file are destroyed. If this flag is given, you cannot specify _O_RDONLY.

_O_WRONLY
Opens file for writing only; if this flag is given, neither _O_RDONLY nor _O_RDWR can be given.

Warning Use the _O_TRUNC flag with care, as it destroys the complete contents of an existing file.

Either _O_RDONLY, _O_RDWR, or _O_WRONLY must be given to specify the access mode. There is no default value for the access mode.

The *pmode* argument is required only when _O_CREAT is specified. If the file exists, *pmode* is ignored. Otherwise, *pmode* specifies the file's permission settings, which are set when the new file is closed for the first time. The *pmode* is an integer expression containing one or both of the manifest constants _S_IWRITE and _S_IREAD, defined in SYS\STAT.H. When both constants are given, they are joined with the bitwise-OR operator (|). The meaning of the *pmode* argument is as follows:

Value	**Meaning**
_S_IWRITE	Writing

	permitted
_S_IREAD	Reading permitted
_S_IREAD | _S_IWRITE	Reading and writing permitted

If write permission is not given, the file is read-only. With MS-DOS, all files are readable; it is not possible to give write-only permission. Thus the modes _S_IWRITE and _S_IREAD | _S_IWRITE are equivalent.

The _open function applies the current file-permission mask to *pmode* before setting the permissions (see _umask).

The *filename* argument used in the _open function is affected by the MS-DOS APPEND command.

Note that with MS-DOS versions 3.0 and later, a problem occurs when SHARE is installed and a new file is opened with *oflag* set to _O_CREAT | _O_RDONLY or _O_CREAT | _O_WRONLY and *pmode* set to _S_IREAD. Under these conditions, the operating system prematurely closes the file during system calls made within _open.

To work around the problem, open the file with the *pmode* argument set to _S_IWRITE. Then close the file and use _chmod to change the access mode back to _S_IREAD. Another workaround is to open the file with *pmode* set to _S_IREAD and *oflag* set to _O_CREAT | _O_RDWR.

Return Value

The _open function returns a file handle for the opened file. A return value of -1 indicates an error, and errno is set to one of the following values:

Value	Meaning
EACCES	Given path is a directory; or an attempt was made to open a read-only file for writing; or a sharing violation occurred (the file's sharing mode does not allow the specified operations).
EEXIST	The _O_CREAT and _O_EXCL flags are specified, but the named file already exists.
EINVAL	An invalid *oflag* or *pmode* argument was given.
EMFILE	No more file handles available (too many open files).
ENOENT	File or path not found.

Use _open for compatibility with ANSI naming conventions of non-ANSI functions. Use open and link with OLDNAMES.LIB for UNIX compatibility.

_chmod
_close
_creat
_dup
_dup2
fopen
_sopen
_umask

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* OPEN.C: This program uses _open to open a file
 * named OPEN.C for input and a file named OPEN.OUT
 * for output. The files are then closed.
 */
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>
void main( void )
{
   int fh1, fh2;
   fh1 = _open( "OPEN.C", _O_RDONLY );
   if( fh1 == -1 )
      perror( "open failed on input file" );
   else
   {
      printf( "open succeeded on input file\n" );
      _close( fh1 );
   }
   fh2 = _open( "OPEN.OUT", _O_WRONLY | _O_CREAT, _S_IREAD | _S_IWRITE );
   if( fh2 == -1 )
      perror( "open failed on output file" );
   else
   {
      printf( "open succeeded on output file\n" );
      _close( fh2 );
   }
}
```

_outgtext

#include <graph.h>

Syntax void __far _outgtext(const char __far *_text_);

Parameter	Description
text	Text string to output

The _outgtext function outputs on the screen the null-terminated string that *text* points to. The text is output using the current font at the current graphics position and in the current color.

No formatting is provided, in contrast to the standard console I/O library routines such as printf.

After it outputs the text, _outgtext updates the current graphics position.

The _outgtext function operates only in graphics video modes (e.g., _MRES4COLOR). Because it is a graphics function, the color of text is set by the _setcolor function, not by the _settextcolor function. Similarly, the position is affected by the _moveto function, not by the _settextposition function.

The _outgtext function operates differently in QuickWin applications. For specific information, see *Programming Techniques*.

Return Value

None.

_moveto functions
_setcolor
_setfont

Standards: None
16-Bit: MS-DOS, QWIN

```
/* OUTGTXT.C illustrates font output using functions:
 *   _registerfonts      _setfont         _outgtext
 *   _unregisterfonts    _getfontinfo     _getgtextextent
 *   _setgtextvector
 */
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <graph.h>
#define NFONTS 4
unsigned char *face[NFONTS] =
{
   "Courier", "Modern", "Script", "Roman"
};
unsigned char *options[NFONTS] =
{
   "courier", "modern", "script", "roman"
};
void main( void )
{
   unsigned char list[20];
   char fondir[_MAX_PATH];
   struct _videoconfig vc;
   struct _fontinfo fi;
   short fontnum, x, y;
   /* Read header info from all .FON files in current or given directory. */
   if( _registerfonts( "*.FON" ) <= 0 )
   {
      _outtext( "Enter full path where .FON files are located: " );
      gets( fondir );
      strcat( fondir, "\\*.FON" );
      if( _registerfonts( fondir ) <= 0 )
      {
         _outtext( "Error: can't register fonts" );
         exit( 1 );
      }
   }
   /* Set highest available graphics mode and get configuration. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   _getvideoconfig( &vc );
   /* Display each font name centered on screen. */
   for( fontnum = 0; fontnum < NFONTS; fontnum++ )
   {
      /* Build options string. */
      strcat( strcat( strcpy( list, "t'" ),
      options[fontnum] ), "'");
      strcat( list, "h30w24b" );
      _clearscreen( _GCLEARSCREEN );
      if( _setfont( list ) >= 0 )
      {
         /* Use length of text and height of font to center text. */
         x = (vc.numxpixels / 2) - (_getgtextextent( face[fontnum] ) / 2);
         y = (vc.numypixels / 2) + (_getgtextextent( face[fontnum] ) / 2);
         if( _getfontinfo( &fi ) )
         {
            _outtext( "Error: Can't get font information" );
            break;
         }
```

```
            _moveto( x, y );
            if( vc.numcolors > 2 )
               _setcolor( fontnum + 2 );
            /* Rotate and display text. */
            _setgtextvector( 1, 0 );
            _outgtext( face[fontnum] );
            _setgtextvector( 0, 1 );
            _outgtext( face[fontnum] );
            _setgtextvector( -1, 0 );
            _outgtext( face[fontnum] );
            _setgtextvector( 0, -1 );
            _outgtext( face[fontnum] );
         }
         else
         {
            _outtext( "Error: Can't set font: " );
            _outtext( list );
         }
         _getch();
      }
      _unregisterfonts();
      _setvideomode( _DEFAULTMODE );
      exit( 0 );
}
```

_outmem

#include <graph.h>

Syntax void __far _outmem(const char __far **text*, short *length*);

Parameter	Description
text	Text string to output
length	Length of string to output

The _outmem function outputs the string that *text* points to. The *length* argument specifies the number of characters to output.

Unlike _outtext, the _outmem function prints all characters literally, including ASCII 10, 13, and 0 as the equivalent graphics characters. No formatting is provided. Text is printed using the current text color, starting at the current text position.

To output text using special fonts, you must use the _outgtext function.

Standard console routines such as printf do not use or maintain information about the display, such as current text position, which is contained in the graphics library. Therefore, you cannot use standard console routines to output graphics text.

Return Value

None.

_outtext
_settextcolor
_settextposition
_settextwindow

Standards: None
16-Bit: MS-DOS, QWIN

```
/* OUTMEM.C illustrates:
 *    _outmem
 */
#include <stdio.h>
#include <graph.h>
void main( void )
{
   int  i, len;
   char tmp[10];
   _clearscreen( _GCLEARSCREEN );
   for( i = 0; i < 256; i++ )
   {
      _settextposition( (i % 24) + 1, (i / 24) * 7 );
      len = sprintf( tmp, "%3d %c", i, i );
      _outmem( tmp, len );
   }
   _settextposition( 24, 1 );
}
```

_outp, _outpw

#include <conio.h> Required only for function declarations

Syntax int _outp(unsigned *port*, int *databyte*);

unsigned _outpw(unsigned *port*, unsigned *dataword*);

Parameter	**Description**
port	Port number
databyte	Output value
dataword	Output value

The _outp and _outpw functions write a byte and a word, respectively, to the specified output port. The *port* argument can be any unsigned integer in the range 0 - 65,535; *databyte* can be any integer in the range 0 - 255; and *dataword* can be any value in the range 0 - 65,535.

Return Value

The functions return the data output. There is no error return.

_inp
_inpw

Standards: None
16-Bit: MS-DOS

```
/* OUTP.C: This program uses _inp and _outp
 * to make sound of variable tone and duration.
 */
#include <conio.h>
#include <stdio.h>
#include <time.h>
void Beep( unsigned duration, unsigned frequency );
void Sleep( clock_t _wait );
void main( void )
{
   Beep( 698, 700 );
   Beep( 523, 500 );
}
/* Sounds the speaker for a time specified in microseconds
 * by duration at a pitch specified in hertz by frequency.
 */
void Beep( unsigned frequency, unsigned duration )
{
   int control;
   /* If frequency is 0, Beep doesn't try to make a sound. */
   if( frequency )
   {
      /* 75 is about the shortest reliable duration of a sound. */
      if( duration < 75 )
         duration = 75;
      /* Prepare timer by sending 10111100 to port 43. */
      _outp( 0x43, 0xb6 );
      /* Divide input frequency by timer ticks per
       * second and write (byte by byte) to timer.
       */
      frequency = (unsigned)(1193180L / frequency);
      _outp( 0x42, (char)frequency );
      _outp( 0x42, (char)(frequency >> 8) );
      /* Save speaker control byte. */
      control = _inp( 0x61 );
      /* Turn on the speaker (with bits 0 and 1). */
      _outp( 0x61, control | 0x3 );
   }
   Sleep( (clock_t)duration );
   /* Turn speaker back on if necessary. */
   if( frequency )
      _outp( 0x61, control );
}
/* Pauses for a specified number of microseconds. */
void Sleep( clock_t _wait )
{
   clock_t goal;
   goal = _wait + clock();
   while( goal > clock() )
      ;
}
```

_outtext

#include <graph.h>

Syntax void __far _outtext(const char __far **text*);

Parameter	**Description**
text	Text string to output

The _outtext function outputs the null-terminated string that *text* points to. No formatting is provided, in contrast to the standard console I/O library routines such as printf. This function will work in any screen mode.

Text output begins at the current text position.

To output text using special fonts, you must use the _outgtext function.

The _outtext function operates differently in QuickWin applications. For specific information, see *Programming Techniques*.

Standard console routines such as printf do not use or maintain information about the display, such as current text position, which is contained in the graphics library. Therefore, you cannot use standard console routines to output graphics text.

Return Value

None.

_outmem
_settextcolor
_settextposition
_settextwindow
_wrapon

Standards: None
16-Bit: MS-DOS, QWIN

```
/* OUTTXT.C: This example illustrates text output functions:
 *   _gettextcolor  _getbkcolor  _gettextposition  _outtext
 *   _settextcolor  _setbkcolor  _settextposition
 */
#include <conio.h>
#include <stdio.h>
#include <graph.h>
char buffer [80];
void main( void )
{
   /* Save original foreground, background, and text position */
   short blink, fgd, oldfgd;
   long  bgd, oldbgd;
   struct _rccoord oldpos;
   /* Save original foreground, background, and text position. */
   oldfgd = _gettextcolor();
   oldbgd = _getbkcolor();
   oldpos = _gettextposition();
   _clearscreen( _GCLEARSCREEN );
   /* First time no blink, second time blinking. */
   for( blink = 0; blink <= 16; blink += 16 )
   {
      /* Loop through 8 background colors. */
      for( bgd = 0; bgd < 8; bgd++ )
      {
         _setbkcolor( bgd );
         _settextposition( (short)bgd + ((blink / 16) * 9) + 3, 1 );
         _settextcolor( 7 );
         sprintf(buffer, "Back: %d Fore:", bgd );
         _outtext( buffer );
         /* Loop through 16 foreground colors. */
         for( fgd = 0; fgd < 16; fgd++ )
         {
            _settextcolor( fgd + blink );
            sprintf( buffer, " %2d ", fgd + blink );
            _outtext( buffer );
         }
      }
   }
   _getch();
   /* Restore original foreground, background, and text position. */
   _settextcolor( oldfgd );
   _setbkcolor( oldbgd );
   _clearscreen( _GCLEARSCREEN );
   _settextposition( oldpos.row, oldpos.col );
}
```

perror

#include <stdio.h> Required only for function declarations

Syntax void perror(const char **string*);

Parameter	**Description**
string	String message to print

The perror function prints an error message to stderr. The *string* argument is printed first, followed by a colon, then by the system error message for the last library call that produced the error, and finally by a newline character. If *string* is a null pointer or a pointer to a null string, perror prints only the system error message.

The actual error number is stored in the variable errno (defined in ERRNO.H). The system error messages are accessed through the variable sys_errlist, which is an array of messages ordered by error number. The perror function prints the appropriate error message by using the errno value as an index to _sys_errlist. The value of the variable _sys_nerr is defined as the maximum number of elements in the _sys_errlist array.

To produce accurate results, perror should be called immediately after a library routine returns with an error. Otherwise, the errno value may be overwritten by subsequent calls.

Under MS-DOS, some of the errno values listed in ERRNO.H are not used. These additional errno values are reserved for UNIX use. See "_doserrno, errno, _sys_errlist, _sys_nerr" for a list of errno values used in MS-DOS and the corresponding error messages. The perror function prints an empty string for any errno value not used under the operating system.

Return Value

None.

clearerr
ferror
strerror

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN

```
/* PERROR.C: This program attempts to open a file named
 * NOSUCHF.ILE. Because this file probably doesn't exist,
 * an error message is displayed. The same message is
 * created using perror, strerror, and _strerror.
 */
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
void main( void )
{
   int  fh;
   if( (fh = _open( "NOSUCHF.ILE", _O_RDONLY )) == -1 )
   {
      /* Three ways to create error message: */
      perror( "perror says open failed" );
      printf( "strerror says open failed: %s\n", strerror( errno ) );
      printf( _strerror( "_strerror says open failed" ) );
   }
   else
   {
      printf( "open succeeded on input file\n" );
      _close( fh );
   }
}
```

_pg_analyzechart Functions

#include <pgchart.h>

Syntax short __far _pg_analyzechart(_chartenv __far **env*, const char __far * const __far **categories*, const float __far **values*, short *n*);

short __far _pg_analyzechartms(_chartenv __far **env*, const char __far * const __far **categories*, const float __far **values*, short *nseries*, short *n*, short *arraydim*, const char __far * const __far **serieslabels*);

Parameter	Description
env	Chart environment variable
categories	Array of category variables
values	Array of data values
nseries	Number of series to chart
n	Number of data values to chart
arraydim	Row dimension of data array
serieslabels	Array of labels for series

The _pg_analyzechart routines analyze a single or multiple series of data without actually displaying the presentation-graphic image.

The _pg_analyzechart function fills the chart environment with default values for a single-series bar, column, or line chart, depending on the type specified by the call to the _pg_defaultchart function. The variables calculated by _pg_analyzechart reflect the data given in the arguments *categories* and *values*. All arguments are the same as those used with the _pg_chart function.

The _pg_analyzechartms function fills the chart environment with default values for a multiseries bar, column, or line chart, depending on which type is specified in the _pg_defaultchart function. The variables calculated by _pg_analyzechartms reflect the data given in the arguments *categories* and *values*. All arguments are the same as those used with the _pg_chartms function.

Boolean flags in the chart environment, such as AUTOSCALE and LEGEND, should be set to TRUE before calling either _pg_analyzechart function. This will ensure that the function will calculate all defaults.

Return Value

The _pg_analyzechart and _pg_analyzechartms functions return 0 if there are no errors. A nonzero value indicates a failure.

_pg_chart functions
_pg_defaultchart
_pg_initchart

Standards: None
16-Bit: MS-DOS

```
/* PGACHART.C: This example illustrates presentation-
 * graphics analyze functions. The example uses:
 *      _pg_analyzechartms
 * The same principles apply for
 *      _pg_analyzepie        _pg_analyzechart
 *      _pg_analyzescatter    _pg_analyzescatterms
 */
#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include <graph.h>
#include <pgchart.h>
#define FALSE  0
#define TRUE   1
/* Note data declared as a single-dimension array. The
 * multiseries chart functions expect only one dimension.
 * See _pg_chartms example for an alternate method using
 * multidimension array.
 */
#define TEAMS  4
#define MONTHS 3
float __far values[TEAMS * MONTHS] = { .435f,   .522f,   .671f,
                                       .533f,   .431f,   .590f,
                                       .723f,   .624f,   .488f,
                                       .329f,   .226f,   .401f   };
char __far *months[MONTHS] =         { "May",  "June", "July" };
char __far *teams[TEAMS] = { "Reds", "Sox", "Cubs", "Mets" };
void main( void )
{
   _chartenv env;
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   _pg_initchart();                    /* Initialize chart system.   */
   /* Default multiseries bar chart */
   _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );
    strcpy( env.maintitle.title, "Little League Records - Default" );
   _pg_chartms( &env, months, values, TEAMS, MONTHS, MONTHS, teams );
   _getch();
   _clearscreen( _GCLEARSCREEN );
   /* Analyze multiseries bar chart with autoscale. This sets all
    * default scale values. We want y axis values to be automatic.
    */
   _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );
   strcpy( env.maintitle.title, "Little League Records - Customized" );
   env.xaxis.autoscale = TRUE;
   _pg_analyzechartms( &env, months, values, TEAMS, MONTHS, MONTHS, teams  );
   /* Now customize some of the x axis values. Then draw the chart.  */
   env.xaxis.autoscale = FALSE;
   env.xaxis.scalemax = 1.0f;        /* Make scale show 0.0 to 1.0.    */
   env.xaxis.ticinterval = 0.2f;     /* Don't make scale too crowded.  */
   env.xaxis.ticdecimals = 3;        /* Show three decimals.           */
   strcpy( env.xaxis.scaletitle.title, "Win/Loss Percentage" );
   _pg_chartms( &env, months, values, TEAMS, MONTHS, MONTHS, teams );
   _getch();
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}
```

_pg_analyzepie

#include <pgchart.h>

Syntax short __far _pg_analyzepie(_chartenv __far **env**, const char __far * const __far ***categories***, const float __far ***values***, const short __far ***explode***, short *n*);

Parameter	Description
env	Chart environment variable
categories	Array of category variables
values	Array of data values
explode	Array of explode flags
n	Number of data values to chart

The _pg_analyzepie function analyzes a single series of data without actually displaying the graphic image.

The _pg_analyzepie function fills the chart environment for a pie chart using the data contained in the array *values*. All arguments are the same as those used in the _pg_chartpie function.

Return Value

The _pg_analyzepie function returns 0 if there are no errors. A nonzero value indicates a failure.

_pg_chartpie
_pg_defaultchart
_pg_initchart

Standards: None

16-Bit: MS-DOS

_pg_analyzescatter Functions

#include <pgchart.h>

Syntax short __far _pg_analyzescatter(_chartenv __far *_env_, float __far *_xvalues_, float __far *_yvalues_, short _n_);

short __far _pg_analyzescatterms(_chartenv __far *_env_, const float __far *_xvalues_, const float __far *_yvalues_, short _nseries_, short _n_, short _rowdim_, const char __far * const __far *_serieslabels_);

Parameter	Description
env	Chart environment structure
xvalues	Array of *x*-axis data values
yvalues	Array of *y*-axis data values
n	Number of data values to chart
nseries	Number of series to chart
rowdim	Row dimension of data array
serieslabels	Array of labels for series

The _pg_analyzescatter set of routines analyzes a single or multiple series of data without actually displaying the graphic image.

The _pg_analyzescatter function fills the chart environment for a single-series scatter diagram. The variables calculated by this function reflect the data given in the arguments *xvalues* and *yvalues*. All arguments are the same as those used in the _pg_chartscatter function.

The _pg_analyzescatterms function fills the chart environment for a multiseries scatter diagram. The variables calculated by _pg_analyzescatterms reflect the data given in the arguments *xvalues* and *yvalues*. All arguments are the same as those used in the _pg_chartscatterms function.

Boolean flags in the chart environment, such as AUTOSCALE and LEGEND, should be set to TRUE before calling _pg_analyzescatterms; this ensures that the function will calculate all defaults.

Return Value

The _pg_analyzescatter and _pg_analyzescatterms functions return 0 if there are no errors. A nonzero value indicates a failure.

_pg_chartscatter functions
_pg_defaultchart
_pg_initchart

Standards: None

16-Bit: MS-DOS

_pg_chart Functions

#include <pgchart.h>

Syntax short __far _pg_chart(_chartenv __far *_env_, const char __far * const __far *_categories_, const float __far *_values_, short _n_);

short __far _pg_chartms(_chartenv __far *_env_, const char __far * const __far *_categories_, const float __far *_values_, short _nseries_, short _n_, short _arraydim_, const char __far * const __far *_serieslabels_);

Parameter	Description
env	Chart environment variable
categories	Array of category variables
values	Array of data values
n	Number of data values to chart
nseries	Number of series to chart
arraydim	Row dimension of data array
serieslabels	Array of labels for series

The _pg_chart function displays a single-series bar, column, or line chart, depending on the type specified in the chart environment variable (*env*).

The _pg_chartms function displays a multiseries bar, column, or line chart, depending on the type specified in the chart environment. All the series must contain the same number of data points, specified by the argument *n*.

The array *values* is a two-dimensional array containing all value data for every series to be plotted on the chart. Each column of *values* represents a single series. The parameter *arraydim* is the integer value used to dimension rows in the array declaration for *values*.

For example, the following code fragment declares the identifier `values` to be a two-dimensional floating-point array with 20 rows and 10 columns:

```
#define ARRAYDIM 20
float values [ARRAYDIM][10];
short rowdim = ARRAYDIM;
```

Note that the number of columns in the *values* array cannot exceed 10, the maximum number of data series on a single chart. Note also that `rowdim` must be greater than or equal to the argument *n*, and the column dimension in the array declaration must be greater than or equal to the argument *nseries*. If *n* and *nseries* are set to values less than the full dimensional size of the *values* array, only part of the data contained in *values* will be plotted.

The array *serieslabels* holds the labels used in the chart legend to identify each series.

Return Value

The _pg_chart and _pg_chartms functions return 0 if there are no errors. A nonzero value indicates a

failure.

_pg_analyzechart functions
_pg_defaultchart
_pg_initchart

Standards: None
16-Bit: MS-DOS

```
/* PGCHART.C: This example illustrates presentation-graphics
 * support routines and single-series chart routines, including
 *  _pg_initchart  _pg_defaultchart  _pg_chart  _pg_chartpie
 */
#include <conio.h>
#include <graph.h>
#include <string.h>
#include <stdlib.h>
#include <pgchart.h>
#define COUNTRIES 5
float __far value[COUNTRIES] =    { 42.5f,   14.3f,   35.2f,  21.3f,   32.6f   };
char __far *category[COUNTRIES] = { "Cuba",  "France","USA",  "UK",    "Other" };
short __far explode[COUNTRIES] =  { 0,       1,        0,      1,      0       };
void main( void )
{
   _chartenv env;
   short mode = _VRES16COLOR;
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   _pg_initchart();                    /* Initialize chart system. */
   /* Single-series bar chart */
   _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );
   strcpy( env.maintitle.title, "Widget Production" );
   _pg_chart( &env, category, value, COUNTRIES );
   _getch();
   _clearscreen( _GCLEARSCREEN );
   /* Single-series column chart */
   _pg_defaultchart( &env, _PG_COLUMNCHART, _PG_PLAINBARS );
   strcpy( env.maintitle.title, "Widget Production" );
   _pg_chart( &env, category, value, COUNTRIES );
   _getch();
   _clearscreen( _GCLEARSCREEN );
   /* Pie chart */
   _pg_defaultchart( &env, _PG_PIECHART, _PG_PERCENT );
   strcpy( env.maintitle.title, "Widget Production" );
   _pg_chartpie( &env, category, value, explode, COUNTRIES );
   _getch();
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}
```

_pg_chartpie

#include <pgchart.h>

Syntax short __far _pg_chartpie(_chartenv __far **env**, const char __far * const __far **categories**, const float __far **values**, const short __far * **explode**, short *n*);

Parameter	Description
env	Chart environment structure
categories	Array of category labels
values	Array of data values
explode	Array of explode flags
n	Number of data values to chart

The _pg_chartpie function displays a pie chart for the data contained in the array *values*. Pie charts are formed from a single series of data; there is no multiseries version of pie charts as there is for other chart types.

The array *explode* must be dimensioned so that its length is greater than or equal to the argument *n*. All entries in *explode* are either 0 or 1. If an entry is 1, the corresponding pie slice is displayed slightly removed from the rest of the pie.

For example, if the *explode* array is initialized as

```
short explode[5] = {0, 1, 0, 0, 0};
```

the pie slice corresponding to the second entry of the *categories* array will be displayed "exploded" from the other four slices.

Return Value

The _pg_chartpie function returns 0 if there are no errors. A nonzero value indicates a failure.

_pg_analyzepie
_pg_defaultchart
_pg_initchart

Standards: None

16-Bit: MS-DOS

_pg_chartscatter Functions

#include <pgchart.h>

Syntax short __far _pg_chartscatter(_chartenv __far *_env_, float __far *_xvalues_, float __far *_yvalues_, short _n_);

short __far _pg_chartscatterms(_chartenv __far *_env_, const float __far *_xvalues_, const float __far *_yvalues_, short _nseries_, short _n_, short _rowdim_, const char __far * const __far *_serieslabels_);

Parameter	Description
env	Chart environment structure
xvalues	Array of *x*-axis data values
yvalues	Array of *y*-axis data values
n	Number of data values to chart
nseries	Number of series to chart
rowdim	Row dimension of data array
serieslabels	Array of labels for series

The _pg_chartscatter function displays a scatter diagram for a single series of data.

The _pg_chartscatterms function displays a scatter diagram for more than one series of data.

The arguments *xvalues* and *yvalues* are two-dimensional arrays containing data for the *x* axis and *y* axis, respectively. Columns for each array hold data for individual series; thus the first columns of *xvalues* and *yvalues* contain plot data for the first series, the second columns contain plot data for the second series, and so forth.

The *n*, *rowdim*, *nseries*, and *serieslabels* arguments fulfill the same purposes as those used in the _pg_chartms function. See _pg_chartms for an explanation of these arguments.

Return Value

The _pg_chartscatter and _pg_chartscatterms functions return 0 if there are no errors. A nonzero value indicates a failure.

_pg_analyzescatter functions
_pg_defaultchart
_pg_initchart

Standards: None

16-Bit: MS-DOS

_pg_defaultchart

#include <pgchart.h>

Syntax short __far _pg_defaultchart(_chartenv __far **env*, short *charttype*, short *chartstyle*);

Parameter	**Description**
env	Chart environment structure
charttype	Chart type
chartstyle	Chart style

The _pg_defaultchart function initializes all necessary variables in the chart environment for the chart type by the variable *charttype*.

All title fields in the environment structure are blanked. Titles should be set in the proper fields after calling _pg_defaultchart.

The *charttype* variable can be set to one of the following manifest constants:

Chart Type	**Description**
_PG_BARCHART	Bar chart
_PG_COLUMNCHART	Column chart
_PG_LINECHART	Line chart
_PG_PIECHART	Pie chart
_PG_SCATTERCHART	Scatter chart

The *chartstyle* variable specifies the style of the chart with either the number "1" or the number "2." Each of the five types of presentation-graphics charts can appear in two different chart styles, as described below:

Chart Type	**Chart Style 1**	**Chart Style 2**
Bar	Side by side	Stacked
Column	Side by side	Stacked
Line	Points with lines	Points only
Pie	Percent	No percent
Scatter	Points with lines	Points only

In a pie chart, the pieces are "exploded" according to the *explode* array argument in the _pg_chartpie function. In the "percent" format, percentages are printed next to each slice. Bar and column charts have only one style when displaying a single series of data. The styles "side by side" and "stacked" are applicable only when more than one series appears on the same chart. The first style arranges the bars or columns for the different series side by side, showing relative heights or lengths. The stacked style emphasizes relative sizes between bars and columns.

Return Value

The _pg_defaultchart function returns 0 if there are no errors. A nonzero value indicates a failure.

_pg_getchardef
_pg_getpalette
_pg_getstyleset
_pg_hlabelchart
_pg_initchart
_pg_resetpalette
_pg_resetstyleset
_pg_setchardef
_pg_setpalette
_pg_setstyleset
_pg_vlabelchart

Standards: None

16-Bit: MS-DOS

_pg_getchardef

#include <pgchart.h>

Syntax short __far _pg_getchardef(short *charnum*, unsigned char __far **chardef*);

Parameter	Description
charnum	ASCII number of character
chardef	Pointer to 8-by-8 bitmap array

The _pg_getchardef function retrieves the current 8-by-8 pixel bitmap for the character having the ASCII number *charnum*. The bitmap is stored in the *chardef* array.

Return Value

The _pg_getchardef function returns 0 if there are no errors. A nonzero value indicates an error.

_pg_defaultchart
_pg_initchart
_pg_setchardef

Standards: None

16-Bit: MS-DOS

_pg_getpalette

#include <pgchart.h>

Syntax short __far _pg_getpalette(_paletteentry __far **palette*);

Parameter	Description
palette	Pointer to first palette structure in array

The _pg_getpalette function retrieves palette colors, line styles, fill patterns, and plot characters for all palettes. The pointer *palette* points to an array of palette structures that will contain the desired palette values.

The palette used by the presentation-graphics routines is independent of the palette used by the low-level graphics routines.

Return Value

The function _pg_getpalette returns 0 if there are no errors, and it returns the value _BADSCREENMODE if current palettes have not been initialized by a previous call to _pg_setpalette.

_pg_defaultchart
_pg_initchart
_pg_resetpalette
_pg_setpalette

Standards: None
16-Bit: MS-DOS

```
/* PGGPAL.C: This example illustrates presentation-graphics
 * palettes and the routines that modify them, including
 *    _pg_getpalette    _pg_resetpalette    _pg_setstyleset
 *    _pg_getstyleset   _pg_resetstyleset   _pg_vlabelchart
 *    _pg_hlabelchart   _pg_setpalette
 */
#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include <graph.h>
#include <pgchart.h>
#define TEAMS  2
#define MONTHS 3
float __far values[TEAMS][MONTHS] =   { { .435f,  .522f,  .671f },
                                        { .533f,  .431f,  .401f } };
char __far *months[MONTHS] =            { "May", "June", "July" };
char __far *teams[TEAMS] = { "Cubs", "Reds" };
_fillmap fill1 = { 0x99, 0x33, 0x66, 0xcc, 0x99, 0x33, 0x66, 0xcc };
_fillmap fill2 = { 0x99, 0xcc, 0x66, 0x33, 0x99, 0xcc, 0x66, 0x33 };
_styleset styles;
_palettetype pal;
void main( void )
{
   _chartenv env;
   short mode = _VRES16COLOR;
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   _pg_initchart();                    /* Initialize chart system.    */
   /* Modify global set of line styles used for borders, grids, and
    * data connectors. Note that this change is used before
    * _pg_defaultchart, which will use the style set.
    */
   _pg_getstyleset( styles );          /* Get styles and modify       */
   styles[1] = 0x5555;                 /*   style 1 (used for         */
   _pg_setstyleset( styles );          /*   borders)-then set new.    */
   _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );
   /* Modify palette for data lines, colors, fill patterns, and
    * characters. Note that the line styles are set in the palette, not
    * in the style set, so that only data connectors will be affected.
    */
   _pg_getpalette( pal );              /* Get default palette.         */
   pal[1].plotchar = 16;               /* Set to ASCII 16 and 17.      */
   pal[2].plotchar = 17;
   memcpy( pal[1].fill, fill1, 8 );    /* Copy fill masks to palette. */
   memcpy( pal[2].fill, fill2, 8 );
   pal[1].color = 3;                   /* Change palette colors.       */
   pal[2].color = 4;
   pal[1].style = 0xfcfc;              /* Change palette line styles. */
   pal[2].style = 0x0303;
   _pg_setpalette( pal );              /* Put modified palette.        */
   /* Multiseries bar chart */
   strcpy( env.maintitle.title, "Little League Records - Customized" );
   _pg_chartms( &env, months, (float __far *)values, TEAMS, MONTHS, MONTHS, teams );
   _getch();
   _clearscreen( _GCLEARSCREEN );
   /* Multiseries line chart */
   _pg_defaultchart( &env, _PG_LINECHART, _PG_POINTANDLINE );
   strcpy( env.maintitle.title, "Little League Records - Customized" );
   _pg_chartms( &env, months, (float __far *)values, TEAMS, MONTHS, MONTHS, teams );
```

```
   /* Print labels. */
   _pg_hlabelchart( &env, (short)(env.chartwindow.x2 * .75),
                          (short)(env.chartwindow.y2 * .10),
                          12, "Up and up!" );
   _pg_vlabelchart( &env, (short)(env.chartwindow.x2 * .75),
                          (short)(env.chartwindow.y2 * .45),
                          13, "Sliding down!" );
   _getch();
   _clearscreen( _GCLEARSCREEN );
   _pg_resetpalette();                /* Restore default palette      */
   _pg_resetstyleset();               /*   and style set.            */
   /* Multiseries bar chart */
   _pg_defaultchart( &env, _PG_BARCHART, _PG_PLAINBARS );
   strcpy( env.maintitle.title, "Little League Records - Default" );
   _pg_chartms( &env, months, (float __far *)values, TEAMS, MONTHS, MONTHS, teams );
   _getch();
   _clearscreen( _GCLEARSCREEN );
   /* Multiseries line chart */
   _pg_defaultchart( &env, _PG_LINECHART, _PG_POINTANDLINE );
   strcpy( env.maintitle.title, "Little League Records - Default" );
   _pg_chartms( &env, months, (float __far *)values, TEAMS, MONTHS, MONTHS, teams );
   _getch();
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}
```

_pg_getstyleset

#include <pgchart.h>

Syntax void __far _pg_getstyleset(unsigned short __far **styleset*)**;**

Parameter	Description
styleset	Pointer to current styleset array

The _pg_getstyleset function retrieves the contents of the current styleset array**.**

Return Value

None**.**

_pg_defaultchart
_pg_initchart
_pg_resetstyleset
_pg_setstyleset

Standards: None

16-Bit: MS-DOS

_pg_hlabelchart

#include <pgchart.h>

Syntax short __far _pg_hlabelchart(_chartenv __far **env*, short *x*, short *y*, short *color*, const char __far **label*);

Parameter	Description
env	Chart environment structure
x	*x*-coordinate for text
y	Pixel *y*-coordinate for text
color	Color code for text
label	Label text

The _pg_hlabelchart function writes text horizontally on the screen. The arguments *x* and *y* are pixel coordinates for the beginning location of text relative to the upper-left corner of the chart window.

Return Value

The _pg_hlabelchart functions return 0 if there are no errors. A nonzero value indicates a failure.

_pg_defaultchart
_pg_initchart
_pg_vlabelchart

Standards: None

16-Bit: MS-DOS

_pg_initchart

#include <pgchart.h>

Syntax short __far _pg_initchart(void);

The _pg_initchart function initializes the presentation-graphics package. It initializes the color and style pools, resets the chartline styleset, builds default palette modes, and reads the presentation-graphics font definition from the disk. This function is required in all programs that use presentation graphics. The _pg_initchart function must be called before any of the other functions in the presentation-graphics library. Also, _pg_initchart must be called after each change of the video mode.

The _pg_initchart function assumes a valid graphics mode has been established. Therefore, it must be called only after a successful call to the library function _setvideomode.

Note The _pg_initchart function can only be called after using the _setvideomode function to establish the video mode. Also, _pg_initchart must be called after each change of the video mode.

Return Value

The _pg_initchart functions return 0 if there are no errors. A nonzero value indicates a failure.

_pg_defaultchart
_pg_getchardef
_pg_getpalette
_pg_getstyleset
_pg_hlabelchart
_pg_resetpalette
_pg_resetstyleset
_pg_setchardef
_pg_setpalette
_pg_setstyleset
_pg_vlabelchart
_setvideomode

Standards: None

16-Bit: MS-DOS

_pg_resetpalette

#include <pgchart.h>

Syntax short __far _pg_resetpalette(void);

The _pg_resetpalette function sets the palette colors, line styles, fill patterns, and plot characters for the palette to the default for the current screen mode.

The palette used by the presentation-graphics routines is independent of the palette used by the low-level graphics routines.

Return Value

The _pg_resetpalette function returns 0 if there are no errors. If the screen mode is not valid, the value _BADSCREENMODE is returned.

_pg_defaultchart
_pg_getpalette
_pg_initchart
_pg_setpalette

Standards: None

16-Bit: MS-DOS

_pg_resetstyleset

#include <pgchart.h>

Syntax void __far _pg_resetstyleset(void);

The _pg_resetstyleset function reinitializes the styleset to the default values for the current screen mode.

Return Value

None.

_pg_defaultchart
_pg_getstyleset
_pg_initchart
_pg_setstyleset

Standards: None

16-Bit: MS-DOS

_pg_setchardef

#include <pgchart.h>

Syntax short __far _pg_setchardef(short *charnum*, unsigned char __far **chardef*);

Parameter	Description
charnum	ASCII number of character
chardef	Pointer to an 8-by-8 bitmap array for the character

The _pg_setchardef function sets the 8-by-8 pixel bitmap for the character with the ASCII number *charnum*. The bitmap is stored in the *chardef* array.

Return Value

The _pg_setchardef function returns 0 if there is no error. A nonzero value indicates an error.

_pg_defaultchart
_pg_getchardef
_pg_initchart

Standards: None

16-Bit: MS-DOS

_pg_setpalette

#include <pgchart.h>

Syntax short __far _pg_setpalette(_paletteentry __far **palette*);

Parameter	Description
palette	Pointer to first palette structure in array

The _pg_setpalette function sets palette colors, line styles, fill patterns, and plot characters for all palettes. The pointer *palette* points to an array of palette structures that contain the desired palette values.

The palette used by the presentation-graphics routines is independent of the palette used by the low-level graphics routines.

Return Value

The _pg_setpalette function returns 0 if there are no errors. If the new palettes are not valid, the value _BADSCREENMODE is returned.

_pg_defaultchart
_pg_getpalette
_pg_initchart
_pg_resetpalette

Standards: None
16-Bit: MS-DOS

_pg_setstyleset

#include <pgchart.h>

Syntax void __far _pg_setstyleset(unsigned short __far **styleset*);

Parameter	**Description**
styleset	Pointer to new styleset

The _pg_setstyleset function sets the current styleset.

Return Value

None.

_pg_defaultchart
_pg_getstyleset
_pg_initchart
_pg_resetstyleset

Standards: None

16-Bit: MS-DOS

_pg_vlabelchart

#include <pgchart.h>

Syntax short __far _pg_vlabelchart(_chartenv __far **env*, short *x*, short *y*, short *color*, const char __far **label*);

Parameter	Description
env	Chart environment structure
x	Pixel *x* coordinate for text
y	Pixel *y* coordinate for text
color	Color code for text
label	Label text

The _pg_vlabelchart function writes text vertically on the screen. The arguments *x* and *y* are pixel coordinates for the beginning location of text relative to the upper-left corner of the chart window.

Return Value

The _pg_vlabelchart function returns 0 if there are no errors. A nonzero value indicates a failure.

_pg_defaultchart
_pg_hlabelchart
_pg_initchart

Standards:	None
16-Bit:	MS-DOS

_pie Functions

#include <graph.h>

Syntax short __far _pie(short *control*, short *x1*, short *y1*, short *x2*, short *y2*, short *x3*, short *y3*, short *x4*, short *y4*);

short __far _pie_w(short *control*, double *x1*, double *y1*, double *x2*, double *y2*, double *x3*, double *y3*, double *x4*, double *y4*);

short __far _pie_wxy(short *control*, struct _wxycoord __far **pwxy1*, struct _wxycoord __far **pwxy2*, struct _wxycoord __far **pwxy3*, struct _wxycoord __far**pwxy4*);

Parameter	Description
control	Fill-control constant
x1, *y1*	Upper-left corner of bounding rectangle
x2, *y2*	Lower-right corner of bounding rectangle
x3, *y3*	Second point of start vector (center of bounding rectangle is first point)
x4, *y4*	Second point of end vector (center of bounding rectangle is first point)
pwxy1	Upper-left corner of bounding rectangle
pwxy2	Lower-right corner of bounding rectangle
pwxy3	Second point of start vector (center of bounding rectangle is first point)
pwxy4	Second point of end vector (center of bounding rectangle is first point)

The _pie functions draw a pie-shaped wedge by drawing an elliptical arc whose center and two endpoints are joined by lines.

The center of the pie is the center of the bounding rectangle, which is defined by points (*x1*, *y1*) and (*x2*, *y2*) for _pie and _pie_w and by points *pwxy1* and *pwxy2* for _pie_wxy. The pie starts where it intersects an imaginary line extending from the center of the arc through (*x3*, *y3*) for _pie and _pie_w and through *pwxy3* for _pie_wxy. It is drawn counterclockwise about the center of the arc, ending where it intersects an imaginary line extending from the center of the arc through (*x4*, *y4*) for _pie and _pie_w and through *pwxy4* for _pie_wxy.

The _pie routine uses the view coordinate system. The _pie_w and _pie_wxy functions use the real-valued window coordinate system. The arc is drawn using the current color. Because an arc does not define a closed area, it is not filled.

The _wxycoord structure is defined in GRAPH.H and contains the following elements:

Element	Description
double wx	Window *x* coordinate
double wy	Window *y* coordinate

The wedge is drawn using the current color moving in a counterclockwise direction. The *control* parameter can be one of the following manifest constants:

Constant	Action
_GFILLINTERIOR	Fills the figure using the current color and fill mask
_GBORDER	Does not fill the figure

The control option given by _GFILLINTERIOR is equivalent to a subsequent call to the _floodfill function using the approximate center of the pie as the starting point and the current color (set by _setcolor) as the boundary color. Use the _getarcinfo function to find the exact starting point.

Return Value

These functions return a nonzero value if successful; otherwise, they return 0.

_arc functions
_ellipse functions
_floodfill
_getarcinfo
_getcolor
_lineto functions
_rectangle functions
_setcolor
_setfillmask

Standards: None
16-Bit: MS-DOS, QWIN

```
/* PIE.C: This program draws a pie-shaped figure. */
#include <stdlib.h>
#include <conio.h>
#include <graph.h>
void main( void )
{
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   _pie( _GBORDER, 80, 50, 240, 150, 240, 12, 0, 150 );
   _getch();
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}
```

_polygon Functions

#include <graph.h>

Syntax short __far _polygon(short *control*, const struct _xycoord __far * *points*, short *numpoints*);

short __far _polygon_w(short *control*, const double __far **points*, short *numpoints*);

short __far _polygon_wxy(short *control*, const struct _wxycoord __far **points*, short *numpoints*);

Parameter	Description
control	Fill flag
points	Pointer to an array of structures or doubles defining the polygon
numpoints	Number of points

The _polygon functions draw polygons. The border of the polygon is drawn in the current color and line style. The _polygon routine uses the view coordinate system (expressed in _xycoord structures), and the _polygon_wxy and _polygon_w routines use real-valued window coordinates (expressed in _wxycoord structures and in pairs of double-precision floating-point values, respectively).

The argument *points* is an array of _xycoord or _wxycoord structures or pairs of doubles, each of which specifies one of the polygon's vertices. (For _polygon_w, *points*[0] and *points*[1] specify the *x* and *y* coordinates, respectively, of the first point.) If the first point does not equal the last point, the _polygon functions use them to provide a closing edge.

The argument *numpoints* indicates the number of elements (the number of vertices) in the *points* array. The minimum number of points is 3; the maximum is 16,381.

The *control* argument can be one of the following manifest constants:

Constant	Action
_GFILLINTERIOR	Fills the polygon with the current fill mask using a scan fill
_GBORDER	Does not fill the polygon

The _setwritemode, _setlinestyle, and _setfillmask functions all affect the output from the_polygon functions.

If you try to fill the polgon with the _floodfill function, the polygon must be bordered by a solid line-style pattern.

Return Value

The _polygon functions return a nonzero value if the arc is successfully drawn; otherwise, they return 0.

_ellipse functions
_floodfill
_lineto functions
_pie functions
_rectangle functions
_setcolor
_setfillmask
_setlinestyle
_setwritemode

Standards: None
16-Bit: MS-DOS, QWIN

```
/* POLYGON.C: This program draws a star-shaped polygon. */
#include <conio.h>
#include <stdlib.h>
#include <graph.h>
#include <math.h>
#include <stdlib.h>
#define PI 3.1415
void main( void )
{
   short  side, radius = 90, x = 0, y = 0;
   double radians;
   struct _xycoord polyside[5];
   struct _videoconfig vc;
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   _getvideoconfig( &vc );
   _setvieworg( (short)(vc.numxpixels / 2), (short)(vc. numypixels / 2) );
   /* Calculate points of star every 144 degrees, then connect them. */
   for( side = 0;side < 5;side++ )
   {
      radians = 144 * PI / 180;
      polyside[side].xcoord = x + (short)(cos( side * radians ) * radius);
      polyside[side].ycoord = y + (short)(sin( side * radians ) * radius);
   }
   _polygon( _GFILLINTERIOR, polyside, 5 );
   _getch();
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}
```

pow Functions

#include <math.h>

Syntax double pow(double *x*, double *y*);

long double powl(long double *x*, long double *y*);

Parameter	Description
x	Number to be raised
y	Power of *x*

The pow and powl functions compute *x* raised to the power of *y*.

The powl function is the 80-bit counterpart, and it uses an 80-bit, 10-byte coprocessor form of arguments and return values. See the long double functions for more details on this data type.

Return Value

The pow and powl functions return the value of *xy*. If *x* is not 0.0 and *y* is 0.0, pow and powl return the value 1. If *x* is 0.0 and *y* is negative, pow and powl set errno to EDOM and return 0.0. If both *x* and *y* are 0.0, or if *x* is negative and *y* is not an integer, the function prints a _DOMAIN error message to stderr, sets errno to EDOM, and returns 0.0. If an overflow results, the function sets errno to ERANGE and returns HUGE_VAL. No message is printed on overflow or underflow.

The pow function does not recognize integral floating-point values greater than 2 raised to the power of 64, such as `1.0E100`.

exp
log functions
sqrt

pow

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

powl

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* POW.C */
#include <math.h>
#include <stdio.h>
void main( void )
{
   double x = 2.0, y = 3.0, z;
   z = pow( x, y );
   printf( "%.1f to the power of %.1f is %.1f\n", x, y, z );
}
```

printf

#include <stdio.h>

Syntax int printf(const char **format* [, *argument*]**...**);

Parameter	**Description**
format	Format control
argument	Optional arguments

The printf function formats and prints a series of characters and values to the standard output stream, stdout. The *format* argument consists of ordinary characters, escape sequences, and (if arguments follow *format*) format specifications. The ordinary characters and escape sequences are copied to stdout in order of their appearance. For example, the line

```
printf("Line one\n\t\tLine two\n");
```

produces the output

```
Line one
        Line two
```

If arguments follow the *format* string, the *format* string must contain specifications that determine the output format for the arguments.

Format specifications always begin with a percent sign (%) and are read left to right. When the first format specification (if any) is encountered, the value of the first argument after *format* is converted and output accordingly. The second format specification causes the second argument to be converted and output, and so on. If there are more arguments than there are format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for all the format specifications.

Return Value

The printf function returns the number of characters printed, or a negative value in the case of an error.

fprintf
scanf
sprintf
vfprintf
vprintf
vsprintf
printf Format Specifications
printf *size* and *distance* Specification
printf *type* Characters
printf *width* Specification
printf *precision* Specification
printf *flag* Directives

printf
printf *size* and *distance* Specification
printf *type* Characters
printf *width* Specification
printf *precision* Specification
printf *flag* Directives

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN

Format Specification Fields

A format specification, which consists of optional and required fields, has the following form:

%[*flags*] [*width*] [.*precision*] [{F | N | h | l | L}]*type*

Each field of the format specification is a single character or a number signifying a particular format option. The simplest format specification contains only the percent sign and a *type* character (for example, %s). The optional fields, which appear before the *type* character, control other aspects of the formatting. The fields in a printf format specification are described in the following list:

Field	Description
type	Required character that determines whether the associated argument is interpreted as a character, a string, or a number. (See *type* Field Characters.)
flags	Optional character or characters that control justification of output and printing of signs, blanks, decimal points, and octal and hexadecimal prefixes. (See *flag* Directives.) More than one flag can appear in a format specification.
width	Optional number that specifies minimum number of characters output. (See *width* Specification.)
precision	Optional number that specifies maximum number of characters printed for all or part of the output field, or minimum number of digits printed for integer values. (See *precision* Specification.)
F, N	Optional prefixes that refer to the "distance" to the object being printed (near or far). F and N are not part of the ANSI definition for printf. They are Microsoft extensions that should not be used if ANSI portability is desired.
h, l, L	Optional prefixes that determine the size of the argument expected, as shown below:

Prefix	Use
h	Used with the integer types d, i, o, x, and X to

	specify that the argument is short int, or with u to specify short unsigned int. If used with %p, it indicates a 16-bit pointer.
l	Used with d, i, o, x, and X types to specify that the argument is long int, or with u to specify long unsigned int; also used with e, E, f, g, and G types to specify double rather than float. If used with %p, it indicates a 32-bit pointer.
L	Used with e, E, f, g, and G types to specify long double.

If a percent sign is followed by a character that has no meaning as a format field, the character is copied to stdout. For example, to print a percent-sign character, use %%.

Type Field Characters

%[*flags*] [*width*] [**.***precision*] [{F | N | h | l | L}]*type*

The *type* character is the only required format field for the printf function; it appears after any optional format fields. The *type* character determines whether the associated argument is interpreted as a character, string, or number (as shown below).

Character	Type	Output Format
d	int	Signed decimal integer.
i	int	Signed decimal integer.
u	int	Unsigned decimal integer.
o	int	Unsigned octal integer.
x	int	Unsigned hexadecimal integer, using "abcdef."
X	int	Unsigned hexadecimal integer, using "ABCDEF."
f	double	Signed value having the form [−]*dddd.dddd*, where *dddd* is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.
e	double	Signed value having the form [−]*d.dddd* e [*sign*]*ddd*, where *d* is a single decimal digit, *dddd* is one or more decimal digits, *ddd* is exactly three decimal digits, and *sign* is + or −.
E	double	Identical to the e format, except that E, rather than e, introduces the exponent.

g	double	Signed value printed in f or e format, whichever is more compact for the given value and precision. The e format is used only when the exponent of the value is less than –4 or greater than or equal to the *precision* argument. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.
G	double	Identical to the g format, except that G, rather than g, introduces the exponent (where appropriate).
c	int	Single character.
s	String	Characters printed up to the first null character ('\0') or until the *precision* value is reached.
n	Pointer to integer	Number of characters successfully written so far to the stream or buffer; this value is stored in the integer whose address is given as the argument.
p	Far pointer to void	Prints the address pointed to by the argument in the form *xxxx*:*yyyy*, where *xxxx* is the segment and *yyyy* is the offset, and the digits *x* and *y* are uppercase hexadecimal digits; %hp indicates a near pointer and prints only the offset of the address.

Flag Directives

%[*flags*] [*width*] [.*precision*] [{F | N | h | l | L}]*type*

The first optional field of the format specification is *flag*. A flag directive is a character that justifies output and prints signs, blanks, decimal points, and octal and hexadecimal prefixes. More than one flag directive may appear in a format specification.

Flag	Meaning	Default
-	Left justify the result within the given field width.	Right justify.
+	Prefix the output value with a sign (+ or -) if the output value is of a signed type.	Sign appears only for negative signed values (-).
0	If *width* is prefixed with 0, zeros are added until the minimum width is reached. If 0 and - appear, the 0 is ignored. If 0 is specified with an integer format (i, u, x, X, o, d) the 0 is ignored.	No padding.
blank (' ')	Prefix the output value with a blank if the output value is signed and positive; the blank is ignored if both the blank and + flags appear.	No blank appears.
#	When used with the o, x, or X format, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.	No blank appears.
	When used with the e, E, or f format, the # flag forces the output value to contain a decimal point in all cases. Decimal point appears only if digits follow it.	
	When used with the g or G format, the # flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros. Decimal point appears only if digits	

follow it. Trailing zeros are truncated.

Ignored when used with c, d, i, u, or s.

Width Specification

%[*flags*] [*width*] [.*precision*] [{F | N | h | l | L}]*type*

The second optional field of the format specification is the width specification. The *width* argument is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified width, blanks are added to the left or the right of the values_depending on whether the - flag (for left justification) is specified

_until the minimum width is reached. If *width* is prefixed with 0, zeros are added until the minimum width is reached (not useful for left-justified numbers).

The width specification never causes a value to be truncated. If the number of characters in the output value is greater than the specified width, or *width* is not given, all characters of the value are printed (subject to the precision specification).

The width specification may be an asterisk (*), in which case an int argument from the argument list supplies the value. The *width* argument must precede the value being formatted in the argument list. A nonexistent or small field width does not cause a truncation of a field; if the result of a conversion is wider than the field width, the field expands to contain the conversion result.

Precision Specification

%[*flags*] [*width*] [.*precision*] [{F | N | h | l | L}]*type*

The third optional field of the format specification is the precision specification. It specifies a nonnegative decimal integer, preceded by a period (.), which specifies the number of characters to be printed, the number of decimal places, or the number of significant digits. (See the table below.) Unlike the width specification, the precision specification can cause truncation of the output value, or rounding in the case of a floating-point value. If *precision* is specified as zero and the value to be converted is zero, the result is no characters output, as shown below:

```
printf( "%.0d", 0 );      /* No characters output */
```

The precision specification may be an asterisk (*), in which case an int argument from the argument list supplies the value. The *precision* argument must precede the value being formatted in the argument list.

The interpretation of the precision value and the default when *precision* is omitted depend on the type, as shown below.

Type	Meaning	Default
d i u o x X	The precision specifies the minimum number of digits to be printed. If the number of digits in the argument is less than *precision*, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds *precision*.	Default precision is 1.
e E	The precision specifies the number of digits to be printed after the decimal point. The last printed digit is rounded.	Default precision is 6; if *precision* is 0 or the period (.) appears without a number following it, no decimal point is printed.
f	The precision value specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.	Default precision is 6; if *precision* is 0, or if the period (.) appears without a number following it, no decimal point is printed.

g G	The precision specifies the maximum number of significant digits printed.	Six significant digits are printed, with any trailing zeros truncated.
c	The precision has no effect.	Character is printed.
s	The precision specifies the maximum number of characters to be printed. Characters in excess of *precision* are not printed.	Characters are printed until a null character is encountered.

If the argument corresponding to a floating-point specifier is infinite, indefinite, or not a number (NAN), the printf function gives the following output:

Value	Output
+ infinity	1.#INF*random-digits*
– infinity	–1.#INF*random-digits*
Indefinite	*digit*.#IND*random-digits*
NAN	*digit*.#NAN*random-digits*

Size and Distance Specification

%[*flags*] [*width*] [.*precision*] [{F | N | h | l | L}]*type*

For printf, the format specification fields F and N refer to the "distance" to the object being read (near or far), and h and l refer to the "size" of the object being read (16-bit short or 32-bit long). The following list clarifies this use of F, N, h, l, and L:

Program Code	Action
printf ("%Ns");	Print near string
printf ("%Fs");	Print far string
printf ("%Nn");	Store char count in near int
printf ("%Fn");	Store char count in far int
printf ("%hp");	Print a 16-bit pointer (*xxxx*)
printf ("%lp");	Print a 32-bit pointer (*xxxx*:*xxxx*)
printf ("%Nhn");	Store char count in near short int
printf ("%Nln");	Store char count in near long int
printf ("%Fhn");	Store char count in far short int
printf ("%Fln");	Store char count in far int

The specifications "%hs" and "%ls" are meaningless to printf. The specifications "%Np" and "%Fp" are aliases for "%hp" and "%lp" for the sake of compatibility with Microsoft C version 4.0.

```
/* PRINTF.C illustrates output formatting with printf. */
#include <stdio.h>
void main( void )
{
   char   ch = 'h', *string = "computer";
   int    count = -9234;
   double fp = 251.7366;
   /* Display integers. */
   printf( "Integer formats:\n"
           "\tDecimal: %d  Justified: %.6d  Unsigned: %u\n",
           count, count, count, count );
   printf( "Decimal %d as:\n\tHex: %Xh  C hex: 0x%x  Octal: %o\n",
            count, count, count, count );
   /* Display in different radixes. */
   printf( "Digits 10 equal:\n\tHex: %i  Octal: %i  Decimal: %i\n",
            0x10, 010, 10 );
   /* Display characters. */
   printf( "Characters in field:\n%10c   %5c\n", ch, ch );
   /* Display strings. */
   printf( "Strings in field:\n%25s\n%25.4s\n", string, string );
   /* Display real numbers. */
   printf( "Real numbers:\n\t%f    %.2f    %e    %E\n", fp, fp, fp, fp );
   /* Display pointers. */
   printf( "Address as:\n\tDefault: %p  Near: %Np  Far: %Fp\n",
            &count, (int __near *)&count, (int __far *)&count );
   /* Count characters printed. */
   printf( "Display to here:\n" );
   printf( "1234567890123456%n78901234567890\n", &count );
   printf( "\tNumber displayed: %d\n\n", count );
}
```

putc, putchar

#include <stdio.h>

Syntax int putc(int *c*, FILE **stream*);

int putchar(int *c*);

Parameter	Description
c	Character to be written
stream	Pointer to FILE structure

The putc routine writes the single character *c* to the output stream at the current position. The putchar routine is identical to putc(*c, stdout)*.

These routines are implemented as both macros and functions. See Choosing Between Functions and Macros for a discussion of how to select between the macro and function forms.

Return Value

The putc and putchar routines return the character written, or EOF in the case of an error. Any integer can be passed to putc, but only the lower 8 bits are written.

putc

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

putchar

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN

fputc
_fputchar
getc
getchar

```
/* PUTC.C: This program uses putc to write buffer
 * to a stream. If an error occurs, the program
 * will stop before writing the entire buffer.
 */
#include <stdio.h>
void main( void )
{
   FILE *stream;
   char *p, buffer[] = "This is the line of output\n";
   int  ch;
   /* Make standard out the stream and write to it. */
   stream = stdout;
   for( p = buffer; (ch != EOF) && (*p != '\0'); p++ )
      ch = putc( *p, stream );
}
```

_putch

#include <conio.h> Required only for function declarations

Syntax int _putch(int *c*);

Parameter	**Description**
c	Character to be output

The _putch function writes the character *c* directly (without buffering) to the console.

Return Value

The function returns *c* if successful, and EOF if not.

Standards: None
16-Bit: MS-DOS

_cprintf
_getch
_getche

```
/* GETCH.C: This program reads characters from
 * the keyboard until it receives a 'Y' or 'y'.
 */
#include <conio.h>
#include <ctype.h>
void main( void )
{
   int ch;
   _cputs( "Type 'Y' when finished typing keys: " );
   do
   {
      ch = _getch();
      ch = toupper( ch );
   } while( ch != 'Y' );
   _putch( ch );
   _putch( '\r' );    /* Carriage return */
   _putch( '\n' );    /* Line feed       */
}
```

_putenv

#include <stdlib.h> Required only for function declarations

Syntax int _putenv(const char *_envstring_);

Parameter	**Description**
envstring	Environment-string definition

The _putenv function adds new environment variables or modifies the values of existing environment variables. Environment variables define the environment in which a process executes (for example, the default search path for libraries to be linked with a program).

The *envstring* argument must be a pointer to a string with the form

varname=string

where *varname* is the name of the environment variable to be added or modified and *string* is the variable's value. If *varname* is already part of the environment, its value is replaced by *string*; otherwise, the new *varname* variable and its *string* value are added to the environment. A variable can be removed from the environment by specifying an empty *string*—that is, by specifying only *varname*=.

This function affects only the environment that is local to the currently running process; it cannot be used to modify the command-level environment. When the currently running process terminates, the environment reverts to the level of the parent process (in most cases, the operating system level). However, the environment affected by _putenv can be passed to any child processes created by _spawn, _exec, or system, and these child processes get any new items added by _putenv.

Never free a pointer to an environment entry, because the environment variable will then point to freed space. A similar problem can occur if you pass _putenv a pointer to a local variable, then exit the function in which the variable is declared.

The _putenv function operates only on data structures accessible to the run-time library and not on the environment "segment" created for a process by the operating system.

Note that environment-table entries must not be changed directly. If an entry must be changed, use _putenv. To modify the returned value without affecting the environment table, use _strdup or strcpy to make a copy of the string.

The getenv and _putenv functions use the global variable _environ to access the environment table. The _putenv function may change the value of environ, thus invalidating the *envp* argument to the main function. Therefore, it is safer to use the _environ variable to access the environment information.

Return Value

The _putenv function returns 0 if it is successful. A return value of -1 indicates an error.

Use _putenv for compatibility with ANSI naming conventions of non-ANSI functions. Use putenv and link with OLDNAMES.LIB for UNIX compatibility.

Standards:	UNIX
16-Bit:	MS-DOS, QWIN, WIN, WIN DLL

getenv
_searchenv

```
/* GETENV.C: This program uses getenv to retrieve
 * the LIB environment variable and then uses
 * _putenv to change it to a new value.
 */
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   char *libvar;
   /* Get the value of the LIB environment variable. */
   libvar = getenv( "LIB" );
   if( libvar != NULL )
      printf( "Original LIB variable is: %s\n", libvar );
   /* Attempt to change path. Note that this only affects the environment
    * variable of the current process. The command processor's environment
    * is not changed.
    */
   _putenv( "LIB=c:\\mylib;c:\\yourlib" );
   /* Get new value. */
   libvar = getenv( "LIB" );
   if( libvar != NULL )
      printf( "New LIB variable is: %s\n", libvar );
}
```

_putimage Functions

#include <graph.h>

Syntax void __far _putimage(short *x*, short *y*, const char __huge **image*, short *action*);

void __far _putimage_w(double *wx*, double *wy*, const char __huge **image* , short *action*);

Parameter	Description
x, *y*	Position of upper-left corner of image
image	Stored image buffer
action	Interaction with existing screen image
wx, *wy*	Position of upper-left corner of image

The _putimage function transfers to the screen the image stored in the buffer that *image* points to.

In the _putimage function, the upper-left corner of the image is placed at the view coordinate point (*x*, *y*). In the _putimage_w function, the upper-left corner of the image is placed at the window coordinate point (*wx*, *wy*).

The *action* argument defines the interaction between the stored image and the one that is already on the screen. It may be any one of the following manifest constants (defined in GRAPH.H):

_GAND
: Transfers the image over an existing image on the screen. The resulting image is the logical-AND product of the two images: points that had the same color in both the existing image and the new one will remain the same color, while points that have different colors are joined by logical-AND.

_GOR
: Superimposes the image onto an existing image. The new image does not erase the previous screen contents.

_GPRESET
: Transfers the data point-by-point onto the screen. Each point has the inverse of the color attribute it had when it was taken from the screen by _getimage, producing a negative image.

_GPSET
: Transfers the data point-by-point onto the screen. Each point has the exact color attribute it had when it was taken from the screen by _getimage.

_GXOR
: Causes the points on the screen to be inverted where a point exists in the *image* buffer. This behavior is like that of the cursor: when an image is put against a complex background twice, the background is restored unchanged. This allows you to move an object around without erasing the background. The _GXOR constant is a special mode often used for animation.

Return Value

None. Use the _grstatus function to check the result of a call to the _putimage functions.

Standards: None
16-Bit: MS-DOS, QWIN

_getimage
_grstatus
_imagesize

puts

#include <stdio.h>

Syntax int puts(const char **string*);

Parameter	**Description**
string	String to be output

The puts function writes *string* to the standard output stream stdout, replacing the string's terminating null character ('\0') with a newline character (\n) in the output stream.

Return Value

The puts function returns a nonnegative value if it is successful. If the function fails, it returns EOF.

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN

fputs
gets

```
/* PUTS.C: This program uses puts
 * to write a string to stdout.
 */
#include <stdio.h>
void main( void )
{
   puts( "Hello world from puts!" );
}
```

_putw

#include <stdio.h>

Syntax int _putw(int *binint*, FILE **stream*);

Parameter	**Description**
binint	Binary integer to be output
stream	Pointer to FILE structure

The _putw function writes a binary value of type int to the current position of *stream*. The _putw function does not affect the alignment of items in the stream, nor does it assume any special alignment.

The _putw function is provided primarily for compatibility with previous libraries. Note that portability problems may occur with _putw, since the size of an int and ordering of bytes within an int differ across systems.

Return Value

The _putw function returns the value written. A return value of EOF may indicate an error. Since EOF is also a legitimate integer value, ferror should be used to verify an error.

Use _putw for compatibility with ANSI naming conventions of non-ANSI functions. Use putw and link with OLDNAMES.LIB for UNIX compatibility.

Standards: UNIX
16-Bit: MS-DOS, QWIN

_getw

```c
/* PUTW.C: This program uses _putw to write a
 * word to a stream, then performs an error check.
 */
#include <stdio.h>
#include <stdlib.h>
void main( void )
{
   FILE *stream;
   unsigned u;
   if( (stream = fopen( "data.out", "wb" )) == NULL )
      exit( 1 );
   for( u = 0; u << 10; u++ )
   {
      _putw( u + 0x2132, stdout );
      _putw( u + 0x2132, stream );   /* Write word to stream. */
      if( ferror( stream ) )         /* Make error check. */
      {
         printf( "_putw failed" );
         clearerr( stream );
         exit( 1 );
      }
   }
   printf( "\nWrote ten words\n" );
   fclose( stream );
}
```

qsort

#include <stdlib.h> For ANSI compatibility

#include <search.h> Required only for function declarations

Syntax void qsort(void **base*, size_t *num*, size_t *width*, int(__cdecl **compare*) (const void **elem1*, const void **elem2*));

Parameter	Description
base	Start of target array
num	Array size in elements
width	Element size in bytes
compare	Comparison function
elem1	Pointer to the key for the search
elem2	Pointer to the array element to be compared with the key

The qsort function implements a quick-sort algorithm to sort an array of *num* elements, each of *width* bytes. The argument *base* is a pointer to the base of the array to be sorted. The qsort function overwrites this array with the sorted elements.

The argument *compare* is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. The qsort function calls the *compare* routine one or more times during the sort, passing pointers to two array elements on each call:

compare((void *) *elem1*, (void *) *elem2*);

The routine must compare the elements, then return one of the following values:

Value	Meaning
< 0	*elem1* less than *elem2*
= 0	*elem1* equivalent to *elem2*
> 0	*elem1* greater than *elem2*

The array is sorted in increasing order, as defined by the comparison function. To sort an array in decreasing order, reverse the sense of " greater than" and "less than" in the comparison function.

Return Value

None.

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN

bsearch
_lsearch

```
/* QSORT.C: This program reads the command-line
 * parameters and uses qsort to sort them. It
 * then displays the sorted arguments.
 */
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
int compare( const void *arg1, const void *arg2 );
void main( int argc, char **argv )
{
   int i;
   /* Eliminate argv[0] from sort: */
   argv++;
   argc--;
   /* Sort remaining args using Quicksort algorithm: */
   qsort( (void *)argv, (size_t)argc, sizeof( char * ), compare );
   /* Output sorted list: */
   for( i = 0; i < argc; ++i )
      printf( "%s ", argv[i] );
   printf( "\n" );
}
int compare( const void *arg1, const void *arg2 )
{
   /* Compare all of both strings: */
   return _stricmp( * ( char** ) arg1, * ( char** ) arg2 );
}
```

raise

#include <signal.h>

Syntax int raise(int *sig*);

Parameter	**Description**
sig	Signal to be raised

The raise function sends *sig* to the executing program. If a signal-handling routine for *sig* has been installed by a prior call to signal, raise causes that routine to be executed. If no handler routine has been installed, the default action (as listed below) is taken.

The signal value *sig* can be one of the following manifest constants:

Signal	**Meaning**	**Default**
SIGABRT	Abnormal termination.	Terminates the calling program with exit code 3.
SIGFPE	Floating-point error.	Terminates the calling program.
SIGILL	Illegal instruction. This signal is not generated by MS-DOS, but is supported for ANSI compatibility.	Terminates the calling program.
SIGINT	CTRL+C interrupt.	Issues INT 23H.
SIGSEGV	Illegal storage access. This signal is not generated by MS-DOS, but is supported for ANSI compatibility.	Terminates the calling program.
SIGTERM	Termination request sent to the program. This signal is not generated by MS-DOS, but is supported for ANSI compatibility.	Ignores the signal.

Return Value

If successful, the raise function returns 0. Otherwise, it returns a nonzero value.

abort
signal

Standards: ANSI

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

rand

#include <stdlib.h> Required only for function declarations

Syntax int rand(void);

The rand function returns a pseudorandom integer in the range 0 to RAND_MAX. The srand routine can be used to seed the pseudorandom-number generator before calling rand.

Return Value

The rand function returns a pseudorandom number, as described above. There is no error return.

srand

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* RAND.C: This program seeds the random-number generator
 * with the time, then displays 10 random integers.
 */
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
void main( void )
{
   int i;
   /* Seed the random-number generator with current time so that
    * the numbers will be different every time we run.
    */
   srand( (unsigned)time( NULL ) );
   /* Display 10 numbers. */
   for( i = 0;   i < 10;i++ )
      printf( "  %6d\n", rand() );
}
```

_read

#include <io.h> Required only for function declarations

Syntax int _read(int *handle*, void **buffer*, unsigned int *count*);

Parameter	Description
handle	Handle referring to open file
buffer	Storage location for data
count	Maximum number of bytes

The _read function attempts to read *count* bytes into *buffer* from the file associated with *handle*. The read operation begins at the current position of the file pointer associated with the given file. After the read operation, the file pointer points to the next unread character.

Return Value

The _read function returns the number of bytes actually read, which may be less than *count* if there are fewer than *count* bytes left in the file, or if the file was opened in text mode (see below). The return value 0 indicates an attempt to read at end-of-file. The return value -1 indicates an error, and errno is set to the following value:

Value	Meaning
EBADF	The given *handle* is invalid; or the file is not open for reading; or (MS-DOS versions 3.0 and later) the file is locked.

For 16-bit platforms, if you are reading more than 32K (the maximum size for type int) from a file, the return value should be of type unsigned int (see the example that follows). However, the maximum number of bytes that can be read from a file in one operation is 65,534, because 65,535 (or 0xFFFF) is indistinguishable from -1, and therefore cannot be distinguished from an error return.

If the file was opened in text mode, the return value may not correspond to the number of bytes actually read. When text mode is in effect, each carriage-return-line-feed (CR-LF) pair is replaced with a single line-feed character. Only the single line-feed character is counted in the return value. The replacement does not affect the file pointer.

Note that when files are opened in text mode, a CTRL+Z character is treated as an end-of-file indicator. When the CTRL+Z is encountered, the read terminates, and the next read returns 0 bytes. The _lseek function will clear the end-of-file indicator.

Use _read for compatibility with ANSI naming conventions of non-ANSI functions. Use read and link with OLDNAMES.LIB for UNIX compatibility.

_creat
fread
_open
_write

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* READ.C: This program opens a file named
 * READ.C and tries to read 60,000 bytes from
 * that file using read. It then displays the
 * actual number of bytes read from READ.C.
 */
#include <fcntl.h>      /* Needed only for _O_RDWR definition */
#include <io.h>
#include <stdlib.h>
#include <stdio.h>
char buffer[60000];
void main( void )
{
   int fh;
   unsigned int nbytes = 60000, bytesread;
   /* Open file for input: */
   if( (fh = _open( "read.c", _O_RDONLY )) == -1 )
   {
      perror( "open failed on input file" );
      exit( 1 );
   }
   /* Read in input: */
   if( ( bytesread = _read( fh, buffer, nbytes ) ) <= 0 )
      perror( "Problem reading file" );
   else
      printf( "Read %u bytes from file\n", bytesread );
   _close( fh );
}
```

realloc Functions

#include <stdlib.h> For ANSI compatibility (realloc only)

#include <malloc.h> Required only for function declarations

Syntax void *realloc(void **memblock*, size_t *size*);

void __based(void) *_brealloc(__segment *seg*, void __based(void) **memblock*, size_t *size*);

void __far *_frealloc(void __far **memblock*, size_t *size*);

void __near *_nrealloc(void __near **memblock*, size_t *size*);

Parameter	Description
memblock	Pointer to previously allocated memory block
size	New size in bytes
seg	Segment selector

The realloc family of functions changes the size of a previously allocated memory block. The *memblock* argument points to the beginning of the memory block. If *memblock* is NULL (_NULLOFF for _brealloc), realloc functions in the same way as malloc and allocates a new block of *size* bytes. If *memblock* is not NULL (_NULLOFF for _brealloc), it should be a pointer returned by a prior call to calloc, malloc, or realloc.

The *size* argument gives the new size of the block, in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes, although the new block may be in a different location. Because the new block can be in a new memory location, the pointer returned by the realloc functions is not guaranteed to be the pointer passed through the *memblock* argument.

In large data models (that is, compact-, large-, and huge-model programs), realloc maps to _frealloc. In small data models (tiny-, small-, and medium-model programs), realloc maps to _nrealloc.

The various realloc functions reallocate memory in the heap as specified in the following list:

Function	Heap
realloc	Depends on data model of program
_brealloc	Based heap specified by *seg* value
_frealloc	Far heap (outside default data segment)
_nrealloc	Near heap (inside default data segment)

Return Value

The realloc functions return a void pointer to the reallocated (and possibly moved) memory block.

The return value is NULL (_NULLOFF for _brealloc) if the size is zero and the buffer argument is not NULL (_NULLOFF for _brealloc), or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second, the original block is unchanged.

The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

realloc

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_brealloc, _frealloc, _nrealloc

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

calloc functions
free functions
malloc functions

```
/* REALLOC.C: This program allocates a block of memory for
 * buffer and then uses _msize to display the size of that
 * block. Next, it uses realloc to expand the amount of
 * memory used by buffer and then calls _msize again to
 * display the new amount of memory allocated to buffer.
 */
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
void main( void )
{
   long *buffer;
   size_t size;
   if( (buffer = (long *)malloc( 1000 * sizeof( long ) )) == NULL )
      exit( 1 );
   size = _msize( buffer );
   printf( "Size of block after malloc of 1000 longs: %u\n", size );
   /* Reallocate and show new size: */
   if( (buffer = realloc( buffer, size + (1000 * sizeof( long )) )) ==  NULL )
      exit( 1 );
   size = _msize( buffer );
   printf( "Size of block after realloc of 1000 more longs: %u\n", size );
   free( buffer );
   exit( 0 );
}
```

_rectangle Functions

#include <graph.h>

Syntax short __far _rectangle(short *control*, short *x1*, short *y1*, short *x2*, short *y2*);

short __far _rectangle_w(short *control*, double *wx1*, double *wy1*, double *wx2*, double *wy2*);

short __far _rectangle_wxy(short *control*, struct _wxycoord __far * *pwxy1*, struct _wxycoord __far **pwxy2*);

Parameter	Description
control	Fill flag
x1, *y1*	Upper-left corner
x2, *y2*	Lower-right corner
wx1, *wy1*	Upper-left corner
wx2, *wy2*	Lower-right corner
pwxy1	Upper-left corner
pwxy2	Lower-right corner

The _rectangle functions draw a rectangle with the current line style. The _rectangle function uses the view coordinate system. The view coordinate points (*x1*, *y1*) and (*x2*, *y2*) are the diagonally opposed corners of the rectangle.

The _rectangle_w function uses the window coordinate system. The window coordinate points (*wx1*, *wy1*) and (*wx2*, *wy2*) are the diagonally opposed corners of the rectangle.

The _rectangle_wxy function uses the window coordinate system. The window coordinate points (*pwxy1*) and (*pwxy2*) are the diagonally opposed corners of the rectangle. The coordinates for the _rectangle_wxy routine are given in terms of a _wxycoord structure (defined in GRAPH.H), which contains the following elements:

Element	Description
double wx	window *x* coordinate
double wy	window *y* coordinate

The *control* parameter can be one of the following manifest constants:

Constant	Action
_GFILLINTERIOR	Fills the figure, using a scanfill algorithm, with the current color using the current fill mask
_GBORDER	Does not fill the rectangle

If the current fill mask is NULL, no mask is used. Instead, the rectangle is filled with the current color.

If you try to fill the rectangle with the _floodfill function, the rectangle must be bordered by a solid line-style pattern.

Return Value

The function returns a nonzero value if the rectangle is drawn successfully, or 0 if not.

_arc functions
_ellipse functions
_floodfill
_getcolor
_lineto functions
_pie functions
_polygon
_setcolor
_setfillmask

Standards: None
16-Bit: MS-DOS, QWIN

```
/* RECT.C: This program draws a rectangle. */
#include <conio.h>
#include <stdlib.h>
#include <graph.h>
void main( void )
{
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   _rectangle( _GBORDER, 80, 50, 240, 150 );
   _getch();
   _setvideomode( _DEFAULTMODE );
}
```

_registerfonts

#include <graph.h>

Syntax short __far _registerfonts(const char __far **pathname*);

Parameter	Description
path	Path specifying .FON files to be registered

The _registerfonts function initializes the fonts graphics system. Font files must be registered with the _registerfonts function before any other font-related library function (_getgtextextent, _outgtext, _setfont, _unregisterfonts) can be used.

The _registerfonts function reads the specified files and loads font header information into memory. Each font header takes up about 140 bytes of memory.

The *path* argument is the path specification and filename of valid .FON files. The *path* can contain standard MS-DOS wildcards.

The font functions affect only the output from the font output function _outgtext; no other run-time output functions are affected by font usage.

The _registerfonts function works differently in QuickWin applications. For specific information, see *Programming Techniques*.

Return Value

The _registerfonts function returns a positive value that indicates the number of fonts successfully registered. A negative return value indicates failure. The following negative values may be returned:

Value	Meaning
-1	No such file or directory.
-2	One or more of the .FON files was not a valid, binary .FON file.
-3	One or more of the .FON files is damaged.

_getfontinfo
_getgtextextent
_grstatus
_outgtext
_setfont
_unregisterfonts

Standards: None
16-Bit: MS-DOS, QWIN

_remapallpalette, _remappalette

#include <graph.h>

Syntax short __far _remapallpalette(long __far **colors*);
long __far _remappalette(short *index*, long *color*);

Parameter	Description
colors	Color value array
index	Color index to reassign
color	Color value to assign color index to

The _remapallpalette function remaps the entire color palette simultaneously to the colors given in the *colors* array. The *colors* array is an array of long integers where the size of the array varies from 16 to 64 to 256, depending on the video mode. The number of colors mapped depends on the number of colors supported by the current video mode. The _remapallpalette function works in all video modes (except _ORESCOLOR mode), but only with EGA, MCGA, VGA, or SVGA hardware.

The default color values for color text or a 16-color graphics mode are shown below:

Number	Color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	White
8	Dark gray
9	Light blue
10	Light green
11	Light cyan
12	Light red
13	Light magenta
14	Yellow
15	Bright white

The first array element specifies the new color value to be associated with color index 0 (the background color in graphics modes). After the call to _remapallpalette, calls to _setcolor will index into the new array of colors. The mapping done by _remapallpalette affects the current display immediately.

The *colors* array can be larger than the number of colors supported by the current video mode, but

only the first *n* elements are used, where *n* is the number of colors supported by the current video mode, as indicated by the numcolors element of the _videoconfig structure.

The long color value is defined by specifying three bytes of data representing the three component colors: red, green, and blue.

Each of the three bytes represents the intensity of one of the red, green, or blue component colors, and must be in the range 0–63. In other words, the low-order six bits of each byte specify the component's intensity and the high-order two bits should be zero. The fourth (high-order) byte in the long is unused and should be set to zero. The diagram below shows the ordering of bytes within the long value.

For example, to create a lighter shade of blue, start with lots of blue, add some green, and maybe a little bit of red. The three-byte color value would be:

```
blue byte            green byte        red byte

00011111             00101111          00011111

high ----------------->  low order
```

Manifest constants are defined in GRAPH.H for the default color values corresponding to color indices 0-15 in color text modes and 16-color graphics modes, as shown below:

Index	Constant
0	_BLACK
1	_BLUE
2	_GREEN
3	_CYAN
4	_RED
5	_MAGENTA
6	_BROWN
7	_WHITE
8	_GRAY
9	_LIGHTBLUE
10	_LIGHTGREEN
11	_LIGHTCYAN
12	_LIGHTRED
13	_LIGHTMAGENTA
14	_YELLOW
15	_BRIGHTWHITE

The VGA supports a palette of 262,144 (256K) colors in color modes, and the EGA supports a palette of only 64 different colors. Color values for EGA are specified in exactly the same way as with the VGA; however, the low-order four bits of each byte are simply ignored.

The _remappalette function assigns a new color value *color* to the color index given by *index*. This remapping affects the current display immediately.

The _remappalette function works in all graphics modes, but only with EGA, MCGA, VGA, or SVGA hardware. An error results if the function is called while using any other configuration.

The color value used in _remappalette is defined and used exactly as noted above for _remapallpalette. The range of color indices used with _remappalette depends on the number of colors supported by the video mode.

The _remapallpalette and _remappalette functions do not affect the presentation-graphics "palettes,"

which are manipulated with the _pg_getpalette, _pg_setpalette, and _pg_resetpalette functions.

If a VGA or MCGA adapter is connected to an analog monochrome monitor, the color value is transformed into its gray-scale equivalent, based on the weighted sum of its red, green, and blue components (30% red + 50% green + 11% blue).

The _remapallpalette and _remappalette functions work differently in QuickWin applications. For specific information, see *Programming Techniques*.

Return Value

If successful, _remapallpalette returns nonzero value (short). In case of an error, _remapallpalette returns 0 (short).

If successful, _remappalette returns the color value previously assigned to *index*, or -1 if the function is inoperative (not EGA, VGA, SVGA, or MCGA), or if the color index is out of range. Note that _remapallpalette returns a short value and _remappalette returns a long value.

_getvideoconfig
_selectpalette
_setbkcolor
_setvideomode

Standards: None
16-Bit: MS-DOS, QWIN

```
/* RMPALPAL.C: This example illustrates functions
 * for assigning color values to color indices.
 * Functions illustrated include:
 *    _remappalette         _remapallpalette
 */
#include <graph.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
/* Macro for mixing Red, Green, and Blue elements of color */
#define RGB(r,g,b) (((long) ((b) << 8 | (g)) << 8) | (r))
long tmp, pal[256];
void main( void )
{
   short  red, blue, green;
   short  inc, i, mode, cells, x, y, xinc, yinc;
   char   buf[40];
   struct _videoconfig vc;
   /* Make sure all palette numbers are valid. */
   for( i = 0; i < 256; i++ )
      pal[i] = _BLACK;
   /* Loop through each graphics mode that supports palettes. */
   for( mode = _MRES4COLOR;mode <= _MRES256COLOR;mode++ )
   {
      if( mode == _ERESNOCOLOR )
         mode++;
      if( !_setvideomode( mode ) )
         continue;
      /* Set variables for each mode. */
      _getvideoconfig( &vc );
      switch( vc.numcolors )
      {
      case 256:          /* Active bits in this order:           */
         cells = 13;
         inc = 12;      /* ???????? ??bbbbbb ??gggggg ??rrrrrr  */
         break;
      case  16:
         cells = 4;
         if( (vc.mode == _ERESCOLOR) || (vc.mode == _VRES16COLOR) )
            inc = 16;   /* ???????? ??bb???? ??gg???? ??rr????  */
         else
            inc = 32;   /* ???????? ??Bb???? ??Gg???? ??Rr????  */
         break;
      case   4:
         cells = 2;
         inc = 32;      /* ???????? ??Bb???? ??Gg???? ??Rr????  */
         break;
      default:
         continue;
      }
      xinc = vc.numxpixels / cells;
      yinc = vc.numypixels / cells;
      /* Fill palette arrays in BGR order. */
      for( i = 0, blue = 0; blue < 64; blue += inc )
         for( green = 0; green < 64; green += inc )
            for( red = 0; red < 64; red += inc )
            {
               pal[i] = RGB( red, green, blue );
               /* Special case of using 6 bits to represent 16 colors.
                * If both bits are on for any color, intensity is set.
```

```
                * If one bit is set for a color, the color is on.
                */
               if( inc == 32 )
                  pal[i + 8] = pal[i] | (pal[i] >> 1);
               i++;
            }
      /* If palettes available, remap all palettes at once. */
      if( !_remapallpalette( pal ) )
      {
         _setvideomode( _DEFAULTMODE );
         _outtext( "Palettes not available with this adapter" );
         exit( 1 );
      }
      /* Draw colored squares. */
      for( i = 0, x = 0; x < ( xinc * cells ); x += xinc )
         for( y = 0; y < ( yinc * cells); y += yinc )
         {
            _setcolor( i++ );
            _rectangle( _GFILLINTERIOR, x, y, x + xinc, y + yinc );
         }
      /* Note that for 256-color mode, not all colors are shown. The number
       * of colors from mixing three base colors can never be the same as
       * the number that can be shown on a two-dimensional grid.
       */
      sprintf( buf, "Mode %d has %d colors", vc.mode, vc.numcolors );
      _setcolor( (short)(vc.numcolors / 2) );
      _outtext( buf );
      _getch();
      /* Change each palette entry separately in GRB order. */
      for( i = 0, green = 0; green < 64; green += inc )
         for( red = 0; red < 64; red += inc )
            for(blue = 0; blue < 64; blue += inc )
            {
               tmp = RGB( red, green, blue );
               _remappalette( i, tmp );
               if( inc == 32 )
                  _remappalette( (short)(i + 8), (long)(tmp | (tmp >> 1)) );
               i++;
            }
      _getch();
   }
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}
```

remove

#include <stdio.h> Required for ANSI compatibility

#include <io.h> Use either IO.H or STDIO.H

Syntax int remove(const char **path*);

Parameter	Description
path	Path of file to be removed

The remove function deletes the file specified by *path*.

Return Value

The function returns 0 if the file is successfully deleted. Otherwise, it returns -1 and sets errno to one of these values:

Value	Meaning
EACCES	Path specifies a read-only file.
ENOENT	Filename or path not found, or path specifies a directory.

_unlink

Standards: ANSI

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* REMOVE.C: This program uses remove to delete REMOVE.OBJ. */
#include <stdio.h>
void main( void )
{
   if( remove( "remove.obj" ) == -1 )
      perror( "Could not delete 'REMOVE.OBJ'" );
   else
      printf( "Deleted 'REMOVE.OBJ'\n" );
}
```

rename

#include <stdio.h> Required for ANSI compatibility

#include <io.h> Use either IO.H or STDIO.H

Syntax int rename(const char *_oldname_, const char *_newname_);

Parameter	**Description**
oldname	Pointer to old name
newname	Pointer to new name

The rename function renames the file or directory specified by *oldname* to the name given by *newname*. The old name must be the path of an existing file or directory. The new name must not be the name of an existing file or directory.

The rename function can be used to move a file from one directory to another by giving a different path in the *newname* argument. However, files cannot be moved from one device to another (for example, from drive A to drive B). Directories can only be renamed, not moved.

Return Value

The rename function returns 0 if it is successful. On an error, it returns a nonzero value and sets errno to one of the following values:

Value	**Meaning**
EACCES	File or directory specified by *newname* already exists or could not be created (invalid path); or oldname is a directory and *newname* specifies a different path.
ENOENT	File or path specified by *oldname* not found.
EXDEV	Attempt to move a file to a different device.

Standards: ANSI

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```c
/* RENAMER.C: This program attempts to rename a file
 * named RENAMER.OBJ to RENAMER.JBO. For this operation
 * to succeed, a file named RENAMER.OBJ must exist and
 * a file named RENAMER.JBO must not exist.
 */
#include <stdio.h>
void main( void )
{
   int  result;
   char old[] = "RENAMER.OBJ", new[] = "RENAMER.JBO";
   /* Attempt to rename file: */
   result = rename( old, new );
   if( result != 0 )
      printf( "Could not rename '%s'\n", old );
   else
      printf( "File '%s' renamed to '%s'\n", old, new );
}
```

rewind

#include <stdio.h>

Syntax void rewind(FILE **stream*);

Parameter	**Description**
stream	Pointer to FILE structure

The rewind function repositions the file pointer associated with *stream* to the beginning of the file. A call to rewind is equivalent to

(void) fseek(*stream*, 0L, SEEK_SET);

except that rewind clears the error indicators for the stream, and fseek does not. Both rewind and fseek clear the end-of-file indicator. Also, fseek returns a value that indicates whether the pointer was successfully moved, but rewind does not return any value.

You can also use the rewind function to clear the keyboard buffer. Use the rewind function with the stream stdin, which is associated with the keyboard by default.

Return Value

The rewind function has no return value.

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* REWIND.C: This program first opens a file named
 * REWIND.OUT for input and output and writes two
 * integers to the file. Next, it uses rewind to
 * reposition the file pointer to the beginning of
 * the file and reads the data back in.
 */
#include <stdio.h>
void main( void )
{
   FILE *stream;
   int data1, data2;
   data1 = 1;
   data2 = -37;
   if( (stream = fopen( "rewind.out", "w+" )) != NULL )
   {
      fprintf( stream, "%d %d", data1, data2 );
      printf( "The values written are: %d and %d\n", data1, data2 );
      rewind( stream );
      fscanf( stream, "%d %d", &data1, &data2 );
      printf( "The values read are: %d and %d\n", data1, data2 );
      fclose( stream );
   }
}
```

_rmdir

#include <direct.h> Required only for function declarations

Syntax int _rmdir(const char **dirname*);

Parameter	Description
dirname	Path of directory to be removed

The _rmdir function deletes the directory specified by *dirname*. The directory must be empty, and it must not be the current working directory or the root directory.

Return Value

The _rmdir function returns the value 0 if the directory is successfully deleted. A return value of -1 indicates an error, and errno is set to one of the following values:

Value	Meaning
EACCES	The given path is not a directory; or the directory is not empty; or the directory is the current working directory or the root directory.
ENOENT	Path not found.

_chdir
_mkdir

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_rmtmp

#include <stdio.h>

Syntax int _rmtmp(void);

The _rmtmp function is used to clean up all the temporary files in the current directory. The function removes only those files created by tmpfile and should be used only in the same directory in which the temporary files were created.

Return Value

The _rmtmp function returns the number of temporary files closed and deleted.

_flushall
tmpfile
tmpnam

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_rotl, _rotr

#include <stdlib.h>

Syntax unsigned int _rotl(unsigned int *value*, int *shift*);

unsigned int _rotr(unsigned int *value*, int *shift*);

Parameter	**Description**
value	Value to be rotated
shift	Number of bits to shift

The _rotl and _rotr functions rotate the unsigned *value* by *shift* bits. The _rotl function rotates the value left. The _rotr function rotates the value right. Both functions "wrap" bits rotated off one end of *value* to the other end.

Return Value

Both functions return the rotated value. There is no error return.

__lrotl
__lrotr

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* ROT.C: This program uses _rotr and _rotl with
 * different shift values to rotate an integer.
 */
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   unsigned val = 0x0fd93;
   printf( "0x%4.4x rotated left three times is 0x%4.4x\n", val, _rotl( val, 3 ) );
   printf( "0x%4.4x rotated right four times is 0x%4.4x\n", val, _rotr( val, 4 ) );
}
```

About the Microsoft Run-Time Library

The Microsoft run-time library is a set of more than 550 ready-to-use functions and macros designed for use in C and C++ programs. The run-time library makes programming easier by providing:

- Fast and efficient routines to perform common programming tasks (such as string manipulation), sparing you the time and effort needed to write such routines
- Reliable methods of performing operating-system functions (such as opening and closing files)

The run-time library is important because it provides basic functions not provided by the C and C++ languages themselves. These functions include input and output, memory allocation, process control, graphics, and many others.

Help describes the run-time library routines included with Visual C++. These comprise all of the routines included with earlier versions of Microsoft C, as well as many new routines.

ANSI C Compatibility

The run-time library routines are designed for compatibility with the ANSI C standard, which the Microsoft C and C++ compilers support. Functions that are ANSI C compatible are marked as ANSI in the compatibility heading in the help topic for the routine.

Type Checking

The major innovation of ANSI C is to permit argument-type lists in function prototypes (declarations). Given the information in the function prototype, the compiler can check later references to the function to make sure that the references use the correct number and type of arguments and the correct return value.

To take advantage of the compiler's type-checking ability, the include files that accompany the run-time library have been expanded. In addition to the definitions and declarations required by library routines, the include files now contain function declarations with argument-type lists. Several new include files have also been added. The names of these files are chosen to maximize compatibility with the ANSI C standard and with UNIX and XENIX names.

Underscores and OLDNAMES.LIB

With Microsoft C/C++, all Microsoft-specific run-time functions, constants, variables, type definitions, structures, and macros (such as, respectively, _open, _VRES16COLOR, _cpumode, _HEAPINFO, _heapinfo, and __isascii) are ANSI compatible. The Microsoft-specific run-time functions, constants, variables, type definitions, and structures begin with a single underscore; Microsoft-specific run-time macros begin with two underscores.

For compatibility with previous versions of Microsoft C, Microsoft C/C++ provides a library named OLDNAMES.LIB, which contains alias records mapping the names to the new names. For instance, open is mapped to _open.

You have to link with OLDNAMES.LIB to link Microsoft C/C++ programs with object files created by previous versions of Microsoft C. However, by default the compiler emits a library search record. The only time you must link explicitly with OLDNAMES.LIB is under one of the following situations:

- Compiling with a combination of the default /Ze option (use Microsoft extensions) and the /Zl option (omit default library name from object file)
- Compiling with the default /Ze option (use Microsoft extensions) and a combination of the /link option (linker-control) and the /NOD option (no default-library search)

Note The compiler views a structure with both an old name and a new name as two different types. You cannot copy from an old type to a new type. Also, old prototypes that take struct pointers use the old struct names in the prototype. So, you must be consistent; match the old names for routines with the old names for the parameters and be similarly consistent with the new routine names and parameters.

UNIX C Compatibility

Most of the functions in the Microsoft run-time library are compatible with like-named UNIX routines. For additional compatibility, the math library functions have been extended to provide exception handling in the same manner as the UNIX System V math functions. Functions that are UNIX and XENIX compatible are marked as UNIX in the compatibility section.

MS-DOS and Windows Programming

Programming Microsoft run-time library routines are designed to maintain maximum compatibility between MS-DOS, Windows, and UNIX or XENIX systems. The run-time library offers a number of operating-system interface routines that allow you to take advantage of specific features of MS-DOS and Windows. Functions that are MS-DOS and Windows compatible are marked, respectively, as MS-DOS and WIN in the compatibility heading in the help topic for the function. Note that for Windows the compatibility heading also contains information on dynamic-link library (DLL) compatibility.

QuickWin

The Microsoft run-time library now contains several QuickWin functions that make it possible to compile non-Windows MS-DOS programs as simple applications for Windows. MS-DOS programs compiled with the /Mq compiler option have a limited Windows user interface, including a standard menu bar, standard online help (describing only the QuickWin features), and a client (or application) window with child (document) windows for the C input/output streams stdin, stdout, and stderr. You can also add other child windows of your own. QuickWin applications support the Windows Clipboard, and you can use standard C functions to write to and read from a QuickWin application's windows, which behave as streams. Functions that are QuickWin compatible are marked as QWIN in the compatibility heading in the help topic for the function.

Extended Graphics Library

The Microsoft run-time library contains more than one hundred graphics routines. The core of this library consists of several dozen low-level graphics routines that allow your programs to select video modes, set points, draw lines, change colors, and draw shapes such as rectangles and ellipses. You can display real-valued data, such as floating-point values, within windows of different sizes by using various coordinate systems.

The graphics library includes presentation graphics and fonts. The presentation-graphics library provides powerful tools for adding presentation-quality graphics to your programs. These routines can display data as a variety of graphs, including pie charts, bar and column charts, line graphs, and scatter diagrams.

The fonts library allows your programs to display various styles and sizes of text in graphics images or charts. You can use font-manipulation routines with any graphics routines that display text, including presentation graphics.

Calling Library Routines

To use a library routine, simply call it in your program, just as if it is defined there. For instance, suppose you write the following program and name it SAMPLE.C:

```
#include <stdio.h>
void main( void )
{
   printf( "Microsoft C/C++" );
}
```

The program prints Microsoft C/C++ by calling the printf routine, which is part of the run-time library. Calling a library routine normally involves two groups of files:

- Header ("include") files that contain declarations, constants, and type definitions required by library routines
- Library files that contain the library routines in compiled form

Header files and library files are both included with Microsoft C/C++. Header files are used when compiling, and library files are used when linking.

You include the necessary header files in your program source code with #include directives. The help topic for each library routine tells you which header file the routine requires. Because printf requires the STDIO.H header file, the SAMPLE.C program contains the following line:

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

This line causes the compiler to insert the contents of STDIO.H into the source file SAMPLE.C.

After you compile the source file, you link the resulting object (.OBJ) file with the appropriate library (.LIB) file to create an executable (.EXE) file. Your object file contains the name of every routine that your program calls, including library routines. If a routine is not defined in your program, the linker searches for its code in a library file and includes that code in the executable file. Normally, the code for standard library routines is contained in the "default library" that you create when installing Microsoft C/C++. Because the linker automatically searches the default library, you do not need to specify that library's name when linking your program. The following command links the example program with the default library:

```
link sample,,,;
```

If you call a library routine that is not contained in the default library, you must give the linker the name of the library file that contains the routine. For instance, if your program uses a Microsoft graphics routine, you would link the program using a line that includes GRAPHICS.LIB:

```
link sample,,, graphics.lib;
```

Using Header Files

You should include header files when using library routines. The reasons why header files are required are described below.

Including Necessary Definitions

Many run-time library routines use constants, type definitions, or macros defined in a header file. To use the routine, you must include the header file containing the needed definition(s). The following list gives examples:

Definition	Example
Macro	If a library routine is implemented as a macro, the macro definition appears in a header file. For instance, the toupper macro is defined in the header file CTYPE.H.
Manifest constant	Many library routines refer to constants that are defined in header files. For instance, the _open routine uses constants such as _O_CREAT, which is defined in the header file FCNTL.H.
Type definition	Some library routines return a structure or take a structure as an argument. For example, stream input/output routines use a structure of type FILE, which is defined in STDIO.H.

Including Function Declarations

The run-time library header files also contain function declarations for every function in the run-time library. These declarations are in the style recommended by the ANSI C standard. Given these declarations, the compiler can perform "type checking" on every reference to a library function, making sure that you have used the correct return type and arguments. Function declarations are sometimes called "prototypes," because the declaration serves as a prototype or template for every subsequent reference to the function.

A function declaration lists the name of the function, its return type, and the number and type of its arguments. For instance, this is the declaration of the pow library function from the header file MATH.H:

```
double pow( double x, double y );
```

The example declares that pow returns a value of type double and takes two arguments of type double. Given this declaration, the compiler can check every reference to pow in your program to ensure that the reference passes two double arguments to pow and takes a return value of type double.

The compiler can perform type checking only for function references that appear after the function declaration. Because of this, function declarations normally appear near the beginning of the source file, prior to any use of the functions they declare.

Function declarations are especially important for functions that return a value of some type other than int, which is the default. For example, the pow function returns a double value. If you do not declare such a function, the compiler treats its return value as int, which can cause unexpected results.

It is also a good practice to provide declarations for functions that you write. If you do not want to type the declarations by hand, you can generate them automatically by using the /Zg compiler option. This option causes the compiler to generate ANSI-standard function declarations for every function defined in the current source file. Redirect this output to a file, then insert the file near the beginning of your source file.

Your program can contain more than one declaration of the same function, as long as the declarations do not conflict. This is important if you have old programs whose function declarations do not contain argument-type lists. For instance, if your program contains the declaration,

```
char *calloc( );
```

you can later include the following declaration:

```
char *calloc(unsigned, unsigned);
```

Because the two declarations are compatible, even though they are not identical, no conflict occurs. The second declaration simply gives more information about function arguments than the first. A conflict would arise, however, if the declarations gave a different number of arguments or gave arguments of different types.

Some library functions can take a variable number of arguments. For instance, the printf function can take one argument or several. The compiler can perform only limited type checking on such functions, a factor that affects the following library functions:

- In calls to _cprintf, _cscanf, printf, and scanf, only the first argument (the format string) is type checked.
- In calls to fprintf, fscanf, sprintf, and sscanf, only the first two arguments (the file or buffer and the format string) are type checked. In calls to _snprintf, only the first three arguments (the file or buffer, the format string, and the maximum number of bytes to store) are type checked.
- In calls to _open, only the first two arguments (the path and the _open flag) are type checked.
- In calls to _sopen, only the first three arguments (the path, the _open flag, and the sharing mode) are type checked.
- In calls to _execl, _execle, _execlp, and _execlpe, only the first two arguments (the path and the first argument pointer) are type checked.
- In calls to _snprintf, _spawnl, _spawnle, _spawnlp, and _spawnlpe, only the first three arguments are type checked.

Paths and Filenames

Many library routines take strings representing paths and filenames as arguments. If you plan to transport your programs to the UNIX (or XENIX) operating system, remember that UNIX uses path and filename conventions different from those used by MS-DOS.

Case Sensitivity

The MS-DOS operating system is not case sensitive (it does not distinguish between uppercase and lowercase letters). Thus, SAMPLE.C and Sample.C refer to the same file. However, the UNIX operating system is case sensitive. In UNIX, SAMPLE.C and Sample.C refer to different files. To transport programs to UNIX, choose paths and filenames that work correctly in UNIX, since either case works in MS-DOS. For instance, the following directives are identical in MS-DOS, but only the second works in UNIX:

```
#include <STDIO.H>
#include <stdio.h>
```

Subdirectory Conventions

Under UNIX, certain header files are normally placed in a subdirectory named SYS. Microsoft C follows this convention to ease the process of transporting programs to UNIX. If you do not plan to transport your programs, you can place the SYS header files elsewhere.

Path Delimiters

UNIX uses the slash (/) in pathnames, while MS-DOS generally uses the backslash (\). MS-DOS accepts either the slash or the backslash. To transport programs to UNIX, use the slash to delimit paths.

Choosing Between Functions and Macros

Help uses the words "routine" and "function" interchangeably. However, the term "routine" actually encompasses functions and macros. Because functions and macros have different properties, pay attention to which form you are using. The descriptions in the help topics indicate whether routines are implemented as functions or as macros.

Most routines in the Microsoft run-time library are functions. They consist of compiled C code or assembled Microsoft Macro Assembler (MASM) code. However, a few library routines are implemented as macros that behave like functions. You can pass arguments to library macros and invoke them in the same way you invoke functions.

The main benefit of using macros is faster execution time. Every library macro is defined with a #define directive in a header file. A macro is expanded (replaced by its definition) during preprocessing, creating inline code. Thus, macros do not have the overhead associated with function calls. On the other hand, each use of a macro inserts the same code in your program, whereas a function definition occurs only once regardless of how many times it is called. Functions and macros thus offer a trade-off between speed and size.

Apart from speed and size issues, macros and functions have some other important differences:

- Some macros treat arguments with side effects incorrectly when the macro evaluates its arguments more than once (see the example that follows this list). Not every macro has this effect. To determine if a macro handles side effects as desired, examine its definition in the appropriate header file.
- A function name evaluates to an address, but a macro name does not. Thus, you cannot use a macro name in contexts requiring a function pointer. For instance, you can declare a pointer to a function, but you cannot declare a pointer to a macro.
- You can declare functions, but you cannot declare macros. Thus, the compiler cannot perform type checking of macro arguments as it does of function arguments. However, the compiler can detect when you pass the wrong number of arguments to a macro.

The following example demonstrates how some macros can produce unwanted side effects. It uses the toupper routine.

```
#include <ctype.h>
int a = 'm';
a = toupper(a++);
```

The example increments `a` when passing it as an argument to the toupper routine, which is implemented as a macro. It is defined in CTYPE.H:

```
#define toupper(c) ( (islower(c)) ? _toupper(c) : (c) )
```

The definition uses the conditional operator (**?** :). The conditional expression evaluates the argument c twice: once to check if it is lowercase and again to create the result. This macro evaluates the argument a++ twice, increasing `a` by 2 instead of 1. As a result, the value operated on by islower differs from the value operated on by _toupper.

Like some other library routines, toupper is provided in macro and function versions. The header file CTYPE.H not only declares the toupper function but also defines the toupper macro.

Choosing between the macro version and function version of such routines is easy. To use the macro version, simply include the header file that contains the macro definition. Because the macro definition of the routine always appears after the function declaration, the macro definition normally takes precedence. Thus, if your program includes CTYPE.H and then calls toupper, the compiler uses the toupper macro:

```
#include <ctype.h>
int a = 'm';
a = toupper(a);
```

You can force the compiler to use the function version of a routine by enclosing the routine's name in parentheses:

```
#include <ctype.h>
int a = 'm';
a = (toupper) (a);
```

Because the name toupper is not immediately followed by a left parenthesis, the compiler cannot interpret it as a macro name. It must use the toupper function.

A second way to do this is to "undefine" the macro definition with the #undef directive:

```
#include <ctype.h>
#undef toupper
```

Because the macro definition no longer exists, subsequent references to toupper use the function version.

A third way, not generally recommended, to make sure the compiler uses the function version is to declare the function explicitly:

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

Because this function declaration appears after the macro definition in CTYPE.H, it causes the compiler to use the toupper function.

Stack Checking on Entry

For certain library routines, the compiler performs stack checking on entry; the "stack" is a memory area used for temporary storage. On entry to such a routine, the stack is checked to determine if it has enough room for the local variables used by that routine. If it does, space is allocated by adjusting the stack pointer. Otherwise, a "stack overflow" run-time error occurs. If stack checking is disabled, the compiler assumes there is enough stack space. If there is not, you might overwrite memory locations in the data segment and receive no warning; unpredictable program behavior may result.

Typically, stack checking is enabled only for functions with large local-variable requirements (more than about 150 bytes), because there is enough free space between the stack and data segments to handle functions with smaller requirements. If the function is called many times, stack checking slows execution slightly.

Stack checking is enabled for the following library functions:

_execvp	_spawnvp	vprintf
_execvpe	_spawnvpe	vsprintf
fprintf	sprintf	_vsnprintf
fscanf	sscanf	_write
printf	system	
scanf	vfprintf	

You can enable or disable stack checking with the /Gs and /Ge compiler options or the check_stack pragma.

Handling Errors

Many library routines return a value that indicates an error condition. To avoid unexpected results, your code should always check such error values and handle all of the possible error conditions. The description of each library routine in the reference section lists the routine's return value(s).

Some library functions do not have a set error return. These include functions that return nothing and functions whose range of return values makes it impossible to return a unique error value.

To aid in error handling, some functions set the value of a global variable named errno. If the reference description of a routine states that it sets the errno variable, you can use errno in two ways:

- Compare errno to the values defined in the header file ERRNO.H.
- Handle errno with the perror or strerror library routine. The perror routine prints a system error message to the standard error (stderr). The strerror routine stores the same information in a string for later use.

When you use errno, perror, and strerror, remember that the value of errno reflects the error value for the last call that set errno. To avoid confusion, you should always test the return value to verify that an error actually occurred. Once you determine that an error has occurred, use strerror or perror immediately. Otherwise, the value of errno may be changed by intervening calls.

Library math routines set errno by calling the _matherr or _matherrl library routine. To handle math errors differently from these routines, you can write your own routine and name it _matherr or _matherrl. Your routine must follow the rules listed in the _matherr help topic.

The ferror library routine allows you to check for errors in stream input/output operations. This routine checks whether an error indicator has been set for a given stream. Closing or rewinding the stream automatically clears the error indicator. You can also reset the error indicator by calling the clearerr library routine.

The feof library routine tests for end-of-file on a given stream. You can check an end-of-file condition in low-level input and output with the _eof routine or when a _read operation returns 0 as the number of bytes read.

The _grstatus library routine enables you to check for errors after calling certain graphics library operations.

Operating-System Considerations

The library routines listed in this section behave differently under different operating-system versions.

Routine	Restrictions
_locking, _sopen, _fsopen	These routines are effective only in MS-DOS versions 3.0 and later.
_dosexterr	This routine provides error handling for system call 0x59 (get extended error) in MS-DOS versions 3.0 and later.
_dup, _dup2	These routines can cause unexpected results in MS-DOS versions earlier than 3.0. If you use them to create a duplicate file handle for stdin, stdout, stderr, _stdaux, or _stdprn, calling the _close function with one handle causes errors in later I/O operations that use the other handle. This anomaly does not occur in MS-DOS versions 3.0 and later.
_exec, _spawn	When using these families of functions under MS-DOS versions earlier than 3.0, the value of the *arg0* argument (or *argv*[0] to the child process) is not available; a null string ("") is stored in that position instead. In MS-DOS versions 3.0 and later, the *arg0* argument contains the complete command path.

Microsoft C/C++ defines global variables that indicate the version of the current operating system. You can use these to determine the operating-system version in which a program is executing.

Using Huge Arrays

In programs that use small, compact, medium, and large memory models, the compiler enables you to use arrays exceeding the 64K (kilobyte) limit of physical memory in these models by explicitly declaring the arrays as __huge. However, generally, you cannot pass huge pointers as arguments to run-time library functions. In the compact-model library used by compact-model programs and in the large-model library used by large-model and huge-model programs, only the functions listed below use pointer arithmetic that works with huge items:

bsearch	_fmemmove	memcmp
fread	_fmemset	memcpy
fwrite	_halloc	_memicmp
_fmemccpy	_hfree	memmove
_fmemchr	_lfind	memset
_fmemcmp	_lsearch	qsort
_fmemcpy	_memccpy	
_fmemicmp	memchr	

With this set of functions, you can read from, write to, search, sort, copy, initialize, compare, or dynamically allocate and free huge arrays; the huge array can be passed without difficulty to any of these functions in a compact-, large-, or huge-model program. The model-independent routines in the previous list (those beginning with _f) are available in all memory models.

The memset, memcpy, and memcmp library routines are available in two versions: as C functions and as intrinsic (inline) code. The function versions of these routines support huge pointers in compact and large memory models, but the intrinsic versions do not support huge pointers. (The function version of such routines generates a call to a library function, whereas the intrinsic version inserts inline code into your program.)

Expanding Wildcard Arguments

You can use either of the two MS-DOS wildcards, the question mark (**?**) and the asterisk (*), to specify filename and path arguments on the command line.

Command-line arguments are handled by a routine called _setargv, which by default does not expand wildcards into separate strings in the *argv* string array. You can replace the normal _setargv with a more powerful version that does handle wildcards by linking with the SETARGV.OBJ file.

To link with SETARGV.OBJ, use the /NOE linker option. For example:

```
cl typeit.c setargv /link /NOE
```

The wildcards are expanded in the same manner as in MS-DOS commands. (See your MS-DOS user's guide if you are unfamiliar with wildcards.) Enclosing an argument in double quotation marks (" ") suppresses the wildcard expansion. Within quoted arguments, you can represent quotation marks literally by preceding the double-quotation-mark character with a backslash (\). If no matches are found for the wildcard argument, the argument is passed literally.

Suppressing Command-Line Processing

If your program does not take command-line arguments, you can save a small amount of space by suppressing use of the library routine that performs command-line processing. This routine is called _setargv. To suppress its use, define a routine that does nothing in the same file that contains the main function, and name it _setargv. The call to _setargv will be satisfied by your definition of _setargv, and the library version will not be loaded.

Similarly, if you never access the environment table through the *envp* argument, you can provide your own empty routine to be used in place of _setenvp, the environment-processing routine.

If your program makes calls to the spawn or exec family of routines in the C run-time library, you should not suppress the environment-processing routine, because this routine is used to pass an environment from the parent process to the child process.

Parsing Command-Line Arguments

Microsoft C startup code uses the following rules when parsing arguments given on the MS-DOS command line:

- Arguments are delimited by white space, which is either a space or a tab.
- A string surrounded by double quotation marks is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.. Note that the caret (^) is not recognized as an escape character or delimiter.
- A double quotation mark preceded by a backslash, \", is interpreted as a literal double quotation mark (").
- Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
- If an even number of backslashes is followed by a double quotation mark, one \ is placed in the *argv* array for every \\ pair, and the " is interpreted as a string delimiter.
- If an odd number of backslashes is followed by a double quotation mark, one is placed in the *argv* array for every \ pair, and the " is escaped by the \ remaining, causing a literal " to be placed in *argv*.

The following list shows the interpreted result passed to `argv` for several examples of command-line arguments. The output listed in the second, third, and fourth columns is from the ARGS.C program.

Command-Line Input	argv[1]	argv[2]	argv[3]
`"a b c" d e`	`a b c`	`d`	`e`
`"ab\"c" "\\" d`	`ab"c`	`\`	`d`
`a\\\b d"e f"g h`	`a\\\b`	`de fg`	`h`
`a\\\"b c d`	`a\"b`	`c`	`d`
`a\\\\"b c" d e`	`a\\b c`	`d`	`e`

Reducing Text-Only Programs

You can reduce the executable-file size by about 8K or 10K for a program that uses only the text-mode functions from GRAPHICS.LIB.

To do so, link your program with TXTONLY.OBJ. If you explicitly name GRAPHICS.LIB in the compile or link command, place it after TXTONLY.OBJ on the command line. Use the /NOE option to avoid multiple symbol definitions. For example:

```
CL TEST.C NOGRAPH /LINK [ GRAPHICS.LIB ] /NOE
```

A program built with TXTONLY.OBJ cannot use graphics modes or graphics functions. It cannot try to change the palette. If the program tries to enter a graphics mode, _setvideomode will return an error.

Controlling File Mode

Most C programs use one or more data files for input and output. Data files are ordinarily processed in text mode, in which carriage-return/linefeed (CR/LF) combinations are translated into a single linefeed (LF) character upon input. LF characters are translated to CR/LF combinations upon output.

In some cases, you may want to process files without making these translations. In binary mode, CR/LF translations are suppressed.

Standard library routines, such as fopen or _open, give you the option of overriding the default mode when you open a particular file. You can also change the default mode for an entire program from text to binary mode. Do this by linking your program with the file BINMODE.OBJ, which is supplied as part of your C compiler software. Add the path of BINMODE.OBJ to the list of object file names when you link your program. For example:

```
CL MYPROG.C BINMODE.OBJ /LINK /NOE
```

When you link with BINMODE.OBJ, all files opened in your program default to binary mode, with the exceptions of stdin, stdout, and stderr. However, linking with BINMODE.OBJ does not force you to process all data files in binary mode. You still have the option of overriding the default mode when you open the file.

Use the _setmode library function to change the default mode of stdin, stdout, or stderr from text to binary, or the default mode of _stdaux or _stdprn from binary to text. The _setmode function can change the current mode for any file and is primarily used for changing the modes of stdin, stdout, stderr, _stdaux, and _stdprn, which are not explicitly opened by users.

Supressing Null-Pointer Checks

An error-checking routine (_nullcheck) is automatically invoked after your program has terminated to determine whether the contents of the NULL segment have changed. If they have, the routine displays the following error message:

```
run-time error R6001
- null pointer assignment
```

This error does not cause your program to terminate. The error message is displayed following normal termination of the program.

The NULL segment is a location in low memory that is normally not used. If the contents of the NULL segment change during program execution, it means the program has written to this area, usually by an inadvertent assignment through a null pointer.

The null-pointer error message reflects a potentially serious program error. Although the program may appear to operate correctly, it is likely to cause problems and may fail to run in a different operating environment.

The _nullcheck library routine works only for near pointers and is therefore not useful for the compact, large, and huge memory models. Its source code (CHKSUM.ASM) is included as part of the startup code that is provided with the compiler.

Controlling Stack and Heap Allocation

You can let the heap allocate memory from unused stack space by linking your program with VARSTCK.OBJ. This file allows the near-memory allocation functions to allocate items in unused stack space if they run out of other memory. The near memory allocation functions are _nmalloc, _nrealloc, _nexpand, and _ncalloc in all memory models, and malloc, realloc, _expand, and calloc in small data models (small and medium memory models).

Programs compiled and linked under Microsoft C/C++ run with a fixed stack size (default size is 2K). The stack resides above static data, and the near heap uses space left above the stack. However, for some programs a fixed-stack model may not be ideal. Having the stack and heap compete for space may be more appropriate. Use VARSTCK.OBJ to do this. When the heap runs out of memory, it uses available stack space until it reaches the top of the stack.

Be aware of the following:

- The stack cannot grow beyond the last heap item allocated in the stack or, if no heap items are in the stack, beyond the size set at link time.
- Once any part of stack memory is incorporated into the near heap, it can never be used as stack space again.
- While the heap can employ unused stack space, the stack cannot take unused heap space. When using VARSTCK.OBJ, be wary of suppressing stack checking, which is done with the #check_stack pragma or with the /Gs or /Ox option. Stack overflow can occur more easily in programs that suppress stack checking, possibly causing errors that are difficult to detect.

The following command line compiles TEST.C, then links the resulting object module with VARSTCK.OBJ. Use the /NOE option to avoid multiple symbol definitions. For example:

```
CL TEST.C VARSTCK /LINK /NOE
```

scanf

#include <stdio.h>

Syntax int scanf(const char **format* [*,argument*]*...*);

Parameter	**Description**
format	Format control
argument	Optional argument

The scanf function reads data from the standard input stream stdin into the locations given by *argument*. Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in *format*. The format controls the interpretation of the input fields. The format can contain one or more of the following:

- White-space characters: blank (' '); tab ('\t'); or newline ('\n'). A white-space character causes scanf to read, but not store, all consecutive white-space characters in the input up to the next non-white-space character. One white-space character in the format matches any number (including 0) and combination of white-space characters in the input.
- Non-white-space characters, except for the percent sign (%). A non-white-space character causes scanf to read, but not store, a matching non-white-space character. If the next character in stdin does not match, scanf terminates.
- Format specifications, introduced by the percent sign (%). A format specification causes scanf to read and convert characters in the input into values of a specified type. The value is assigned to an argument in the argument list. See scanf Format Specifiers for more information.

The format is read from left to right. Characters outside format specifications are expected to match the sequence of characters in stdin; the matching characters in stdin are scanned but not stored. If a character in stdin conflicts with the format specification, scanf terminates. The character is left in stdin as if it had not been read.

When the first format specification is encountered, the value of the first input field is converted according to this specification and stored in the location that is specified by the first *argument*. The second format specification causes the second input field to be converted and stored in the second *argument*, and so on through the end of the format string.

An input field is defined as all characters up to the first white-space character (space, tab, or newline), or up to the first character that cannot be converted according to the format specification, or until the field width (if specified) is reached. If there are too many arguments for the given specifications, the extra arguments are evaluated but ignored. The results are unpredictable if there are not enough arguments for the format specification. See Input Fields for more information.

See Type Characters for a description of the type characters and their meanings.

Return Value

The scanf function returns the number of fields that were successfully converted and assigned. The return value may be less than the number requested in the call to scanf. The return value does not include fields that were read but not assigned.

The return value is EOF if the end-of-file or end-of-string is encountered in the first attempt to read a character.

fscanf
printf
sscanf
vfprintf
vprintf
vsprintf
Format Specifiers
scanf Prefixes
scanf Input Prefixes
scanf Type Characters

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN

```
/* SCANF.C: This program receives formatted input using scanf. */
#include <stdio.h>
void main( void )
{
   int   i;
   float fp;
   char  c, s[81];
   int   result;
   printf( "Enter an integer, a floating-point number, "
           "a character and a string:\n" );
   result = scanf( "%d %f %c %s", &i, &fp, &c, s );
   printf( "\nThe number of fields input is %d\n", result );
   printf( "The contents are: %d %f %c %s\n", i, fp, c, s );
}
```

scanf Format Specifiers

A format specification has the following form:

%[*] [*width*] [{*F* | N}] [{h | l}]*type*

Each field of the format specification is a single character or a number signifying a particular format option. The *type* character, which appears after the last optional format field, determines whether the input field is interpreted as a character, a string, or a number. The simplest format specification contains only the percent sign and a *type* character (for example, `%s`).

See Prefixes and Types for more information.

Each field of the format specification is discussed in detail below. If a percent sign (%) is followed by a character that has no meaning as a format-control character, that character and the following characters (up to the next percent sign) are treated as an ordinary sequence of characters, that is, a sequence of characters that must match the input. For example, to specify that a percent-sign character is to be input, use `%%`.

An asterisk (*) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified type. The field is scanned but not stored.

The *width* is a positive decimal integer controlling the maximum number of characters to be read from stdin. No more than *width* characters are converted and stored at the corresponding *argument*. Fewer than *width* characters may be read if a white-space character (space, tab, or newline) or a character that cannot be converted according to the given format occurs before *width* is reached.

scanf Prefixes

%[*] [*width*] [**{**F | N**}**] [**{**h | l**}**]*type*

The optional F and N prefixes allow the user to specify whether the argument is far or near, respectively. F should be prefixed to an *argument* pointing to a far object, while N should be prefixed to an *argument* pointing to a near object. Note also that the F and N prefixes are not part of the ANSI definition for scanf, but are instead Microsoft extensions, which should not be used when ANSI portability is desired.

The optional prefix l indicates that the long version of the following type is to be used, while the prefix h indicates that the short version is to be used. The corresponding *argument* should point to a long or double object (with the l character) or a short object (with the h character). The l and h modifiers can be used with the d, i, n, o, x, and u type characters. The l modifier can also be used with the e, f, and g type characters. The l and h modifiers are ignored if specified for any other type.

For scanf, N and F refer to the "distance" to the object being read in (near or far) and h and l refer to the "size" of the object being read in (16-bit short or 32-bit long). The list below clarifies this use of N, F, l, and h:

Program Code	**Action**
scanf(`"%Ns", &x`);	Read a string into near memory
scanf(`"%Fs", &x`);	Read a string into far memory
scanf(`"%Nd", &x`);	Read an int into near memory
scanf(`"%Fd", &x`);	Read an int into far memory
scanf(`"%Nld", &x`);	Read a long int into near memory
scanf(`"%Fld", &x`);	Read a long int into far memory
scanf(`"%Nhp", &x`);	Read a 16-bit pointer into near memory
scanf(`"%Nlp", &x`);	Read a 32-bit pointer into near memory
scanf(`"%Fhp", &x`);	Read a 16-bit pointer into far memory
scanf(`"%Flp", &x`);	Read a 32-bit pointer into far memory

Note that %[a-z] and %[z-a] are interpreted as equivalent to %[abcde...z]. This is a common scanf extension, but note that it is not required by the ANSI standard.

To store a string without storing a terminating null character ('\0'), use the specification %*n*c, where *n* is a decimal integer. In this case, the c type character indicates that the argument is a pointer to a character array. The next *n* characters are read from the input stream into the specified location, and no null character ('\0') is appended. If *n* is not specified, the default value for it is 1.

scanf Type Characters

%[*] [*width*] [**{**F | N**}**] [**{**h | l**}**]*type*

Char-acter	Type of Input Expected	Type of Argument
d	Decimal integer	Pointer to int
o	Octal integer	Pointer to int
x	Hexadecimal integer	Pointer to int
i	Decimal, hexadecimal, or octal integer	Pointer to int
u	Unsigned decimal integer	Pointer to unsigned int
e, E f g, G	Floating-point value consisting of an optional sign (+ or –), a series of one or more decimal digits containing a decimal point, and an optional exponent ("e" or "E") followed by an optionally signed integer value.	Pointer to float
c	Character. White-space characters that are ordinarily skipped are read when c is specified; to read the next non-white-space character, use `%1s`.	Pointer to char
s	String	Pointer to character array large enough for input field plus a terminating null character ('\0'), which is automaticall y appended.
n	No input read from stream or buffer.	Pointer to int, into which is stored the number of characters successfully read from the stream or buffer up to that point

		in the current call to scanf.
p	Value in the form *xxxx*:*yyyy*, where the digits *x* and *y* are uppercase hexadecimal digits.	Pointer to far pointer to void

To read strings not delimited by space characters, a set of characters in brackets ([ ]) can be substituted for the s (string) type character. The corresponding input field is read up to the first character that does not appear in the bracketed character set. If the first character in the set is a caret (^), the effect is reversed: the input field is read up to the first character that does appear in the rest of the character set.

Input Fields

The scanf function scans each input field, character by character. It may stop reading a particular input field before it reaches a space character for a variety of reasons:

- the specified width has been reached;
- the next character cannot be converted as specified;
- the next character conflicts with a character in the control string that it is supposed to match; or
- the next character fails to appear in a given character set.

For whatever reason, when scanf stops reading an input field, the next input field is considered to begin at the first unread character. The conflicting character, if there is one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on stdin.

_scrolltextwindow

#include <graph.h>

Syntax void __far _scrolltextwindow(short *lines*);

Parameter	Description
lines	Number of lines to scroll

The _scrolltextwindow function scrolls a text window (previously defined by the _settextwindow function). The *lines* argument specifies the number of lines to scroll. A positive value of *lines* scrolls the window up (the usual direction); a negative value scrolls the window down. Specifying a number larger than the height of the current text window is equivalent to calling _clearscreen(_GWINDOW). A value of 0 for *lines* has no effect on the text.

Return Value

None.

_gettextposition
_outmem
_outtext
_settextposition
_settextwindow

Standards: None
16-Bit: MS-DOS, QWIN

```
/* SCRTXWIN.C: This program displays text in text
 * windows and then scrolls, inserts, and deletes lines.
 */
#include <stdio.h>
#include <conio.h>
#include <graph.h>
void deleteline( void );
void insertline( void );
void status( char *msg );
void main( void )
{
   short row;
   char  buf[40];
   /* Set up screen for scrolling, and put text window around
   scroll  area. */ _settextrows( 25 );
   _clearscreen( _GCLEARSCREEN );
   for( row = 1; row <= 25; row++ )
   {
      _settextposition( row, 1 );
      sprintf( buf, "Line %c             %2d", row + 'A' - 1, row );
      _outtext( buf );
   }
   _getch();
   _settextwindow( 1, 1, 25, 10 );
   /* Delete some lines. */ _settextposition( 11, 1 );
   for( row = 12; row < 20; row++ )
      deleteline();
   status( "Deleted 8 lines" );
   /* Insert some lines. */ _settextposition( 5, 1 );
   for( row = 1; row < 6; row++ )
      insertline();
   status( "Inserted 5 lines" );
   /* Scroll up and down. */
   _scrolltextwindow( -7 );
   status( "Scrolled down 7 lines" );
   _scrolltextwindow( 5 );
   status( "Scrolled up 5 lines" );
   _setvideomode( _DEFAULTMODE );
}
/* Delete lines by scrolling them off the top of the current text
 * window. Save and restore original window.
 */
void deleteline()
{
   short left, top, right, bottom;
   struct _rccoord rc;
   _gettextwindow( &top, &left, &bottom, &right );
   rc = _gettextposition();
   _settextwindow( rc.row, left, bottom, right );
   _scrolltextwindow( _GSCROLLUP );
   _settextwindow( top, left, bottom, right );
   _settextposition( rc.row, rc.col );
}
/* Insert some lines by scrolling in blank lines from the top of the
 * current text window. Save and restore original window.
 */
 void insertline()
{
   short left, top, right, bottom;
   struct _rccoord rc;
```

```
   _gettextwindow( &top, &left, &bottom, &right );
   rc = _gettextposition();
   _settextwindow( rc.row, left, bottom, right );
   _scrolltextwindow( _GSCROLLDOWN );
   _settextwindow( top, left, bottom, right );
   _settextposition( rc.row, rc.col );
}
/* Display and clear status in its own window. */
void status( char *msg )
{
   short left, top, right, bottom;
   _gettextwindow( &top, &left, &bottom, &right );
   _settextwindow( 1, 50, 2, 80 );
   _outtext( msg );
   _getch();
   _clearscreen( _GWINDOW );
   _settextwindow( top, left, bottom, right );
}
```

_searchenv

#include <stdlib.h>

Syntax void _searchenv(const char **filename*, const char **varname*, char **pathname*);

Parameter	Description
filename	Name of file to search for
varname	Environment to search
path	Buffer to store complete path

The _searchenv routine searches for the target file in the specified domain. The *varname* variable can be any environment variable that specifies a list of directory paths, such as PATH, LIB, INCLUDE, or other user-defined variables. The _searchenv function is case-sensitive, so the *varname* variable should match the case of the environment variable.

The routine first searches for the file in the current working directory. If it doesn't find the file, it next looks through the directories specified by the environment variable.

If the target file is found in one of the directories, the newly created path is copied into the buffer pointed to by *path*. You must ensure that there is sufficient space for the constructed path. If the *filename* file is not found, *path* will contain an empty null-terminated string.

Return Value

The _searchenv function does not return a value.

getenv
_putenv

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* SEARCHEN.C: This program searches for a file in
 * a directory specified by an environment variable.
 */
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   char pathbuffer[_MAX_PATH];
   char searchfile[] = "CL.EXE";
   char envvar[] = "PATH";
   /* Search for file in PATH environment variable: */
   _searchenv( searchfile, envvar, pathbuffer );
   if( *pathbuffer != '\0' )
      printf( "Path for %s: %s\n", searchfile, pathbuffer );
   else
      printf( "%s not found\n", searchfile );
}
```

_segread

#include <dos.h>

Syntax void _segread(struct _SREGS **segregs*);

Parameter	**Description**
segregs	Segment-register values

The _segread function fills the structure pointed to by *segregs* with the current contents of the segment registers. The _SREGS union is described in the reference section for _int86x. This function is intended to be used with the _intdosx and _int86x functions to retrieve segment-register values for later use.

Return Value

None.

_FP_SEG
_intdosx
_int86x

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* SEGREAD.C: This program gets the
 * current segment values with _segread.
 */
#include <dos.h>
#include <stdio.h>
void main( void )
{
   struct   _SREGS segregs;
   unsigned cs, ds, es, ss;
   /* Read the segment register values */ _segread( &segregs );
   cs = segregs.cs;
   ds = segregs.ds;
   es = segregs.es;
   ss = segregs.ss;
   printf( "CS = 0x%.4x    DS = 0x%.4x    ES = 0x%.4x    SS = 0x%.4x\n", cs, ds, es,
ss );
}
```

_selectpalette

#include <graph.h>

Syntax short __far _selectpalette(short *number*);

Parameter	**Description**
number	Palette number

The _selectpalette function works only under the video modes _MRES4COLOR, _MRESNOCOLOR, and _ORESCOLOR. A CGA palette consists of a selectable background color (Color 0) and three set colors. Under the _MRES4COLOR mode, the *number* argument selects one of the four predefined palettes shown in the following table.

	----------Color Index----------		
Palette Number	**Color 1**	**Color 2**	**Color 3**
0	Green	Red	Brown
1	Cyan	Magenta	White
2	Lt. green	Lt. red	Yellow
3	Lt. cyan	Lt. magenta	Bright white

The _MRESNOCOLOR video mode is used with black-and-white displays, producing palettes consisting of various shades of gray. It will also produce color when used with a color display. The number of palettes available depends upon whether a CGA or EGA hardware package is employed. Under a CGA configuration, only the palettes shown in the following table are available. Note that although four palette numbers are listed, palettes 0 and 1 are identical, as are palettes 2 and 3.

	----------Color Index----------		
Palette Number	**Color 1**	**Color 2**	**Color 3**
0	Blue	Red	White
1	Blue	Red	White
2	Lt. blue	Lt. red	Bright white
3	Lt. blue	Lt. red	Bright white

Under the EGA configuration, the three palettes shown in the following table are available in the _MRESNOCOLOR video mode. Note that although four palette numbers are listed, palettes 1 and 3 are identical.

	----------Color Index----------		
Palette Number	**Color 1**	**Color 2**	**Color 3**
0	Green	Red	Brown
1	Cyan	Magenta	White
2	Lt. green	Lt. red	Yellow
3	Cyan	Magenta	White

You can use the _ORESCOLOR high resolution video mode for the Olivetti graphics adapters found in most Olivetti computers and in the AT&T 6300 series computers. In _ORESCOLOR mode, an argument number in the range 0–15 selects one of the colors listed in the following table. The background color is always black in this mode.

Index	Color	Index	Color
0	Black	8	Dark Grey
1	Blue	9	Lt. Blue
2	Green	10	Lt. Green
3	Cyan	11	Lt. Cyan
4	Red	12	Lt. Red
5	Magenta	13	Lt. Magenta
6	Brown	14	Yellow
7	White	15	Bright White

Return Value

The function returns the value of the previous palette. There is no error return.

_getvideoconfig
_remappalette
_setbkcolor
_setvideomode

Standards: None
16-Bit: MS-DOS, QWIN

```
/* SELPAL.C: This program changes the current CGA palette. */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <graph.h>
long bkcolor[8] = { _BLACK,  _BLUE,     _GREEN,  _CYAN,
                    _RED,    _MAGENTA,  _BROWN,  _WHITE };
char *bkname [] = { "BLACK", "BLUE",    "GREEN", "CYAN",
                    "RED",   "MAGENTA", "BROWN", "WHITE" };
void main( void )
{
   short i, j, k;
   if ( !_setvideomode( _MRES4COLOR ) )
   {
      printf( "No palettes available" );
      exit( 1 );
   }
   for( i = 0; i < 4; i++ )          /* Palette loop           */
   {
      _selectpalette( i );
      for( k = 0; k < 8; k++ )       /* Background color  loop */
      {
         _clearscreen( _GCLEARSCREEN );
         _setbkcolor( bkcolor[k] );
         _settextposition( 1, 1 );
         printf( "Background: %s\tPalette: %d", bkname[k], i );
         for( j = 1; j < 4; j++ )       /* Foreground color loop  */
         {
            _setcolor( j );
            _ellipse( _GFILLINTERIOR, 100, j * 30, 220, 80 + (j * 30) );
         }
         _getch();
      }
   }
    _setvideomode( _DEFAULTMODE );
   exit( 0 );
}
```

_setactivepage

#include <graph.h>

Syntax short __far _setactivepage(short *page*);

Parameter	**Description**
page	Memory page number

For hardware and mode configurations with enough memory to support multiple screen pages, _setactivepage specifies the area in memory in which output is written. The *page* argument selects the current active page. The default page number is 0.

Screen animation can be done by alternating the graphics pages displayed. Use the _setvisualpage function to display a completed graphics or text page while executing graphics statements in another active page.

These functions can also be used to control text output if you use the text functions _gettextcursor, _settextcursor, _outtext, _settextposition, _gettextposition, _settextcolor, _gettextcolor, _settextwindow, and _wrapon instead of the standard C-language I/O functions.

The CGA hardware configuration has only 16K of RAM available to support multiple video pages, and only in the text mode. The EGA and VGA configurations may be equipped with up to 256K of RAM for multiple video pages in graphics mode.

This function works differently in QuickWin applications. For specific information, see *Programming Techniques*.

Return Value

If successful, the function returns the page number of the previous active page. If the function fails, it returns a negative value.

_getactivepage
_getvisualpage
_setvisualpage

Standards: None
16-Bit: MS-DOS, QWIN

```
/* PAGE.C illustrates video page functions including:
 *  _getactivepage  _getvisualpage
 *  _setactivepage  _setvisualpage
*/
#include <conio.h>
#include <graph.h>
#include <stdlib.h>
void main( void )
{
   short  oldvpage, oldapage, page;
   struct _videoconfig vc;
   _getvideoconfig( &vc );
   if( vc.numvideopages < 4 )
      exit( 1 );              /* Fail for or monochrome. */
   oldapage  = _getactivepage();
   oldvpage  = _getvisualpage();
   _displaycursor( _GCURSOROFF );
   /* Draw arrows in different place on each page. */
   for( page = 0; page < vc.numvideopages; page++ )
   {
      _setactivepage( page );
      _setvisualpage( page );
      _clearscreen( _GCLEARSCREEN );
      _settextposition( 12, 10 * page );
      _outtext( ">>>>>>>>>>>" );
   }
   while( !_kbhit() )
      /* Cycle through pages 1 to 3 to show moving image. */
      for( page = 0; page < vc.numvideopages; page++ )
         _setvisualpage( page );
   _getch();
   /* Restore original page (normally 0) to restore screen. */
   _setactivepage( oldapage );
   _setvisualpage( oldvpage );
   _displaycursor( _GCURSORON );
}
```

_setbkcolor

#include <graph.h>

Syntax long __far _setbkcolor(long *color*);

Parameter	**Description**
color	Desired color

The _setbkcolor function sets the current background color to the color value *color*.

In a color text mode (such as _TEXTC80), _setbkcolor accepts (and _getbkcolor returns) a color index. The value for the default colors is given in a table in the description of the _settextcolor function. For example, _setbkcolor(2L) sets the background color to color index 2. The actual color displayed depends on the palette mapping for color index 2. The default is green in a color text mode.

In a color graphics mode (such as _ERESCOLOR), _setbkcolor accepts (and _getbkcolor returns) a color value. The value for the background color is given by the manifest constants defined in the GRAPH.H include file. For example, _setbkcolor(_GREEN) sets the background color in a graphics mode to green. These manifest constants are provided as a convenience in defining and manipulating the most common colors. The actual range of colors is, in general, much greater.

In general, whenever a color argument is long, it refers to a color value, and whenever it is short, it refers to a color index. The two exceptions are _setbkcolor and _getbkcolor.

Because the background color is color index 0, the _remappalette function will act identically to the _setbkcolor function. Unlike _remappalette, however, _setbkcolor does not require an EGA or VGA environment.

In a text mode, the _setbkcolor function does not affect anything already appearing on the display; only the subsequent output is affected. In a graphics mode, it immediately changes all background pixels.

This function works differently in QuickWin applications. For specific information, see *Programming Techniques*.

Return Value

In text modes, _setbkcolor returns the color index of the old background color. In graphics modes, _setbkcolor returns the old color value of color index 0. There is no error return. Use the _grstatus function to check the status after a call to _setbkcolor.

_getbkcolor
_grstatus
_remappalette
_selectpalette

Standards: None
16-Bit: MS-DOS, QWIN

setbuf

#include <stdio.h>

Syntax void setbuf(FILE **stream*, char **buffer*);

Parameter	**Description**
stream	Pointer to FILE structure
buffer	User-allocated buffer

The setbuf function allows the user to control buffering for *stream*. The *stream* argument must refer to an open file that has not been read or written. If the *buffer* argument is NULL, the stream is unbuffered. If not, the buffer must point to a character array of length BUFSIZ, where BUFSIZ is the buffer size as defined in STDIO.H. The user-specified buffer, instead of the default system-allocated buffer for the given stream, is used for I/O buffering.

The stderr and (in MS-DOS only) _stdaux streams are unbuffered by default, but can be assigned buffers with setbuf.

The setbuf function has been subsumed by the setvbuf function, which should be the preferred routine for new code. The setbuf function is retained for compatibility with existing code.

Return Value

None.

fclose
fflush
fopen
setvbuf

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* SETBUF.C: This program first opens files named DATA1 and
 * DATA2. Then it uses setbuf to give DATA1 a user-assigned
 * buffer and to change DATA2 so that it has no buffer.
 */
#include <stdio.h>
void main( void )
{
   char buf[BUFSIZ];
   FILE *stream1, *stream2;
   if( ((stream1 = fopen( "data1", "a" )) != NULL) &&
       ((stream2 = fopen( "data2", "w" )) != NULL) )
   {
      /* "stream1" uses user-assigned buffer: */
      setbuf( stream1, buf );
      printf( "stream1 set to user-defined buffer at: %Fp\n", buf );
      /* "stream2" is unbuffered              */
      setbuf( stream2, NULL );
      printf( "stream2 buffering disabled\n" );
      _fcloseall();
   }
}
```

_setcliprgn

#include <graph.h>

Syntax void __far _setcliprgn(short *x1*, short *y1*, short *x2*, short *y2*);

Parameter	Description
x1, *y1*	Upper-left corner of clip region
x2, *y2*	Lower-right corner of clip region

The _setcliprgn function limits the display of subsequent graphics output and font text output to an area of the screen called the "clipping region." The physical points (*x1*, *y1*) and (*x2*, *y2*) are the diagonally opposed sides of a rectangle that defines the clipping region. This function does not change the view coordinate system. Rather, it merely masks the screen.

Note that the _setcliprgn function affects graphics and font text output only. To mask the screen for text output, use the _settextwindow function.

Return Value

None.

_settextwindow
_setvieworg
_setviewport
_setwindow

Standards: None
16-Bit: MS-DOS, QWIN

```
/* SCLIPRGN.C */
#include <stdlib.h>
#include <conio.h>
#include <graph.h>
void main( void )
{
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   /* Set clip region, then draw an ellipse larger than the region. */
   _setcliprgn( 0, 0, 200, 125 );
   _ellipse( _GFILLINTERIOR, 80, 50, 240, 190 );
   _getch();
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}
```

_setcolor

#include <graph.h>

Syntax short __far _setcolor(short *color*);

Parameter	**Description**
color	Desired color index

The _setcolor function sets the current color index to *color*. The color parameter is masked but always within range. The following graphics functions use the current color: _arc, _ellipse, _floodfill, _lineto, _outgtext, _pie, _polygon, _rectangle, and _setpixel.

The _setcolor function accepts a short value as an argument. It is a color index.

The default color index is the highest numbered color index in the current palette.

Note that the _setcolor function does not affect the output of the presentation-graphics functions.

Return Value

This function returns the previous color. If the function fails (e.g., if used in a text mode), it returns -1.

_arc functions
_ellipse functions
_floodfill
_getcolor
_lineto functions
_outgtext
_pie functions
_polygon functions
_rectangle functions
_selectpalette
_setpixel functions

Standards: None
16-Bit: MS-DOS, QWIN

```
/* GPIXEL.C: This program assigns different
 * colors to randomly selected pixels.
 */
#include <conio.h>
#include <stdlib.h>
#include <graph.h>
void main( void )
{
   short xvar, yvar;
   struct _videoconfig vc;
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXCOLORMODE ) )
      exit( 1 );
   _getvideoconfig( &vc );
   /* Draw filled ellipse to turn on certain pixels. */
   _ellipse( _GFILLINTERIOR, vc.numxpixels / 6, vc.numypixels / 6,
                             vc.numxpixels / 6 * 5, vc.numypixels / 6 * 5 );
   /* Draw random pixels in random colors... */
   while( !_kbhit() )
   {
      /* ...but only if they are already on (inside the ellipse). */
      xvar = rand() % vc.numxpixels;
      yvar = rand() % vc.numypixels;
      if( _getpixel( xvar, yvar ) != 0 )
      {
         _setcolor( rand() % 16 );
         _setpixel( xvar, yvar );
      }
   }
   _getch();          /* Throw away the keystroke. */
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}
```

_setfillmask

#include <graph.h>

Syntax void __far _setfillmask(unsigned char __far **mask*);

Parameter	Description
mask	Mask array

The _setfillmask function sets the current fill mask, which determines the fill pattern. The mask is an 8-by-8 array of bits in which each bit represents a pixel. A 1 bit sets the corresponding pixel to the current color, while a 0 bit leaves the pixel unchanged. The pattern is repeated over the entire fill area.

If no fill mask is set (*mask* is NULL_the default), a solid (unpatterned) fill is performed using the current color.

Return Value

None.

_ellipse functions
_floodfill
_getfillmask
_pie functions
_polygon functions
_rectangle functions

Standards: None
16-Bit: MS-DOS, QWIN

```
/* GFILLMSK.C: This program illustrates
 * _getfillmask and _setfillmask
 */
#include <conio.h>
#include <stdlib.h>
#include <graph.h>
void ellipsemask( short x1, short y1, short x2, short y2,
   const unsigned char __far *newmask );
unsigned char mask1[8] =
   { 0x43, 0x23, 0x7c, 0xf7, 0x8a, 0x4d, 0x78,  0x39 };
unsigned char mask2[8] =
   { 0x18, 0xad, 0xc0, 0x79, 0xf6, 0xc4, 0xa8,  0x23 };
char oldmask[8];
void main( void )
{
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   /* Set first fill mask and draw rectangle. */
   _setfillmask( mask1 );
   _rectangle( _GFILLINTERIOR, 20, 20, 100, 100 );
   _getch();
   /* Call routine that saves and restores mask. */
   ellipsemask( 60, 60, 150, 150, mask2 );
   _getch();
   /* Back to original mask. */
   _rectangle( _GFILLINTERIOR, 120, 120, 190, 190 );
   _getch();
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}

/* Draw an ellipse with a specified fill mask. */
void ellipsemask( short x1, short y1, short x2, short y2,
   const unsigned char __far *newmask )
{
   unsigned char savemask[8];
   _getfillmask( savemask );                    /* Save mask        */
   _setfillmask( newmask );                     /* Set new mask     */
   _ellipse( _GFILLINTERIOR, x1, y1, x2, y2 );  /* Use new mask     */
   _setfillmask( savemask );                    /* Restore original */
}
```

_setfont

#include <graph.h>

Syntax short __far _setfont(const char __far *_options_);

Parameter	Description
options	String describing font characteristics

The _setfont function finds a single font, from the set of registered fonts, that has the characteristics specified by the *options* string. If a font is found, it is made the current font. The current font is used in all subsequent calls to the _outgtext function. There can be only one active font at any time.

The *options* string is a set of characters that specifies the desired characteristics of the font. The _setfont function searches the list of registered fonts for a font matching the specified characteristics.

The characteristics that may be specified in the *options* string are shown in the list below. Characteristics specified in the *options* string are not case-sensitive or position-sensitive.

Characteristic	Description
t*'fontname'*	Typeface.
h*x*	Character height, where *x* is the number of pixels.
w*y*	Character width, where *y* is the number of pixels.
f	Find only a fixed-space font (should not be used with the p characteristic).
p	Find only a proportionally spaced font (should not be used with the f characteristic).
v	Find only a vector font (should not be used with the r characteristic).
r	Find only a raster-mapped (bitmapped) font (should not be used with the v characteristic).
b	Select a best fit font.
n*x*	Select font number *x*, where *x* is less than or equal to the value returned by the _registerfonts function. Use this option to "step through" an entire set of fonts or to save or restore a previously set font.

You can request as many options as desired, except with n*x*, which should be used alone. If mutually exclusive options are requested (such as the pair f/p or r/v), the _setfont function ignores them. There is no error detection for incompatible parameters used with n*x*.

Options can be separated by blanks in the *options* string. Any other character is ignored by _setfont.

The t (the typeface specification) in *options* is specified as a "t" followed by *fontname* in single quotation marks.

A b in the *options* field causes the _setfont routine to automatically select the "best fit" font that matches the other characteristics you have specified. If the b parameter is specified and at least one font is registered, _setfont will always be able to set a font and will return 0 to indicate success.

You can also specify a pixel width and height for fonts. If a nonexistent value is chosen for either, and the b option is specified, the _setfont function will choose the closest match. A smaller font size has precedence over a larger size. For example, if _setfont requests Courier 12 with best fit, and only Courier 10 and Courier 14 are available, _setfont will select Courier 10.

In selecting a font, the _setfont routine uses the following precedence (rated from highest precedence to lowest):

1. Pixel height
2. Typeface
3. Pixel width
4. Fixed or proportional font

If a nonexistent value is chosen for pixel height and width, the _setfont function will apply a magnification factor to a vector-mapped font to obtain a suitable font size. This automatic magnification does not apply if the r (raster-mapped font) option is specified, or if a specific typeface is requested and no best fit (b) option is specified.

If you specify the n*x* parameter, _setfont will ignore any other specified options and supply only the font number corresponding to *x*.

Note that the font functions affect only the output from the font output function _outgtext; no other run-time output functions are affected by font usage.

Return Value

The _setfont function returns an index that is suitable for use with n*x* to indicate success or a negative value to indicate an error. An error occurs if a request for a specific font fails and the b option was not specified, or if fonts have not yet been registered.

_getfontinfo
_getgtextextent
_outgtext
_registerfonts
_unregisterfonts

Standards: None

16-Bit: MS-DOS, QWIN

_setgtextvector

#include <graph.h>

Syntax struct _xycoord __far _setgtextvector(short *x*, short *y*);

Parameter	Description
x, y	Integers specifying font rotation

The _setgtextvector function sets the current orientation for font text output to the vector specified by *x* and *y*. The current orientation is used in calls to the _outgtext function.

The values of *x* and *y* define the vector that determines the direction of rotation of font text on the screen. The text-rotation options are shown below:

(x, y)	Text Orientation
(0, 0)	Unchanged
(1, 0)	Horizontal text (default)
(0, 1)	Rotated 90 degrees counterclockwise
(-1, 0)	Rotated 180 degrees
(0, -1)	Rotated 270 degrees counterclockwise

If other values are input, only the sign of the input is used. For example, (-3, 0) is interpreted as (-1, 0).

This function works differently in QuickWin applications. For specific information, see the *Programming Techniques* manual.

Return Value

The _setgtextvector function returns the previous vector in a structure of _xycoord type. If you pass the _setgtextvector function the values (0, 0), the function returns the current vector values in the _xycoord structure.

_getfontinfo
_getgtextextent
_grstatus
_outgtext
_registerfonts
_setfont
_unregisterfonts

Standards: None

16-Bit: MS-DOS, QWIN

setjmp

#include <setjmp.h>

Syntax int setjmp(jmp_buf *env*);

Parameter	**Description**
env	Variable in which environment is stored

The setjmp function saves a stack environment that can be subsequently restored using longjmp. Used together this way, setjmp and longjmp provide a way to execute a "non-local goto." They are typically used to pass execution control to error-handling or recovery code in a previously called routine without using the normal calling or return conventions.

A call to setjmp causes the current stack environment to be saved in *env*. A subsequent call to longjmp restores the saved environment and returns control to the point just after the corresponding setjmp call. All variables (except register variables) accessible to the routine receiving control contain the values they had when longjmp was called.

Warning Neither the setjmp nor the longjmp function is compatible with the C++ language.

Return Value

The setjmp function returns 0 after saving the stack environment. If setjmp returns as a result of a longjmp call, it returns the *value* argument of longjmp, or if the *value* argument of longjmp is 0, setjmp returns 1. There is no error return.

longjmp

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_setlinestyle

#include <graph.h>

Syntax void __far _setlinestyle(unsigned short *mask*);

Parameter	**Description**
mask	Desired line-style mask

Some graphics routines (_lineto, _polygon, and _rectangle) draw straight lines on the screen. The type of line is controlled by the current line-style mask.

The _setlinestyle function selects the mask used for line drawing. The *mask* argument is a 16-bit array, where each bit represents a pixel in the line being drawn. If a bit is 1, the corresponding pixel is set to the color of the line (the current color). If a bit is 0, the corresponding pixel is left unchanged. The template is repeated for the entire length of the line.

The default mask is 0xFFFF (a solid line).

This function works differently in QuickWin applications. For specific information, see the *Programming Techniques* manual.

Return Value

None.

_getlinestyle
_lineto functions
_polygon functions
_rectangle functions

Standards: None

16-Bit: MS-DOS, QWIN

setlocale

#include <locale.h>

Syntax char *setlocale(int *category*, const char **locale*);

Parameter	Description
category	Category affected by locale
locale	Name of the locale that will control the specified category

The setlocale function sets the categories specified by *category* to the locale specified by *locale*. The "locale" refers to the locality (country and language) for which certain aspects of your program can be customized. Some locale-dependent aspects include the formatting of dates and the display format for monetary values.

The setlocale function is used to set or get the program's current entire locale or simply portions of the locale information. The *category* argument specifies which portion of a program's locale information will be affected. The macros used for the *category* argument are listed below:

Category	Parts of Program Affected
LC_ALL	All categories listed below.
LC_COLLATE	The strcoll and strxfrm functions.
LC_CTYPE	The character-handling functions (except for isdigit, isxdigit, mbstowcs, and mbtowc, which are unaffected).
LC_MONETARY	Monetary formatting information returned by the localeconv function.
LC_NUMERIC	Decimal point character for the formatted output routines (such as printf), for the data conversion routines, and for the nonmonetary formatting information returned by the localeconv function.
LC_TIME	The strftime function.

The *locale* argument is a pointer to a string that specifies the name of the locale. If *locale* points to an empty string, the locale is the implementation-defined native environment. A value of "C" specifies the minimal ANSI conforming environment for C translation. This is the only locale supported in Microsoft C version 6.0, Microsoft C/C++ version 7.0, and this product.

If the *locale* argument is a null pointer, setlocale returns a pointer to the string associated with the category of the program's locale. The program's current locale setting is not changed.

Return Value

If a valid locale and category are given, setlocale returns a pointer to the string associated with the specified category for the previous locale. If the locale or category is invalid, the setlocale function returns a null pointer and the program's current locale settings are not changed.

The pointer to a string returned by setlocale can be used in subsequent calls to restore that part of the program's locale information, assuming that your program does not alter the pointer or the string. Later calls to setlocale will overwrite the string; you can use the _strdup function to save a specific locale string.

localeconv
mblen
mbstowcs
mbtowc
strcoll
strftime
strxfrm
wcstombs
wctomb
setlocale pragma

Standards: ANSI

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_setmode

#include <fcntl.h>

#include <io.h> Required only for function declarations

Syntax int _setmode (int *handle*, int *mode*);

Parameter	**Description**
handle	File handle
mode	New translation mode

The _setmode function sets to *mode* the translation mode of the file given by *handle*. The mode must be one of the following manifest constants:

_O_TEXT
Sets text (translated) mode. Carriage-return-line-feed (CR-LF) combinations are translated into a single line-feed (LF) character on input. Line-feed characters are translated into CR-LF combinations on output.

_O_BINARY
Sets binary (untranslated) mode. The above translations are suppressed.

The _setmode function is typically used to modify the default translation mode of stdin, stdout, stderr, _stdaux, and _stdprn, but can be used on any file. If _setmode is applied to the file handle for a stream, the _setmode function should be called before any input or output operations are performed on the stream.

Return Value

If successful, _setmode returns the previous translation mode. A return value of -1 indicates an error, and errno is set to one of the following values:

Value	**Meaning**
EBADF	Invalid file handle
EINVAL	Invalid *mode* argument (neither _O_TEXT nor _O_BINARY)

_creat
fopen
_open

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* SETMODE.C: This program uses _setmode to change
 * stdin from text mode to binary mode.
 */
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
void main( void )
{
   int result;
   /* Set "stdin" to have binary mode: */
   result = _setmode( _fileno( stdin ), _O_BINARY );
   if( result == -1 )
      perror( "Cannot set mode" );
   else
      printf( "'stdin' successfully changed to binary mode\n" );
}
```

_set_new_handler Functions

#include <new.h>

Syntax _PNH _set_new_handler(_PNH *pNewHandler*);

_PNH _set_nnew_handler(_PNH *pNewHandler*);

_PNH _set_fnew_handler(_PNH *pNewHandler*);

_PNHH _set_hnew_handler(_PNHH *pNewHandler*);

_PNHB _set_bnew_handler(_PNHB *pNewHandler*);

Parameter	Description
pNewHandler	Pointer to a function that you write

Use the C++ _set_new_handler function to gain control if the new operator fails to allocate memory. The run-time system automatically calls _set_new_handler when new fails.

To use _set_new_handler, you must write an exception-handling function and then pass it as an argument to _set_new_handler. To facilitate the easy declaration of this new handler, three pointer-to-function types, _PNH, _PNHH, and _PNHB, are defined in NEW.H:

_PNH
: Pointer to a function that returns type int and takes an argument of type size_t. Use size_t to specify the amount of space to be allocated.

_PNHH
: Pointer to a function that returns type int and takes two arguments—the type unsigned long and the type size_t arguments specified to the huge new operator.

_PNHB
: Pointer to a function that returns type int and takes two arguments—the type __segment and the type size_t arguments specified to the based new operator. Your function must ensure the correct binding of the segment variable to its return value.

Basically, _set_new_handler is a garbage collection scheme. The run-time system retries allocation each time your function returns a nonzero value and fails new if your function returns 0.

An occurrence of one of the _set_new_handler functions in a program registers the exception-handling function specified in the argument list with the run-time system:

```
#include <new.h>
int handle_program_memory_depletion( size_t )
{
   // Your code
}
void main( void )
{
   _set_new_handler( handle_program_memory_depletion );
   int *pi = new int[BIG_NUMBER];
}
```

You can save the function address that was last passed to the _set_new_handler function and then reinstate it at a later time:

```
_PNH old_handler = _set_new_handler( my_handler );
   // Code that requires my_handler
   _set_new_handler( old_handler )
   // Code that requires old_handler
```

The _set_new_handler function is defined in five different forms that allow you to manage the heap for five different memory models:

Prototype	Purpose
_PNH _set_new_handler(_PNH);	Default new handler
_PNH _set_nnew_handler(_PNH);	Manages the near heap
_PNH _set_fnew_handler(_PNH);	Manages the far heap
_PNHH _set_hnew_handler(_PNHH);	Manages the huge heap
_PNHB _set_bnew_handler(_PNHB);	Manages based heaps

The _set_new_handler function automatically maps to either the _set_nnew_handler or the _set_fnew_handler function, depending on the default data model.

If the default memory model is either small or medium, you can call _set_fnew_handler to manage the far heap. If the default memory model is either compact or large, you can call _set_nnew_handler to manage the near heap.

You can explicitly call the _set_hnew_handler and the _set_bnew_handler functions to manage both the huge and based heaps.

In a multithreaded environment, handlers are maintained separately for each process and thread. Each new process lacks installed handlers. Each new thread gets a copy of its parent thread's new handlers. Thus, each process and thread is in charge of its own free-store error handling.

Return Value

The _set_new_handler function returns a pointer to the allocated program memory if successful. It returns a 0 if it's unsuccessful.

_set_new_handler

Standards: None

16-Bit: MS-DOS, WIN, WIN DLL

_set_bnew_handler, _set_fnew_handler,
_set_hnew_handler, _set_nnew_handler

Standards: None

16-Bit: MS-DOS, WIN, WIN DLL

_bfreeseg
_bheapseg
calloc
delete
free
malloc
new
realloc

```
/* HANDLER.CPP: This program uses _set_new_handler to
 * print an error message if the new operator fails.
 */
#include <stdio.h>
#include <new.h>
/* Allocate memory in chunks of size MemBlock. */
const size_t MemBlock = 1024;
/* Allocate a memory block for the printf function to use in case
 * of memory allocation failure; the printf function uses malloc.
 * The failsafe memory block must be visible globally because the
 * handle_program_memory_depletion function can take one
 * argument only.
 */
char * failsafe = new char[128];
/* Declare a customized function to handle memory-allocation failure.
 * Pass this function as an argument to _set_new_handler.
 */
int handle_program_memory_depletion( size_t );
void main( void )
{
   // Register existence of a new memory handler.
   _set_new_handler( handle_program_memory_depletion );
   size_t *pmemdump = new size_t[MemBlock];
   for( ; pmemdump != 0; pmemdump = new size_t[MemBlock] );
}
int handle_program_memory_depletion( size_t size )
{
   // Release character buffer memory.
   delete failsafe;
   printf( "Allocation failed, " );
   printf( "%u bytes not available.\n", size );
   // Tell new to stop allocation attempts.
   return 0;
}
```

_setpixel Functions

#include <graph.h>

Syntax short __far _setpixel(short *x*, short *y*);

short __far _setpixel_w(double *wx*, double *wy*);

Parameter	Description
x, *y*	Target pixel
wx, *wy*	Target pixel

The _setpixel and the _setpixel_w functions set a pixel at a specified location to the current color.

The _setpixel function sets the pixel at the view-coordinate point (*x*, *y*) to the current color.

The _setpixel_w function sets the pixel at the window-coordinate point (*wx*, *wy*) to the current color.

Return Value

The function returns the previous value of the target pixel. If the function fails (for example, the point lies outside of the clipping region), it will return -1.

_getpixel functions
_setcolor

Standards: None
16-Bit: MS-DOS, QWIN

```
/* GPIXEL.C: This program assigns different
 * colors to randomly selected pixels.
 */
#include <conio.h>
#include <stdlib.h>
#include <graph.h>
void main( void )
{
   short xvar, yvar;
   struct _videoconfig vc;
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXCOLORMODE ) )
      exit( 1 );
   _getvideoconfig( &vc );
   /* Draw filled ellipse to turn on certain pixels. */
   _ellipse( _GFILLINTERIOR, vc.numxpixels / 6, vc.numypixels / 6,
                             vc.numxpixels / 6 * 5, vc.numypixels / 6 * 5 );
   /* Draw random pixels in random colors... */
   while( !_kbhit() )
   {
      /* ...but only if they are already on (inside the ellipse). */
      xvar = rand() % vc.numxpixels;
      yvar = rand() % vc.numypixels;
      if( _getpixel( xvar, yvar ) != 0 )
      {
         _setcolor( rand() % 16 );
         _setpixel( xvar, yvar );
      }
   }
   _getch();          /* Throw away the keystroke. */
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}
```

_settextcolor

#include <graph.h>

Syntax short __far _settextcolor(short *index*);

Parameter	Description
index	Desired color index

The _settextcolor function sets the current text color to the color index specified by *index*. The default text color is the same as the maximum color index for the current video mode.

The _settextcolor routine sets the color for the _outtext and _outmem functions only. It does not affect the color of the printf function or the color of text output with the _outgtext font routine. Use the _setcolor function to change the color of font output.

In text color mode, you can specify a color index in the range 0-31. The colors in the range 0-15 are interpreted as normal (non-blinking). The normal color range is defined below:

Index	Color	Index	Color
0	Black	8	Dark gray
1	Blue	9	Light blue
2	Green	10	Light green
3	Cyan	11	Light cyan
4	Red	12	Light red
5	Magenta	13	Light magenta
6	Brown	14	Yellow
7	White	15	Bright white

Blinking is selected by adding 16 to the normal color value.

In every text mode, including monochrome, _getvideoconfig returns the value 32 for the number of available colors. The value 32 indicates the range of values (0-31) accepted by the _settextcolor function. This includes sixteen normal colors (0-15) and sixteen blinking colors (16-31). Monochrome text mode has fewer unique display attributes, so some color values are redundant. However, because blinking is selected in the same manner, monochrome text mode has the same range (0-31) as other text modes.

This function works differently in QuickWin applications. For specific information, see the *Programming Techniques* manual.

Return Value

The function returns the color index of the previous text color. There is no error return. Use the _grstatus function to check the status after a call to _settextcolor.

_gettextcolor
_grstatus
_outmem
_outtext

Standards:	None
16-Bit:	MS-DOS, QWIN

_settextcursor

#include <graph.h>

Syntax short __far _settextcursor(short *attr*);

Parameter	Description
attr	Cursor attribute

The _settextcursor function sets the cursor attribute (i.e., the shape) to the value specified by *attr*. The high-order byte of *attr* determines the top line of the cursor within the character cell. The low-order byte of *attr* determines the bottom line of the cursor.

The _settextcursor function uses the same format as the BIOS routines in setting the cursor. Typical values for the cursor attribute are listed below:

Attribute	Cursor Shape
0x0707	Underline
0x0007	Full block cursor
0x0607	Double underline
0x2000	No cursor

Note that this function works only in text video modes.

This function works differently in QuickWin applications. For specific information, see the *Programming Techniques* manual.

Return Value

The function returns the previous cursor attribute, or -1 if an error occurs (such as calling the function in a graphics screen mode).

_displaycursor
_gettextcursor

Standards: None

16-Bit: MS-DOS, QWIN

```
/* DISCURS.C: This program changes the cursor
 * shape using _gettextcursor and _settextcursor,
 * and hides the cursor using _displaycursor.
 */
#include <conio.h>
#include <graph.h>
void main( void )
{
   short oldcursor;
   short newcursor = 0x007;        /* Full block cursor */
   /* Save old cursor shape and make sure cursor is on */
   oldcursor = _gettextcursor();
   _clearscreen( _GCLEARSCREEN );
   _displaycursor( _GCURSORON );
   _outtext( "\nOld cursor shape: " );
   _getch();
   /* Change cursor shape */
   _outtext( "\nNew cursor shape: " );
   _settextcursor( newcursor );
   _getch();
   /* Restore original cursor shape */
   _outtext( "\n" );
   _settextcursor( oldcursor );
}
```

_settextposition

#include <graph.h>

Syntax struct _rccoord __far _settextposition(short *row*, short *column*);

Parameter	**Description**
row, *column*	New output start position

The _settextposition function sets the current text position to the display point (*row*, *column*). The _outtext and _outmem functions (and standard console I/O routines, such as printf) output text at that point. Note that _settextposition does not affect the text position for the _outgtext function; use the _moveto function instead.

The _rccoord structure, defined in GRAPH.H, contains the following elements:

Element	**Description**
short row	Row coordinate
short col	Column coordinate

Standard console routines such as printf do not use or maintain information about the display, such as current text position, which is contained in the graphics library. Therefore, you cannot use standard console routines to output graphics text.

Return Value

The function returns the previous text position in an _rccoord structure, defined in GRAPH.H.

_gettextposition
_moveto
_outmem
_outtext
_settextwindow

Standards: None

16-Bit: MS-DOS, QWIN

_settextrows

#include <graph.h>

Syntax short __far _settextrows(short *rows*);

Parameter	**Description**
rows	Number of text rows

The _settextrows function specifies the number of screen rows to be used in text modes.

If the constant _MAXTEXTROWS is specified for the *rows* argument, the _settextrows function will choose the maximum number of rows available. In text modes, this is 50 rows on VGA, 43 on EGA, and 25 on others. In graphics modes that support 30 or 60 rows, _MAXTEXTROWS specifies 60 rows. In SVGA modes, _MAXTEXTROWS specifies the vertical resolution (as returned in a _videoconfig struct by the _getvideoconfig function) divided by 8.

This function works differently in QuickWin applications. For specific information, see the *Programming Techniques* manual.

Return Value

This function returns the numbers of rows set. The function returns 0 if an error occurs (such as calling the function in a graphics screen mode).

_getvideoconfig
_outtext
_setvideomode
_setvideomoderows

Standards: None
16-Bit: MS-DOS, QWIN

```
/* STXTROWS.C: This program attempts to set the screen
 * height. It returns an errorlevel code of 1 (fail) or
 * 0 (success) that could be tested in a batch file.
 */
#include <graph.h>
#include <stdlib.h>
void main( int argc, char **argv )
{
   short rows;
   if( (rows = (short)atoi( argv[1] )) == 0 )
   {
      _outtext( "\nSyntax: STXTROWS [ 25 | 43 | 50 ]\n" );
      exit( 1 );
   }
   /* Make sure new rows are the same as requested rows. */
   if( _settextrows( rows ) != rows )
   {
      _outtext( "\nInvalid rows\n" );
      exit( 1 );
   }
   else
      exit( 0 );
}
```

_settextwindow

#include <graph.h>

Syntax void __far _settextwindow(short *r1*, short *c1*, short *r2*, short *c2*);

Parameter	Description
r1, *c1*	Upper-left corner of window
r2, *c2*	Lower-right corner of window

The _settextwindow function specifies a window in row and column coordinates where the text output to the screen by the _outtext or _outmem function is displayed. The arguments (*r1*, *c1*) specify the upper-left corner of the text window, and the arguments (*r2*, *c2*) specify the lower-right corner of the text window.

Text is output from the top of the text window down. When the text window is full, the uppermost line scrolls up out of it.

Note that this function does not affect the output of presentation-graphics text (e.g., labels, axis marks, etc.), the output of the font display routine _outgtext, or the output of the standard I/O routine printf. Use the _setviewport function to control the display area for presentation graphics or fonts.

Return Value

None. Use the _grstatus function to check conditions of success or failure.

_gettextposition
_gettextwindow
_grstatus
_outmem
_outtext
_scrolltextwindow
_settextposition

Standards: None
16-Bit: MS-DOS, QWIN

setvbuf

#include <stdio.h>

Syntax int setvbuf(FILE **stream*, char **buffer*, int *mode*, size_t *size*);

Parameter	Description
stream	Pointer to FILE structure
buffer	User-allocated buffer
mode	Mode of buffering: _IOFBF (full buffering), _IOLBF (line buffering), _IONBF (no buffer)
size	Size of buffer

The setvbuf function allows the program to control both buffering and buffer size for *stream*. The *stream* must refer to an open file that has not been read from or written to since it was opened. The array pointed to by *buffer* is used as the buffer, unless it is NULL, and an automatically allocated buffer *size* bytes long is used.

The mode must be _IOFBF, _IOLBF, or _IONBF. If *mode* is _IOFBF or _IOLBF, then *size* is used as the size of the buffer. If *mode* is _IONBF, the stream is unbuffered and *size* and *buffer* are ignored.

Values for *mode* and their meanings are:

Type	Meaning
_IOFBF	Full buffering; that is, *buffer* is used as the buffer and *size* is used as the size of the buffer. If *buffer* is NULL, an automatically allocated buffer *size* bytes long is used.
_IOLBF	With MS-DOS, the same as _IOFBF.
_IONBF	No buffer is used, regardless of *buffer* or *size*.

The legal values for *size* are greater than 0 and less than 32,768.

Return Value

The return value for setvbuf is 0 if successful, and a nonzero value if an illegal type or buffer size is specified.

fclose
fflush
fopen
setbuf

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* SETVBUF.C: This program opens two streams named stream1
 * and stream2. It then uses setvbuf to give stream1 a
 * user-defined buffer of 1024 bytes and stream2 no buffer.
 */
#include <stdio.h>
void main( void )
{
   char buf[1024];
   FILE *stream1, *stream2;
   if( ((stream1 = fopen( "data1", "a" )) != NULL) &&
       ((stream2 = fopen( "data2", "w" )) != NULL) )
   {
      if( setvbuf( stream1, buf, _IOFBF, sizeof( buf ) ) != 0 )
         printf( "Incorrect type or size of buffer for stream1\n" );
      else
         printf( "'stream1' now has a buffer of 1024 bytes\n" );
      if( setvbuf( stream2, NULL, _IONBF, 0 ) != 0 )
         printf( "Incorrect type or size of buffer for stream2\n" );
      else
         printf( "'stream2' now has no buffer\n" );
      _fcloseall();
   }
}
```

_setvideomode

#include <graph.h>

Syntax short __far _setvideomode(short *mode*);

Parameter	**Description**
mode	Desired mode

The _setvideomode function selects a screen mode appropriate for a particular hardware/display configuration. The *mode* argument can be one of the manifest constants shown in the following two tables and defined in GRAPH.H. The table below describes only standard hardware; however, display hardware that is strictly compatible with IBM, Hercules, or Olivetti hardware should also work as described.

Mode	**Type***	**Size****	**Colors*****	**Adapter*****
_DEFAULTMODE	Mode existing at startup			
_MAXRESMODE	Highest resolution in graphics mode			
_MAXCOLORMODE	Maximum colors in graphics mode			
_TEXTBW40	BW/T	40 columns	32	CGA
_TEXTC40	C/T	40 columns	32	CGA
_TEXTBW80	BW/T	80 columns	32	CGA
_TEXTC80	C/T	80 columns	32	CGA
_MRES4COLOR	C/G	320 200	4	CGA
_MRESNOCOLOR	BW/G	320 200	4	CGA
_HRESBW	BW/G	640 200	2	CGA
_TEXTMONO	M/T	80 columns	32	MDPA
_HERCMONO*****	M/G/Hercules graphics	720 348	2	HGC
_MRES16COLOR	C/G	320 200	16	EGA
_HRES16COLOR	C/G	640 200	16	EGA
_ERESNOCOLOR	M/G	640 350	4	EGA
_ERESCOLOR	C/G	640 350	16/4	EGA
_VRES2COLOR	C/G	640 480	2	VGA
_VRES16COLOR	C/G	640 480	16	VGA
_MRES256COLOR	C/G	320 200	256	VGA
_ORESCOLOR	C/G	640 400	1 of 16	OGA

* M indicates monochrome, BW indicates monochrome, C indicates color output, T indicates text, and G indicates graphics generation.

** For text modes, size is given in characters (number of columns). For graphics modes, size is given in pixels (horizontal / vertical).

*** For monochrome displays, the number of colors is the number of attributes or shades of gray.

**** Adapters are the IBM (and compatible) Monochrome Adapter (MDPA), Color Graphics Adapter (CGA), Enhanced Graphics Adapter (EGA), Video Graphics Array (VGA), Hercules-compatible adapter (HGC), and Olivetti-compatible adapter (OGA).

***** In _HERCMONO mode, the text dimensions are 80 columns by 25 rows, with a 9 by 14 character box. The bottom two scan lines of row 25 are not visible.

Super VGA Support

Super VGA (SVGA) does not describe a standard display adapter. Instead, it refers to any VGA-compatible video adapter that also provides higher resolution modes. SVGA adapters made by different manufacturers may support different extended resolution modes.

To allow your programs to use different SVGA adapters, the graphics libraries support the Video Electronics Standards Association (VESA) interface. This standard interface accesses the extended features of different SVGA adapters. The interface is widely supported by video hardware manufacturers, which allows your graphics program to run with virtually any VESA-compliant adapter.

The _setvideomode function supports eight extended resolution modes. Some or all are available on VESA-compliant SVGA adapters: _ORES256COLOR, _VRES256COLOR, _SRES16COLOR, _SRES256COLOR, _XRES16COLOR, _XRES256COLOR, _ZRES16COLOR and _ZRES256COLOR. (These modes represent BIOS numbers 0x0100 through 0x0107, respectively, in the VESA standard.) Consult the owners manual to see which modes your adapter supports.

Having an SVGA adapter does not suffice for using one of these extended modes. You must have a display monitor that supports the higher resolution. Only the first two extended modes can be displayed on a standard VGA analog monitor. The other extended modes require special monitors.

You can use _setvideomode to select a nonstandard graphics mode specific to a particular manufacturers adapter, if the adapter is VESA-compliant. Consult your adapters owner's manual for the BIOS number of a given mode, and pass that number as the argument to _setvideomode. The BIOS number must be between 0x15 and 0x7F. Typically, these additional modes differ from the eight modes listed above only in resolution, not in number of colors. The _setvideomode function may not support all of an adapters extended modes.

VESA TSRs

Some SVGA adapters provide the VESA interface in ROM. Other adapters require that you install a TSR (terminate-and-stay resident) program. Microsoft C/C++ includes TSRs for several adapters (see the file PACKING.LST for a list of TSRs supplied). If the TSR for your adapter is not included, contact your dealer or video adapter manufacturer.

These TSRs are executable programs with names of the form *xxx*VESA.COM or *xxx*VESA.EXE. You install the TSR by running the program. You must install the TSR before you run a graphics program that uses one of the VESA extended modes. If you dont install the TSR, your SVGA adapter behaves like a standard VGA adapter.

Warning These drivers are provided on an as-is, unsupported basis, without any claim as to their correctness or suitability. Neither Microsoft nor the TSR vendor makes any representations or warranties regarding the capabilities or performance of the TSR software. Should you want to distribute any of the supplied TSRs with a software program developed using Microsoft C/C++, it is your responsibility to obtain permission from VESA and/or the TSR vendor.

Limitations of VESA Support

The graphics libraries may not work with all hardware and TSR combinations. Furthermore, VESA support has the following limitations:

- Version 1.0 of the VESA Super VGA Standard (#VS891001) is supported.
- Only color graphics modes are supported.
- Super VGA BIOS functions 2, 3, 6, 7, and 9 must be supported. See section 6 of the VESA Super VGA Standard for more information.
- Three types of hardware windows are supported: single window systems with a readable/writable 64K window beginning at A000h, dual overlapping 64K windows beginning at A000h, and dual adjacent 32K readable/writable windows beginning at A000h and A800h. See section 5.2.1 of the VESA Super

VGA Standard for more information.

- The only window size (`WinSize`) supported is 64K. Some adapters have an option of using either a 64K single-window, or a 32K double-window mode. Your adapter must be configured for 64K mode.
- The window granularity (`WinGranularity`) must be a power of 2.
- The memory model must be either 4-plane planar (16-color) or packed pixel (256-color).

Consult the VESA Super VGA Standard and your adapters owners manual for details. For a copy of the Super VGA Standard, write to the Video Electronics Standards Association (VESA) in San Jose, California.

The table below lists the manifest constants that support the Super VGA screen modes specified by VESA. Other nonstandard Super VGA modes may also be supported. Note that some, or all, of these manifest constants may be supported by graphics cards that support the VESA Super Video standard VS891001. Other modes may also be supported; a TSR driver may be required.

Mode	VESA No.	Type*	Size		Colors	Adapter
_ORES256COLOR	0x0100	C/G	640	400	256	SVGA
_VRES256COLOR	0x0101	C/G	640	480	256	SVGA
_SRES16COLOR**	0x0102	C/G	800	600	16	SVGA
_SRES256COLOR**	0x0103	C/G	800	600	256	SVGA
_XRES16COLOR***	0x0104	C/G	1024	768	16	SVGA
_XRES256COLOR***	0x0105	C/G	1024	768	256	SVGA
_ZRES16COLOR****	0x0106	C/G	1280	1024	16	SVGA
_ZRES256COLOR****	0x0107	C/G	1280	1024	256	SVGA

* C indicates color output and G indicates graphics generation.

** Requires NEC MultiSync 3D or equivalent or better.

*** Requires NEC MultiSync 4D or equivalent or better.

**** Requires NEC MultiSync 5D or equivalent or better.

Warning Do not attempt to set _SRES16COLOR, _SRES256COLOR, _XRES16COLOR, _XRES256COLOR, _ZRES16COLOR, or _ZRES256COLOR without ensuring that your monitor can safely handle that resolution. Otherwise, you may risk damaging your display monitor! Consult your owner's manual for details.

_MAXRESMODE and _MAXCOLORMODE

The special mode _MAXRESMODE selects the highest resolution available with the current hardware. _MAXCOLORMODE selects the greatest number of colors available with the current hardware. These modes fail for adapters that do not support graphics modes. They never select _SRES, _XRES, or _ZRES mode. The following table lists the video mode selected for different adapter and monitor combinations when _MAXRESMODE or _MAXCOLORMODE is specified.

Adapter/Monitor	_MAXRESMODE	_MAXCOLORMODE
MDPA	fails	fails
HGC	_HERCMONO	_HERCMONO
CGA color*	_HRESBW	_MRES4COLOR
CGA noncolor*	_HRESBW	_MRESNOCOLOR
OCGA	_ORESCOLOR	_MRES4COLOR
OEGA color	_ORESCOLOR	_ERESCOLOR
EGA color 256K	_HRES16COLOR	_HRES16COLOR
EGA color 64K	_HRES16COLOR	_HRES16COLOR
EGA ecd 256K	_ERESCOLOR	_ERESCOLOR

EGA ecd 64K	_ERESCOLOR	_HRES16COLOR
EGA mono	_ERESNOCOLOR	_ERESNOCOLOR
MCGA	_VRES2COLOR	_MRES256COLOR
VGA	_VRES16COLOR	_MRES256COLOR
OVGA	_VRES16COLOR	_MRES256COLOR
SVGA	_VRES256COLOR**	_VRES256COLOR**

* Color monitor is assumed if the startup text mode was _TEXTC80 or _TEXTC40 or if the startup mode was graphics mode. Composite or other noncolor CGA monitor is assumed if startup mode was _TEXTBW80 or _TEXTBW40.

** If _VRES256COLOR is supported by the adapter/monitor combination. If not, _MAXCOLORMODE will be either _ORES256COLOR (if supported) or _MRES256COLOR and _MAXRESMODE will be _VRES16COLOR.

Hercules Support

You must install the Hercules driver MSHERC.COM before running your program. Type MSHERC to load the driver. This can be automated by adding a line to your AUTOEXEC.BAT file.

If you have both a Hercules monochrome card and a color video card, you should install MSHERC.COM with the /H (/HALF) option. The /H option causes the driver to use one instead of two graphics pages. This prevents the two video cards from attempting to use the same memory. You do not need to use the /H option if you have only a Hercules card. See your Hercules hardware manuals for more details on compatibility.

To use a mouse, you must follow special instructions for Hercules cards in *Microsoft Mouse Programmer's Reference Guide*. (This is sold separately; it is not supplied with either this product or the mouse package.)

This function works differently in QuickWin applications. For specific information, see the *Programming Techniques* manual.

Return Value

The function returns the number of text rows if the function is successful. If an error is encountered (that is, the mode selected is not supported by the current hardware configuration), the function returns 0.

_getvideoconfig
_settextrows
_setvideomoderows

Standards: None

16-Bit: MS-DOS, QWIN

```
/* SVIDMODE.C: This program sets a video mode
 * from a string given on the command line.
 */
#include <graph.h>
#include <stdlib.h>
#include <string.h>
short modes[] = { _TEXTBW40,     _TEXTC40,      _TEXTBW80,
                  _TEXTC80,      _MRES4COLOR,   _MRESNOCOLOR,
                  _HRESBW,       _TEXTMONO,     _HERCMONO,
                  _MRES16COLOR,  _HRES16COLOR,  _ERESNOCOLOR,
                  _ERESCOLOR,    _VRES2COLOR,   _VRES16COLOR,
                  _MRES256COLOR, _ORESCOLOR
                };
char *names[] = { "TEXTBW40",     "TEXTC40",     "TEXTBW80",
                  "TEXTC80",      "MRES4COLOR",  "MRESNOCOLOR",
                  "HRESBW",       "TEXTMONO",    "HERCMONO",
                  "MRES16COLOR", "HRES16COLOR", "ERESNOCOLOR",
                  "ERESCOLOR",    "VRES2COLOR",  "VRES16COLOR",
                  "MRES256COLOR","ORESCOLOR" };
void error( char *msg );
void main( int argc, char *argv[] )
{
   short i, num = sizeof( modes ) / sizeof( short );
   if( argc < 2 )
      error( "No argument given" );
   /* If matching name found, change to corresponding mode. */
   for( i = 0; i < num; i++ )
   {
      if( !_strcmpi( argv[1], names[i] ) )
      {
         _setvideomode( modes[i] );
         _outtext( "New mode is: " );
         _outtext( names[i] );
         exit( 0 );
      }
   }
   error( "Invalid mode string" );
}
void error( char *msg )
{
   _outtext( msg );
   exit( 1 );
}
```

_setvideomoderows

#include <graph.h>

Syntax short __far _setvideomoderows(short *mode*, short *rows*);

Parameter	**Description**
mode	Desired mode
rows	Number of text rows

The _setvideomoderows function selects a screen mode for a particular hardware/display combination. The manifest constants for the screen mode are given in the description of _setvideomode. The _setvideomoderows function also specifies the number of text rows to be used in a text mode. If the constant _MAXTEXTROWS is specified for the *rows* argument, the _setvideomoderows function will choose the maximum number of rows available. In text modes, this is 50 rows on VGA, 43 on EGA, and 25 on others. In graphics modes that support 30 or 60 rows, _MAXTEXTROWS specifies 60 rows. In SVGA modes, _MAXTEXTROWS specifies the vertical resolution (as returned in a _videoconfig struct by the _getvideoconfig function) divided by 8.

This function works differently in QuickWin applications. For specific information, see the *Programming Techniques* manual.

Return Value

The _setvideomoderows function returns the numbers of rows set. The function returns 0 if an error occurs (e.g., if the mode is not supported).

_getvideoconfig
_settextrows
_setvideomode

Standards: None
16-Bit: MS-DOS, QWIN

```
/* SVMROWS.C */
#include <stdlib.h>
#include <conio.h>
#include <graph.h>
void main( void )
{
   struct _videoconfig config;
   /* Set 43-line graphics mode if available. */
   if( !_setvideomoderows( _ERESCOLOR, 43 ) )
   {
      _outtext( "EGA or VGA required" );
      exit( 1 );
   }
   _getvideoconfig( &config );
   /* Set logical origin to center and draw a rectangle. */
   _setlogorg( (short)(config.numxpixels / 2 - 1), (short)(config.numypixels / 2 -
1) );
   _rectangle( _GBORDER, -80, -50, 80, 50 );
   _getch();
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}
```

_setvieworg

#include <graph.h>

Syntax struct _xycoord __far _setvieworg(short *x*, short *y*);

Parameter	Description
x, *y*	New origin point

The _setvieworg function moves the view-coordinate origin (0, 0) to the physical point (*x*, *y*). It replaces the _setlogorg function of Microsoft C version 5.1.

The _xycoord structure, defined in GRAPH.H, contains the following elements:

Element	Description
short xcoord	*x* coordinate
short ycoord	*y* coordinate

Return Value

The function returns the physical coordinates of the previous view origin in an _xycoord structure, defined in GRAPH.H.

_getphyscoord
_getviewcoord
_getwindowcoord
_setcliprgn
_setviewport

Standards: None
16-Bit: MS-DOS, QWIN

```
/* SVORG.C: This program sets the view origin
 * to the center of the screen, then draws a
 * rectangle using the new origin.
 */
#include <stdlib.h>
#include <conio.h>
#include <graph.h>
void main( void )
{
   struct _videoconfig config;
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   _getvideoconfig( &config );
   /* Set view origin to the center of the screen. */
   _setvieworg( (short)(config.numxpixels / 2), (short)(config.numypixels / 2) );
   _rectangle( _GBORDER, -80, -50, 80, 50 );
   _getch();
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}
```

_setviewport

#include <graph.h>

Syntax void __far _setviewport(short *x1*, short *y1*, short *x2*, short *y2*);

Parameter	Description
x1, *y1*	Upper-left corner of viewport
x2, *y2*	Lower-right corner of viewport

The _setviewport function redefines the graphics viewport. The _setviewport function defines a clipping region in exactly the same manner as _setcliprgn, and then sets the view-coordinate origin to the upper-left corner of the region. The physical points (*x1*, *y1*) and (*x2*, *y2*) are the diagonally opposed corners of the rectangular clipping region. Any window transformation done with the _setwindow function applies only to the viewport and not to the entire screen. The default viewport is the entire screen.

Return Value

None. Use the _grstatus function to check for conditions of success or failure.

_grstatus
_setcliprgn
_setvieworg
_setwindow

Standards: None
16-Bit: MS-DOS, QWIN

```
/* SVIEWPRT.C: This program sets a viewport and then
 * draws a rectangle around it and an ellipse in it.
 */
#include <conio.h>
#include <stdlib.h>
#include <graph.h>
void main( void )
{
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   _setviewport( 100, 100, 200, 200 );
   _rectangle( _GBORDER, 0, 0, 100, 100 );
   _ellipse( _GFILLINTERIOR, 10, 10, 90, 90 );
   _getch();
   _setvideomode( _DEFAULTMODE );
   exit( 0 );
}
```

_setvisualpage

#include <graph.h>

Syntax short __far _setvisualpage(short *page*);

Parameter	**Description**
page	Visual page number

For hardware configurations that have enough memory to support multiple-screen pages, the _setvisualpage function selects the current visual page. The *page* argument specifies the current visual page. The default page number is 0.

This function works differently in QuickWin applications. For specific information, see the *Programming Techniques* manual.

Return Value

The function returns the number of the previous visual page. If the function fails, it returns a negative value.

_getactivepage
_getvisualpage
_setactivepage
_setvideomode

Standards: None

16-Bit: MS-DOS, QWIN

_setwindow

#include <graph.h>

Syntax short __far _setwindow(short *finvert*, double *wx1*, double *wy1*, double *wx2*, double *wy2*);

Parameter	Description
finvert	Invert flag
wx1, *wy1*	Upper-left corner of window
wx2, *wy2*	Lower-right corner of window

The _setwindow function defines a window viewport. The arguments (*wx1*, *wy1*) specify the upper-left corner of the window, and the arguments (*wx2*, *wy2*) specify the lower-right corner of the window.

The *finvert* argument specifies the direction of the coordinates. If *finvert* is TRUE, the *y* axis increases from the screen bottom to the screen top (Cartesian coordinates). If *finvert* is FALSE, the *y* axis increases from the screen top to the screen bottom (screen coordinates).

Any window transformation done with the _setwindow function applies only to the viewport and not to the entire screen.

If *wx1* equals *wx2* or *wy1* equals *wy2*, the function will fail.

Note that this function only affects output functions suffixed with _w or _wxy.

Return Value

The function returns a nonzero value if successful. If the function fails (e.g., if it is not in a graphics mode), it returns 0.

_arc functions, _ellipse functions, _getwindowcoord, _lineto functions, _pie functions, _setviewport

other functions suffixed with _w or _wxy:
floodfill_w, _getcurrentposition_w, _getimage_w and _wxy, _getpixel_w, _getviewcoord_w and _wxy, _imagesize_w and _wxy, _moveto_w, _polygon_w and _wxy, _putimage_w, _rectangle_w and _wxy, _setpixel_w

Standards: None
16-Bit: MS-DOS, QWIN

```
/* SWINDOW.C: This program illustrates translation
 * between window, view, and physical coordinates.
 * Functions used include:
 *     _setwindow        _getwindowcoord
 *     _getphyscoord     _getviewcoord_wxy
 */
#include <conio.h>
#include <stdlib.h>
#include <graph.h>
enum boolean { FALSE, TRUE };
enum display { MOVE, DRAW, ERASE };
void main( void )
{
   struct _xycoord view, phys;
   struct _wxycoord oldwin, newwin;
   struct _videoconfig vc;
   double xunit, yunit, xinc, yinc;
   short color, fintersect = FALSE, fdisplay = TRUE;
   int key;
   /* Find a valid graphics mode. */
   if( !_setvideomode( _MAXRESMODE ) )
      exit( 1 );
   _getvideoconfig( &vc );
   /* Set a window using real numbers. */
   _setwindow( FALSE, -125.0, -100.0, 125.0, 100.0 );
   /* Calculate the size of one pixel in window coordinates.
    * Then get the current window coordinates and color.
    */
   oldwin = _getwindowcoord( 1, 1 );
   newwin = _getwindowcoord( 2, 2 );
   xunit = xinc = newwin.wx - oldwin.wx;
   yunit = yinc = newwin.wy - oldwin.wy;
   newwin = oldwin = _getcurrentposition_w();
   color = _getcolor();
   while( 1 )
   {
      /* Set flag according to whether current pixel is on, then
       * turn pixel on.
       */
      if( _getpixel_w( oldwin.wx, oldwin.wy ) == color )
         fintersect = TRUE;
      else
         fintersect = FALSE;
      _setcolor( color );
      _setpixel_w( oldwin.wx, oldwin.wy );
      /* Get and test key. */
      key = _getch();
      switch( key )
      {
      case 27:                      /* ESC Quit */
         _setvideomode( _DEFAULTMODE );
         exit( 0 );
      case 32:                      /* SPACE Move no color   */
         fdisplay = MOVE;
         continue;
      case 0:                       /* Extended code - get next */
         key = _getch();
         switch( key )
         {
         case 72:                   /* UP       -y */
```

```
            newwin.wy -= yinc;
            break;
         case 77:                  /* RIGHT    +x */
            newwin.wx += xinc;
            break;
         case 80:                  /* DOWN     +y */
            newwin.wy += yinc;
            break;
         case 75:                  /* LEFT     -x */
            newwin.wx -= xinc;
            break;
         case 82:                  /* INS      Draw white */
            fdisplay = DRAW;
            continue;
         case 83:                  /* DEL      Draw black */
            fdisplay = ERASE;
            continue;
         }
         break;
      }
      /* Translate window coordinates to view, view to physical.
       * Then check physical to make sure we're on screen.
       * Update screen and position if we are. Ignore if not.
       */
      view = _getviewcoord_wxy( &newwin );
      phys = _getphyscoord( view.xcoord, view.ycoord );
      if( (phys.xcoord >= 0) && (phys.xcoord < vc.numxpixels) &&
          (phys.ycoord >= 0) && (phys.ycoord < vc.numypixels) )
      {
         /* If display on, draw to new position, else move to new. */
         if( fdisplay != MOVE )
         {
            if( fdisplay == ERASE )
               _setcolor( 0 );
            _lineto_w( newwin.wx, newwin.wy );
         }
         else
         {
            _setcolor( 0 );
            _moveto_w( newwin.wx, newwin.wy );
            /* If there was no intersect, erase old pixel. */
            if( !fintersect )
               _setpixel_w( oldwin.wx, oldwin.wy );
         }
         oldwin = newwin;
      }
      else
         newwin = oldwin;
   }
   exit( 0 );
}
```

_setwritemode

#include <graph.h>

Syntax short __far _setwritemode(short *action*);

Parameter	Description
action	Interaction with existing screen image

The _setwritemode function sets the current logical write mode, which is used when drawing lines with the _lineto, _polygon, and _rectangle functions.

The *action* argument defines the write mode. The possible values are _GAND, _GOR, _GPRESET, _GPSET, and _GXOR. See the description of the _putimage functions for more details on these manifest constants.

Return Value

The _setwritemode function returns the previous write mode, or -1 if an error occurs.

_getwritemode
_grstatus
_lineto functions
_polygon functions
_putimage functions
_rectangle functions
_setcolor
_setlinestyle

Standards: None

16-Bit: MS-DOS, QWIN

signal

#include <signal.h>

Syntax void (__cdecl *signal(int *sig*, void(__cdecl **func*) (int *sig* [, int *subcode*]))) (int *sig*);

Parameter	Description
sig	Signal value
func	Function to be executed
subcode	Optional subcode to the signal number

The signal function allows a process to choose one of several ways to handle an interrupt signal from the operating system. The signal function takes two arguments: *sig* and *func*.

The *sig* argument is the interrupt for which you want the signal function to respond; it must be one of the manifest constants (defined in SIGNAL.H) described in the following table.

Value	Mode	Meaning	Default Action
SIGABRT	Real	Abnormal termination	Terminates the calling program with exit code 3
SIGFPE	Real	Floating-point error	Terminates the calling program with exit code 3
SIGILL	Real	Illegal instruction	Terminates the calling program with exit code 3
SIGINT	Real	CTRL+C signal	Terminates the calling program with exit code 3
SIGSEGV	Real	Illegal storage access	Terminates the calling program with exit code 3
SIGTERM	Real	Termination request	Terminates the calling program

with exit
code 3

The *func* argument is either an address to s signal handler that you write or it is one of the manifest constants defined in SIGNAL.H, SIG_DFL or SIG_IGN.

If *func* is a function, it is installed as the signal handler for the given signal. The signal handler's prototype requires one formal argument, *sig*, of type int. The operating system provides the actual argument through *sig* when an interrupt occurs; the argument will be the signal that generated the interrupt. Thus, you can use the six manifest constants (SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, and SIGTERM) inside your signal handler to determine which interrupt occurred and take appropriate action.

For example, you could call the signal function twice to assign the same handler to two different signals, then test the *sig* argument inside the handler to take different actions based on the signal received.

If you are testing for floating-point exceptions (SIGFPE), the function pointed to by *func* takes an optional second argument that is one of several manifest constants of the form FPE_*xxx* (defined in FLOAT.H). When a SIGFPE signal occurs, you can test the value of the second argument to determine the type of floating-point exception and then take appropriate action. This argument and its possible values are Microsoft extensions.

For floating-point exceptions, the value of *func* is not reset upon receiving the signal. To recover from floating-point exceptions, use setjmp in conjunction with longjmp. If the function returns, the calling process resumes execution with the floating-point state of the process left undefined.

If the signal handler returns, the calling process resumes execution immediately following the point at which it received the interrupt signal. This is true regardless of the type of signal or operating mode.

Before the specified function is executed under MS-DOS version 3.x or earlier, the value of *func* is set to SIG_DFL. The next interrupt signal is treated as described for SIG_DFL, unless an intervening call to signal specifies otherwise. This allows the user to reset signals in the called function if desired.

Because signal-handler routines are usually called asynchronously when an interrupt occurs, it is possible that your signal-handler function will get control when a run-time operation is incomplete and in an unknown state. There are certain restrictions as to which functions you can use in your signal-handler routine. The list below summarizes these restrictions:

- Do not issue low-level or STDIO.H I/O routines (e.g., printf, fread).
- Do not call heap routines or any routine that uses the heap routines (e.g., malloc, _freect, _strdup, _putenv). See malloc for more information.
- Do not use any function that generates a system call (e.g., _getcwd, time).
- Do not use the longjmp function unless the interrupt is caused by a floating-point exception (i.e., *sig* is SIGFPE). In this case, the program should first reinitialize the floating-point package by means of a call to _fpreset.
- Do not use any overlay routines.

Under MS-DOS, a program must contain floating-point code if it is to trap the SIGFPE exception with the signal function. If your program does not have floating-point code and it requires the run-time library's signal-handling code, simply declare a volatile double and initialize it to zero:

```
volatile double d = 0.0f;
```

The SIGILL, SIGSEGV, and SIGTERM signals are not generated under MS-DOS. They are included for ANSI compatibility. Thus, you can set signal handlers for these signals via signal, and you can also explicitly generate these signals by calling raise.

Signal settings are not preserved in child processes created by calls to _exec or _spawn. The signal settings are reset to the default in the child process.

Return Value

The signal function returns the previous value of *func* associated with the given signal. For example, if

the previous value of *func* was SIG_IGN, the return value will be SIG_IGN.
A return value of SIG_ERR indicates an error, and errno is set to EINVAL.

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* SIGNAL.C illustrates setting up signal interrupt routines.
 * Functions illustrated include signal and raise.
 *
 * Because C I/O functions are not safe inside signal routines,
 * the code uses conditionals to use system-level MS-DOS
 * services. Another option is to set global flags and do any
 * I/O operations outside the signal handler.
 */
#include <stdio.h>
#include <conio.h>
#include <signal.h>
#include <process.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
void ctrlchandler( int sig );         /* Prototypes */
void safeout( char *str );
int  safein( void );
void main( void )
{
   int ch;
   /* Install signal handler to modify CTRL+C behavior. */
   if( signal( SIGINT, ctrlchandler ) == SIG_ERR )
   {
      fprintf( stderr, "Couldn't set SIGINT\n" );
      abort();
   }
   /* Loop prints message to screen asking user to
    * enter Cntl+C--at which point the ctrlchandler
    * signal handler takes control.
    */
   do
   {
      printf( "Press Ctrl+C to enter handler.\n" );
   }
   while( ch = _getch());   /* Discard keystokes */
}
/* A signal handler must take a single argument. The argument can be
 * tested within the handler and thus allows a single signal handler
 * to handle several different signals. In this case, the parameter
 * is included to keep the compiler from generating a warning but is
 * ignored because this signal handler only handles one interrupt:
 * SIGINT (Ctrl+C).
 */
void ctrlchandler( int sig )
{
   int c;
   char str[] = " ";
   /* Disallow CTRL+C during handler. */
   signal( SIGINT, SIG_IGN );
   safeout( "User break - abort processing (y|n)? " );
   c = safein();
   str[0] = (char)c;
   safeout( str );
   safeout( "\r\n" );
   if( (c == 'y') || (c == 'Y') )
      abort();
   else
   {
      /* The CTRL+C interrupt must be reset to our handler because
```

```
       * by default it is reset to the system handler.
       */
      signal( SIGINT, ctrlchandler );
      safeout( "Press Ctrl+C to enter handler.\r\n" );
   }
}
/* Outputs a string using system level calls. */
void safeout( char *str )
{
   union _REGS inregs, outregs;
   inregs.h.ah = 0x0e;
   while( *str )
   {
      inregs.h.al = *str++;
      _int86( 0x10, &inregs, &outregs );
   }
}
/* Inputs a character using system level calls. */
int safein()
{
   return _bios_keybrd( _KEYBRD_READ ) & 0xff;
}
```

sin Functions

#include <math.h>

Syntax double sin(double *x*);

double sinh(double *x*);

long double sinl(long double *x*);

long double sinhl(long double *x*);

Parameter	Description
x	Angle in radians

The sin and sinh functions find the sine and hyperbolic sine of *x*, respectively. The sinl and sinhl functions are the 80-bit counterparts and use an 80-bit, 10-byte coprocessor form of arguments and return values. See the long double functions for more details on this data type.

Return Value

The sin functions return the sine of *x*. If *x* is greater than or equal to 2 raised to the power of 27, a partial loss of significance in the result may occur, and sin generates a _PLOSS error. If *x* is greater than or equal to 2 raised to the power of 31, significance is completely lost, and the sin function prints a _TLOSS message to stderr and returns 0. In both cases, errno is set to ERANGE.

The sinh function returns the hyperbolic sine of *x*. If the result is too large, sinh sets errno to ERANGE and returns ±HUGE_VAL (±_LHUGE_VAL for the long double functions). Error handling can be changed with the _matherr function.

sin, sinh

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

sinl, sinhl

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

acos functions
asin functions
atan functions
cos functions
tan functions

```
/* SINCOS.C: This program displays the sine, hyperbolic
 * sine, cosine, and hyperbolic cosine of pi / 2.
 */
#include <math.h>
#include <stdio.h>
void main( void )
{
   double pi = 3.1415926535;
   double x, y;
   x = pi / 2;
   y = sin( x );
   printf( "sin( %f ) = %f\n", x, y );
   y = sinh( x );
   printf( "sinh( %f ) = %f\n",x, y );
   y = cos( x );
   printf( "cos( %f ) = %f\n", x, y );
   y = cosh( x );
   printf( "cosh( %f ) = %f\n",x, y );
}
```

_sopen

#include <fcntl.h>, <sys\types.h>, <sys\stat.h>, and <share.h>

#include <io.h> Required only for function declarations

Syntax int _sopen(const char **filename*, int *oflag*, int *shflag* [, int *pmode*]);

Parameter	Description
filename	Filename
oflag	Type of operations allowed
shflag	Type of sharing allowed
pmode	Permission setting

The _sopen function opens the file specified by *filename* and prepares the file for subsequent shared reading or writing, as defined by *oflag* and *shflag*. The integer expression *oflag* is formed by combining one or more of the following manifest constants, defined in the file FCNTL.H. When two or more constants are used to form the argument *oflag*, the constants are combined with the bitwise-OR operator (|).

_O_APPEND
  Repositions the file pointer to the end of the file before every write operation.

_O_BINARY
  Opens file in binary (untranslated) mode. (See fopen for a description of binary mode.)

_O_CREAT
  Creates and opens a new file. This has no effect if the file specified by *filename* exists.

_O_EXCL
  Returns an error value if the file specified by *filename* exists. This applies only when used with _O_CREAT.

_O_RDONLY
  Opens file for reading only. If this flag is given, neither the _O_RDWR flag nor the _O_WRONLY flag can be given.

_O_RDWR
  Opens file for both reading and writing. If this flag is given, neither _O_RDONLY nor _O_WRONLY can be given.

_O_TEXT
  Opens file in text (translated) mode. (See fopen for a description of text mode.)

_O_TRUNC
  Opens and truncates an existing file to 0 bytes. The file must have write permission; the contents of the file are destroyed.

_O_WRONLY
  Opens file for writing only. If this flag is given, neither _O_RDONLY nor _O_RDWR can be given.

The argument *shflag* is a constant expression consisting of one of the following manifest constants, defined in SHARE.H. If SHARE.COM (or SHARE.EXE for some versions of MS-DOS) is not installed, MS-DOS ignores the sharing mode. (See your system documentation for detailed information about sharing modes.)

_SH_COMPAT
  Sets compatibility mode. This is the sharing mode used in the _open function in MS-DOS.

_SH_DENYRW
  Denies read and write access to file.

_SH_DENYWR
Denies write access to file.

_SH_DENYRD
Denies read access to file.

_SH_DENYNO
Permits read and write access.

The _sopen function should be used only with MS-DOS version 3.0 and later. Under earlier versions of MS-DOS, the *shflag* argument is ignored.

The *pmode* argument is required only when _O_CREAT is specified. If the file does not exist, *pmode* specifies the file's permission settings, which are set when the new file is closed for the first time. Otherwise, the *pmode* argument is ignored. The *pmode* argument is an integer expression that contains one or both of the manifest constants _S_IWRITE and _S_IREAD, defined in SYS\STAT.H. When both constants are given, they are combined with the bitwise-OR operator (|). The meaning of the *pmode* argument is as follows:

Value	Meaning
_S_IWRITE	Writing permitted
_S_IREAD	Reading permitted
_S_IREAD | _S_IWRITE	Reading and writing permitted

If write permission is not given, the file is read-only. With MS-DOS, all files are readable; it is not possible to give write-only permission. Thus, the modes _S_IWRITE and _S_IREAD | _S_IWRITE are equivalent.

Note that with MS-DOS versions 3.x with SHARE installed, a problem occurs when opening a new file with _sopen under the following sets of conditions:

- With *oflag* set to _O_CREAT | _O_RDONLY or _O_CREAT | _O_WRONLY, *pmode* set to _S_IREAD, and *shflag* set to _SH_COMPAT.
- With *oflag* set to any combination that includes _O_CREAT | _O_RDWR, *pmode* set to _S_IREAD, and *shflag* set to anything other than _SH_COMPAT.

In either case, the operating system will prematurely close the file during system calls made within _sopen, or the system will generate a sharing violation (INT 24H). To avoid the problem, open the file with *pmode* set to _S_IWRITE. After closing the file, call _chmod and change the mode back to _S_IREAD. Another solution is to open the file with *pmode* set to _S_IREAD, *oflag* set to _O_CREAT | _O_RDWR, and *shflag* set to _SH_COMPAT.

The _sopen function applies the current file-permission mask to *pmode* before setting the permissions (see _umask).

Return Value

The _sopen function returns a file handle for the opened file. A return value of -1 indicates an error, and errno is set to one of the following values:

EACCES
Given path is a directory, or the file is read-only but an open for writing was attempted, or a sharing violation occurred (the file's sharing mode does not allow the specified operations; MS-DOS versions 3.0 and later only).

EEXIST
The _O_CREAT and _O_EXCL flags are specified, but the named file already exists.

EINVAL
An invalid *oflag* or *shflag* argument was given.

EMFILE

No more file handles available (too many open files).

ENOENT

File or path not found.

_close
_creat
fopen
_fsopen
_open
_umask

Standards:	None
16-Bit:	MS-DOS, QWIN, WIN, WIN DLL

_spawn Functions

#include <stdio.h>, <process.h>

Syntax int _spawnl(int *mode*, const char **cmdname*, const char **arg0*, const char **arg1*, ... const char **argn*, NULL);

int _spawnle(int *mode*, const char **cmdname*, const char **arg0*, const char **arg1*, ... const char **argn*, NULL, const char * const **envp*);

int _spawnlp(int *mode*, const char **cmdname*, const char **arg0*, const char **arg1*, ... const char **argn*, NULL);

int _spawnlpe(int *mode*, const char **cmdname*, const char **arg0*, const char **arg1*, ... const char **argn*, NULL, const char * const **envp*);

int _spawnv(int *mode*, const char **cmdname*, const char * const **argv*);

int _spawnve(int *mode*, const char **cmdname*, const char * const **argv*, const char * const **envp*);

int _spawnvp(int *mode*, const char **cmdname*, const char * const **argv*);

int _spawnvpe(int *mode*, const char **cmdname*, const char * const **argv*, const char * const **envp*);

Parameter	Description
mode	Execution mode for parent process
cmdname	Path of file to be executed
arg0, ... *argn*	List of pointers to arguments
argv	Array of pointers to arguments
envp	Array of pointers to environment settings

The _spawn family of functions creates and executes a new child process for MS-DOS. Enough memory must be available for loading and executing the child process. The *mode* argument determines the action taken by the parent process before and during _spawn. The following values for *mode* are defined in PROCESS.H:

Value	Meaning
_P_OVERLAY	Overlays parent process with child, destroying the parent (same effect as _exec calls).
_P_WAIT	Suspends parent process until execution of child process is complete (synchronous _spawn).

The *cmdname* argument specifies the file that is executed as the child process and can specify a full path (from the root), a partial path (from the current working directory), or just a filename. If *cmdname* does not have a filename extension or does not end with a period (.), the _spawn function first tries the .COM extension, then the .EXE extension, and finally the .BAT extension. This ability to spawn batch files is new beginning with Microsoft C version 6.0.

If *cmdname* has an extension, only that extension is used. If *cmdname* ends with a period, the _spawn calls search for *cmdname* with no extension. The _spawnlp, _spawnlpe, _spawnvp, and _spawnvpe

routines search for *cmdname* (using the same procedures) in the directories specified by the PATH environment variable.

If *cmdname* contains a drive specifier or any slashes (that is, if it is a relative path), the _spawn call searches only for the specified file and no path searching is done.

Note To ensure proper overlay initialization and termination, do not use the setjmp or longjmp function to enter or leave an overlay routine.

Arguments for the Child Process

Arguments are passed to the child process by giving one or more pointers to character strings as arguments in the _spawn call. These character strings form the argument list for the child process. The combined length of the strings forming the argument list for the child process must not exceed 128 bytes in real mode. The terminating null character ('\0') for each string is not included in the count, but space characters (automatically inserted to separate arguments) are included.

The argument pointers may be passed as separate arguments (_spawnl, _spawnle, _spawnlp, and _spawnlpe) or as an array of pointers (_spawnv, _spawnve, _spawnvp, and _spawnvpe). At least one argument, *arg0* or *argv*[0], must be passed to the child process. By convention, this argument is the name of the program as it might be typed on the command line by the user. (A different value does not produce an error.) In real mode, the *argv*[0] value is supplied by the operating system and is the fully qualified path of the executing program. In protected mode, it is usually the program name as it would be typed on the command line.

The _spawnl, _spawnle, _spawnlp, and _spawnlpe calls are typically used in cases where the number of arguments is known in advance. The *arg0* argument is usually a pointer to *cmdname*. The arguments *arg1* through *argn* are pointers to the character strings forming the new argument list. Following *argn*, there must be a NULL pointer to mark the end of the argument list.

The _spawnv, _spawnve, _spawnvp, and _spawnvpe calls are useful when the number of arguments to the child process is variable. Pointers to the arguments are passed as an array, *argv*. The argument *argv*[0] is usually a pointer to a path in real mode or to the program name in protected mode, and *argv*[1] through *argv*[*n*] are pointers to the character strings forming the new argument list. The argument *argv*[*n* +1] must be a NULL pointer to mark the end of the argument list.

Environment of the Child Process

Files that are open when a _spawn call is made remain open in the child process. In the _spawnl, _spawnlp, _spawnv, and _spawnvp calls, the child process inherits the environment of the parent. The _spawnle, _spawnlpe, _spawnve, and _spawnvpe calls make it possible for the user to alter the environment for the child process by passing a list of environment settings through the *envp* argument. The argument *envp* is an array of character pointers, each element of which (except for the final element) points to a null-terminated string defining an environment variable. Such a string usually has the form

NAME=*value*

where NAME is the name of an environment variable and *value* is the string value to which that variable is set. (Note that *value* is not enclosed in double quotation marks.) The final element of the *envp* array should be NULL. When *envp* itself is NULL, the child process inherits the environment settings of the parent process.

The _spawn functions can pass the child process all information about open files, including the translation mode, through the C_FILE_INFO entry in the environment that is passed in real mode. The startup code normally processes this entry and then deletes it from the environment. However, if a _spawn function spawns a non-C process, this entry remains in the environment. Printing the environment shows graphics characters in the definition string for this entry, because the environment information is passed in binary form in real mode. It should not have any other effect on normal operations. In protected mode, the environment information is passed in text form and therefore contains no graphics characters.

You must explicitly flush (using fflush or _flushall) or close any stream prior to the _spawn function call.

With this product, as well as with Microsoft C version 6.0 and Microsoft C/C++ version 7.0, you can control whether or not the open file information of a process is passed to its child processes. The external variable _fileinfo (declared in STDLIB.H) controls the passing of C_FILE_INFO information. If _fileinfo is 0, the C_FILE_INFO information is not passed to the child processes. If _fileinfo is not 0, C_FILE_INFO is passed to child processes.

By default, _fileinfo is 0 and thus the C_FILE_INFO information is not passed to child processes. There are two ways to modify the default value of _fileinfo. You can link the supplied object file FILEINFO.OBJ into your program, using the /NOE option to avoid multiple symbol definitions, or you can set the _fileinfo variable to a nonzero value directly within your C program.

Return Value

The return value from a synchronous _spawn (_P_WAIT specified for *mode*) is the exit status of the child process.

The exit status is 0 if the process terminated normally. The exit status can be set to a nonzero value if the child process specifically calls the exit routine with a nonzero argument. If the child process did not explicitly set a positive exit status, a positive exit status indicates an abnormal exit with an abort or an interrupt. A return value of -1 indicates an error (the child process is not started). In this case, errno is set to one of the following values:

Value	Meaning
E2BIG	In MS-DOS, the argument list exceeds 128 bytes, or the space required for the environment information exceeds 32K.
EINVAL	The *mode* argument is invalid.
ENOENT	The file or path is not found.
ENOEXEC	The specified file is not executable or has an invalid executable-file format.
ENOMEM	Not enough memory is available to execute the child process.

Note that signal settings are not preserved in child processes created by calls to _spawn routines. The signal settings are reset to the default in the child process.

abort
atexit
_exec functions
exit
_exit
_onexit
system

Standards: None
16-Bit: MS-DOS

```
/* SPAWN.C: This program accepts a number in the range
 * 1-8 from the command line. Based on the number it receives,
 * it executes one of  the eight different procedures that
 * spawn the process named child. For some of these procedures,
 * the CHILD.EXE file must be in the same directory; for
 * others, it only has to be in the same path.
 */
#include <stdio.h>
#include <process.h>
char *my_env[] =
{
   "THIS=environment will be",
   "PASSED=to child.exe by the",
   "_SPAWNLE=and",
   "_SPAWNLPE=and",
   "_SPAWNVE=and",
   "_SPAWNVPE=functions",
   NULL
};
void main( int argc, char *argv[] )
{
   char *args[4];
   /* Set up parameters to be sent: */
   args[0] = "child";
   args[1] = "spawn??";
   args[2] = "two";
   args[3] = NULL;
   switch (argv[1][0])   /* Based on first letter of argument */
   {
   case '1':
      _spawnl( _P_WAIT, argv[2], argv[2], "_spawnl", "two", NULL );
      break;
   case '2':
      _spawnle( _P_WAIT, argv[2], argv[2], "_spawnle", "two", NULL, my_env );
      break;
   case '3':
      _spawnlp( _P_WAIT, argv[2], argv[2], "_spawnlp", "two", NULL );
      break;
   case '4':
      _spawnlpe( _P_WAIT, argv[2], argv[2], "_spawnlpe", "two", NULL, my_env );
      break;
   case '5':
      _spawnv( _P_OVERLAY, argv[2], args );
      break;
   case '6':
      _spawnve( _P_OVERLAY, argv[2], args, my_env );
      break;
   case '7':
      _spawnvp( _P_OVERLAY, argv[2], args );
      break;
   case '8':
      _spawnvpe( _P_OVERLAY, argv[2], args, my_env );
      break;
   default:
      printf( "SYNTAX: SPAWN <1-8> <childprogram>\n" );
      exit( 1 );
   }
   printf( "from SPAWN!\n" );
}
```

_splitpath

#include <stdlib.h>

Syntax void _splitpath(const char **path*, char **drive*, char **dir*, char **fname*, char **ext*);

Parameter	Description
path	Full path
drive	Drive letter
dir	Directory path
fname	Filename
ext	File extension

The _splitpath routine breaks a full path into its four components. The *path* argument should point to a buffer containing the complete path. The maximum size necessary for each buffer is specified by the manifest constants _MAX_DRIVE, _MAX_DIR, _MAX_FNAME, and _MAX_EXT, defined in STDLIB.H. The other arguments point to the buffers used to store the path elements:

drive
Contains the drive letter followed by a colon (:) if a drive is specified in *path*.

dir
Contains the path of subdirectories, if any, including the trailing slash. Forward slashes (/), backslashes (\ ), or both may be present in *path*.

fname
Contains the base filename without any extensions.

ext
Contains the filename extension, if any, including the leading period (.).

The return parameters will contain empty strings for any path components not found in *path*. You can pass a NULL pointer to _splitpath for any component you don't wish to receive.

Return Value

None.

_fullpath
_makepath

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

sprintf, _snprintf

#include <stdio.h>

Syntax int sprintf(char *_buffer_, const char *_format_ [, _argument_] ...);

int _snprintf(char *_buffer_, size_t _count_, const char *_format_ [, _argument_] ...);

Parameter	Description
buffer	Storage location for output
format	Format-control string
argument	Optional arguments
count	Maximum number of bytes to store

The sprintf function formats and stores a series of characters and values in *buffer*. Each *argument* (if any) is converted and output according to the corresponding format specification in the *format*. The format consists of ordinary characters and has the same form and function as the *format* argument for the printf function. (See printf for a description of the format and arguments.) A null character is appended to the end of the characters written but is not counted in the return value.

The _snprintf function differs from sprintf in that it stores no more than *count* characters to *buffer*.

Return Value

Both the sprintf and _snprintf functions return the number of characters stored in *buffer*, not counting the terminating null character. For _snprintf, if the number of bytes required to store the data exceeds *count*, then *count* bytes of data are stored in *buffer* and -1 is returned.

sprintf

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN

_snprintf

Standards: None
16-Bit: MS-DOS, QWIN, WIN

fprintf
printf
sscanf

```
/* SPRINTF.C: This program uses sprintf to format various
 * data and place them in the string named buffer.
 */
#include <stdio.h>
void main( void )
{
   char  buffer[200], s[] = "computer", c = 'l';
   int   i = 35, j;
   float fp = 1.7320534f;
   /* Format and print various data: */
   j  = sprintf( buffer,     "\tString:    %s\n", s );
   j += sprintf( buffer + j, "\tCharacter: %c\n", c );
   j += sprintf( buffer + j, "\tInteger:   %d\n", i );
   j += sprintf( buffer + j, "\tReal:      %f\n", fp );
   printf( "Output:\n%s\ncharacter count = %d\n", buffer, j );
}
```

sqrt, sqrtl

#include <math.h>

Syntax double sqrt(double *x*);

long double sqrtl(long double *x*);

Parameter	Description
x	Nonnegative floating-point value

The sqrt functions calculate the square root of *x*. The sqrtl function is the 80-bit counterpart and uses an 80-bit, 10-byte coprocessor form of arguments and return values.

Return Value

The sqrt functions return the square-root result. If *x* is negative, the function prints a _DOMAIN error message to stderr, sets errno to EDOM, and returns 0.

Error handling can be modified by using the _matherr or _matherrl routine.

sqrt

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

sqrtl

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

exp
log
_matherr
pow

```c
/* SQRT.C: This program calculates a square root. */
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
void main( void )
{
   double question = 45.35, answer;
   answer = sqrt( question );
   if( errno == EDOM )
      printf( "Domain error\n" );
   else
      printf( "The square root of %.2f is %.2f\n", question, answer );
}
```

srand

#include <stdlib.h> Required only for function declarations

Syntax void srand(unsigned int *seed*);

Parameter	Description
seed	Seed for random-number generation

The srand function sets the starting point for generating a series of pseudorandom integers. To reinitialize the generator, use 1 as the *seed* argument. Any other value for *seed* sets the generator to a random starting point.

The rand function is used to retrieve the pseudorandom numbers that are generated. Calling rand before any call to srand will generate the same sequence as calling srand with *seed* passed as 1.

Return Value

None.

rand

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

32-Bit:

sscanf

#include <stdio.h>

Syntax int sscanf(const char **buffer*, const char **format* [, *argument*] ...);

Parameter	Description
buffer	Stored data
format	Format-control string
argument	Optional arguments

The sscanf function reads data from *buffer* into the locations given by each *argument*. Every *argument* must be a pointer to a variable with a type that corresponds to a type specifier in *format*. The format controls the interpretation of the input fields and has the same form and function as the *format* argument for the scanf function; see scanf for a complete description of *format*.

Return Value

The sscanf function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF for an attempt to read at end-of-string. A return value of 0 means that no fields were assigned.

fscanf
scanf
sprintf

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN

```
/* SSCANF.C: This program uses sscanf to read data items
 * from a string named tokenstring, then displays them.
 */
#include <stdio.h>
void main( void )
{
   char  tokenstring[] = "15 12 14...";
   char  s[81];
   char  c;
   int   i;
   float fp;
   /* Input various data from tokenstring: */
   sscanf( tokenstring, "%s", s );
   sscanf( tokenstring, "%c", &c );
   sscanf( tokenstring, "%d", &i );
   sscanf( tokenstring, "%f", &fp );
   /* Output the data read */
   printf( "String    = %s\n", s );
   printf( "Character = %c\n", c );
   printf( "Integer:  = %d\n", i );
   printf( "Real:     = %f\n", fp );
}
```

_stackavail

#include <malloc.h> Required only for function declarations

Syntax size_t _stackavail(void);

The _stackavail function returns the approximate size (in bytes) of the stack space available for dynamic memory allocation with _alloca.

Return Value

The _stackavail function returns the size in bytes as an unsigned integer value.

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* ALLOCA.C: This program checks the stack
 * space available before and after using the
 * _alloca function to allocate space on the stack.
 */
#include <malloc.h>
#include <stdio.h>
void main( void )
{
   char *buffer;
   printf( "Bytes available on stack: %u\n", _stackavail() );
   /* Allocate memory for string. */
   buffer = _alloca( 120 * sizeof( char ) );
   printf( "The _alloca function just allocated" );
   printf( " memory from the program stack.\n" );
   printf( "Enter a string: " );
   gets( buffer );
   printf( "\"%s\" was stored in the program stack.\n", buffer );
   printf( "Bytes available on stack: %u\n", _stackavail() );
}
```

_stat

#include <sys\types.h>, <sys\stat.h>

Syntax int _stat(const char **pathname,* struct _stat **buffer*);

Parameter	Description
path	Path of existing file
buffer	Pointer to structure that receives results

The _stat function obtains information about the file or directory specified by *path* and stores it in the structure pointed to by *buffer*. The _stat structure, defined in the file SYS\STAT.H, includes the following fields:

st_atime
Time of last access of file.

st_ctime
Time of creation of file.

st_dev
Drive number of the disk containing the file (same as st_rdev).

st_mode
Bit mask for file-mode information. The _S_IFDIR bit is set if *pathname* specifies a directory; the _S_IFREG bit is set if *path* specifies an ordinary file or a device. User read/write bits are set according to the file's permission mode; user execute bits are set according to the filename extension.

st_mtime
Time of last modification of file.

st_nlink
Always 1.

st_rdev
Drive number of the disk containing the file (same as st_dev).

st_size
Size of the file in bytes.

Note that if *path* refers to a device, the size, time, _dev, and _rdev fields in the _stat structure are meaningless. Also, as STAT.H uses the _dev_t type, which is defined in TYPES.H, you must include TYPES.H before STAT.H in your code.

Return Value

The _stat function returns 0 if the file-status information is obtained. A return value of -1 indicates an error; also, errno is set to ENOENT, indicating that the filename or path could not be found.

Use _stat for compatibility with ANSI naming conventions of non-ANSI functions. Use stat and link with OLDNAMES.LIB for UNIX compatibility.

__access
__fstat

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* STAT.C: This program uses the _stat function to
 * report information about the file named STAT.C.
 */
#include <time.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <stdio.h>
void main( void )
{
   struct _stat buf;
   int result;
   char buffer[] = "A line to output";
   /* Get data associated with "stat.c": */
   result = _stat( "stat.c", &buf );
   /* Check if statistics are valid: */
   if( result != 0 )
      perror( "Problem getting information" );
   else
   {
      /* Output some of the statistics: */
      printf( "File size     : %ld\n", buf.st_size );
      printf( "Drive         : %c:\n", buf.st_dev + 'A' );
      printf( "Time modified : %s", ctime( &buf.st_atime ) );
   }
}
```

_status87

#include <float.h>

Syntax unsigned int _status87(void);

The _status87 function gets the floating-point status word. The status word is a combination of the 8087/80287/80387 status word and other conditions detected by the 8087/80287/80387 exception handler, such as floating-point stack overflow and underflow.

Return Value

The bits in the value returned indicate the floating-point status. See the FLOAT.H include file for a complete definition of the bits returned by _status87.

Note that many of the math library functions modify the 8087/80287 status word, with unpredictable results. Return values from _clear87 and _status87 become more reliable as fewer floating-point operations are performed between known states of the floating-point status word.

_clear87
_control87

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* This program places the name of the current directory in
 * the buffer array, then displays the name of the current
 * directory on the screen. Specifying a length of _MAX_DIR
 * leaves room for the longest legal directory name.
 */
#include <direct.h>
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   char buffer[_MAX_DIR];
   /* Get the current working directory: */
   if( _getcwd( buffer, _MAX_DIR ) == NULL )
      perror( "_getcwd error" );
   else
      printf( "%s\n", buffer );
}
```

strcat, _fstrcat

#include <string.h> Required only for function declarations

Syntax char *strcat(char **string1*, const char **string2*);

char __far * __far _fstrcat(char __far **string1*, const char __far * *string2*);

Parameter	**Description**
string1	Destination string
string2	Source string

The strcat and _fstrcat functions append *string2* to *string1*, terminate the resulting string with a null character, and return a pointer to the concatenated string (*string1*).

The strcat and _fstrcat functions operate on null-terminated strings. The string arguments to these functions are expected to contain a null character ('\0') marking the end of the string. No overflow checking is performed when strings are copied or appended.

The _fstrcat function is a model-independent (large-model) form of the strcat function. The behavior and return value of _fstrcat are identical to those of the model-dependent function strcat, with the exception that the arguments and return values are far pointers.

Return Value

The return values for these functions are described above.

strcat

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrcat

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strncat
strncmp
strncpy
_strnicmp
strrchr
strspn

```c
/* STRCPY.C: This program uses strcpy
 * and strcat to build a phrase.
 */
#include <string.h>
#include <stdio.h>
void main( void )
{
   char string[80];
   strcpy( string, "Hello world from " );
   strcat( string, "strcpy " );
   strcat( string, "and " );
   strcat( string, "strcat!" );
   printf( "String = %s\n", string );
}
```

strchr, _fstrchr

#include <string.h> Required only for function declarations

Syntax char *strchr(const char **string*, int *c*);

char __far * __far _fstrchr(const char __far **string*, int *c*);

Parameter	**Description**
string	Source string
c	Character to be located

The strchr and _fstrchr functions return a pointer to the first occurrence of *c* (converted to char) in *string*. The converted character *c* may be the null character ('\0'); the terminating null character of *string* is included in the search. The function returns NULL if the character is not found.

The strchr and _fstrchr functions operate on null-terminated strings. The string arguments to these functions are expected to contain a null character ('\0') marking the end of the string.

The _fstrchr function is a model-independent (large-model) form of the strchr function. The behavior and return value of _fstrchr are identical to those of the model-dependent function strchr, with the exception that the arguments and return values are far.

Return Value

The return values for these functions are described above.

strchr

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrchr

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strcspn
strncat
strncmp
strncpy
_strnicmp
strpbrk
strrchr
strspn
strstr

```
/* STRCHR.C: This program illustrates searching for a character
 * with strchr (search forward) or strrchr (search backward).
 */
#include <string.h>
#include <stdio.h>
int  ch = 'r';
char string[] = "The quick brown dog jumps over the lazy fox";
char fmt1[] =   "         1         2         3         4         5";
char fmt2[] =   "12345678901234567890123456789012345678901234567890";
void main( void )
{
   char *pdest;
   int result;
   printf( "String to be searched: \n\t\t%s\n", string );
   printf( "\t\t%s\n\t\t%s\n\n", fmt1, fmt2 );
   printf( "Search char:\t%c\n", ch );
   /* Search forward. */
   pdest = strchr( string, ch );
   result = pdest - string + 1;
   if( pdest != NULL )
      printf( "Result:\tfirst %c found at position %d\n\n", ch, result );
   else
      printf( "Result:\t%c not found\n" );
   /* Search backward. */
   pdest = strrchr( string, ch );
   result = pdest - string + 1;
   if( pdest != NULL )
      printf( "Result:\tlast %c found at position %d\n\n", ch, result );
   else
      printf( "Result:\t%c not found\n" );
}
```

strcmp, _fstrcmp

#include <string.h> Required only for function declarations

Syntax int strcmp(const char *_string1_, const char *_string2_);

int __far _fstrcmp(const char __far *_string1_, const char __far * _string2_);

Parameter	Description
string1	String to compare
string2	String to compare

The strcmp and _fstrcmp functions compare *string1* and *string2* lexicographically and return a value indicating their relationship, as follows:

Value	Meaning
< 0	*string1* less than *string2*
= 0	*string1* identical to *string2*
> 0	*string1* greater than *string2*

The strcmp and _fstrcmp functions operate on null-terminated strings. The string arguments to these functions are expected to contain a null character ('\0') marking the end of the string.

The _fstrcmp function is a model-independent (large-model) form of the strcmp function. The behavior and return value of _fstrcmp are identical to those of the model-dependent function strcmp, with the exception that the arguments are far pointers.

Both the _stricmp function and the _strcmpi function compare strings by first converting them to their lowercase forms.

Note that two strings containing characters located between 'Z' and 'a' in the ASCII table ('**[**', '\'_'**]**', '**^**', '_', and '`') compare differently depending on their case. For example, the two strings, `"ABCDE"` and `"ABCD^"`, compare one way if the comparison is lowercase (`"abcde" > "abcd^"`) and compare the other way (`"ABCDE" < "ABCD^"`) if it is uppercase.

Return Value

The return values for these functions are described above.

strcmp

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrcmp

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

memcmp
_memicmp
strncat
strncmp
strncpy
_strnicmp
strrchr
strspn

```
/* STRCMP.C */
#include <string.h>
#include <stdio.h>
char string1[] = "The quick brown dog jumps over the lazy fox";
char string2[] = "The QUICK brown dog jumps over the lazy fox";
void main( void )
{
   char tmp[20];
   int result;
   /* Case sensitive */
   printf( "Compare strings:\n\t%s\n\t%s\n\n", string1, string2 );
   result = strcmp( string1, string2 );
   if( result > 0 )
      strcpy( tmp, "greater than" );
   else if( result < 0 )
      strcpy( tmp, "less than" );
   else
      strcpy( tmp, "equal to" );
   printf( "\tstrcmp:   String 1 is %s string 2\n", tmp );
   /* Case insensitive (could use equivalent _stricmp) */
   result = _stricmp( string1, string2 );
   if( result > 0 )
      strcpy( tmp, "greater than" );
   else if( result < 0 )
      strcpy( tmp, "less than" );
   else
      strcpy( tmp, "equal to" );
   printf( "\t_stricmp:  String 1 is %s string 2\n", tmp );
}
```

strcoll

#include <string.h> Required only for function declarations

Syntax int strcoll(const char **string1*, const char **string2*);

Parameter	Description
string1	String to compare
string2	String to compare

The strcoll function compares *string1* and *string2* in a manner determined by the LC_COLLATE macro and returns a value indicating their relationship, as follows:

Value	Meaning
< 0	*string1* less than *string2*
= 0	*string1* identical to *string2*
> 0	*string1* greater than *string2*

For more information on the LC_COLLATE macro, see the setlocale function.

The strcoll function operates on null-terminated strings. The string arguments to these functions are expected to contain a null character ('\0') marking the end of the string.

The strcoll function differs from strcmp in that it uses locale-specific information to provide locale-specific collating sequences.

Return Value

The return value for this function is described above.

localeconv
setlocale
strcmp
strncmp
strxfrm

Standards: ANSI

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strcpy, _fstrcpy

#include <string.h> Required only for function declarations

Syntax char *strcpy(char **string1*, const char **string2*);

char __far * __far _fstrcpy(char __far **string1*, const char __far * *string2*);

Parameter	**Description**
string1	Destination string
string2	Source string

The strcpy function copies *string2*, including the terminating null character, to the location specified by *string1*, and returns *string1*.

The strcpy and _fstrcpy functions operate on null-terminated strings. The string arguments to these functions are expected to contain a null character ('\0') marking the end of the string. No overflow checking is performed when strings are copied or appended.

The _fstrcpy function is a model-independent (large-model) form of the strcpy function. The behavior and return value of _fstrcpy are identical to those of the model-dependent function strcpy, with the exception that the arguments and return values are far pointers.

Return Value

The return values for these functions are described above.

strcpy

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrcpy

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strcat
strcmp
strncat
strncmp
strncpy
_strnicmp
strrchr
strspn

```c
/* STRCPY.C: This program uses strcpy
 * and strcat to build a phrase.
 */
#include <string.h>
#include <stdio.h>
void main( void )
{
   char string[80];
   strcpy( string, "Hello world from " );
   strcat( string, "strcpy " );
   strcat( string, "and " );
   strcat( string, "strcat!" );
   printf( "String = %s\n", string );
}
```

strcspn, _fstrcspn

#include <string.h> Required only for function declarations

Syntax size_t strcspn(const char *_string1_, const char *_string2_);

size_t __far _fstrcspn(const char __far *_string1_, const char __far * _string2_);

Parameter	Description
string1	Source string
string2	Character set

The strcspn functions return the index of the first character in *string1* belonging to the set of characters specified by *string2*. This value is equivalent to the length of the initial substring of *string1* consisting entirely of characters not in *string2*. Terminating null characters are not considered in the search. If *string1* begins with a character from *string2*, strcspn returns 0.

The strcspn and _fstrcspn functions operate on null-terminated strings. The string arguments to these functions are expected to contain a null character ('\0') marking the end of the string.

The _fstrcspn function is a model-independent (large-model) form of the strcspn function. The behavior and return value of _fstrcspn are identical to those of the model-dependent function strcspn, with the exception that the arguments and return values are far.

Return Value

The return values for these functions are described above.

strcspn

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrcspn

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strncat
strncmp
strncpy
_strnicmp
strrchr
strspn

```c
/* STRCSPN.C */
#include <string.h>
#include <stdio.h>
void main( void )
{
   char string[] = "xyzabc";
   int  pos;
   pos = strcspn( string, "abc" );
   printf( "First a, b or c in %s is at character %d\n", string, pos );
}
```

_strdate

#include <time.h>

Syntax char *_strdate(char **datestr*);

Parameter	**Description**
datestr	Current date

The _strdate function copies the date to the buffer pointed to by *datestr*, formatted

```
mm/dd/yy
```

where mm is two digits representing the month, `dd` is two digits representing the day of the month, and `yy` is the last two digits of the year. For example, the string

```
12/05/99
```

represents December 5, 1999.

The buffer must be at least nine bytes long.

Return Value

The _strdate function returns a pointer to the resulting text string *datestr*.

asctime
ctime
gmtime
localtime
mktime
time
_tzset

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_strdup Functions

#include <string.h> Required only for function declarations

Syntax char *_strdup(const char **string*);

char __far * __far _fstrdup(const char __far **string*);

char __near * __far _nstrdup(const char __far **string*);

Parameter	Description
string	Source string

The _strdup function allocates storage space (with a call to malloc) for a copy of *string* and returns a pointer to the storage space containing the copied string. The function returns NULL if storage cannot be allocated.

The _fstrdup and _nstrdup functions provide complete control over the heap used for string duplication. The _strdup function returns a pointer to a copy of the string argument. The space for the string is allocated from the heap specified by the memory model in use. In large data models (that is, compact-, large-, and huge-model programs), _strdup allocates space from the far heap. In small data models (tiny-, small-, and medium-model programs), _strdup allocates space from the near heap.

The _strdup, _fstrdup, and _nstrdup functions operate on null-terminated strings. The string arguments to these functions are expected to contain a null character ('\0') marking the end of the string.

The _fstrdup function returns a far pointer to a copy of the string allocated in far memory (the far heap). As with the other model-independent functions, the syntax and semantics of these functions correspond to those of _strdup except for the sizes of the arguments and return values. The _nstrdup function returns a near pointer to a copy of the string allocated in the near heap (in the default data segment).

Return Value

The return values for these functions are described above.

_strdup

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrdup, _nstrdup

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strcat
strcmp
strncat
strncmp
strncpy
_strnicmp
strrchr
strspn

```
/* STRDUP.C */
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <dos.h>
void main( void )
{
   char buffer[] = "This is the buffer text";
   char *newstring;
   printf( "Original: %s\n", buffer );
   newstring = _strdup( buffer );
   printf( "Copy:     %s\n", newstring );
}
```

strerror, _strerror

#include <string.h> Required only for function declarations

Syntax char *strerror(int *errnum*);

char *_strerror(const char **string*);

Parameter	**Description**
errnum	Error number
string	User-supplied message

The strerror function maps *errnum* to an error-message string, returning a pointer to the string. The function itself does not actually print the message; for that, you need to call an output function such as fprintf:

```
if (( _access( "datafile",2 )) == -1 )
   fprintf( stderr, strerror(NULL) );
```

If *string* is passed as NULL, _strerror returns a pointer to a string containing the system error message for the last library call that produced an error. The error-message string is terminated by the newline character ('\n').

If *string* is not equal to NULL, then _strerror returns a pointer to a string containing (in order) your string message, a colon, a space, the system error message for the last library call producing an error, and a newline character. Your string message can be a maximum of 94 bytes long.

Unlike perror, _strerror alone does not print any messages. To print the message returned by _strerror to stderr, your program will need an fprintf statement, as shown in the following lines:

```
if (( _access( "datafile",2 )) == -1 )
   fprintf( stderr, _strerror(NULL) );
```

The actual error number for _strerror is stored in the variable errno. The system error messages are accessed through the variable _sys_errlist, which is an array of messages ordered by error number. The _strerror function accesses the appropriate error message by using the errno value as an index to the variable _sys_errlist. The value of the variable _sys_nerr is defined as the maximum number of elements in the _sys_errlist array.

To produce accurate results, _strerror should be called immediately after a library routine returns with an error. Otherwise, the errno value may be overwritten by subsequent calls.

Note that the _strerror function under Microsoft C version 5.0 is identical to the version 4.0 strerror function. The name was altered to permit the inclusion in Microsoft C version 5.0 of the ANSI-conforming strerror function. The _strerror function is not part of the ANSI definition but is instead a Microsoft extension to it; it should not be used where portability is desired. For ANSI compatibility, use strerror instead.

Return Value

The strerror and _strerror functions return a pointer to the error-message string. The string can be overwritten by subsequent calls to strerror or _strerror, respectively.

strerror

Standards: ANSI
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_strerror

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

clearerr
ferror
perror

strftime

#include <time.h> Required only for function declarations

Syntax size_t strftime(char **string*, size_t *maxsize*, const char **format*, const struct tm **timeptr*);

Parameter	Description
string	Output string
maxsize	Maximum length of string
format	Format control string
timeptr	tm data structure

The strftime function formats the tm time value in *timeptr* according to the supplied *format* argument and stores the result in the buffer *string*. At most, *maxsize* characters are placed in the string.

The *format* argument consists of one or more codes; as in printf, the formatting codes are preceded by a % sign. Characters that do not begin with a % sign are copied unchanged to *string*. The LC_TIME category of the current locale affects the output formatting of strftime.

The formatting codes for strftime are listed below:

Format	Description
%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Date and time representation appropriate for the locale
%d	Day of the month as a decimal number (01-31)
%H	Hour in 24-hour format (00-23)
%I	Hour in 12-hour format (01-12)
%j	Day of the year as a decimal number (001-366)
%m	Month as a decimal number (01-12)
%M	Minute as a decimal number (00-59)
%p	Current locale's AM/PM indicator for a 12-hour clock
%S	Second as a decimal number (00-59)
%U	Week of the year as a decimal number, with Sunday as the first day of the week (00-51)
%w	Weekday as a decimal number (0-6; Sunday is 0)
%W	Week of the year as a decimal number, with Monday as the first day of the week (00-51)
%x	Date representation for current

	locale
%X	Time representation for current locale
%y	Year without the century as a decimal number (00-99)
%Y	Year with the century as a decimal number
%Z	Time zone name or abbreviation; no characters if time zone is unknown
%%	Percent sign

Return Value

The strftime function returns the number of characters placed in *string* if the total number of resulting characters, including the terminating null, is not more than *maxsize*. Otherwise, strftime returns 0, and the contents of the string are indeterminate.

localeconv
setlocale
strcoll
strxfrm

Standards:	ANSI
16-Bit:	MS-DOS, QWIN, WIN

_stricmp, _fstricmp

#include <string.h> Required only for function declarations

Syntax int _stricmp(const char *_string1_, const char *_string2_);

int __far _fstricmp(const char __far *_string1_, const char __far * _string2_);

Parameter	Description
string1	String to compare
string2	String to compare

The _stricmp and _fstricmp functions perform a lexicographical comparison of lowercase versions of *string1* and *string2* and return a value indicating their relationship, as follows:

Value	Meaning
< 0	*string1* less than *string2*
= 0	*string1* identical to *string2*
> 0	*string1* greater than *string2*

The _stricmp and _fstricmp functions operate on null-terminated strings. The string arguments to these functions are expected to contain a null character ('\0') marking the end of the string.

The _fstricmp function is a model-independent (large-model) form of the _stricmp function. The behavior and return value of _fstricmp are identical to those of the model-dependent function _stricmp, with the exception that the arguments are far pointers.

The _strcmpi function is functionally equivalent to _stricmp. It is included in STRING.H for compatibility with previous versions of Microsoft C. The preferred form is _stricmp.

The strcmp function is a case-sensitive version of _stricmp.

Return Value

The return values for these functions are described above.

_stricmp

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstricmp

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

memcmp
_memicmp
strcat
strcpy
strncat
strncmp
strncpy
_strnicmp
strrchr
_strset
strspn

strlen, _fstrlen

#include <string.h> Required only for function declarations

Syntax size_t strlen(const char **string*);

size_t _fstrlen(const char __far **string*);

Parameter	**Description**
string	Null-terminated string

The strlen and _fstrlen functions return the length, in bytes, of *string*, not including the terminating null character ('\0').

The _fstrlen function is a model-independent (large-model) form of the strlen function. The behavior and return value of _fstrlen are identical to those of the model-dependent function strlen, with the exception that the argument is a far pointer.

Return Value

These functions return the string length. There is no error return.

strlen

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrlen

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/*  STRLEN.C */
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <dos.h>
void main( void )
{
   char buffer[61] = "How long am I?";
   int  len;
   len = strlen( buffer );
   printf( "'%s' is %d characters long\n", buffer, len );
}
```

_strlwr, _fstrlwr

#include <string.h> Required only for function declarations

Syntax char *_strlwr(char **string*);

char __far * __far _fstrlwr(char __far **string*);

Parameter	**Description**
string	String to be converted

The _strlwr and _fstrlwr functions convert any uppercase letters in the given null-terminated *string* to lowercase. Other characters are not affected.

The _fstrlwr function is a model-independent (large-model) form of the _strlwr function. The behavior and return value of _fstrlwr are identical to those of the model-dependent function _strlwr, with the exception that the argument and return values are far pointers.

Return Value

These functions return a pointer to the converted string. There is no error return.

_strlwr

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrlwr

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_strupr

```c
/* STRLWR.C: This program uses _strlwr and _strupr to create
 * uppercase and lowercase copies of a mixed-case string.
 */
#include <string.h>
#include <stdio.h>
void main( void )
{
   char string[100] = "The String to End All Strings!";
   char *copy1, *copy2;
   copy1 = _strlwr( _strdup( string ) );
   copy2 = _strupr( _strdup( string ) );
   printf( "Mixed: %s\n", string );
   printf( "Lower: %s\n", copy1 );
   printf( "Upper: %s\n", copy2 );
}
```

strncat, _fstrncat

#include <string.h> Required only for function declarations

Syntax char *strncat(char **string1*, const char **string2*, size_t *count*);

char __far * __far _fstrncat(char __far **string1*, const char __far * *string2*, size_t *count*);

Parameter	Description
string1	Destination string
string2	Source string
count	Number of characters appended

The strncat and _fstrncat functions append, at most, the first *count* characters of *string2* to *string1*, terminate the resulting string with a null character ('\0'), and return a pointer to the concatenated string (*string1*). If *count* is greater than the length of *string2*, the length of *string2* is used in place of *count*.

The _fstrncat function is a model-independent (large-model) form of the strncat function. The behavior and return value of _fstrncat are identical to those of the model-dependent function strncat, with the exception that all the pointer arguments and return values are far pointers.

Return Value

The return values for these functions are described above.

strncat

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrncat

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strcat
strcmp
strcpy
strncmp
strncpy
_strnicmp
strrchr
_strset
strspn

```c
/* STRNCAT.C */
#include <string.h>
#include <stdio.h>
void main( void )
{
   char string[80] = "This is the initial string!";
   char suffix[] = " extra text to add to the string...";
   /* Combine strings with no more than 19 characters of suffix: */
   printf( "Before: %s\n", string );
   strncat( string, suffix, 19 );
   printf( "After:  %s\n", string );
}
```

strncmp, _fstrncmp

#include <string.h> Required only for function declarations

Syntax int strncmp(const char **string1*, const char **string2*, size_t *count*);

int __far _fstrncmp(const char __far **string1*, const char __far * *string2*, size_t *count*);

Parameter	Description
string1	String to compare
string2	String to compare
count	Number of characters compared

The strncmp and _fstrncmp functions lexicographically compare, at most, the first *count* characters of *string1* and *string2* and return a value indicating the relationship between the substrings, as listed below:

Value	Meaning
< 0	*string1* less than *string2*
= 0	*string1* identical to *string2*
> 0	*string1* greater than *string2*

The _strnicmp function is a case-insensitive version of strncmp.

The _fstrncmp function is a model-independent (large-model) form of the strncmp function. The behavior and return value of _fstrncmp are identical to those of the model-dependent function strncmp, with the exception that all the arguments and return values are far.

Return Value

The return values for these functions are described above.

strncmp

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrncmp

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strcat
strcmp
strcpy
strncat
strncpy
strrchr
_strset
strspn

```
/* STRNCMP.C */
#include <string.h>
#include <stdio.h>
char string1[] = "The quick brown dog jumps over the lazy fox";
char string2[] = "The QUICK brown fox jumps over the lazy dog";
void main( void )
{
   char tmp[20];
   int result;
   printf( "Compare strings:\n\t\t%s\n\t\t%s\n\n", string1, string2 );
   printf( "Function:\tstrncmp (first 10 characters only)\n" );
   result = strncmp( string1, string2 , 10 );
   if( result > 0 )
      strcpy( tmp, "greater than" );
   else if( result < 0 )
      strcpy( tmp, "less than" );
   else
      strcpy( tmp, "equal to" );
   printf( "Result:\t\t1 is %s string 2\n\n", tmp );
   printf( "Function:\tstrnicmp _strnicmp (first 10 characters only)\n" );
   result = _strnicmp( string1, string2, 10 );
   if( result > 0 )
      strcpy( tmp, "greater than" );
   else if( result < 0 )
      strcpy( tmp, "less than" );
   else
      strcpy( tmp, "equal to" );
   printf( "Result:\t\tString 1 is %s string 2\n\n", tmp );
}
```

strncpy, _fstrncpy

#include <string.h> Required only for function declarations

Syntax char *strncpy(char **string1*, const char **string2*, size_t *count*);

char __far * __far _fstrncpy(char __far **string1*, const char __far * *string2*, size_t *count*);

Parameter	Description
string1	Destination string
string2	Source string
count	Number of characters copied

The strncpy and _fstrncpy functions copy *count* characters of *string2* to *string1* and return *string1*. If *count* is less than the length of *string2*, a null character ('\0') is not appended automatically to the copied string. If *count* is greater than the length of *string2*, the *string1* result is padded with null characters ('\0') up to length *count*.

Note that the behavior of strncpy and _fstrncpy is undefined if the address ranges of the source and destination strings overlap.

The _fstrncpy function is a model-independent (large-model) form of the strncpy function. The behavior and return value of _fstrncpy are identical to those of the model-dependent function strncpy, with the exception that all the arguments and return values are far.

Return Value

The return values for these functions are described above.

strncpy

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrncpy

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strcat
strcmp
strcpy
strncat
strncmp
_strnicmp
strrchr
_strset
strspn

```c
/* STRNCPY.C */
#include <string.h>
#include <stdio.h>
void main( void )
{
   char string[100] = "Cats are nice usually";
   printf("Before: %s\n", string );
   strncpy( string, "Dogs", 4 );
   strncpy( string + 9, "mean", 4 );
   printf("After:  %s\n", string );
}
```

_strnicmp, _fstrnicmp

#include <string.h> Required only for function declarations

Syntax int _strnicmp(const char **string1*, const char **string2*, size_t *count*);

int __far _fstrnicmp(const char __far **string1*, const char __far * *string2*, size_t *count*);

Parameter	Description
string1	String to compare
string2	String to compare
count	Number of characters compared

The _strnicmp and _fstrnicmp functions lexicographically compare (without regard to case), at most, the first *count* characters of *string1* and *string2* and return a value indicating the relationship between the substrings, as listed below:

Value	Meaning
< 0	*string1* less than *string2*
= 0	*string1* identical to *string2*
> 0	*string1* greater than *string2*

The strncmp function is a case-sensitive version of _strnicmp.

Note that two strings containing characters located between 'Z' and 'a' in the ASCII table ('**[**', '**_**', '**]**', '**^**', '**_**', and '`') compare differently depending on their case. For example, the two strings, `"ABCDE"` and `"ABCD^"`, compare one way if the comparison is lowercase (`"abcde"` > `"abcd^"`) and compare the other way (`"ABCDE"` < `"ABCD^"`) if it is uppercase.

The _fstrnicmp function is a model-independent (large-model) form of the _strnicmp function. The behavior and return value of _fstrnicmp are identical to those of the model-dependent function _strnicmp, with the exception that all the arguments and return values are far.

Return Value

The return values for these functions are described above.

_strnicmp

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrnicmp

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strcat
strcmp
strcpy
strncat
strncpy
strrchr
_strset
strspn

_strnset, _fstrnset

#include <string.h> Required only for function declarations

Syntax char *_strnset(char **string*, int *c*, size_t *count*);

char __far * __far _fstrnset(char __far **string*, int *c*, size_t *count*);

Parameter	**Description**
string	String to be initialized
c	Character setting
count	Number of characters set

The _strnset and _fstrnset functions set, at most, the first *count* characters of *string* to *c* (converted to char) and return a pointer to the altered string. If *count* is greater than the length of *string*, the length of *string* is used in place of *count*.

The _fstrnset function is a model-independent (large-model) form of the _strnset function. The behavior and return value of _fstrnset are identical to those of the model-dependent function _strnset, with the exception that all the arguments and return values are far.

Return Value

The return values for these functions are described above.

_strnset

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrnset

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strcat
strcmp
strcpy
_strset

```c
/* STRNSET.C */
#include <string.h>
#include <stdio.h>
void main( void )
{
   char string[15] = "This is a test";
   /* Set not more than 4 characters of string to be *'s */
   printf( "Before: %s\n", string );
   _strnset( string, '*', 4 );
   printf( "After:  %s\n", string );
}
```

strpbrk, _fstrpbrk

#include <string.h> Required only for function declarations

Syntax char *strpbrk(const char **string1*, const char **string2*);

char __far * __far _fstrpbrk(const char __far **string1*, const char __far **string2*);

Parameter	Description
string1	Source string
string2	Character set

The strpbrk function finds the first occurrence in *string1* of any character from *string2*. The terminating null character ('\0') is not included in the search.

The _fstrpbrk function is a model-independent (large-model) form of the strpbrk function. The behavior and return value of _fstrpbrk are identical to those of the model-dependent function strpbrk, with the exception that all the arguments and return values are far.

Return Value

These functions return a pointer to the first occurrence of any character from *string2* in *string1*. A NULL return value indicates that the two string arguments have no characters in common.

strpbrk

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrpbrk

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strchr
strrchr

```c
/* STRPBRK.C */
#include <string.h>
#include <stdio.h>
void main( void )
{
   char string[100] = "The 3 men and 2 boys ate 5 pigs\n";
   char *result;
   /* Return pointer to first 'a' or 'b' in "string" */
   printf( "1: %s\n", string );
   result = strpbrk( string, "0123456789" );
   printf( "2: %s\n", result++ );
   result = strpbrk( result, "0123456789" );
   printf( "3: %s\n", result++ );
   result = strpbrk( result, "0123456789" );
   printf( "4: %s\n", result );
}
```

strrchr, _fstrrchr

#include <string.h> Required only for function declarations

Syntax char *strrchr(const char **string*, int *c*);

char __far * __far _fstrrchr(const char __far **string*, int *c*);

Parameter	**Description**
string	Searched string
c	Character to be located

The strrchr function finds the last occurrence of *c* (converted to char) in *string.* The string's terminating null character ('\0') is included in the search. (Use strchr to find the first occurrence of *c* in *string*.)

The _fstrrchr function is a model-independent (large-model) form of the strrchr function. The behavior and return value of _fstrrchr are identical to those of the model-dependent function strrchr, with the exception that all the pointer arguments and return values are far pointers.

Return Value

These functions return a pointer to the last occurrence of the character in the string. A NULL pointer is returned if the given character is not found.

strrchr

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrrchr

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strchr
strcspn
strncat
strncmp
strncpy
_strnicmp
strpbrk
strspn

```
/* STRCHR.C: This program illustrates searching for a character
 * with strchr (search forward) or strrchr (search backward).
 */
#include <string.h>
#include <stdio.h>
int  ch = 'r';
char string[] = "The quick brown dog jumps over the lazy fox";
char fmt1[] =   "         1         2         3         4         5";
char fmt2[] =   "12345678901234567890123456789012345678901234567890";
void main( void )
{
   char *pdest;
   int result;
   printf( "String to be searched: \n\t\t%s\n", string );
   printf( "\t\t%s\n\t\t%s\n\n", fmt1, fmt2 );
   printf( "Search char:\t%c\n", ch );
   /* Search forward. */
   pdest = strchr( string, ch );
   result = pdest - string + 1;
   if( pdest != NULL )
      printf( "Result:\tfirst %c found at position %d\n\n", ch, result );
   else
      printf( "Result:\t%c not found\n" );
   /* Search backward. */
   pdest = strrchr( string, ch );
   result = pdest - string + 1;
   if( pdest != NULL )
      printf( "Result:\tlast %c found at position %d\n\n", ch, result );
   else
      printf( "Result:\t%c not found\n" );
}
```

_strrev, _fstrrev

#include <string.h> Required only for function declarations

Syntax char *_strrev(char **string*);

char __far * __far _fstrrev(char __far **string*);

Parameter	**Description**
string	String to be reversed

The _strrev function reverses the order of the characters in *string*. The terminating null character ('\0') remains in place.

The _fstrrev function is a model-independent (large-model) form of the _strrev function. The behavior and return value of _fstrrev are identical to those of the model-dependent function _strrev, with the exception that the argument and return value are far pointers.

Return Value

These functions return a pointer to the altered string. There is no error return.

_strrev

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

fstrrev

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strcpy
_strset

```
/* STRREV.C: This program checks an input string to
 * see whether it is a palindrome: that is, whether
 * it reads the same forward and backward.
 */
#include <string.h>
#include <stdio.h>
void main( void )
{
   char string[100];
   int result;
   printf( "Input a string and I will tell you if it is a palindrome:\n" );
   gets( string );
   /* Reverse string and compare (ignore case): */
   result = _strcmpi( string, _strrev( _strdup( string ) ) );
   if( result == 0 )
      printf( "The string \"%s\" is a palindrome\n\n", string );
   else
      printf( "The string \"%s\" is not a palindrome\n\n", string );
}
```

_strset, _fstrset

#include <string.h> Required only for function declarations

Syntax char *_strset(char **string*, int *c*);

char __far * __far _fstrset(char __far **string*, int *c*);

Parameter	**Description**
string	String to be set
c	Character setting

The _strset function sets all of the characters of *string* to *c* (converted to char), except the terminating null character ('\0').

The _fstrset function is a model-independent (large-model) form of the _strset function. The behavior and return value of _fstrset are identical to those of the model-dependent function _strset, with the exception that the pointer arguments and return value are far pointers.

Return Value

These functions return a pointer to the altered string. There is no error return.

_strset

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrset

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

memset
strcat
strcmp
strcpy
_strnset

```c
/* STRSET.C */
#include <string.h>
#include <stdio.h>
void main( void )
{
   char string[] = "Fill the string with something";
   printf( "Before: %s\n", string );
   _strset( string, '*' );
   printf( "After:  %s\n", string );
}
```

strspn, _fstrspn

#include <string.h> Required only for function declarations

Syntax size_t strspn(const char *_string1_, const char *_string2_);

size_t __far _fstrspn(const char __far *_string1_, const char __far * _string2_);

Parameter	Description
string1	Searched string
string2	Character set

The strspn function returns the index of the first character in *string1* that does not belong to the set of characters specified by *string2*. This value is equivalent to the length of the initial substring of *string1* that consists entirely of characters from *string2* . The null character ('\0') terminating *string2* is not considered in the matching process. If *string1* begins with a character not in *string2*, strspn returns 0.

The _fstrspn function is a model-independent (large-model) form of the strspn function. The behavior and return value of _fstrspn are identical to those of the model-dependent function strspn, with the exception that the arguments are far pointers.

Return Value

These functions return an integer value specifying the length of the segment in *string1* consisting entirely of characters in *string2*.

strspn

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrspn

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strcspn
strncat
strncmp
strncpy
_strnicmp
strrchr

```c
/* STRSPN.C: This program uses strspn to determine
 * the length of the segment in the string "cabbage"
 * consisting of a's, b's, and c's. In other words,
 * it finds the first non-abc letter.
 */
#include <string.h>
#include <stdio.h>
void main( void )
{
   char string[] = "cabbage";
   int  result;
   result = strspn( string, "abc" );
   printf( "The portion of '%s' containing only a, b, or c "
           "is %d bytes long\n", string, result );
}
```

strstr, _fstrstr

#include <string.h> Required only for function declarations

Syntax char *strstr(const char **string1*, const char **string2*);

char __far * __far _fstrstr(const char __far **string1*, const char __far **string2*);

Parameter	**Description**
string1	Searched string
string2	String to search for

The strstr function returns a pointer to the first occurrence of *string2* in *string1*.

The _fstrstr function is a model-independent (large-model) form of the strstr function. The behavior and return value of _fstrstr are identical to those of the model-dependent function strstr, with the exception that the arguments and return value are far pointers.

Return Value

These functions return either a pointer to the first occurrence of *string2* in *string1*, or NULL if they do not find the string.

strstr

Standards: ANSI
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrstr

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strcspn
strncat
strncmp
strncpy
_strnicmp
strpbrk
strrchr
strspn

```
/* STRSTR.C */
#include <string.h>
#include <stdio.h>
char str[] =    "lazy";
char string[] = "The quick brown dog jumps over the lazy fox";
char fmt1[] =   "         1         2         3         4         5";
char fmt2[] =   "12345678901234567890123456789012345678901234567890";
void main( void )
{
   char *pdest;
   int  result;
   printf( "String to be searched:\n\t%s\n", string );
   printf( "\t%s\n\t%s\n\n", fmt1, fmt2 );
   pdest = strstr( string, str );
   result = pdest - string + 1;
   if( pdest != NULL )
      printf( "%s found at position %d\n\n", str, result );
   else
      printf( "%s not found\n", str );
}
```

_strtime

#include <time.h>

Syntax char *_strtime(char **timestr*);

Parameter	**Description**
timestr	Time string

The _strtime function copies the current time into the buffer pointed to by *timestr.* The time is formatted as

```
hh:mm:ss
```

where `hh` is two digits representing the hour in 24-hour notation, `mm` is two digits representing the minutes past the hour, and `ss` is two digits representing seconds. For example, the string

```
18:23:44
```

represents 23 minutes and 44 seconds past 6:00 PM.

The buffer must be at least nine bytes long.

Return Value

The _strtime function returns a pointer to the resulting text string *timestr*.

asctime
ctime
gmtime
localtime
mktime
time
_tzset

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* STRTIME.C */
#include <time.h>
#include <stdio.h>
void main( void )
{
   char dbuffer [9];
   char tbuffer [9];
   _strdate( dbuffer );
   printf( "The current date is %s \n", dbuffer );
   _strtime( tbuffer );
   printf( "The current time is %s \n", tbuffer );
}
```

strtod, strtol, _strtold, strtoul

#include <stdlib.h>

Syntax double strtod(const char **nptr*, char ***endptr*);

long strtol(const char **nptr*, char ***endptr*, int *base*);

long double _strtold(const char **nptr*, char ***endptr*);

unsigned long strtoul(const char **nptr*, char ***endptr*, int *base*);

Parameter	Description
nptr	String to convert
endptr	Pointer to character that stops scan
base	Number base to use

The strtod, _strtold, strtol, and strtoul functions convert a character string to a double-precision value, a long-double value, a long-integer value, or an unsigned long-integer value, respectively. The input string is a sequence of characters that can be interpreted as a numerical value of the specified type.

These functions stop reading the string at the first character they cannot recognize as part of a number. This may be the null character ('\0') at the end of the string. With strtol or strtoul, this terminating character can also be the first numeric character greater than or equal to *base*. If *endptr* is not NULL, a pointer to the character that stopped the scan is stored at the location pointed to by *endptr*. If no conversion could be performed (no valid digits were found or an invalid base was specified), the value of *nptr* is stored at the location pointed to by *endptr*.

The strtod and _strtold functions expect *nptr* to point to a string with the following form:

[*whitespace*] [*sign*] [*digits*] [.*digits*] [{d | D | e | E}[*sign*]*digits*]

A *whitespace* consists of space and tab characters, which are ignored; *sign* is either plus (+) or minus (–); and *digits* are one or more decimal digits. If no digits appear before the decimal point, at least one must appear after the decimal point. The decimal digits can be followed by an exponent, which consists of an introductory letter (b, D, e, or E) and an optionally signed decimal integer.

The first character that does not fit this form stops the scan.

The strtol and strtoul functions expect *nptr* to point to a string with the following form:

[*whitespace*] [{ + | –}] [0 [{ x | X }]] [*digits*]

If *base* is between 2 and 36, then it is used as the base of the number. If *base* is 0, the initial characters of the string pointed to by *nptr* are used to determine the base. If the first character is 0 and the second character is not 'x' or 'X', then the string is interpreted as an octal integer; otherwise, it is interpreted as a decimal number. If the first character is '0' and the second character is 'x' or 'X', then the string is interpreted as a hexadecimal integer. If the first character is '1' through '9', then the string is interpreted as a decimal integer. The letters 'a' through 'z' (or 'A' through 'Z') are assigned the values 10 through 35; only letters whose assigned values are less than *base* are permitted.

The strtoul function allows a plus (+) or minus (–) sign prefix; a leading minus sign indicates that the return value is negated.

Return Value

The strtod and _strtold functions return the value of the floating-point number, except when the representation would cause an overflow, in which case they return +/-HUGE_VAL. The functions return 0 if no conversion could be performed or an underflow occurred.

The strtol function returns the value represented in the string, except when the representation would cause an overflow, in which case it returns LONG_MAX or LONG_MIN. The function returns 0 if no conversion could be performed.

The strtoul function returns the converted value, if any. If no conversion can be performed, the function returns 0. The function returns ULONG_MAX on overflow.

In all four functions, errno is set to ERANGE if overflow or underflow occurs.

atof
atol

strtod, strtol

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strtold

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strtoul

Standards: ANSI
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* STRTOD.C: This program uses strtod to convert a
 * string to a double-precision value; strtol to
 * convert a string to long integer values; and strtoul
 * to convert a string to unsigned long-integer values.
 */
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   char   *string, *stopstring;
   double x;
   long   l;
   int    base;
   unsigned long ul;
   string = "3.1415926This stopped it";
   x = strtod( string, &stopstring );
   printf( "string = %s\n", string );
   printf("   strtod = %f\n", x );
   printf("   Stopped scan at: %s\n\n", stopstring );
   string = "-10110134932This stopped it";
   l = strtol( string, &stopstring, 10 );
   printf( "string = %s", string );
   printf("   strtol = %ld", l );
   printf("   Stopped scan at: %s", stopstring );
   string = "10110134932";
   printf( "string = %s\n", string );
   /* Convert string using base 2, 4, and 8: */
   for( base = 2; base <= 8; base *= 2 )
   {
      /* Convert the string: */
      ul = strtoul( string, &stopstring, base );
      printf( "   strtol = %ld (base %d)\n", ul, base );
      printf( "   Stopped scan at: %s\n", stopstring );
   }
}
```

strtok, _fstrtok

#include <string.h> Required only for function declarations

Syntax char *strtok(char **string1*, const char **string2*);

char __far * __far _fstrtok(char __far **string1*, const char __far * *string2*);

Parameter	**Description**
string1	String containing token(s)
string2	Set of delimiter characters

The strtok function reads *string1* as a series of zero or more tokens and *string2* as the set of characters serving as delimiters of the tokens in *string1*. The tokens in *string1* may be separated by one or more of the delimiters from *string2*.

The tokens can be broken out of *string1* by a series of calls to strtok. In the first call to strtok for *string1*, strtok searches for the first token in *string1*, skipping leading delimiters. A pointer to the first token is returned. To read the next token from *string1*, call strtok with a NULL value for the *string1* argument. The NULL *string1* argument causes strtok to search for the next token in the previous token string. The set of delimiters may vary from call to call, so *string2* can take any value.

The _fstrtok function is a model-independent (large-model) form of the strtok function. The behavior and return value of _fstrtok are identical to those of the model-dependent function strtok, with the exception that the arguments and return value are far pointers.

Note that calls to these functions will modify *string1*, because each time strtok is called it inserts a null character ('\0') after the token in *string1*.

Return Value

The first time strtok is called, it returns a pointer to the first token in *string1*. In later calls with the same token string, strtok returns a pointer to the next token in the string. A NULL pointer is returned when there are no more tokens. All tokens are null-terminated.

strtok

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrtok

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

strcspn
strspn

```
/* STRTOK.C: In this program, a loop uses strtok
 * to print all the tokens (separated by commas
 * or blanks) in the string named "string".
 */
#include <string.h>
#include <stdio.h>
char string[] = "A string\tof ,,tokens\nand some  more tokens";
char seps[]   = " ,\t\n";
char *token;
void main( void )
{
   printf( "%s\n\nTokens:\n", string );
   /* Establish string and get the first token: */
   token = strtok( string, seps );
   while( token != NULL )
   {
      /* While there are tokens in "string" */
      printf( " %s\n", token );
      /* Get next token: */
      token = strtok( NULL, seps );
   }
}
```

_strupr, _fstrupr

#include <string.h> Required only for function declarations

Syntax char *_strupr(char **string*);

char __far * __far _fstrupr(char __far **string*);

Parameter	**Description**
string	String to be capitalized

These functions convert any lowercase letters in the string to uppercase. Other characters are not affected.

The _fstrupr function is a model-independent (large-model) form of the _strupr function. The behavior and return value of _fstrupr are identical to those of the model-dependent function _strupr, with the exception that the argument and return value are far pointers.

Return Value

These functions return a pointer to the converted string. There is no error return.

_strupr

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fstrupr

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_strlwr

```c
/* STRLWR.C: This program uses _strlwr and _strupr to create
 * uppercase and lowercase copies of a mixed-case string.
 */
#include <string.h>
#include <stdio.h>
void main( void )
{
   char string[100] = "The String to End All Strings!";
   char *copy1, *copy2;
   copy1 = _strlwr( _strdup( string ) );
   copy2 = _strupr( _strdup( string ) );
   printf( "Mixed: %s\n", string );
   printf( "Lower: %s\n", copy1 );
   printf( "Upper: %s\n", copy2 );
}
```

strxfrm

#include <string.h> Required only for function declarations

Syntax size_t strxfrm(char **string1*, const char **string2*, size_t *count*);

Parameter	Description
string1	String to which transformed version of *string2* is returned
string2	String to transform
count	Maximum number of characters to be placed in *string1*

The strxfrm function transforms the string pointed to by *string2* into a new collated form that is stored in *string1*. No more than *count* characters (including the null character) are transformed and placed into the resulting string.

The transformation is made using the locale-specific information set by the setlocale function.

After the transformation, a call to strcmp with the two transformed strings will yield identical results to a call to strcoll applied to the original two strings.

The value of the following expression is the size of the array needed to hold the transformation of the source string:

```
1 + strxfrm( NULL, string, 0 )
```

Currently, the run-time library supports the "C" locale only; thus strxfrm is equivalent to the following:

```
strncpy( _string1, _string2, _count );
return( strlen( _string1 ) );
```

Return Value

The strxfrm function returns the length of the transformed string, not counting the terminating null character. If the return value is greater than or equal to *count*, the contents of *string1* are unpredictable.

localeconv
setlocale
strcmp
strncmp
strcoll

Standards: ANSI

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_swab

#include <stdlib.h> Required only for function declarations

Syntax void _swab(char **src*, char **dest*, int *n*);

Parameter	Description
src	Data to be copied and swapped
dest	Storage location for swapped data
n	Number of bytes to be copied and swapped

The _swab function copies *n* bytes from *src*, swaps each pair of adjacent bytes, and stores the result at *dest*. The integer *n* should be an even number to allow for swapping. The _swab function is typically used to prepare binary data for transfer to a machine that uses a different byte order.

Return Value

None.

Use _swab for compatibility with ANSI naming conventions of non-ANSI functions. Use swab and link with OLDNAMES.LIB for UNIX compatibility.

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* SWAB.C */
#include <stdlib.h>
#include <stdio.h>
char from[] = "BADCFEHGJILKNMPORQTSVUXWZY";
char to[] =   ".........................";
void main( void )
{
   printf( "Before:\t%s\n\t%s\n\n", from, to );
   _swab( from, to, sizeof( from ) );
   printf( "After:\t%s\n\t%s\n\n", from, to );
}
```

system

#include <process.h> Required only for function declarations

#include <stdlib.h> Use STDLIB.H for ANSI compatibility

Syntax int system(const char **command*);

Parameter	**Description**
command	Command to be executed

The system function passes *command* to the command interpreter, which executes the string as an operating-system command. The system function refers to the COMSPEC and PATH environment variables that locate the command-interpreter file (the file named COMMAND.COM in MS-DOS). If *command* is a pointer to an empty string, the function simply checks to see whether or not the command interpreter exists.

Return Value

If *command* is NULL and the command interpreter is found, the function returns a nonzero value. If the command interpreter is not found, it returns the value 0 and sets errno to ENOENT. If *command* is not NULL, the system function returns the value 0 if the command interpreter is successfully started.

A return value of -1 indicates an error, and errno is set to one of the following values:

E2BIG
: In MS-DOS, the argument list exceeds 128 bytes, or the space required for the environment information exceeds 32K.

ENOENT
: The command interpreter cannot be found.

ENOEXEC
: The command-interpreter file has an invalid format and is not executable.

ENOMEM
: Not enough memory is available to execute the command; the available memory has been corrupted; or an invalid block exists, indicating that the process making the call was not allocated properly.

Standards:	ANSI, UNIX
16-Bit:	MS-DOS

_exec functions
exit
_exit
_spawn functions

```
/* SYSTEM.C: This program uses
 * system to TYPE its source file.
 */
#include <process.h>
void main( void )
{
   system( "type system.c" );
}
```

tan Functions

#include <math.h>

Syntax double tan(double *x*);

double tanh(double *x*);

long double tanl(long double *x*);

long double tanhl(long double *x*);

Parameter	Description
x	Angle in radians

The tan functions return the tangent or hyperbolic tangent of their arguments. The list below describes the differences between the various tangent functions:

Function	Description
tan	Calculates tangent of *x*
tanh	Calculates hyperbolic tangent of *x*
tanl	Calculates tangent of *x* (80-bit version)
tanhl	Calculates hyperbolic tangent of *x* (80-bit version)

The tanl and tanhl functions are the 80-bit counterparts and use an 80-bit, 10-byte coprocessor form of arguments and return values. See long double functions for more details on this data type.

Return Value

The tan function returns the tangent of *x*. If *x* is greater than or equal to 2 raised to the power of 27, a partial loss of significance in the result may occur; in this case, tan sets errno to ERANGE and generates a _PLOSS error. If *x* is greater than or equal to 2 raised to the power of 31, significance is totally lost, tan prints a _TLOSS error message to stderr, sets errno to ERANGE, and returns 0. Error handling can be modified by using the _matherr function.

There is no error return for tanh.

tan, tanh

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

tanl, tanhl

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

acos functions
asin functions
atan functions
cos functions
sin functions

```
/* TAN.C:  This program displays the tangent of pi / 4
 * and the hyperbolic tangent of the result.
 */
#include <math.h>
#include <stdio.h>
void main( void )
{
   double pi = 3.1415926535;
   double x, y;
   x = tan( pi / 4 );
   y = tanh( x );
   printf( "tan( %f ) = %f\n", x, y );
   printf( "tanh( %f ) = %f\n", y, x );
}
```

_tell

#include <io.h> Required only for function declarations

Syntax long _tell(int *handle*);

Parameter	**Description**
handle	Handle referring to open file

The _tell function gets the current position of the file pointer (if any) associated with the *handle* argument. The position is expressed as the number of bytes from the beginning of the file.

Return Value

A return value of -1L indicates an error, and errno is set to EBADF to indicate an invalid file-handle argument. On devices incapable of seeking, the return value is undefined.

ftell
_lseek

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* TELL.C:  This program uses _tell to tell the
 * file pointer position after a file read.
 */
#include <io.h>
#include <stdio.h>
#include <fcntl.h>
void main( void )
{
   int  fh;
   char buffer[500];
   if( (fh = _open( "tell.c", _O_RDONLY )) != -1 )
   {
      if( _read( fh, buffer, 500 ) > 0 )
         printf( "Current file position is: %d\n", _tell( fh ) );
      _close( fh );
   }
}
```

_tempnam, tmpnam

#include <stdio.h>

Syntax char *_tempnam(char **dir*, char **prefix*);

char *tmpnam(char **string*);

Parameter	Description
string	Pointer to temporary name
dir	Target directory to be used if TMP not defined
prefix	Filename prefix

The tmpnam function generates a temporary filename that can be used to open a temporary file without overwriting an existing file. This name is stored in *string*. If *string* is NULL, then tmpnam leaves the result in an internal static buffer. Thus, any subsequent calls destroy this value. If *string* is not NULL, it is assumed to point to an array of at least L_tmpnam bytes (the value of L_tmpnam is defined in STDIO.H). The function will generate unique filenames for up to TMP_MAX calls.

The character string that tmpnam creates consists of the path prefix, defined by the entry P_tmpdir in the file STDIO.H, followed by a sequence consisting of the digit characters '0' through '9'; the numerical value of this string can range from 1 to 65,535. Changing the definitions of L_tmpnam or P_tmpdir in STDIO.H does not change the operation of tmpnam.

The _tempnam function allows the program to create a temporary filename for use in another directory. This filename will be different from that of any existing file. The *prefix* argument is the prefix to the filename. The _tempnam function uses malloc to allocate space for the filename; the program is responsible for freeing this space when it is no longer needed. The _tempnam function looks for the file with the given name in the following directories, listed in order of precedence:

Directory Used	Conditions
Directory specified by TMP	TMP environment variable is set, and directory specified by TMP exists.
dir argument to _tempnam	TMP environment variable is not set, or directory specified by TMP does not exist.
P_tmpdir in STDIO.H	The *dir* argument is NULL, or *dir* is name of nonexistent directory.
Current working directory	P_tmpdir does not exist.

If the search through the locations listed above fails, _tempnam returns the value NULL.

Return Value

The tmpnam and _tempnam functions both return a pointer to the name generated, unless it is impossible to create this name or the name is not unique. If the name cannot be created or if a file with that name already exists, tmpnam and _tempnam return the value NULL.

Use _tempnam for compatibility with ANSI naming conventions of non-ANSI functions. Use tempnam and link with OLDNAMES.LIB for UNIX compatibility.

_tempnam

Standards: UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

tmpnam

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

tmpfile

```
/* TEMPNAM.C: This program uses tmpnam to create a unique
 * filename in the current working directory, then uses
 * _tempnam to create a unique filename with a prefix of stq.
 */
#include <stdio.h>
void main( void )
{
   char *name1, *name2;
   /* Create a temporary filename for the current working directory: */
   if( ( name1 = tmpnam( NULL ) ) != NULL )
      printf( "%s is safe to use as a temporary file.\n", name1 );
   else
      printf( "Cannot create a unique filename\n" );
   /* Create a temporary file name in temporary directory with the
    * prefix "stq". The actual destination directory may vary
    * depending on the state of the TMP environment variable and
    * the global variable P_tmpdir.
    */
   if( ( name2 = _tempnam( "c:\\tmp", "stq" ) ) != NULL )
      printf( "%s is safe to use as a temporary file.\n", name2 );
   else
      printf( "Cannot create a unique filename\n" );
}
```

time

#include <time.h> Required only for function declarations

Syntax time_t time(time_t **timer*);

Parameter	**Description**
timer	Storage location for time

The time function returns the number of seconds elapsed since midnight (00:00:00), January 1, 1970, Universal Coordinated Time, according to the system clock. The system time is adjusted according to the _timezone system variable, which is explained under _tzset.

The return value is stored in the location given by *timer*. This parameter may be NULL, in which case the return value is not stored.

Return Value

The time function returns the time in elapsed seconds. There is no error return.

asctime
_ftime
gmtime
localtime
_tzset
_utime

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* TIMES.C illustrates various time and date functions including:
 *      time            _ftime          ctime       asctime
 *      localtime       gmtime          mktime      _tzset
 *      _strtime        _strdate        strftime
 *
 * Also the global variable:
 *      _tzname
 */

#include <time.h>
#include <stdio.h>
#include <sys\types.h>
#include <sys\timeb.h>
#include <string.h>

void main()
{
    char tmpbuf[128], ampm[] = "AM";
    time_t ltime;
    struct _timeb tstruct;
    struct tm *today, *gmt, xmas = { 0, 0, 12, 25, 11, 91 };

    /* Set time zone from TZ environment variable. If TZ is not set,
     * PST8PDT is used (Pacific standard time, daylight savings).
     */
    _tzset();

    /* Display MS-DOS-style date and time. */
    _strtime( tmpbuf );
    printf( "MS-DOS time:\t\t\t\t%s\n", tmpbuf );
    _strdate( tmpbuf );
    printf( "MS-DOS date:\t\t\t\t%s\n", tmpbuf );

    /* Get UNIX-style time and display as number and string. */
    time( &ltime );
    printf( "Time in seconds since UCT 1/1/70:\t%ld\n", ltime );
    printf( "UNIX time and date:\t\t\t%s", ctime( &ltime ) );

    /* Display UCT. */
    gmt = gmtime( &ltime );
    printf( "Universal coordinated time:\t\t%s", asctime( gmt ) );

    /* Convert to time structure and adjust for PM if necessary. */
    today = localtime( &ltime );
    if( today->tm_hour > 12 )
    {
   strcpy( ampm, "PM" );
   today->tm_hour -= 12;
    }
    if( today->tm_hour == 0 )  /* Adjust if midnight hour. */
   today->tm_hour = 12;

    /* Note how pointer addition is used to skip the first 11 characters
     * and printf is used to trim off terminating characters.
     */
    printf( "12-hour time:\t\t\t\t%.8s %s\n",
       asctime( today ) + 11, ampm );

    /* Print additional time information. */
    _ftime( &tstruct );
```

```
    printf( "Plus milliseconds:\t\t\t%u\n", tstruct.millitm );
    printf( "Zone difference in seconds from UCT:\t%u\n", tstruct.timezone );
    printf( "Time zone name:\t\t\t\t%s\n", _tzname[0] );
    printf( "Daylight savings:\t\t\t%s\n", tstruct.dstflag ? "YES" : "NO" );

    /* Make time for noon on Christmas, 1992. */
    if( mktime( &xmas ) != (time_t)-1 )
   printf( "Christmas\t\t\t\t%s\n", asctime( &xmas ) );

    /* Use time structure to build a customized time string. */
    today = localtime( &ltime );

    /* Use strftime to build a customized time string. */
    strftime( tmpbuf, 128,
         "Today is %A, day %d of %B in the year %Y.\n", today );
    printf( tmpbuf );
}
```

tmpfile

#include <stdio.h>

Syntax FILE *tmpfile(void);

The tmpfile function creates a temporary file and returns a pointer to that stream. If the file cannot be opened, tmpfile returns a NULL pointer.

This temporary file is automatically deleted when the file is closed, when the program terminates normally, or when _rmtmp is called, assuming that the current working directory does not change. The temporary file is opened in w+b (binary read/write) mode.

Return Value

If successful, the tmpfile function returns a stream pointer. Otherwise, it returns a NULL pointer.

_rmtmp
_tempnam
tmpnam

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* TMPFILE.C: This program uses tmpfile to create a
 * temporary file, then deletes this file with _rmtmp.
 */
#include <stdio.h>
void main( void )
{
   FILE *stream;
   char tempstring[] = "String to be written";
   int  i;
   /* Create temporary files. */
   for( i = 1; i <= 10; i++ )
   {
      if( (stream = tmpfile()) == NULL )
         perror( "Could not open new temporary file\n" );
      else
         printf( "Temporary file %d was created\n", i );
   }
   /* Remove temporary files. */
   printf( "%d temporary files deleted\n", _rmtmp() );
}
```

__toascii, tolower, toupper Functions

#include <ctype.h>

Syntax int __toascii(int *c*);
int tolower(int *c*);
int _tolower(int *c*);
int toupper(int *c*);
int _toupper(int *c*);

Parameter	**Description**
c	Character to be converted

The __toascii, tolower, _tolower, toupper, and _toupper routines and their associated macros convert a single character, as described below:

Function	**Macro**	**Description**
__toascii	_toascii	Converts *c* to ASCII character
tolower	tolower	Converts *c* to lowercase if appropriate
_tolower	_tolower	Converts *c* to lowercase
toupper	toupper	Converts *c* to uppercase if appropriate
_toupper	_toupper	Converts *c* to uppercase

The __toascii routine sets all but the low-order 7 bits of *c* to 0, so that the converted value represents a character in the ASCII character set. If *c* already represents an ASCII character, *c* is unchanged.

The tolower routine converts *c* to lowercase if *c* represents an uppercase letter. Otherwise, *c* is unchanged.

The _tolower routine is a version of tolower to be used only when *c* is known to be uppercase. The result of _tolower is undefined if *c* is not an uppercase letter.

The toupper routine convers *c* to uppercase if *c* represents an lowercase letter. Otherwise, *c* is unchanged.

The _toupper routine is a version of toupper to be used only when *c* is known to be lowercase. The result of _toupper is undefined if *c* is not a lowercase letter.

These routines are implemented both as functions and as macros. To conform to the ANSI specification, the tolower and toupper routines are also implemented as functions. The function versions can be used by removing the macro definitions through #undef directives or by not including CTYPE.H. Function declarations of tolower and toupper are given in STDLIB.H.

If the /Za compile option is used, the macro form of toupper or tolower is not used because it evaluates its argument more than once. Because the arguments are evaluated more than once, arguments with side effects would produce potentially bad results.

Return Value

The __toascii, tolower, _tolower, toupper, and _toupper routines return the converted character *c*. There is no error return.

Use __toascii for compatibility with ANSI naming conventions of non-ANSI functions. Use toascii and link with OLDNAMES.LIB for UNIX compatibility.

__toascii, _tolower, _toupper

Standards: UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

tolower, toupper

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

is functions

```
/* TOUPPER.C: This program uses toupper and tolower to
 * analyze all characters between 0x0 and 0x7F. It also
 * applies _toupper and _tolower to any code in this
 * range for which these functions make sense.
 */
#include <conio.h>
#include <ctype.h>
#include <string.h>
char msg[] = "Some of THESE letters are Capitals\r\n";
char *p;
void main( void )
{
   _cputs( msg );
   /* Reverse case of message. */
   for( p = msg; p < msg + strlen( msg ); p++ )
   {
      if( islower( *p ) )
         _putch( _toupper( *p ) );
      else if( isupper( *p ) )
         _putch( _tolower( *p ) );
      else
         _putch( *p );
   }
}
```

_tzset

#include <time.h> Required only for function declarations

Syntax void _tzset(void);

int _daylight
long_timezone
char *_tzname[2] Global variables set by function

The _tzset function uses the current setting of the environment variable TZ to assign values to three global variables: _daylight, _timezone, and _tzname. These variables are used by the _ftime and localtime functions to make corrections from Universal Coordinated Time (UCT) to local time, and by time to compute UCT from system time.

Use the following syntax to set the TZ environment variable:

set TZ=*tzn*[+ | −]*hh*[:*mm*[:*ss*]][*dzn*]

The *tzn* must be a three-letter time-zone name, such as PST, followed by an optionally signed number, + −*hh*, giving the difference in hours between UCT and local time. To specify the exact local time, the hours can be followed by minutes, :*mm*; seconds, :*ss*; and a three-letter daylight-saving-time zone, *dzn*, such as PDT. Separate hours, minutes, and seconds with colons (:). If daylight saving time is never in effect, as is the case in certain states and localities, set TZ without a value for *dzn*.

If the TZ value is not currently set, the default is PST8PDT, which corresponds to the Pacific time zone.

Based on the TZ environment variable value, the following values are assigned to the variables _daylight, _timezone, and _tzname when _tzset is called:

Variable	Value
_daylight	Nonzero value if a daylight-saving-time zone is specified in the TZ setting; otherwise, 0
_timezone	Difference in seconds between UCT and local time
_tzname[0]	String value of the three-letter time-zone name from the TZ environmental variable
_tzname[1]	String value of the daylight-saving-time zone, or an empty string if the daylight-saving-time zone is omitted from the TZ environmental variable

The default for _daylight is 1; for _timezone, 28,800; for _tzname[0], PST; and for _tzname[1], PDT. This corresponds to "PST8PDT."

If the DST zone is omitted from the TZ environmental variable, the _daylight variable will be 0 and the _ftime, gmtime, and localtime functions will return 0 for their DST flags.

Return Value

None.

Use _tzset for compatibility with ANSI naming conventions of non-ANSI functions. Use tzset and link with OLDNAMES.LIB for UNIX compatibility.

asctime
_ftime
gmtime
localtime
time

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* TZSET.C: This program first sets up the time zone by
 * placing the variable named TZ=EST5 in the environment
 * table. It then uses _tzset to set the global variables
 * named _daylight, _timezone, and _tzname.
 */
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   if( _putenv( "TZ=EST5EDT" ) == -1 )
   {
      printf( "Unable to set TZ\n" );
      exit( 1 );
   }
   else
   {
      _tzset();
      printf( "_daylight = %d\n", _daylight );
      printf( "_timezone = %ld\n", _timezone );
      printf( "_tzname[0] = %s\n", _tzname[0] );
   }
   exit( 0 );
}
```

_ultoa

#include <stdlib.h> Required only for function declarations

Syntax char *_ultoa(unsigned long *value*, char **string*, int *radix*);

Parameter	Description
value	Number to be converted
string	String result
radix	Base of *value*

The _ultoa function converts *value* to a null-terminated character string and stores the result (up to 33 bytes) in *string*. No overflow checking is performed. The *radix* argument specifies the base of *value*; it must be in the range 2-36.

Return Value

The _ultoa function returns a pointer to *string*. There is no error return.

__itoa
__ltoa

Standards: None

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_umask

#include <sys\types.h>, <sys\stat.h>

#include <io.h> Required only for function declarations

Syntax int _umask(int *pmode*);

Parameter	**Description**
pmode	Default permission setting

The _umask function sets the file-permission mask of the current process to the mode specified by *pmode*. The file-permission mask is used to modify the permission setting of new files created by _creat, _open, or _sopen. If a bit in the mask is 1, the corresponding bit in the file's requested permission value is set to 0 (disallowed). If a bit in the mask is 0, the corresponding bit is left unchanged. The permission setting for a new file is not set until the file is closed for the first time.

The argument *pmode* is a constant expression containing one or both of the manifest constants _S_IREAD and _S_IWRITE, defined in SYS.H. When both constants are given, they are joined with the bitwise-OR operator (|). The meaning of the *pmode* argument is as follows:

Value	**Meaning**
_S_IREAD	Reading not allowed (file is write-only)
_S_IWRITE	Writing not allowed (file is read-only)

For example, if the write bit is set in the mask, any new files will be read-only.

Note that with MS-DOS, all files are readable; it is not possible to give write-only permission. Therefore, setting the read bit with _umask has no effect on the file's modes.

Return Value

The _umask function returns the previous value of *pmode*. There is no error return.

Use _umask for compatibility with ANSI naming conventions of non-ANSI functions. Use umask and link with OLDNAMES.LIB for UNIX compatibility.

_chmod
_creat
_mkdir
_open

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* UMASK.C: This program uses _umask to set
 * the file-permission mask so that all future
 * files will be created as read-only files.
 * It also displays the old mask.
 */
#include <sys\stat.h>
#include <sys\types.h>
#include <io.h>
#include <stdio.h>
void main( void )
{
   int oldmask;
   /* Create read-only files: */
   oldmask = _umask( _S_IWRITE );
   printf( "Oldmask = 0x%.4x\n", oldmask );
}
```

ungetc

#include <stdio.h>

Syntax int ungetc(int *c*, FILE **stream*);

Parameter	**Description**
c	Character to be pushed
stream	Pointer to FILE structure

The ungetc function pushes the character *c* back onto *stream* and clears the end-of-file indicator. The stream must be open for reading. A subsequent read operation on the stream starts with *c*. An attempt to push EOF onto the stream using ungetc is ignored. The ungetc function returns an error value if nothing has yet been read from *stream* or if *c* cannot be pushed back.

Characters placed on the stream by ungetc may be erased if fflush, fseek, fsetpos, or rewind is called before the character is read from the stream. The file-position indicator will have the same value it had before the characters were pushed back. On a successful ungetc call against a text stream, the file-position indicator is unspecified until all the pushed-back characters are read or discarded. On each successful ungetc call against a binary stream, the file-position indicator is stepped down; if its value was 0 before a call, the value is undefined after the call.

Results are unpredictable if the ungetc function is called twice without a read operation between the two calls. After a call to the fscanf function, a call to ungetc may fail unless another read operation (such as the getc function) has been performed. This is because the fscanf function itself calls the ungetc function.

Return Value

The ungetc function returns the character argument *c*. The return value EOF indicates a failure to push back the specified character.

getc
getchar
putc
putchar

Standards: ANSI, UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* UNGETC.C: This program first converts a character
 * representation of an unsigned integer to an integer. If
 * the program encounters a character that is not a digit,
 * the program uses ungetc to replace it in the  stream.
 */
#include <stdio.h>
#include <ctype.h>
void main( void )
{
   int ch;
   int result = 0;
   printf( "Enter an integer: " );
   /* Read in and convert number: */
   while( ((ch = getchar()) != EOF) && isdigit( ch ) )
      result = result * 10 + ch - '0';    /* Use digit. */
   if( ch != EOF )
      ungetc( ch, stdin );                /* Put nondigit back. */
   printf( "Number = %d\nNextcharacter in stream = '%c'", result, getchar() );
}
```

_ungetch

#include <conio.h> Required only for function declarations

Syntax int _ungetch(int *c*);

Parameter	**Description**
c	Character to be pushed

The _ungetch function pushes the character *c* back to the console, causing *c* to be the next character read by _getch or _getche. The _ungetch function fails if it is called more than once before the next read. The *c* argument may not be EOF.

Return Value

The _ungetch function returns the character *c* if it is successful. A return value of EOF indicates an error.

_cscanf
_getch
_getche

Standards: None
16-Bit: MS-DOS

```
/* UNGETCH.C: In this program, a white-space delimited
 * token is read from the keyboard. When the program
 * encounters a delimiter, it uses _ungetch to replace
 * the character in the keyboard buffer.
 */
#include <conio.h>
#include <ctype.h>
#include <stdio.h>
void main( void )
{
   char buffer[100];
   int count = 0;
   int ch;
   ch = _getche();
   while( isspace( ch ) )      /* Skip preceding white space. */
      ch = _getche();
   while( count < 99 )         /* Gather token. */
   {
      if( isspace( ch ) )      /* End of token. */
         break;
      buffer[count++] = (char)ch;
      ch = _getche();
   }
   _ungetch( ch );            /* Put back delimiter. */
   buffer[count] = '\0';      /* Null terminate the token. */
   printf( "\ntoken = %s\n", buffer );
}
```

_unlink

#include <io.h> Required only for function declarations

#include <stdio.h>

Syntax int _unlink(const char **filename*);

Parameter	**Description**
filename	Name of file to remove

The _unlink function deletes the file specified by *filename*.

Return Value

If successful, _unlink returns 0; otherwise, it returns -1 and sets errno to one of the following constants:

Value	**Meaning**
EACCES	Path specifies a read-only file
ENOENT	Filename or path not found, or path specified a directory

Use _unlink for compatibility with ANSI naming conventions of non-ANSI functions. Use unlink and link with OLDNAMES.LIB for UNIX compatibility.

_close
remove

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* UNLINK.C: This program uses _unlink to delete UNLINK.OBJ. */
#include <stdio.h>
void main( void )
{
   if( _unlink( "_unlink.obj" ) == -1 )
      perror( "Could not delete 'UNLINK.OBJ'" );
   else
      printf( "Deleted 'UNLINK.OBJ'\n" );
}
```

_unregisterfonts

#include <graph.h>

Syntax void __far _unregisterfonts(void);

The _unregisterfonts function frees memory previously allocated and used by the _registerfonts function. The _unregisterfonts function removes the header information for all fonts and unloads the currently selected font data from memory.

Any attempt to use the _setfont function or the _outgtext function after calling _unregisterfonts results in an error.

Return Value

None.

_getfontinfo
_getgtextextent
_outgtext
_registerfonts
_setfont

Standards: None

16-Bit: MS-DOS

_utime

#include <sys\types.h>, <sys\utime.h>

Syntax int _utime(const char *_filename_, struct _utimbuf *_times_);

Parameter	Description
filename	Filename
times	Pointer to stored time values

The _utime function sets the modification time for the file specified by *filename*. The process must have write access to the file; otherwise, the time cannot be changed.

Although the _utimbuf structure contains a field for access time, only the modification time is set with MS-DOS. If *times* is a NULL pointer, the modification time is set to the current time. Otherwise, *times* must point to a structure of type _utimbuf, defined in SYS\UTIME.H. The modification time is set from the modtime field in this structure.

Return Value

The _utime function returns the value 0 if the file-modification time was changed. A return value of -1 indicates an error, and errno is set to one of the following values:

Value	Meaning
EACCES	Path specifies directory or read-only file
EINVAL	Invalid argument; the *times* argument is invalid
EMFILE	Too many open files (the file must be opened to change its modification time)
ENOENT	Filename or path not found

Use _utime for compatibility with ANSI naming conventions of non-ANSI functions. Use utime and link with OLDNAMES.LIB for UNIX compatibility.

asctime
ctime
_fstat
_ftime
gmtime
localtime
_stat
time

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* UTIME.C: This program uses _utime to set the
 * file-modification time to the current time.
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys\types.h>
#include <sys\utime.h>
void main( void )
{
   /* Show file time before and after. */
   system( "dir utime.c" );
   if( _utime( "utime.c", NULL ) == -1 )
      perror( "_utime failed\n" );
   else
      printf( "File time modified\n" );
   system( "dir utime.c" );
}
```

va_arg, va_end, va_start

#include <stdarg.h> Required for ANSI compatibility

#include <varargs.h> Required for UNIX V compatibility

#include <stdio.h>

Syntax *type* va_arg(va_list *arg_ptr*, *type*);

void va_end(va_list *arg_ptr*);

void va_start(va_list *arg_ptr*); (UNIX version)

void va_start(va_list *arg_ptr*, *prev_param*); (ANSI)

Parameter	Description
arg_ptr	Pointer to list of arguments
prev_param	Parameter preceding first optional argument (ANSI only)
type	Type of argument to be retrieved

The va_arg, va_end, and va_start macros provide a portable way to access the arguments to a function when the function takes a variable number of arguments. Two versions of the macros are available: the macros defined in STDARG.H conform to the ANSI C standard, and the macros defined in VARARGS.H are compatible with the UNIX System V definition. The macros are listed below:

Macro	Description
va_alist	Name of parameter to called function (UNIX version only)
va_arg	Macro to retrieve current argument
va_dcl	Declaration of va_alist (UNIX version only)
va_end	Macro to reset *arg_ptr*
va_list	The typedef for the pointer to list of arguments
va_start	Macro to set *arg_ptr* to beginning of list of optional arguments (UNIX version only)

Both versions of the macros assume that the function takes a fixed number of required arguments, followed by a variable number of optional arguments. The required arguments are declared as ordinary parameters to the function and can be accessed through the parameter names. The optional arguments are accessed through the macros in STDARG.H or VARARGS.H, which set a pointer to the first optional argument in the argument list, retrieve arguments from the list, and reset the pointer when argument processing is completed.

The ANSI C standard macros, defined in STDARG.H, are used as follows:

1. All required arguments to the function are declared as parameters in the usual way. The va_dcl macro is not used with the STDARG.H macros.
2. The va_start macro sets *arg_ptr* to the first optional argument in the list of arguments passed to the function. The argument *arg_ptr* must have va_list type. The argument *prev_param* is the name of the required parameter immediately preceding the first optional argument in the argument list. If

prev_param is declared with the register storage class, the macro's behavior is undefined. The va_start macro must be used before va_arg is used for the first time.

3. The va_arg macro does the following:

- Retrieves a value of *type* from the location given by *arg_ptr*
- Increments *arg_ptr* to point to the next argument in the list, using the size of *type* to determine where the next argument starts

The va_arg macro can be used any number of times within the function to retrieve arguments from the list.

4. After all arguments have been retrieved, va_end resets the pointer to NULL.

The UNIX System V macros, defined in VARARGS.H, operate in a slightly different manner, as follows:

1. Any required arguments to the function can be declared as parameters in the usual way.
2. The last (or only) parameter to the function represents the list of optional arguments. This parameter must be named va_alist (not to be confused with va_list, which is defined as the type of va_alist).
3. The va_dcl macro appears after the function definition and before the opening left brace of the function. This macro is defined as a complete declaration of the va_alist parameter, including the terminating semicolon; therefore, no semicolon should follow va_dcl.
4. Within the function, the va_start macro sets *arg_ptr* to the beginning of the list of optional arguments passed to the function. The va_start macro must be used before va_arg is used for the first time. The argument *arg_ptr* must have va_list type.
5. The va_arg macro does the following:

- Retrieves a value of *type* from the location given by *arg_ptr*
- Increments *arg_ptr* to point to the next argument in the list, using the size of *type* to determine where the next argument starts

The va_arg macro can be used any number of times within the function to retrieve the arguments from the list.

6. After all arguments have been retrieved, va_end resets the pointer to NULL.

Return Value

The va_arg macro returns the current argument; va_start and va_end do not return values.

vfprintf

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* VA.C: The program below illustrates passing a variable
 * number of arguments using the following macros:
 *      va_start            va_arg              va_end
 *      va_list             va_decl (UNIX only)
 */
#include <stdio.h>
#define ANSI            /* Comment out for UNIX version     */
#ifdef ANSI             /* ANSI compatible version          */
#include <stdarg.h>
int average( int first, ... );
#else                   /* UNIX compatible version          */
#include <varargs.h>
int average( va_list );
#endif
void main( void )
{
   /* Call with 3 integers (-1 is used as terminator). */
   printf( "Average is: %d\n", average( 2, 3, 4, -1 ) );
   /* Call with 4 integers. */
   printf( "Average is: %d\n", average( 5, 7, 9, 11, -1 ) );
   /* Call with just -1 terminator. */
   printf( "Average is: %d\n", average( -1 ) );
}
/* Returns the average of a variable list of integers. */
#ifdef ANSI             /* ANSI compatible version    */
int average( int first, ... )
{
   int count = 0, sum = 0, i = first;
   va_list marker;
   va_start( marker, first );     /* Initialize variable arguments. */
   while( i != -1 )
   {
      sum += i;
      count++;
      i = va_arg( marker, int);
   }
   va_end( marker );              /* Reset variable arguments.      */
   return( sum ? (sum / count) : 0 );
}
#else       /* UNIX compatible version must use old-style definition.  */
int average( va_alist )
va_dcl
{
   int i, count, sum;
   va_list marker;
   va_start( marker );            /* Initialize variable arguments. */
   for( sum = count = 0; (i = va_arg( marker, int)) != -1; count++ )
      sum += i;
   va_end( marker );              /* Reset variable arguments.      */
   return( sum ? (sum / count) : 0 );
}
#endif
```

vfprintf, vprintf, vsprintf, _vsnprintf

#include <stdio.h>

#include <varargs.h> Required for UNIX System V compatibility

#include <stdarg.h> Required for ANSI compatibility

Syntax int vfprintf(FILE **stream*, const char **format*, va_list *argptr*);

int vprintf(const char **format*, va_list *argptr*);

int vsprintf(char **buffer*, const char **format*, va_list *argptr*);

int _vsnprintf(char **buffer*, size_t *count*, const char **format*, va_list *argptr*);

Parameter	**Description**
stream	Pointer to FILE structure
format	Format control
argptr	Pointer to list of arguments
buffer	Storage location for output
count	Maximum number of bytes

The vfprintf, vprintf, and vsprintf functions format data and output data to the file specified by *stream*, to standard output, and to the memory pointed to by *buffer*, respectively. The _vsnprintf function differs from vsprintf in that it writes not more than *count* bytes to *buffer*. These functions are similar to their counterparts fprintf, printf, and sprintf, but each accepts a pointer to a list of arguments instead of an argument list.

The *format* argument has the same form and function as the *format* argument for the printf function; see printf for a description of *format*.

The *argptr* parameter has type va_list, which is defined in the include files VARARGS.H and STDARG.H. The *argptr* parameter points to a list of arguments that are converted and output according to the corresponding format specifications in the format.

Return Value

The return value for vprintf, vsprintf, and _vsnprintf is the number of characters written, not counting the terminating null character. For _vsnprintf, if the number of bytes to write exceeds *buffer*, then *count* bytes are written and -1 is returned. If successful, the vfprintf return value is the number of characters written. If an output error occurs, it is a negative value.

vfprintf, vsprintf

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN, WIN

vprintf

Standards: ANSI, UNIX
16-Bit: MS-DOS, QWIN

_vsnprintf

Standards: None
16-Bit: MS-DOS, QWIN

fprintf
printf
sprintf
va_arg
va_end
va_start

```
/* WPRINTF.C shows how to use vprintf functions to write new
 * versions of printf. Functions in this catagory include:
 *      vsprintf     vprintf     vfprintf     _vsnprintf
 *
 * The vsprintf function is used in the example. The other
 * variations can be used similarly. For other examples of
 * formatted output, see EXTDIR.C (sprintf), TABLE.C
 * (fprintf), and PRINTF.C.
 */

#include <stdio.h>
#include <graph.h>
#include <string.h>
#include <stdarg.h>
#include <malloc.h>

int wprintf( short row, short col, short clr, long bclr, char *fmt, ... );

void main()
{
    short fgd = 0;
    long  bgd = 0L;

    _clearscreen( _GCLEARSCREEN );
    _outtext( "Color text example:\n\n" );

    /* Loop through 8 background colors. */
    for( bgd = 0L; bgd < 8; bgd++ )
    {
        wprintf( (short)(bgd + 3), 1, 15, bgd, "Back: %d Fore:", bgd );

        /* Loop through 16 foreground colors. */
        for( fgd = 0; fgd < 16; fgd++ )
            wprintf( -1, -1, fgd, -1L, " %2d ", fgd );
    }
}

/* Full-screen window version of printf that takes row, column, textcolor,
 * and background color as its first arguments, followed by normal printf
 * format strings (except that \t is not handled). You can specify -1 for
 * any of the first arguments to use the current value. The function returns
 * the number of characters printed, or a negative number for errors.
 */
int wprintf( short row, short col, short clr, long bclr, char *fmt, ... )
{
    struct  _rccoord tmppos;
    va_list marker;
    char    *buffer;
    int     increment = 256, size = increment, written = -1;

    /* Set text position. */
    tmppos = _gettextposition();
    if( row < 1 )
        row = tmppos.row;
    if( col < 1 )
        col = tmppos.col;
    _settextposition( row, col );

    /* Set foreground and background colors. */
    if( clr >= 0 )
```

```
        _settextcolor( clr );
    if( bclr >= 0 )
        _setbkcolor( bclr );

    /* Write text to a string and output the string. */
    va_start( marker, fmt );
    while ( written == -1 )
    {
        if(( buffer = (char *)malloc( size )) == NULL )
            return -1;
        written = _vsnprintf( buffer, size, fmt, marker );
        if ( written == -1 )
        {
             size += increment;
        }
     }
    va_end( marker );
    _outtext( buffer );
    free( buffer );
    return written;
}
```

_vfree

#include <vmemory.h>

Syntax void __far _vfree(_vmhnd_t *handle*);

Parameter	Description
handle	Handle to previously allocated virtual memory block

The _vfree function deallocates a virtual memory block. The argument *handle* points to a virtual memory block previously allocated through a call to _vmalloc or _vrealloc. The number of bytes freed is the number of bytes specified when the block was allocated (or reallocated, in the case of _vrealloc). The block must be unlocked before it is freed; use _vlockcnt to ensure that the block is unlocked. After the call, the freed block is available for reuse by the virtual heap.

Return Value

None.

_vlock
_vlockcnt
_vmalloc
_vrealloc
_vunlock

Standards:	None
16-Bit:	MS-DOS

_vheapinit

#include <vmemory.h>

Syntax int __far _vheapinit(unsigned int *dosmin*, unsigned int *dosmax*, unsigned int *swaparea*);

Parameter	Description
dosmin	Minimum amount of MS-DOS memory that must be available for the virtual memory manager to install itself, in paragraphs
dosmax	Maximum amount of MS-DOS memory that the virtual memory manager can use, in paragraphs
swaparea	Type of auxiliary memory to use

The _vheapinit routine initializes the virtual memory manager in preparation for future allocations. It must be called before any virtual memory blocks are requested.

The _vheapinit function may round up the minimum value specified. After rounding, if the minimum amount of MS-DOS memory is not available, _vheapinit does not initialize the virtual memory manager and returns 0. The virtual memory manager requires several kilobytes to function effectively.

If _VM_ALLDOS is specified for the *dosmax* argument, the virtual memory manager uses all available MS-DOS memory.

The *swaparea* argument specifies which types of auxiliary memory the virtual memory manager can use to hold blocks of memory that are swapped out. The argument can be one or more of the following manifest constants, combined with the bitwise-OR operator (|):

Value	Meaning
_VM_EMS	Use expanded memory
_VM_XMS	Use extended memory
_VM_DISK	Use disk space
_VM_ALLSWAP	(_VM_EMS | _VM_XMS | _VM_DISK)

If not all of the specified forms of storage are available, the virtual memory manager uses what is available.

After the program is done using virtual memory, it must call _vheapterm to terminate the virtual memory manager. A program can contain multiple pairs of _vheapinit/_vheapterm calls.

Warning If the program terminates without a call to _vheapterm, various system memory resources may not be available to subsequent programs.

To specify that no minimum amount of memory is required for installation of the virtual memory manager and to use all available MS-DOS memory in the virtual heap and all auxiliary storage, use the following command:

```
if( _vheapinit( 0, _VM_ALLDOS, _VM_ALLSWAP) == 0 )
   /* Error */
```

Return Value

The _vheapinit function returns a nonzero value if the virtual memory manager was successfully initialized. Otherwise, it returns 0.

_vheapterm

Standards: None

16-Bit: MS-DOS

_vheapterm

#include <vmemory.h>

Syntax void __far _vheapterm(void);

The _vheapterm function terminates the virtual memory manager and releases all resources that it used.

Warning If the program terminates without a call to _vheapterm, various system memory resources may not be available to subsequent programs.

If the virtual memory manager has not been initialized or has already been terminated when _vheapterm is called, the function returns immediately.

Return Value

None.

_vheapinit

Standards: None

16-Bit: MS-DOS

_vload

#include <vmemory.h>

Syntax void __far *__far _vload(_vmhnd_t *handle*, int *dirty*);

Parameter	Description
handle	Handle to previously allocated virtual memory block
dirty	Flag indicating whether the block should be written out or discarded when swapping occurs

The _vload function loads a virtual memory block into MS-DOS memory and returns a far pointer to it. The argument *handle* points to a virtual memory block previously allocated through a call to _vmalloc or _vrealloc.

The block of memory is not locked and may be swapped out if the virtual memory manager needs the memory. Consequently, the pointer returned by _vload is valid only until the next call to the virtual memory manager.

The *dirty* flag indicates whether the block of memory should be written out or discarded when swapping occurs. It can have one of the following values:

Value	Meaning
_VM_CLEAN	Discard contents of block when swapping occurs
_VM_DIRTY	Write contents of block to auxiliary memory when swapping occurs

Return Value

The _vload function returns a far pointer to MS-DOS memory if the virtual memory block is successfully loaded. If insufficient MS-DOS memory is available, _vload returns NULL.

_vlock
_vmalloc
_vunlock

Standards: None
16-Bit: MS-DOS

```
/* VLOAD.C: This program loads a block of virtual
 * memory with _vload, writes to it, and loads in
 * a new block. It then reloads the first block and
 * verifies that its contents haven't changed.
 */
#include <stdio.h>
#include <stdlib.h>
#include <vmemory.h>
void main( void )
{
   int i, flag;
   _vmhnd_t handle1,
            handle2;
   int __far *buffer1;
   int __far *buffer2;
   if ( !_vheapinit( 0, _VM_ALLDOS, _VM_XMS | _VM_EMS ) )
   {
      printf( "Could not initialize virtual memory manager. \n" );
      exit( -1 );
   }
   if ( ( (handle1 = _vmalloc( 100 * sizeof(int) )) == _VM_NULL ) ||
      ( (handle2 = _vmalloc( 100 * sizeof(int) )) == _VM_NULL )   )
   {
      _vheapterm();
      exit( -1 );
   }
   printf( "Two blocks of virtual memory allocated.\n" );
   if ( (buffer1 = (int __far *)_vload( handle1, _VM_DIRTY )) == NULL )
   {
      _vheapterm();
      exit( -1 );
   }
   printf( "buffer1 loaded: valid until next call to VM manager.\n" );
   for ( i = 0; i < 100; i++ )      /* write to buffer1 */
      buffer1[i] = i;
   if ( (buffer2 = (int __far *)_vload( handle2, _VM_DIRTY )) == NULL )
   {
      _vheapterm();
      exit( -1 );
   }
   printf( "buffer2 loaded. buffer 1 no longer valid.\n" );
   if ( (buffer1 = (int __far *)_vload( handle1, _VM_CLEAN )) == NULL )
   {
      _vheapterm();
      exit( -1 );
   }
   printf( "buffer1 reloaded.\n" );
   flag = 0;
   for ( i = 0; i < 100; i++ )
      if ( buffer1[i] != i )
         flag = 1;
   if ( !flag )
      printf( "Contents of buffer1 verified.\n" );
   _vfree( handle1 );
   _vfree( handle2 );
   _vheapterm();
   exit( 0 );
}
```

_vlock

#include <vmemory.h>

Syntax void __far *__far _vlock(_vmhnd_t *handle*);

Parameter	Description
handle	Handle to previously allocated virtual memory block

The _vlock function loads a virtual memory block into MS-DOS memory, locks it, and returns a far pointer to it. The argument *handle* points to a virtual memory block previously allocated through a call to _vmalloc or _vrealloc.

A locked virtual memory block will not be swapped out until it is unlocked. A virtual memory block can be locked up to 255 times. The pointer returned by _vlock remains valid until an equal number of unlock operations is performed.

Because MS-DOS memory may be scarce, try to keep the number of blocks locked at one time to a minimum and use _vunlock to unlock them as soon as possible.

Return Value

The _vlock function returns a far pointer to MS-DOS memory if the virtual memory block is successfully loaded and locked. If insufficient MS-DOS memory is available, _vload returns NULL.

_vlockcnt
_vmalloc
_vunlock

Standards: None
16-Bit: MS-DOS

```
/* VLOCK.C: This program locks a block of virtual
 * memory using _vlock, writes to it, loads in a
 * new block with _vload, and then verifies that
 * the contents of the locked block are still
 * accessible. It then unlocks the block with _vunlock.
 */
#include <stdio.h>
#include <stdlib.h>
#include <vmemory.h>
void main( void )
{
   int i, flag;
   _vmhnd_t handle1, handle2;
   int __far *buffer1;
   int __far *buffer2;
   if ( !_vheapinit( 0, _VM_ALLDOS, _VM_XMS | _VM_EMS ) )
   {
      printf( "Could not initialize virtual memory manager. \n" );
      exit( -1 );
   }
   if ( ( (handle1 = _vmalloc( 100 * sizeof(int) )) == _VM_NULL ) ||
      ( (handle2 = _vmalloc( 100 * sizeof(int) )) == _VM_NULL )   )
   {
      _vheapterm();
      exit( -1 );
   }
   printf( "Two blocks of virtual memory allocated.\n" );
   if ( (buffer1 = (int __far *)_vlock( handle1 )) == NULL )
   {
      _vheapterm();
      exit( -1 );
   }
   printf( "buffer1 locked: valid until unlocked.\n" );
   for ( i = 0; i < 100; i++ )      /* write to buffer1 */
      buffer1[i] = i;
   if ( (buffer2 = (int __far *)_vload( handle2, _VM_DIRTY )) == NULL )
   {
      _vheapterm();
      exit( -1 );
   }
   printf( "buffer2 loaded. buffer 1 still valid.\n" );
   flag = 0;
   for ( i = 0; i < 100; i++ )
      if ( buffer1[i] != i )
         flag = 1;
   if ( !flag )
      printf( "Contents of buffer1 verified.\n" );
   _vunlock( handle1, _VM_DIRTY );
   _vfree( handle1 );
   _vfree( handle2 );
   _vheapterm();
   exit( 0 );
}
```

_vlockcnt

#include <vmemory.h>

Syntax unsigned int __far _vlockcnt(_vmhnd_t *handle*)**;**

Parameter	Description
handle	Handle to previously allocated virtual memory block

The _vlockcnt function returns the number of times a virtual memory block has been locked**.** The argument *handle* points to a virtual memory block previously allocated through a call to _vmalloc or _vrealloc**.** Use the _vlockcnt function to ensure that a block is unlocked before it is freed (using _vfree)**.**

Return Value

The _vlockcnt function returns the number of locks held on the specified virtual memory block**.**

_vlock
_vmalloc
_vunlock

Standards: None
16-Bit: MS-DOS

```
/* VCNT.C: This program locks a block of virtual memory
 * five times with _vlock, and then unlocks it five
 * times with _vunlock, calling _vlockcnt after each
 * operation to report the number of locks held.
 */
#include <stdio.h>
#include <stdlib.h>
#include <vmemory.h>
void main( void )
{
   int i, count;
   _vmhnd_t handle;
   int __far *buffer;
   if ( !_vheapinit( 0, _VM_ALLDOS, _VM_XMS | _VM_EMS ) )
   {
      printf( "Could not initialize virtual memory manager. \n" );
      exit( -1 );
   }
   if ( (handle = _vmalloc( 100 * sizeof(int) )) == _VM_NULL )
   {
      _vheapterm();
      exit( -1 );
   }
   printf( "Block of virtual memory allocated.\n" );
   printf( "Locking...\n" );
   for ( i = 0; i < 5; i++ )
   {
      if ( (buffer = (int __far *)_vlock( handle )) == NULL )
      {
         _vheapterm();
         exit( -1 );
      }
      count = _vlockcnt( handle );
      printf( "%d locks held.\n", count );
   }
   printf("Unlocking...\n" );
   for ( i = 0; i < 5; i++ )
   {
      _vunlock( handle, _VM_CLEAN );
      count = _vlockcnt( handle );
      printf( "%d locks held.\n", count );
   }
   _vfree( handle );
   _vheapterm();
   exit( 0 );
}
```

_vmalloc

#include <vmemory.h>

Syntax _vmhnd_t __far _vmalloc(unsigned long *size*);

Parameter	**Description**
size	Bytes to allocate

The _vmalloc function allocates a virtual memory block of at least *size* bytes. The actual size of the allocated block may be larger than *size* bytes to allow the virtual memory manager to operate more efficiently; use _vmsize to find the actual size of the block.

The value returned by _vmalloc is a handle that uniquely identifies the virtual memory block. This value is not an address and cannot be used to access memory directly. The value must be passed to either the _vload or _vlock function to obtain a valid address.

Return Value

The _vmalloc function returns a handle to the allocated virtual memory block, or _VM_NULL if insufficient memory is available or if the requested block size is too large to load into MS-DOS memory.

_vfree
_vmsize
_vrealloc

Standards: None
16-Bit: MS-DOS

```
/* VMALLOC.C: This program allocates a block of virtual
 * memory with _vmalloc and uses _vmsize to display the
 * size of that block. Next, it uses _vrealloc to expand
 * the amount of virtual memory and calls _vmsize again
 * to display the new amount of memory allocated.
 */
#include <stdio.h>
#include <stdlib.h>
#include <vmemory.h>

void main( void )
{
    _vmhnd_t handle;
    unsigned long block_size;

    if ( !_vheapinit( 0, _VM_ALLDOS, _VM_XMS | _VM_EMS ) )
    {
        printf( "Could not initialize virtual memory manager.\n" );
        exit( -1 );
    }

    printf( "Requesting 100 bytes of virtual memory.\n" );
    if ( (handle = _vmalloc( 100 )) == _VM_NULL )
    {
        _vheapterm();
        exit( -1 );
    }

    block_size = _vmsize( handle );
    printf( "Received %lu bytes of virtual memory.\n", block_size );

    printf( "Resizing block to 200 bytes. \n" );
    if ( (handle = _vrealloc( handle, 200 )) == _VM_NULL )
    {
        _vheapterm();
        exit( -1 );
    }

    block_size = _vmsize( handle );
    printf( "Block resized to %lu bytes.\n", block_size );

    _vfree( handle );
    _vheapterm();
}
```

_vmsize

#include <vmemory.h>

Syntax unsigned long __far _vmsize(_vmhnd_t *handle*);

Parameter	Description
handle	Handle to previously allocated virtual memory block

The _vmsize function returns the size, in bytes, of a virtual memory block. The argument *handle* points to a virtual memory block previously allocated through a call to _vmalloc or _vrealloc. The size returned may be larger than the size requested in the call to _vmalloc or _vrealloc.

Return Value

The _vmsize function returns the size (in bytes) of the specified virtual memory block as an unsigned long.

__vmalloc

Standards: None

16-Bit: MS-DOS

_vrealloc

#include <vmemory.h>

Syntax _vmhnd_t __far _vrealloc(_vmhnd_t *handle*, unsigned long *size*);

Parameter	Description
handle	Handle to previously allocated virtual memory block
size	New size in bytes

The _vrealloc function changes the size of a virtual memory block. If *handle* is _VM_NULL, _vrealloc behaves in the same way as _vmalloc and allocates a new block of *size* bytes. If *handle* is not _VM_NULL, it must point to a virtual memory block previously allocated through a call to _vmalloc or _vrealloc.

The *size* argument gives the new size of the block, in bytes. The size of the block may be larger than *size* bytes to allow the virtual memory manager to operate more efficiently; use _vmsize to find the actual size of the block. The contents of the block are unchanged up to the shorter of the new and old sizes, although the new block may be in a different location.

Return Value

The _vrealloc function returns a handle to the reallocated (and possibly moved) virtual memory block.

The return value is _VM_NULL if the size specified is zero and the handle argument is not _VM_NULL. In this case, the original block is freed.

The return value is also _VM_NULL if there is not enough available memory to expand the block to the requested size, if the requested block size is too large to load into MS-DOS memory, or if the given handle is still locked. In these cases, the original block is still valid.

_vfree
_vmalloc
_vmsize

Standards: None
16-Bit: MS-DOS

```
/* VRSIZE.C: This program allocates a block of virtual
 * memory with _vmalloc and uses _vmsize to display the
 * size of that block. Next, it uses _vrealloc to expand
 * the amount of virtual memory and calls _vmsize again
 * to display the new amount of memory allocated.
 */
#include <stdio.h>
#include <stdlib.h>
#include <vmemory.h>
void main( void )
{
   _vmhnd_t handle;
   unsigned long block_size;
   if ( !_vheapinit( 0, _VM_ALLDOS, _VM_XMS | _VM_EMS ) )
   {
      printf( "Could not initialize virtual memory manager.\n" );
      exit( -1 );
   }
   printf( "Requesting 100 bytes of virtual memory.\n" );
   if ( (handle = _vmalloc( 100 )) == _VM_NULL )
   {
      _vheapterm();
      exit( -1 );
   }
   block_size = _vmsize( handle );
   printf( "Received %d bytes of virtual memory.\n", block_size );
   printf( "Resizing block to 200 bytes.\n" );
   if ( (handle = _vrealloc( handle, 200 )) == _VM_NULL )
   {
      _vheapterm();
      exit( -1 );
   }
   block_size = _vmsize( handle );
   printf( "Block resized to %d bytes.\n", block_size );
   _vfree( handle );
   _vheapterm();
   exit( 0 );
}
```

_vunlock

#include <vmemory.h>

Syntax void __far _vunlock(_vmhnd_t *handle*, int *dirty*);

Parameter	Description
handle	Handle to previously allocated virtual memory block
dirty	Flag indicating whether block should be written out or discarded when swapping occurs

The _vunlock function unlocks a virtual memory block. The argument *handle* points to a virtual memory block previously allocated through a call to _vmalloc or _vrealloc and locked through a call to _vlock.

If multiple locks are held on the virtual memory block, the block's lock count is decremented by one. If the block's lock count goes to zero, the block can be swapped out by the virtual memory manager. The pointer returned by _vlock when the block was first locked then becomes invalid.

The *dirty* flag indicates whether the block should be written out or discarded when swapping occurs. It can have one of the following values:

Value	Meaning
_VM_CLEAN	Discard contents of block when swapping occurs
_VM_DIRTY	Write contents of block to auxiliary memory when swapping occurs

Return Value

None.

_vlock
_vlockcnt
_vmalloc

Standards:	None
16-Bit:	MS-DOS

_wabout

#include <io.h>

Syntax int _wabout(char **string*);

Parameter	Description
string	Pointer to a null-terminated string

The _wabout function sets the string that appears in the About dialog box of a QuickWin program. This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see *Programming Techniques*.

When the user chooses the About command from the Help menu, a dialog box appears containing the string set with _wabout. If a QuickWin program does not include a call to _wabout, information about QuickWin itself is displayed by default.

The maximum string length is 256 bytes.

Return Value

If successful, _wabout returns 0. A nonzero return value indicates an error.

```
/* WABOUT.C - Demonstrate setting the
 * About dialog box string with _wabout
 */
#include <stdio.h>
#include <io.h>
char string[512];
void main( void )
{
   int nRes;
   for ( ; ; )
   {
      printf( "\nEnter the About string: " );
      gets(string);
      printf( "\nAbout string = %s\n", string );
      printf( "Setting about string..." );
      nRes = _wabout( string );
      printf( "_wabout result = %i", nRes );
      printf( "\nTry 'About' in the Help menu\n" );
   }
}
```

_wclose

#include <io.h>

Syntax int _wclose(int *wfh*, int *persist*);

Parameter	Description
wfh	File handle to a QuickWin window
persist	Flag indicating whether the window stays on the screen after closing

The _wclose function closes a QuickWin window. The window must have been previously opened with the QuickWin function _wopen. This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see *Programming Techniques*.

To close a window opened with _wopen, pass its file handle to _wclose. To close a window opened with _fwopen, call the STDIO.H function fclose.

The *persist* flag can have one of the following values:

Value	Meaning
_WINNOPERSIST	Erase the closed window
_WINPERSIST	Leave the window on the screen

If the window remains on the screen, another _wclose call to the same file handle with _WINNOPERSIST removes it. While the window remains visible, the user can copy and paste text in it, choose QuickWin menus, and operate the window's scroll bars.

Regardless of which *persist* option is used, the window's file handle is closed to all further I/O. If a window is opened with the same title as a window closed with persistence, it will be a different window. Windows closed with persistence count against the total number of open windows (20 by default).

Return Value

If successful, _wclose returns 0. A return value of -1 indicates an error; errno is set to EBADF, indicating an invalid file-handle argument.

```
/* WCLOSE.C - Demonstrate closing QuickWin windows */
#include <fcntl.h>
#include <stdio.h>
#include <io.h>
#include <stdlib.h>
#define PERSISTFLAG _WINNOPERSIST
#define OPENFLAGS   _O_RDWR
void main( void )
{
   int wfh;   /* File handle for window */
   int nRes;  /* Window write results */
   int wc;    /* Window closure results */
   struct _wopeninfo wininfo; /* Open information */
   /* Set up window open information */
   wininfo._version = _QWINVER;
   wininfo._title = "Window Closing";
   wininfo._wbufsize = _WINBUFDEF;
   /* Open a window with _wopen */
   wfh = _wopen( &wininfo, NULL, OPENFLAGS );
   if( wfh == -1 )
   {
      printf( "***ERROR: On _wopen\n" );
      exit( -1 );
   }
   /* Write in the window */
   nRes = write( wfh, "Windows Everywhere!\n", 20 );
   /* Close the window with _wclose */
   wc = _wclose( wfh, PERSISTFLAG );
   exit( 0 );
}
```

wcstombs, _fwcstombs

#include <stdlib.h>

Syntax size_t wcstombs(char *_mbstr_, const wchar_t *_wcstr_, size_t _count_);

size_t __far _fwcstombs(char __far *_mbstr_, const wchar_t __far *_wcstr_ , size_t _count_);

Parameter	Description
mbstr	The address of a sequence of multibyte characters
wcstr	The address of a sequence of wide characters
count	The number of bytes to convert

The wcstombs function converts *count* or fewer wide characters pointed to by *wcstr* to the corresponding multibyte characters and stores the results in the *mbstr* array.

If wcstombs encounters the wide-character null character (L'\0') either before or when *count* occurs, it converts it to the multibyte null character (a 16-bit 0) and stops. Thus, the multibyte character string at *mbstr* is null-terminated only if wcstombs encounters a wide-character null character during conversion. If the sequences pointed to by *wcstr* and *mbstr* overlap, the behavior of wcstombs is undefined.

The _fwcstombs function is a model-independent (large-model) form of the wcstombs function.

Return Value

If either wcstombs or _fwcstombs successfully converts the multibyte string, it returns the number of converted multibyte characters, excluding the wide-character null character. If either function encounters a wide character that cannot be converted to a multibyte character, it returns -1 cast to type size_t.

wcstombs

Standards: ANSI
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fwcstombs

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

mblen
mbstowcs
mbtowc
wctomb
MB_CUR_MAX
MB_LEN_MAX

```
/* WCSTOMBS.CPP illustrates the behavior of the wcstombs function */
#include <stdio.h>
#include <stdlib.h>
void main( void )
{
   int      i;
   char    *pmbbuf   = (char *)malloc( MB_CUR_MAX );
   wchar_t *pwcEOL   = L'\0';
   wchar_t *pwchello = L"Hello, world.";
   printf( "Convert entire wide-character string:\n" );
   i = wcstombs( pmbbuf, pwchello, MB_CUR_MAX );
   printf( "\tCharacters converted: %u\n", i );
   printf( "\tMultibyte character: %s\n\n", pmbbuf );
   printf( "Attempt to convert null character:\n" );
   i = wcstombs( pmbbuf, pwcEOL, MB_CUR_MAX );
   printf( "\tCharacters converted: %u\n", i );
   printf( "\tMultibyte character: %s\n\n", pmbbuf );
}
```

wctomb, _fwctomb

#include <stdlib.h>

Syntax int wctomb(char **mbchar*, wchar_t *wchar*);

int __far _fwctomb(char __far **mbchar*, wchar_t *wchar*);

Parameter	Description
mbchar	The address of a multibyte character
wchar	A wide character

The wctomb function converts its *wchar* argument to the corresponding multibyte character and stores the result at *mbchar*.

The _fwctomb function is a model-independent (large-model) form of the wctomb function. It can be called from any point in any program.

Return Value

If either wctomb or _fwctomb converts the wide character to a multibyte character, it returns the number of bytes (which is never greater than MB_CUR_MAX) in the wide character. If *wchar* is the wide-character null character (L'\0'), wctomb returns 0. If the conversion is not possible in the current locale, wctomb returns -1.

wctomb

Standards: ANSI
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

_fwctomb

Standards: None
16-Bit: MS-DOS, QWIN, WIN, WIN DLL

mblen
mbstowcs
mbtowc
wcstombs
MB_CUR_MAX
MB_LEN_MAX

```
/* WCTOMB.CPP illustrates the behavior of the wctomb function */
#include <stdio.h>
#include <stdlib.h>
void main( void )
{
   int i;
   wchar_t wc = L'a';
   char *pmbnull = NULL;
   char *pmb = (char *)malloc( sizeof( char ) );

   printf( "Convert a wide character:\n" );
   i = wctomb( pmb, wc );
   printf( "\tCharacters converted: %u\n", i );
   printf( "\tMultibyte character: %.1s\n\n", pmb );
   printf( "Attempt to convert when target is NULL:\n" );
   i = wctomb( pmbnull, wc );
   printf( "\tCharacters converted: %u\n", i );
   printf( "\tMultibyte character: %.1s\n", pmbnull );
}
```

_wgclose

#include <stdio.h>, <graph.h>

Syntax short __far __cdecl _wgclose(int *handle*);

Parameter	Description
handle	Handle of the window to be closed

The _wgclose function closes an existing graphics child window, freeing all memory associated with the window. Closing the active graphics child window causes all graphics output to fail until a new window is activated. See _wgsetactive for more information. This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see *Programming Techniques*.

Return Value

If successful, _wgclose returns 1; otherwise, it returns 0.

_wgopen

_wgetexit

#include <stdio.h>

Syntax int _wgetexit(void);

QuickWin programs can optionally keep their windows on the screen after termination. How a program will behave at exit time depends on its current exit behavior setting. The _wgetexit function lets you examine the current exit behavior setting. This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see *Programming Techniques*.

If the companion function _wsetexit has been called previously, _wgetexit returns the value that it set. This can be one of the following values:

_WINEXITPROMPT
  Prompt the user at exit time to determine whether the windows stay on the screen

_WINEXITNOPERSIST
  The windows do not stay on the screen and there is no prompt to the user

_WINEXITPERSIST
  The windows stay on the screen at exit

If _wsetexit has not been called previously, the _wgetexit function returns _WINEXITPERSIST, the default exit behavior. For a description of how to use this exit behavior, see _wsetexit.

Return Value

If successful, _wgetexit returns the current exit behavior setting value: _WINEXITPROMPT, _WINEXITNOPERSIST, or _WINEXITPERSIST. A return value of -1 indicates an error.

```
/* This program places the name of the current directory in
 * the buffer array, then displays the name of the current
 * directory on the screen. Specifying a length of _MAX_DIR
 * leaves room for the longest legal directory name.
 */
#include <direct.h>
#include <stdlib.h>
#include <stdio.h>
void main( void )
{
   char buffer[_MAX_DIR];
   /* Get the current working directory: */
   if( _getcwd( buffer, _MAX_DIR ) == NULL )
      perror( "_getcwd error" );
   else
      printf( "%s\n", buffer );
}
```

_wgetfocus

#include <io.h>

Syntax int _wgetfocus(void);

The _wgetfocus function determines which of a QuickWin program's child (document) windows is active (has the program's "focus"). The routine returns the file handle of the active child window. If the entire application is not active, the routine returns the handle of the child window that would be active if the application were active. This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see *Programming Techniques*.

If the active window is a closed child window kept on the screen with the _WINPERSIST flag (see _wclose), _wgetfocus fails.

Return Value

If successful, _wgetfocus returns the file handle of the active child window. A return value of -1 indicates an error.

```
/* WGETFOC.C - Demonstrate testing which QuickWin
 * window is the active window with _wgetfocus
 */
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMWINS      4    /* Number of windows */
#define OPENFLAGS   "w"   /* Access permission */
void main( void )
{
   int i, nRes;
   int sf, gf;          /* Set/Get focus results */
   FILE *wins[NUMWINS]; /* Array of file pointers */
   /* Open NUMWINS windows */
   /* NULL arguments accept default characteristics */
   for( i = 0; i < NUMWINS; i++ )
   {
      wins[i] = _fwopen( NULL, NULL, OPENFLAGS );
      if( wins[i] == NULL )
      {
         printf( "***ERROR: On _fwopen #%i\n", i );
         exit( -1 );
       }
       /* Write in each window */
       nRes = fprintf( wins[i], "Windows!\n" );
   }
   /* Tile child windows with _wmenuclick */
   nRes = _wmenuclick( _WINTILE );
   if( nRes == -1 )
   {
      printf( "***ERROR: _wmenuclick\n" );
      exit( -1 );
   }
   /* Pass the focus from window to window */
   for( i = 0; i < NUMWINS; i++ )
   {
      sf = _wsetfocus( _fileno( wins[i] ) );
      gf = _wgetfocus();
      if(( sf == -1 ) || ( gf == -1 )  || ( gf != _fileno( wins[i] ) ) )
      {
         printf( "***ERROR: _wsetfocus/_wgetfocus\n" );
         exit( -1 );
      }
   }
   nRes = _fcloseall();
   exit( 0 );
}
```

_wgetscreenbuf

#include <io.h>

Syntax long _wgetscreenbuf(int *wfh*);

Parameter	Description
wfh	File handle to a QuickWin window

The _wgetscreenbuf function returns the size of a QuickWin window screen buffer. This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see *Programming Techniques*.

Each QuickWin child window has a buffer in which the screen-display text for the window is stored. The buffer size determines how much text is retained and thus how much output can be viewed by scrolling back through the window.

By default, the screen-buffer size is 2,048 bytes, but this value can be changed. See _wsetscreenbuf.

Return Value

If successful, the _wgetscreenbuf function returns the current screen-buffer size (in bytes) or the value _WINBUFINF. (A value of _WINBUFINF signifies that the size of the screen buffer is unlimited.) A return value of -1 indicates an error.

```
/* WGSCRBUF.C - Demonstrate examining the current
 * size of a QuickWin window's screen buffer
 */
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMWINS      4    /* Number of windows */
#define OPENFLAGS   "w"   /* Access permission */
void main( void )
{
   long nSize;           /* Size of screen buffer */
   int nRes;             /* Write result */
   FILE *wp;             /* File pointer */
   /* Open a window */
   /* NULL arguments accept default characteristics */
   wp = _fwopen( NULL, NULL, OPENFLAGS );
   if( wp == NULL )
   {
      printf( "***ERROR:_fwopen\n" );
      exit( -1 );
   }
   /* Get the size of its screen buffer */
   nSize = _wgetscreenbuf( _fileno( wp ) );
   nRes = fprintf( wp, "Screen buffer holds %li chars\n", nSize );
   nRes = _wclose( _fileno( wp ), _WINPERSIST );
   exit( 0 );
}
```

_wgetsize

#include <io.h>

Syntax int _wgetsize(int *wfh*, int *reqtype*, struct _wsizeinfo **wsize*);

Parameter	Description
wfh	File handle to a QuickWin window
reqtype	Type of request
wsize	Pointer to a _wsizeinfo structure

The _wgetsize function returns the size and position of the specified child window. This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see *Programming Techniques*.

The *wfh* argument is a handle to the window file. Use the manifest constant _WINFRAMEHAND as the value of *wfh* to query the size and position of the parent frame (client or application window). The maximum size of the parent frame may vary according to the hardware specifications of your terminal.

The *reqtype* argument is the type of request, which can have one of two values:

Value	Meaning
_WINCURRREQ	Return the current size of the window
_WINMAXREQ	Return the maximum size that the window can grow to (which cannot exceed the current size of the parent frame)

The *wsize* argument is a pointer to a _wsizeinfo structure (declared in IO.H) that returns the size and position information. The structure contains a _type field that has one of the following values on return:

Value	Meaning
_WINSIZEMIN	Window is minimized
_WINSIZEMAX	Window is maximized
_WINSIZECHAR	Window is of the size specified in the structure's remaining members

If the type returned is _WINSIZECHAR, the _x, _y, _h, and _w values in the remainder of the structure specify the coordinates of the upper-left corner and the height and width of the window (in characters). Size returned always indicates the "client space" available in the parent frame, which means that it does not include space occupied by title bars and other parts of the window.

Return Value

If successful, _wgetsize returns 0 and fills in the _wsizeinfo structure. A return value of -1 indicates an error.

```
/* WGETSIZE.C - Demonstrate getting the
 * size of a QuickWin window on the screen
 */
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#define OPENFLAGS    "w"         /* Access permission */
#define PERSISTFLAG _WINPERSIST  /* Keep on screen */
void main( void )
{
   int nRes;               /* Result */
   FILE *wp;               /* File pointer */
   struct _wsizeinfo ws;   /* Size information */
   /* Open a window */
   /* NULL arguments accept default characteristics */
   wp = _fwopen( NULL, NULL, OPENFLAGS );
   if( wp == NULL )
   {
      printf( "***ERROR:_fwopen\n" );
      exit( -1 );
   }
   /* Get the window's size and screen position */
   ws._version = _QWINVER;
   nRes = _wgetsize( _fileno( wp ), _WINCURRREQ, &ws );
   if( nRes == -1 )
   {
      printf( "***ERROR: _wgetsize\n" );
      exit( -1 );
   }
   nRes = fprintf( wp, "Size:\n" );
   nRes = fprintf( wp, "  Upper Left: x = %d\n", ws._x );
   nRes = fprintf( wp, "              y = %d\n", ws._y );
   nRes = fprintf( wp, "  Width:      w = %d\n", ws._w );
   nRes = fprintf( wp, "  Height:     h = %d\n", ws._h );
   nRes = _wclose( _fileno( wp ), PERSISTFLAG );
   exit( 0 );
}
```

_wggetactive_wgetsize

#include <stdio.h>, <graph.h>

Syntax short __far __cdecl _wggetactive(void);

The _wggetactive function returns the handle of the currently active graphics child window. This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see *Programming Techniques* .

Return Value

This function returns the handle of the currently active graphics child window; otherwise, -1 (if no graphics child window is active).

_wgsetactive

```
/* QWGDEMO.C - opens multiple Graphics Child Windows using:
 *    _wgopen   _wgclose   _wggetactive   _wgsetactive
 * and demonstrates pausing in a graphics child window using:
 *    _inchar
 * This program creates three windows, pausing after each to
 * allow you to examine the window before continuing. Try
 * using the scroll bars and arrow keys, switching the GCWs
 * to Full Screen or Size to Fit mode, and resizing the windows.
 */
#include <stdio.h>
#include <graph.h>
void main()
{
   int status, handle;
   short color;
   long bgd, prvbgd;
   char name[] = "SMILEY";
   char name2[] = "HELLO";

   /* Send a message to the text window (stdin/stderr/stdout) */
   printf( "QuickWin Graphics Demo Program.\n" );

   /* Pause before continuing */
   printf( "\nThis is a text window. Press Enter to continue.\n" );
   status = getchar( );

   /* Close the default "Graphic1" GCW (we'll name our own) */
   status = _setvideomode( _MAXRESMODE );
   handle = _wggetactive( );
   status = _wgclose( handle );

   /* Open a GCW named "SMILEY" in which to draw a smiley face */
   handle = _wgopen( name );
   status = _wgsetactive( handle );
   status = _setvideomode( _MAXRESMODE );

   /* Draw Smiley's head */
   color = 13;
   status = _setcolor( color );
   status = _ellipse( _GFILLINTERIOR, 40, 20, 240, 140 );

   /* Draw Smiley's eyes, nose, and mouth */
   status = _setcolor( (short)(color + 1) );
   status = _ellipse( _GFILLINTERIOR, 80, 40, 100, 60 );
   status = _ellipse( _GFILLINTERIOR, 180, 40, 200, 60 );
   status = _ellipse( _GBORDER, 130, 70, 150, 90 );
   status = _arc( 90, 80, 190, 120, 115, 100, 200, 100 );

   /* Pause before continuing */
   _settextposition( 1, 6 );
   _outtext( "Press any key to continue." );
   status = _inchar( );

   /* Open a new GCW called "HELLO" and set it for text output */
   handle = _wgopen( name2 );
   status = _wgsetactive( handle );
   status = _setvideomode( _TEXTC80 );

   /* Set background and text colors for the message */
   bgd = 1;
```

```
   prvbgd = _setbkcolor( bgd );
   status = _settextcolor( color );

   /* Display a text message in a GCW */
   _settextposition( 6, 25 );
   _outtext( "HELLO, WORLD!" );

   /* Return to default text color & background color */
   bgd = _setbkcolor( prvbgd );
   status = _settextcolor( 15 );

   /* Pause before continuing */
   _settextposition( 1, 6 );
   _outtext("Press any key to continue.");
   status = _inchar( );

   /* Leave the cursor at the start of the line below the message */
   _settextposition( 7, 1 );
}
```

_wgopen

#include <stdio.h>, <graph.h>

Syntax short __far __cdecl _wgopen(char **name*);

Parameter	Description
name	Name of window to be opened

The _wgopen functions opens a graphics child window. The argument *name*, which specifies the name of the opened window, is a null-terminated C string. QuickWin Graphics generates a default name if the argument *name* is NULL. The window is initially set to _TEXTC80 mode. You can have up to 20 graphics child windows opened. Graphics child windows can be in either text mode or graphics mode. You need to make an opened graphics child window active (see _wgsetactive) before it can accept any graphics call. All standard input and output (READ and WRITE statements) go to the text child windows, not the GCW in text mode.This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see *Programming Techniques*.

Open windows occupy a large amount of memory. Keep in mind that the system's performance decreases with the addition of every new window.

Return Value

If successful, this functions returns the handle of the opened graphics child window; otherwise it returns 0.

_wgclose

_wgsetactive

#include <stdio.h>, <graph.h>

Syntax short __far __cdecl __wgsetactive(int *handle*)

Parameter	Description
handle	Handle of the window to be made active

The _wgsetactive functions makes a graphics child window active. All graphics calls are directed to the active graphics child window. Setting the handle to -1 causes all graphics child windows to be inactive and any call to the graphics APIs to fail.

Return Value

If successful, this function returns 1; otherwise it returns 0.

_wggetactive

_wmenuclick

#include <io.h>

Syntax int _wmenuclick(int *menuitem*);

Parameter	Description
menuitem	Constant specifying which menu command to execute

The _wmenuclick function emulates the user choosing a command from the QuickWin Window menu. This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see *Programming Techniques*.

The *menuitem* argument is a manifest constant specifying one of four available menu commands:

Value	Meaning
_WINTILE	Tile the program's child windows
_WINCASCADE	Cascade the program's child windows
_WINARRANGE	Arrange icons at the bottom of the client window area
_WINSTATBAR	Toggle the status bar

These are the only menu commands you can choose. Calling the function with one of these values performs the menu action.

Return Value

If successful, _wmenuclick returns 0. A return value of -1 indicates an error.

```
/* WMENUCLK.C - Demonstrate choosing a menu
 * command with the QuickWin _wmenuclick function
 */
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMWINS      4    /* Number of windows */
#define OPENFLAGS   "w"   /* Access permission */
void main( void )
{
   int i, nRes;
   int wm;              /* Menu click result */
   int sf, gf;          /* Set/Get focus results */
   FILE *wins[NUMWINS]; /* Array of file pointers */
   /* Open NUMWINS windows */
   /* NULL arguments accept default characteristics */
   for( i = 0; i < NUMWINS; i++ )
   {
      wins[i] = _fwopen( NULL, NULL, OPENFLAGS );
      if( wins[i] == NULL )
      {
         printf( "***ERROR: On _fwopen #%i\n", i );
         exit( -1 );
      }
      /* Write in each window */
      nRes = fprintf( wins[i], "Windows!" );
   }
   /* Tile child windows with _wmenuclick */
   wm = _wmenuclick( _WINTILE );
   if( wm == -1 )
   {
      printf( "***ERROR: _wmenuclick\n" );
      exit( -1 );
   }
   /* Pass the focus from window to window */
   for( i = 0; i < NUMWINS; i++ )
   {
      sf = _wsetfocus( _fileno( wins[i] ) );
      gf = _wgetfocus();
      if(( sf == -1 ) || ( gf == -1 )  || ( gf != _fileno( wins[i] ) ) )
      {
          printf( "***ERROR: _wsetfocus/_wgetfocus\n" );
          exit( -1 );
      }
   }
   nRes = _fcloseall();
   exit( 0 );
}
```

_wopen

#include <io.h>

Syntax int _wopen(struct _wopeninfo *_wopeninfo_, struct _wsizeinfo *_wsizeinfo_, int _oflag_);

Parameter	Description
wopeninfo	Pointer to a _wopeninfo structure
wsizeinfo	Pointer to a _wsizeinfo structure
oflag	Type of operations allowed

The _wopen function opens a QuickWin window, returning a file handle to the window. This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see *Programming Techniques*.

The _wopeninfo and _wsizeinfo structures, declared in IO.H, are used to pass window initialization information, including the window's initial size and position on the screen. You can pass NULL for the _wsizeinfo argument to accept QuickWin size and positioning defaults, or you can declare a variable of type _wsizeinfo and fill in its fields with initial values. You must declare a variable of type _wopeninfo and fill in its fields.

For both the _wopeninfo and _wsizeinfo variables, set the _version field to _QWINVER, which is defined in IO.H.

For the _wopeninfo variable, assign a null-terminated string to the _title field containing the desired window title. You can also optionally set the size of the window's screen buffer in the _wbufsize field. The default is 2,048 bytes, but you can pass some other number or the value _WINBUFINF. The value _WINBUFINF imposes no limit on the buffer size.

For the _wsizeinfo variable, if you choose to pass size information, assign one of the following values to the _type field:

Value	Meaning
_WINSIZEMIN	Minimize the window
_WINSIZEMAX	Maximize the window
_WINSIZECHAR	Use character coordinates for the window size

If the type is _WINSIZECHAR, you must supply the _x, _y, _h, and _w values in the remainder of the structure. They specify the upper-left corner and the height and width of the window (in characters).

The _wopen function is a low-level I/O call. It accepts the following access flags: _O_BINARY, _O_RDONLY, _O_RDWR, _O_TEXT, _O_WRONLY.

These flags can be combined with the bitwise-OR operator (|). See _open for additional information about the flags.

Unlike the _open function, _wopen does not accept the _O_CREAT, _O_TRUNC, or _O_EXCL flag. Using one of these flags results in an error.

Return Value

If successful, _wopen returns a QuickWin file handle. A return value of -1 indicates an error; errno is set to one of the following values:

Value	Meaning

EINVAL	An invalid *oflag* argument was given
EMFILE	No more file handles available (too many open files)

```
/* WOPEN.C - Demonstrate opening
 * a QuickWin window with _wopen
 */
#include <fcntl.h>
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#define PERSISTFLAG   _WINNOPERSIST
#define OPENFLAGS     _O_RDWR
void main( void )
{
   int wfh;                  /* File handle for window */
   int nRes;                 /* Window write results */
   struct _wopeninfo wininfo;/* Open information */
   /* Set up window open information */
   wininfo._version = _QWINVER;
   wininfo._title = "Window Closing";
   wininfo._wbufsize = _WINBUFDEF;
   /* Open a window with _wopen */
   /* NULL second argument accepts default size */
   wfh = _wopen( &wininfo, NULL, OPENFLAGS );
   if( wfh == -1 )
   {
      printf( "***ERROR: On _wopen\n" );
      exit( -1 );
   }
   /* Write in the window */
   nRes = write ( wfh, "Windows Everywhere!\n", 20 );
   /* Close the window with _wclose */
   nRes = _wclose( wfh, PERSISTFLAG );
   exit( 0 );
}
```

_wrapon

#include <graph.h>

Syntax short __far _wrapon(short *option*);

Parameter	**Description**
option	Wrap condition

The _wrapon function controls whether text output with both the _outmem and the _outtext functions wraps to a new line or is simply clipped when the text output reaches the edge of the defined text window. The *option* argument can be one of the following manifest constants:

Constant	**Meaning**
_GWRAPOFF	Truncates lines at window border
_GWRAPON	Wraps lines at window border

Note that this function does not affect the output of presentation-graphics routines or font routines.

The _wrapon function works differently in QuickWin applications. For specific information, see the *Programming Techniques* manual.

Return Value

The function returns the previous value of *option*. There is no error return.

Standards: None
16-Bit: MS-DOS, WIN, WIN DLL

_outtext
_outmem
_scrolltextwindow
_settextwindow

```
/* WRAPON.C */
#include <conio.h>
#include <graph.h>
void main( void )
{
   _wrapon( _GWRAPON );
   while( !_kbhit() )
      _outtext( "Wrap on!   " );
   _getch();
   _outtext( "\n\n" );
   _wrapon( _GWRAPOFF );
   while( !_kbhit() )
      _outtext( "Wrap off!  " );
   _getch();
   _outtext( "\n\n" );
}
```

_write

#include <io.h> Required only for function declarations

Syntax int _write(int *handle*, void **buffer*, unsigned int *count*);

Parameter	Description
handle	Handle of file into which data is written
buffer	Data to be written
count	Number of bytes

The _write function writes *count* bytes from *buffer* into the file associated with *handle*. The write operation begins at the current position of the file pointer (if any) associated with the given file. If the file is open for appending, the operation begins at the current end of the file. After the write operation, the file pointer is increased by the number of bytes actually written.

Return Value

The _write function returns the number of bytes actually written if successful. If the actual space remaining on the disk is less than the size of the buffer the function is trying to write to the disk, the function fails and does not flush any of the buffer's contents to the disk. A return value of -1 indicates an error. In this case, errno is set to one of the following values:

Value	Meaning
EBADF	Invalid file handle or file not opened for writing
ENOSPC	No space left on device

For 16-bit platforms, if you are writing more than 32K (the maximum size for type int) to a file, the return value should be of type unsigned int. (See the example.) However, the maximum number of bytes that can be written to a file at one time is 65,534, since 65,535 (or OxFFFF) is indistinguishable from -1 and would return an error.

If the file is opened in text mode, each line-feed character is replaced with a carriage-return-line-feed pair in the output. The replacement does not affect the return value.

When writing to files opened in text mode, the _write function treats a CTRL+Z character as the logical end-of-file. When writing to a device, _write treats a CTRL+Z character in the buffer as an output terminator.

Use _write for compatibility with ANSI naming conventions of non-ANSI functions. Use write and link with OLDNAMES.LIB for UNIX compatibility.

fwrite
_open
_read

Standards: UNIX

16-Bit: MS-DOS, QWIN, WIN, WIN DLL

```
/* WRITE.C: This program opens a file for output
 * and uses _write to write some bytes to the file.
 */
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
char buffer[] = "This is a test of 'write' function";
void main( void )
{
   int fh;
   unsigned byteswritten;
   if( (fh = _open( "write.o", _O_RDWR | _O_CREAT, _S_IREAD | _S_IWRITE )) != -1 )
   {
      if(( byteswritten = _write( fh, buffer, sizeof( buffer ))) == -1 )
         perror( "Write failed" );
      else
         printf( "Wrote %u bytes to file\n", byteswritten );
      _close( fh );
   }
}
```

_wsetexit

#include <io.h>

Syntax int _wsetexit(int *exb*);

Parameter	**Description**
exb	Desired exit behavior type

QuickWin programs can optionally keep their windows on the screen after termination. How a program behaves at exit time depends on its current exit behavior setting. The _wsetexit function sets the exit behavior setting. This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see *Programming Techniques*.

The _wsetexit function takes one of three arguments:

_WINEXITPROMPT
  Prompt the user at exit time to determine whether the windows stay on the screen

_WINEXITNOPERSIST
  The windows do not stay on the screen and there is no prompt to the user

_WINEXITPERSIST
  The windows stay on the screen at exit

If _WINEXITPERSIST is passed, or if _WINEXITPROMPT is passed and the user chooses to keep the windows on the screen, the windows stay visible, their contents can be copied and pasted, and their scroll bars can be used, but the windows are closed to further I/O. See _wclose for more information. The default exit behavior is _WINEXITPERSIST if you do not call _wsetexit.

Return Value

If successful, _wsetexit returns 0. A return value of -1 indicates an error.

```
/* FWOPEN.C - Demonstrate opening QuickWin windows
 * with _fwopen
 */

#include <io.h>
#include <stdio.h>
#include <stdlib.h>

#define OPENFLAGS "w"              /* Access permission */

void main()
{
    struct _wopeninfo wininfo;     /* Open information */
    char wintitle[32]="QuickWin "; /* Title for window */
    FILE *wp;                      /* FILE ptr to window */
    int nRes;                      /* I/O result */

    /* Set up window info structure for _fwopen */
    wininfo._version = _QWINVER;
    wininfo._title = wintitle;
    wininfo._wbufsize = _WINBUFDEF;

    /* Check current 'exit behavior' setting */
    /* Test should be true, since default is _WINEXITPERSIST */
    /* So set new behavior to prompt user */
    if( _wgetexit() == _WINEXITPERSIST )
       _wsetexit( _WINEXITPROMPT );

    /* Create a new window */
    /* NULL second argument accepts default size/position */
    wp = _fwopen( &wininfo, NULL, OPENFLAGS );
    if( wp == NULL )
    {
        printf( "***ERROR: _fwopen\n" );
        exit( -1 );
    }

    /* Write in the window */
    nRes = fprintf( wp, "Hello, QuickWin!\n" );

    /* Close the window */
    nRes = fclose( wp );

    /* On exiting anywhere, user is prompted
     * to keep window on screen or not
     */
    exit( 0 );

}
```

_wsetfocus

#include <io.h>

Syntax int _wsetfocus(int *wfh*);

Parameter	Description
wfh	File handle to a QuickWin window

The _wsetfocus function makes a QuickWin window the active window (sets the program's focus to the window). This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see *Programming Techniques*.

If the application has focus, the window gets focus. If not, the window will get the focus when the application gets focus.

If the program has other child windows, the focused window moves in front of them and is highlighted. This does not automatically direct I/O to the window. All I/O calls specify which window they are directed to by passing a stream pointer or file handle as an argument.

Return Value

If successful, _wsetfocus returns 0. A return value of -1 indicates that the focus failed to change.

```
/* WSETFOC.C - Demonstrate making a new QuickWin
 * window the active window with _wsetfocus
 */
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMWINS      4     /* Number of windows */
#define OPENFLAGS   "w"    /* Access permission */
void main( void )
{
   int i, nRes, wm;
   int sf, gf;          /* Set/Get focus results */
   FILE *wins[NUMWINS]; /* Array of file pointers */
   /* Open NUMWINS windows */
   /* NULL arguments accept default characteristics */
   for( i = 0; i < NUMWINS; i++ )
   {
      wins[i] = _fwopen( NULL, NULL, OPENFLAGS );
      if( wins[i] == NULL )
      {
         printf( "***ERROR: On _fwopen #%i\n", i );
         exit( -1 );
      }
       /* Write in each window */
       nRes = fprintf( wins[i], "Windows!\n" );
   }
   /* Tile child windows with _wmenuclick */
   wm = _wmenuclick( _WINTILE );
   if( wm == -1 )
   {
      printf( "***ERROR: _wmenuclick\n" );
      exit( -1 );
   }
   /* Pass the focus from window to window */
   for( i = 0; i < NUMWINS; i++ )
   {
      sf = _wsetfocus( _fileno( wins[i] ) );
      gf = _wgetfocus();
      if(( sf == -1 ) || ( gf == -1 )  || ( gf != _fileno( wins[i] ) ) )
      {
         printf( "***ERROR: _wsetfocus/_wgetfocus\n" );
         exit( -1 );
      }
   }
   nRes = _fcloseall();
   exit( 0 );
}
```

_wsetscreenbuf

#include <io.h>

Syntax int _wsetscreenbuf(int *wfh*, long *bufsiz*);

Parameter	Description
wfh	File handle to a QuickWin window
bufsiz	Desired size of the window's screen buffer (in bytes)

The _wsetscreenbuf function sets the size of a QuickWin window's screen buffer to *bufsiz* bytes. This size determines how much text is retained in the buffer and thus how much text you can scroll back through. This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see *Programming Techniques*.

The *bufsiz* argument can be specified as a number or as one of the following values:

Value	Meaning
_WINBUFDEF	Use the default window screen-buffer size (2,048 bytes)
_WINBUFINF	Use a window screen buffer of unlimited size

The buffer size simply limits how big the buffer can become. The buffer is always allocated dynamically, so that it fits its contents. Specifying _WINBUFINF puts no upper limit on buffer size. The buffer may grow within the limits of available memory.

Return Value

If successful, _wsetscreenbuf returns 0. A return value of -1 indicates an error.

```
/* WSSCRBUF.C - Demonstrate setting the size of a
 * QuickWin window's screen buffer
 * Note: The size is set here to an amount smaller than
 * the default size, but you can set it larger as well
 */
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMWINS      4        /* Number of windows */
#define OPENFLAGS   "w"       /* Access permission */
#define NUMLINES    100       /* Lines of text to write */
void main( void )
{
   int i;                    /* Loop variable */
   long nSize;                /* Old size of screen buffer */
   int nWinBufSize = 1500L; /* New size */
   int nRes;                 /* Result */
   FILE *wp;                 /* File pointer */
   /* Open a window */
   /* NULL arguments accept default characteristics */
   wp = _fwopen( NULL, NULL, OPENFLAGS );
   if( wp == NULL )
   {
      printf( "***ERROR:_fwopen\n" );
      exit( -1 );
   }
   /* Get the size of its screen buffer */
   nSize = _wgetscreenbuf( _fileno( wp ) );
   nRes = fprintf( wp, "Screen buffer holds %li chars\n", nSize );
   /* Reset the screen buffer size */
   nRes = _wsetscreenbuf( _fileno( wp ), nWinBufSize );
   /* Write many lines in the window */
   for( i = 0; i < NUMLINES; i++ )
   {
      nRes = fprintf( wp, "%i Lines (Windows!)\n", i );
   }
   nRes = fprintf( wp, "\nWhen the program ends, try using the scroll bars\n" );
   nRes = _wclose( _fileno( wp ), _WINPERSIST );
   exit( 0 );
}
```

_wsetsize

#include <io.h>

Syntax int _wsetsize(int wfh, struct _wsizeinfo **wsize*);

Parameter	Description
wfh	File handle to a QuickWin window
wsize	Pointer to a _wsizeinfo structure

The _wsetsize function sets the size and position of a QuickWin window. This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see *Programming Techniques*.

The *wsize* argument points to a _wsizeinfo structure (declared in IO.H) containing the new size and position information. The structure contains a _type field that can have one of the following values:

Value	Meaning
_WINSIZEMIN	Minimize the window
_WINSIZEMAX	Maximize the window
_WINSIZRESTORE	Restore a previously minimized window
_WINSIZECHAR	Use character coordinates for the window size

If the type is _WINSIZECHAR, you must supply the _x, _y, _h, and _w values in the remainder of the structure. They specify the upper-left corner and the height and width of the window (in characters).

Return Value

If successful, _wsetsize returns 0. A return value of -1 indicates an error.

```
/* WSETSIZE.C - Demonstrate setting the
 * size of a QuickWin window on the screen
 */
#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#define OPENFLAGS    "w"         /* Access permission */
#define PERSISTFLAG _WINPERSIST  /* Keep on screen */
void main( void )
{
   int nRes;               /* Result */
   FILE *wp;               /* File pointer */
   struct _wsizeinfo ws;   /* Size information */
   /* Open a window */
   /* NULL arguments accept default characteristics */
   wp = _fwopen( NULL, NULL, OPENFLAGS );
   if( wp == NULL )
   {
      printf( "***ERROR:_fwopen\n" );
      exit( -1 );
   }
   /* Minimize the window to an icon */
   ws._version = _QWINVER;
   ws._type = _WINSIZEMIN;
   nRes = _wsetsize( _fileno( wp ), &ws );
   if( nRes == -1 )
   {
      printf( "***ERROR: _wsetsize\n" );
      exit( -1 );
   }
   nRes = _wclose( _fileno( wp ), PERSISTFLAG );
   exit( 0 );
}
```

_wyield

#include <io.h>

Syntax void _wyield(void);

The _wyield function yields control to Windows in order to give processor time to other Windows applications. This routine is used only in QuickWin programs; it is not part of the Windows API. For full details about QuickWin, see *Programming Techniques*.

A Windows application must service its message queue periodically to ensure smooth appearance and performance. Well-behaved QuickWin applications yield time to other applications and allow the user to switch tasks without having to wait for the QuickWin program to complete lengthy processing.

The compiler attempts to issue "yield for queue servicing" calls at appropriate times. But in some cases a program requires additional yield calls, particularly during lengthy processing loops. If Windows appears sluggish when running a QuickWin program, insert _wyield calls into the program to improve Windows' responsiveness. Note that when an application is servicing the message queue (yielding) it can be told to stop so the user can work with another running Windows application.

Return Value

None.

```
/* WYIELD.C - Demonstrate yielding processor time from a
 * QuickWin program so that other Windows programs can
 * process their message queues; uses _wyield
 */
#include <io.h>
void compute(void);        /* Function prototype */
void main( void )
{
   int l;
   for( l = 0; l <= 10000; l++ )
   {
      compute();       /* Time-consuming function you supply */
      if( l % 1000 )
         _wyield();        /* Yield once every 1000 loops */
   }
}
void compute(void)
{
   /* Intensive computations */
}
```

Constant "a", "a+", "r", "r+", "w", "w+"

Include <stdio.h>

Context _fdopen, fopen, freopen, _fsopen, _fwopen

These constants specify the access type ("a", "r", or "w") requested for the file. Both the translation mode ("b" or "t") and the commit-to-disk mode ("c" or "n") can be specified with the type of access.

The access types are described below.

"a"
: Opens for writing at the end of the file (appending); creates the file first if it does not exist. All write operations occur at the end of the file. Although the file pointer can be repositioned using fseek or rewind, it is always moved back to the end of the file before any write operation is carried out.

"a+"
: Same as above, but also allows reading.

"r"
: Opens for reading. If the file does not exist or cannot be found, the call to open the file will fail.

"r+"
: Opens for both reading and writing. If the file does not exist, or cannot be found, the call to open the file will fail.

"w"
: Opens an empty file for writing. If the given file exists, its contents are destroyed.

"w+"
: Opens an empty file for both reading and writing. If the given file exists, its contents are destroyed.

When the "r+", "w+", or "a+" type is specified, both reading and writing are allowed (the file is said to be open for update). However, when switching between reading and writing, you must reposition the file pointer, using fsetpos, fseek, or rewind. You can specify the current position, if desired.

Constant "b", "t"

Include <stdio.h>

Context _fdopen, fopen, freopen, _fwopen

These constants specify the mode of translation ("b" or "t"). The mode is included in the string specifying the type of access ("r", "w", "a", "r+", "w+", "a+").

The translation modes are described below:

Mode	Meaning
"t"	Opens in text (translated) mode. In this mode, carriage-return/linefeed (CR-LF) combinations are translated into single linefeeds (LF) on input, and LF characters are translated into CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input.

In files opened for reading or reading/writing, fopen checks for CTRL+Z at the end of the file and removes it, if possible. This is done because using the fseek and ftell functions to move within a file ending with CTRL+Z may cause fseek to behave improperly near the end of the file.

Note The "t" option is not part of the ANSI standard for fopen and freopen, but is a Microsoft extension and should not be used where ANSI portability is desired.

Mode	Meaning
"b"	Opens in binary (untranslated) mode. The above translations are suppressed.

If "t" or "b" is not given in type, the translation mode is defined by the default-mode variable.

See Controlling File Mode Using BINMODE.OBJ for more information.

Constant "c", "n"

Include <stdio.h>

Context _fdopen, fopen

These constants specify whether the buffer associated with the open file is flushed to the operating system's buffers or to disk. The mode is included in the string specifying the type of access ("r", "w", "a", "r+", "w+", "a+").

The modes are described below.

"c"
Write the unwritten contents of the specified buffer to disk. This commit-to-disk functionality only occurs at explicit calls to either the fflush or the _flushall function. This mode is useful when dealing with sensitive data. For example, if your program dies after a call to fflush or _flushall, you can be sure that your data reached the operating system's buffers. However, unless a file is opened with the "c" option, the data might never make it to disk if the operating system also dies.

"n"
Write the unwritten contents of the specified buffer to the operating system's buffers. The operating system can cache data and then determine an optimal time to write to disk. Under many conditions, this behavior makes for efficient program behavior. However, if the retention of data is critical (such as bank transactions or airline ticket information) consider using the "c" option. The "n" mode is the default.

Note The "c" and "n" options are not part of the ANSI standard for fopen, but are Microsoft extensions and should not be used where ANSI portability is desired.

See COMMODE.OBJ for more information.

File-Attribute Constants

Constant _A_ARCH, _A_HIDDEN, _A_NORMAL, _A_RDONLY, _A_SUBDIR, _A_SYSTEM, _A_VOLID

Include <dos.h>

Context _dos_findfirst, _dos_findnext, _dos_getfileattr, _dos_setfileattr

These constants specify the current attributes of the file or directory specified by the function.

The attributes are represented by the following manifest constants:

_A_ARCH
Archive. Set whenever the file is changed. Cleared by the DOS BACKUP command.

_A_HIDDEN
Hidden file. Cannot be found with the DOS DIR command. Returns information about normal files as well as files with this attribute.

_A_NORMAL
Normal. File can be read or written to without restriction.

_A_RDONLY
Read-only. File cannot be opened for writing, and a file with the same name cannot be created. Returns information about normal files as well as files with this attribute.

_A_SUBDIR
Subdirectory. Returns information about normal files as well as files with this attribute.

_A_SYSTEM
System file. Cannot be found with the DOS DIR command. Returns information about normal files as well as files with this attribute.

_A_VOLID
Volume ID. Only one file can have this attribute, and it must be in the root directory.

Multiple constants can be combined with the OR operator (|).

Constant BUFSIZ

Include <stdio.h>

Context Stream I/O functions.

Buffers used in stream I/O are of size BUFSIZ by default. This value is generally used to establish the size of system-allocated buffers. It is also the required size that you must pass to the setbuf routine when allocating a buffer.

_bios_serialcom Initialization Constants

Constant _COM_CHR7, _COM_CHR8, _COM_STOP1, _COM_STOP2, _COM_NOPARITY, _COM_EVENPARITY, _COM_ODDPARITY, _COM_110, _COM_150, _COM_300, _COM_600, _COM_1200, _COM_2400, _COM_4800, _COM_9600

Include <bios.h>

Context _bios_serialcom

The data argument for the _COM_INIT service is created by combining one or more constants with the OR operator.

The constants and their meanings are listed below:

Constant	Meaning
_COM_CHR7	7 data bits
_COM_CHR8	8 data bits
_COM_300	300 baud
_COM_600	600 baud
_COM_1200	1200 baud
_COM_NOPARITY	No parity
_COM_EVENPARITY	Even parity
_COM_ODDPARITY	Odd parity
_COM_110	110 baud
_COM_150	150 baud
_COM_STOP1	1 stop bit
_COM_STOP2	2 stop bits
_COM_2400	2400 baud
_COM_4800	4800 baud
_COM_9600	9600 baud

The default values of *data* are 1 stop bit, no parity, and 110 baud.

_bios_serialcom Service Constants

Constant _COM_INIT, _COM_SEND, _COM_RECEIVE, _COM_STATUS

Include <bios.h>

Context _bios_serialcom

The *service* argument can be set to one of the following constants:

Constant	Service
_COM_INIT	Sets the port to the parameters specified in the data argument
_COM_SEND	Transmits the data characters over the selected serial port
_COM_RECEIVE	Accepts an input character from the selected serial port
_COM_STATUS	Returns the current status of the selected serial port

_bios_disk Constants

Constant _DISK_FORMAT, _DISK_READ, _DISK_RESET, _DISK_STATUS, _DISK_VERIFY, _DISK_WRITE

Context _bios_disk

The *service* argument of _bios_disk selects the disk function desired.

The argument is set with one of the following constants:

_DISK_FORMAT
Formats the track specified by *diskinfo*. The *head* and *track* fields indicate the track to format. Only one track can be formatted in a single call. The *buffer* field points to a set of sector markers. The format of the markers depends on the type of disk drive; see a technical reference to the PC BIOS to determine the marker format. The high-order byte (AH) of the return value contains the status of the call; 0 equals success. If there is an error, the high-order byte will contain a set of status flags, as defined below under Return Value.

_DISK_READ
Reads one or more disk sectors into memory. This service uses all fields of the structure pointed to by *diskinfo*. If no error occurs, the function returns 0 in the high-order byte and the number of sectors read in the low-order byte. If there is an error, the high-order byte (AH) will contain a set of status flags, as defined below under Return Value.

_DISK_RESET
Forces the disk controller to do a hard reset, preparing for floppy-disk I/O. This is useful after an error occurs in another operation, such as a read. If this service is specified, the *diskinfo* argument is ignored. Status is returned in the 8 high-order bits (AH) of the return value. If there is an error, the high-order byte will contain a set of status flags, as defined below under Return Value.

_DISK_STATUS
Obtains the status of the last disk operation. If this service is specified, the *diskinfo* argument is ignored. Status is returned in the 8 low-order bits (AL) of the return value. If there is an error, the low-order byte (AL) will contain a set of status flags, as defined below under Return Value.

_DISK_VERIFY

Checks the disk to be sure the specified sectors exist and can be read. It also runs a CRC (cyclic redundancy check) test. This service uses all fields (except *buffer*) of the structure pointed to by *diskinfo*. If no error occurs, the function returns 0 in the high-order byte (AH) and the number of sectors compared in the low- order byte (AL), as defined below under Return Value.

_DISK_WRITE

Writes data from memory to one or more disk sectors. This service uses all fields of the structure pointed to by *diskinfo*. If no error occurs, the function returns 0 in the high- order byte (AH) and the number of sectors written in the low-order byte (AL). If there is an error, the high-order byte will contain a set of status flags, as defined below under Return Value.

Return Value

Bits	Meaning
0X00	No error
0X01	Invalid request or a bad command
0X02	Address mark not found
0X03	Disk write protected
0X04	Sector not found
0X05	Reset failed
0X06	Floppy disk removed
0X07	Drive parameter activity failed
0X08	Direct Memory Access (DMA) overrun
0X09	DMA crossed 64K boundary
0X0A	Bad sector flag detected
0X0B	Bad track flag detected
0X0C	Media type not found
0X0D	Invalid number of sectors on format
0X0E	Control data access mark detected
0X0F	DMA arbitration level out of range
0X10	Data read (CRC or ECC) error
0X11	Corrected data read (ECC) error
0X0	Controller failure
0X40	Seek error
0X80	Disk timed out or failed to respond
0XAA	Drive not ready
0XBB	Undefined error
0XCC	Write fault on drive
0XE0	Status error
0XFF	Sense operation failed

errno Values

Constant ECHILD, EAGAIN, E2BIG, EACCES, EBADF, EDEADLOCK, EDOM, EEXIST, EINVAL, EMFILE, ENOENT, ENOEXEC, ENOMEM, ENOSPC, ERANGE, EXDEV

Include <errno.h>

The errno values are constants assigned to errno in the event of various error conditions.

The include file ERRNO.H contains the definitions of the errno values. However, not all of the definitions given in ERRNO.H are used in DOS. Some of the values in ERRNO.H are present to maintain compatibility with the UNIX (and XENIX) operating system.

The errno values in DOS are a subset of the values for errno in XENIX systems. Thus, the errno value is not necessarily the same as the actual error code returned by a DOS system call. To access the actual DOS error code, use the _doserrno variable, which contains this value.

The following errno values are supported:

ECHILD
No child processes.

EAGAIN
No more processes. An attempt to create a new process failed, because there are no more process slots, or there is not enough memory, or the maximum nesting level has been reached.

E2BIG
Arg list too long. Under DOS: The argument list exceeds 128 bytes, or the space required for the environment information exceeds 32K.

EACCES
Permission denied. The file's permission setting does not allow the specified access. This error signifies that an attempt was made to access a file (or, in some cases, a directory) in a way that is incompatible with the file's attributes.

For example, the error can occur when an attempt is made to read from a file that is not open, to open an existing read-only file for writing, or to open a directory instead of a file. Under MS-DOS versions 3.0 and later, EACCES may also indicate a locking or sharing violation.

The error can also occur in an attempt to rename a file or directory or to remove an existing directory.

EBADF
Bad file number. The specified file handle is not a valid file-handle value or does not refer to an open file; or an attempt was made to write to a file or device opened for read-only access (or vice versa).

EDEADLOCK
Resource deadlock would occur. The argument to a math function is not in the domain of the function.

EDOM
Math argument.

EEXIST
Files exist. An attempt has been made to create a file that already exists. For example, the _O_CREAT and _O_EXCL flags are specified in an open call, but the named file already exists.

EINVAL
Invalid argument. An invalid value was given for one of the arguments to a function. For example, the value given for the origin when positioning a file pointer (by means of a call to fseek) is before the beginning of the file.

EMFILE
Too many open files. No more file handles are available, so no more files can be opened.

ENOENT
No such file or directory. The specified file or directory does not exist or cannot be found. This message can occur whenever a specified file does not exist or a component of a path name does not specify an existing directory.

ENOEXEC

Exec format error. An attempt was made to execute a file that is not executable or that has an invalid executable-file format.

ENOMEM

Not enough core. Not enough memory is available for the attempted operator. For example, this message can occur when insufficient memory is available to execute a child process, or when the allocation request in a _getcwd call cannot be satisfied.

ENOSPC

No space left on device. No more space for writing is available on the device (for example, when the disk is full).

ERANGE

Result too large. An argument to a math function is too large, resulting in partial or total loss of significance in the result. This error can also occur in other functions when an argument is larger than expected (for example, when the path-name argument to the _getcwd function is longer than expected).

EXDEV

Cross-device link. An attempt was made to move a file to a different device (using the rename function).

errno

Constant EOF

Context Returned by stream input and output functions such as putc, ungetc, scanf, fflush, and _fcloseall as well as non- stream functions such as _ungetch, _putch, _and __isascii.

This value is returned by an I/O routine when the end-of-file (or in some cases, an error) is encountered.

_hardresume Constants

Constant _HARDERR_ABORT, _HARDERR_FAIL, _HARDERR_IGNORE, _HARDERR_RETRY

Include <dos.h>

Context _hardresume

The result value supplied to _hardresume must be one of the following constants:

Constant	Action
_HARDERR_ABORT	Aborts the program issuing INT 0x23.
_HARDERR_FAIL	Fails the system call in progress. (Constant is not supported on MS-DOS version 2.x.)
_HARDERR_IGNORE	Ignores the error.
_HARDERR_RETRY	Retries the operation.

_heap... Constants

Constant _HEAPBADBEGIN, _HEAPBADNODE, _HEAPBADPTR, _HEAPEMPTY, _HEAPEND, _HEAPOK

Include <malloc.h>

Context _heapchk, _heapset, _heapwalk

These constants give the return value indicating status of the heap.

The meaning of the constants is listed below:

Constant	Meaning
_HEAPBADBEGIN	The initial header information was not found, or it was invalid.
_HEAPBADNODE	A bad node was found, or the heap is damaged.
_HEAPBADPTR	The _pentry field of the _HEAPINFO structure does not contain a valid pointer into the heap (_heapwalk routines only).
_HEAPEMPTY	The heap has not been initialized.
_HEAPEND	The end of the heap was reached successfully (_heapwalk routines only).
_HEAPOK	The heap is consistent (_heapset and _heapchk routines only).

The heap is OK so far, and the _HEAPINFO structure contains information about the next entry (_heapwalk routines only).

Constant _FREEENTRY, _USEDENTRY

Include <malloc.h>

Context _heapwalk

These constants represent values assigned by the _heapwalk routines to the _useflag element of the _HEAPINFO structure. They indicate the status of the heap entry.

setvbuf Constants

Constant _IOFBF, _IOLBF, _IONBF

Include <stdio.h>

Context setvbuf

These constants represent the type of buffer for setvbuf.

The possible values are given by the following manifest constants:

Constant	Meaning
_IOFBF	Full buffering: The buffer specified in the call to the setvbuf function is used and its size is as specified in the setvbuf call. If the buffer pointer is NULL, an automatically allocated buffer of the specified size is used.
_IOLBF	Same as _IOFBF.
_IONBF	No buffer is used, regardless of the arguments in the call to setvbuf.

_bios_keybrd Constants

Constant _KEYBRD_READ, _KEYBRD_READY, _KEYBRD_SHIFTSTATUS, _NKEYBRD_READ, _NKEYBRD_READY, _NKEYBRD_SHIFTSTATUS

Include <bios.h>

Context _bios_keybrd

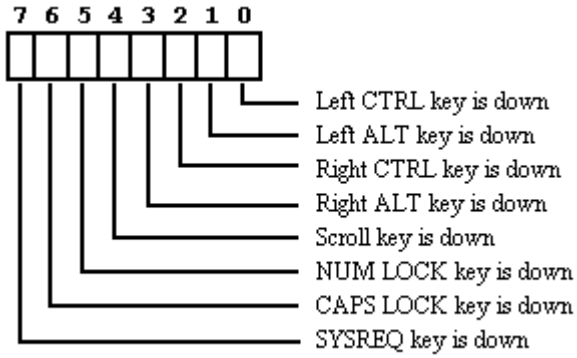
The *service* argument can be any one of the following manifest constants:

_KEYBRD_READ, _NKEYBRD_READ
Reads the next character read from the keyboard. The _NKEYBRD_READ constant is used with enhanced keyboards to obtain the scan codes for function keys F11 and F12 and the cursor control keys. If no character has been typed, the call will wait for one. If the low-order byte of the return value is nonzero, it contains the ASCII value of the character typed. The high-order byte contains the keyboard scan code for the character. See SCAN.H for a list of keyboard scan codes.

_KEYBRD_READY, _NKEYBRD_READY
Checks to see whether a keystroke is waiting to be read and, if so, reads it. The _NKEYBRD_READY constant is for use with enhanced keyboards. The return value is 0 if no keystroke is waiting; otherwise, the return value is the character waiting to be read, in the same format as the _KEYBRD_READ or _NKEYBRD_READ return. This service does not remove the waiting character from the input buffer, as does the _KEYBRD_READ or _NKEYBRD_READ service.

_KEYBRD_SHIFTSTATUS, _NKEYBRD_SHIFTSTATUS
Returns the current shift-key (SHIFT) status. The KEYBRD_SHIFTSTATUS uses only the low-order byte of the return value:

The _NKEYBRD_SHIFTSTATUS uses the low-order and also the high-order byte:



See SHIFT.H for a list of keyboard shift codes.

Locale Categories

Constant LC_ALL, LC_COLLATE, LC_CTYPE, LC_MAX, LC_MIN, LC_MONETARY, LC_NUMERIC, LC_TIME

Include <locale.h>

Context localeconv, setlocale, strcoll, strftime, strxfrm

Locale categories are manifest constants used by the localization routines to specify which portion of a program's locale information will be used. The "locale" refers to the locality (or country) for which certain aspects of your program can be customized. Locale- dependent areas include, for example, the formatting of dates or the display format for monetary values.

The various locale categories, and the areas of a program that they affect, are listed below:

Locale Category	Parts of Program Affected
LC_ALL	All locale-specific behavior (all categories)
LC_COLLATE	Behavior of strcoll and strxfrm functions
LC_CTYPE	Behavior of the character-handling functions (except for isdigit, isxdigit, mbstowcs, and mbtowc, which are unaffected)
LC_MAX	Same as LC_TIME
LC_MIN	Same as LC_ALL
LC_MONETARY	Monetary formatting information returned by the localeconv function
LC_NUMERIC	Decimal-point character for the formatted output routines (for example, printf), the data conversion routines, and the nonmonetary formatting information returned by the localeconv function
LC_TIME	Behavior of the strftime function

_locking Constants

Constant _LK_LOCK, _LK_RLCK, _LK_NBLCK, _LK_NBRLCK, _LK_UNLCK

Include <sys\locking.h>

Context _locking

The *mode* argument in the call to the _locking function specifies the locking action to be performed.

The *mode* argument must be one of the following manifest constants:

_LK_LOCK
Locks the specified bytes. If the bytes cannot be locked, the function tries again after one second. If,

after ten attempts, the bytes cannot be locked, the function returns an error.

_LK_RLCK
Same as _LK_LOCK.

_LK_NBLCK
Locks the specified bytes. If bytes cannot be locked, the function returns an error.

_LK_NBRLCK
Same as _LK_NBLCK.

_LK_UNLCK
Unlocks the specified bytes. (The bytes must have been previously locked.)

Path Field Limits

Constant _MAX_DIR, _MAX_DRIVE, _MAX_EXT, _MAX_FNAME, _MAX_PATH

Include <stdlib.h>

These constants define the maximum length for the path, and for the individual fields within the path.

The constants and their meanings are as follows:

Constant	Meaning
_MAX_DIR	Maximum length of directory component
_MAX_DRIVE	Maximum length of drive component
_MAX_EXT	Maximum length of extension component
_MAX_FNAME	Maximum length of filename component
_MAX_PATH	Maximum length of full path name

The sum of the fields should not exceed _MAX_PATH. The _MAX_PATH limit is larger than current operating systems will handle.

Math-Error Constants

Constant Math errors

Include <math.h>

Context _matherr

The math error constants can be generated by the math routines of the run-time library.

These errors, described as follows, correspond to the exception types defined in MATH.H and are returned by the _matherr function when a math error occurs.

Constant	Meaning
_DOMAIN	An argument to the function is outside the domain of the function.
_OVERFLOW	The result is too large to be represented in the function's return type.
_PLOSS	A partial loss of significance occurred.
_SING	Argument singularity: an argument to the function has an illegal value. (For example, the value 0 is passed to a function that requires a nonzero value.)
_TLOSS	A total loss of significance occurred.
_UNDERFLOW	The result is too small to be represented. (This condition is not currently supported.)

Constant NULL

Context Used with many pointer operations and functions.

The NULL value is the null-pointer value.

File Constants

Constant _O_APPEND, _O_CREAT, _O_EXCL, _O_RDONLY, _O_RDWR, _O_TRUNC, _O_WRONLY

Include <fcntl.h>

Context _open, _sopen, _dos_open

The integer expression formed from one or more of these constants determines the type of reading or writing operations permitted. It is formed by combining one or more constants with a translation-mode constant. See _O_BINARY and _O_TEXT for more information.

The file constants are described below:

_O_APPEND
Repositions the file pointer to the end of the file before every write operation.

_O_CREAT
Creates and opens a new file for writing; this has no effect if the file specified by the path name exists.

_O_EXCL
Returns an error value if the file specified by the path name exists. Only applies when used with _O_CREAT.

_O_RDONLY
Opens file for reading only; if this flag is given, neither _O_RDWR nor _O_WRONLY can be given.

_O_RDWR
Opens file for both reading and writing; if this flag is given, neither _O_RDONLY nor _O_WRONLY can be given.

_O_TRUNC
Opens and truncates an existing file to zero length; the file must have write permission. The contents of the file are destroyed. If this flag is given, _O_RDONLY cannot be given.

_O_WRONLY
Opens file for writing only, if this flag is given, neither _O_RDONLY nor _O_RDWR can be given.

File-Mode Constants

Constant _O_TEXT, _O_BINARY, _O_RAW

Include <fcntl.h>

Context _open, _sopen (*oflag* argument) _setmode (*mode* argument)

The _O_BINARY and _O_TEXT manifest constants determine the translation mode for files (_open and _sopen) or the translation mode for streams (_setmode).

The allowed values are shown below:

_O_TEXT
: Sets text (translated) mode. Carriage-return_linefeed combinations (CR-LF) are translated into a single linefeed (LF) on input. Linefeed characters are translated into CR-LF combinations on output. Also, CTRL+Z is interpreted as an end-of-file character on input. In files opened for reading and reading/writing, fopen checks for CTRL+Z at the end of the file and removes it, if possible. This is done because using the fseek and ftell functions to move within a file ending with CTRL+Z may cause fseek to behave improperly near the end of the file.

_O_BINARY
: Sets binary (untranslated) mode. The above translations are suppressed.

_O_RAW
: Same as _O_BINARY. Supported for C 2.0 compatibility.

File-Permission Setting Constants

Constant _S_IREAD, _S_IWRITE

Include <sys\stat.h>

Context _open, _sopen, _umask, _stat structure

One of these constants is required when _O_CREAT (_open, _sopen) is specified.

The *pmode* argument specifies the file's permission settings, as shown below:

Constant	Meaning
_S_IREAD	Reading permitted
_S_IWRITE	Writing permitted
_S_IREAD | _S_IWRITE	Both reading and writing permitted

When used as the *pmode* argument for _umask, the manifest constant sets the permission setting, as shown below:

Constant	Meaning
_S_IREAD	Writing not permitted (file is read-only)
_S_IWRITE	Reading not permitted (file is write-only)
_S_IREAD | _S_IWRITE	Neither reading nor writing permitted

File Attribute Setting Constants

Constant _S_IFMT, _S_IFDIR, _S_IFCHR, _S_IFREG, _S_IREAD, _S_IWRITE, _S_IEXEC

Include <sys\stat.h>

Context _stat, _fstat, _stat structure

These constants are used to indicate file type in the st_mode field of the _stat structure.

The bit mask constants are described below:

Constant	Meaning
_S_IFMT	File type mask
_S_IFDIR	Directory
_S_IFCHR	Character special (indicates a device if set)
_S_IFREG	Regular
_S_IREAD	Read permission
_S_IWRITE	Write permission
_S_IEXEC	Execute/search permission

Sharing Constants

Constant SH_COMPAT, _SH_DENYRW, _SH_DENYWR, _SH_DENYRD, _SH_DENYNO,

_O_NOINHERIT

Include <share.h>

Context _sopen, _dos_open, _fsopen

The *shflag* argument determines the sharing mode, which consists of one or more manifest constants. These can be combined with the *oflag* arguments above when used in _dos_open.

The constants and their meanings are listed below:

Constant	Meaning
_SH_COMPAT	Sets compatibility mode
_SH_DENYRW	Denies read and write access to file
_SH_DENYWR	Denies write access to file
_SH_DENYRD	Denies read access to file
_SH_DENYNO	Permits read and write access
_O_NOINHERIT	File is not inherited by child process (_dos_open only)

_bios_printer Constants

Constant _PRINTER_WRITE, _PRINTER_INIT, _PRINTER_STATUS

Include <bios.h>

Context _bios_printer

The *service* argument specifies the printer service requested.

The *service* argument can be any of the following manifest constants:

Constant	Meaning
_PRINTER_INIT	Initializes the selected printer. The *data* argument is ignored.
_PRINTER_STATUS	Returns the printer status. The *data* argument is ignored.
_PRINTER_WRITE	Sends the low-order byte of *data* to the printer specified by *printer*.

spawn... Constants

Constant _P_OVERLAY, _P_WAIT

Include <process.h>

Context _spawn family of functions

The *modeflag* argument determines the action taken by the parent process before and during spawn.

The following values for *modeflag* are possible:

Constant	Meaning
_P_OVERLAY	Overlays parent process

	with child, destroying the parent (same effect as exec calls).
_P_WAIT	Suspends parent process until execution of child process is complete (synchronous spawn).

Constant RAND_MAX

Include <stdlib.h>

Context rand

The constant RAND_MAX is the maximum value that can be returned by the rand function. RAND_MAX is defined as the value 0x7fff.

fseek, lseek Constants

Constant SEEK_CUR, SEEK_END, SEEK_SET

Include <stdio.h>

Context fseek, _lseek

The *origin* argument specifies the initial position and can be one of the manifest constants shown below:

Constant	Meaning
SEEK_END	End of file
SEEK_CUR	Current position of file pointer
SEEK_SET	Beginning of file

signal Constants

Constant SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERM

Include <signal.h>

Context signal, raise

The *sig* argument must be one of the manifest constants listed below (defined in SIGNAL.H).

SIGABRT
: Abnormal termination. The default action terminates the calling program with exit code 3.

SIGFPE
: Floating-point error, such as overflow, division by zero, or invalid operation. The default action terminates the calling program. SIGFPE is the only signal constant available when the _WINDOWS constant is defined. The _WINDOWS constant is defined by CL options /GA, /GD, /GE, /GW, /Gw, and /Mq.

SIGILL
: Illegal instruction. This signal is not generated by DOS, but is supported for ANSI compatibility. The default action terminates the calling program.

SIGINT
: CTRL+C interrupt. The default action issues INT 23H.

SIGSEGV
: Illegal storage access. This signal is not generated by DOS, but is supported for ANSI compatibility. The default action terminates the calling program.

SIGTERM
: Termination request sent to the program. This signal is not generated by DOS, but is supported for ANSI compatibility. The default action terminates the calling program.

signal Action Constants

Constant SIG_DFL, SIG_IGN

Include <signal.h>

Context signal

The action taken when the interrupt signal is received depends on the value of *func*.

The *func* argument must be either a function address or one of the manifest constants listed below and defined in SIGNAL.H.

SIG_DFL
: Uses system-default response. Under DOS versions 3.x or earlier, the calling process is terminated and control returns to the DOS command level. If the calling program uses stream I/O, buffers created by the run-time library are not flushed. (DOS buffers are flushed.)

SIG_IGN
: Ignores interrupt signal. This value should never be given for SIGFPE, since the floating-point state of the process is left undefined.

Type size_t

Include <direct.h> <malloc.h> <memory.h> <new.h> <search.h> <stddef.h> <stdio.h> <stdlib.h> <string.h> <time.h>

This is the type returned by sizeof operator. This type is also used as an argument or return type by numerous functions. Generally, size_t is used for any value that represents a count of bytes.

stdin, **stdout**, **stderr**, **_stdaux**, **_stdprn**

Include <stdio.h>

Syntax FILE * stdin;
FILE * stdout;
FILE * stderr;
FILE * _stdaux;
FILE * _stdprn;

Context I/O (Streams) Functions

These are standard streams for input, output, and error output.

By default, standard input is read from the keyboard, while standard output and standard error are printed to the screen.

Two additional streams, standard auxiliary and standard print, are provided by MS-DOS. The assignment of these streams depends on the machine configuration; they usually refer to the first serial port and the first parallel port, respectively, if those ports exist.

The following stream pointers are available to access the standard streams:

Pointer	Stream
stdin	Standard input
stdout	Standard output
stderr	Standard error
_stdaux	Standard auxiliary (MS-DOS only)
_stdprn	Standard print (MS-DOS only)

These pointers can be used as arguments to functions; some functions, such as getchar and putchar, use stdin and stdout automatically.

These pointers are constants, and cannot be assigned new values. The freopen function can be used to redirect the streams to disk files or to other devices. MS-DOS allows you to redirect a program's standard input and output at the command level.

_bios_timeofday Constants

Constant _TIME_GETCLOCK, _TIME_SETCLOCK

Include <bios.h>

Context _bios_timeofday

The *service* argument can be either of the following manifest constants:

_TIME_GETCLOCK
: Copies the current value of the clock count to the location pointed to by *timeval*. If midnight has not passed since the last time the system clock was read or set, the function returns 0; otherwise, it returns 1.

_TIME_SETCLOCK
: Sets the current value of the system clock to the value in the location pointed to by *timeval*. There is no return value.

Type clock_t

Include <time.h>

Context clock

This is the type for representing time elapsed since program invocation**.**

Type time_t

Include <time.h>

Context time, ctime, localtime, gmtime, mktime, difftime, _utime

This is the type for representing calendar time in seconds.

Type va_list

Include <stdarg.h> or <stdio.h>

Context va_start

This is the type of the rightmost parameter of a function that accepts a variable number of arguments of varying types. The parameter holds information used by va_start, va_arg, and va_end for access to the variable arguments.

Type _vmhnd_t

Include <vmemory.h>

Context _vfree, _vload, _vlock, _vlockcnt, _vmalloc, _vmsize, _vrealloc, _vunlock

This is the handle to a block of virtual memory. A handle cannot be used to access memory directly. A handle must be passed to _vload or _vlock in order to get a pointer to memory.

Data-Type Constants

Constant Range limits for data types

Context Used to specify the ranges of C data types.

These are the implementation-dependent ranges of values allowed for C data types.

The constants listed below give the ranges for the integral data types and are defined in LIMITS.H:

Constant	Value	Meaning
SCHAR_MAX	127	Maximum signed char value
SCHAR_MIN	-127	Minimum signed char value
UCHAR_MAX	255	Maximum unsigned char value
CHAR_BIT	8	Number of bits in a char
USHRT_MAX	0xffff	Maximum unsigned short value
SHRT_MAX	32767	Maximum (signed) short value
SHRT_MIN	-32767	Minimum (signed) short value
UINT_MAX	0xffff	Maximum unsigned int value
ULONG_MAX	0xffffffff	Maximum unsigned long value
INT_MAX	32767	Maximum (signed) int value
INT_MIN	-32767	Minimum (signed) int value
LONG_MAX	2147483647	Maximum (signed) long value
LONG_MIN	-2147483647	Minimum (signed) long value
CHAR_MAX	127	Maximum char value
CHAR_MIN	-127	Minimum char value
MB_LEN_MAX	2	Maximum

		number of bytes in multibyte char

Note The /J compiler option changes the default char type to unsigned, with the following limits:

Constant	Value	Meaning
CHAR_MAX	255	Maximum char value
CHAR_MIN	0	Minimum char value

The constants listed below give the range and other characteristics of the double, float, and long double data types, and are defined in FLOAT.H:

Constant	Value	Description
DBL_DIG	15	# of decimal digits of precision
DBL_EPSILON	2.2204460492503131e-016	Smallest such that 1.0+DBL_EPSILON !=1.0
DBL_MANT_DIG	53	# of bits in mantissa
DBL_MAX	1.7976931348623158e+308	Max. value
DBL_MAX_10_EXP	308	Max. decimal exponent
DBL_MAX_EXP	1024	Max. binary exponent
DBL_MIN	2.2250738585072014e-308	Min. positive value
DBL_MIN_10_EXP	(-307)	Min. decimal exponent
DBL_MIN_EXP	(-1021)	Min. binary exponent
_DBL_RADIX	2	Exponent radix
_DBL_ROUNDS	1	Addition rounding: near
FLT_DIG	7	# of decimal digits of precision
FLT_EPSILON	1.192092896e-07	Smallest such that 1.0+FLT_EPSILON !=1.0
FLT_GUARD	0	
FLT_MANT_DIG	24	# of bits in mantissa
FLT_MAX	3.402823466e+38	Max. value
FLT_MAX_10_EXP	38	Max. decimal exponent
FLT_MAX_EXP	128	Max. binary exponent
FLT_MIN	1.175494351e-38	Min. positive value
FLT_MIN_10_EXP	(-37)	Min. decimal exponent
FLT_MIN_EXP	(-125)	Min. binary exponent
FLT_NORMALIZE	0	
FLT_RADIX	2	Exponent radix
FLT_ROUNDS	1	Addition rounding: near
LDBL_DIG	18	# of decimal digits of precision
LDBL_EPSILON	1.084202172485504434e-019L	Smallest such that 1.0+LDBL_EPSILON ! =1.0
LDBL_MANT_DIG	64	# of bits in mantissa
LDBL_MAX	1.189731495357231765e+4932L	Max. value

LDBL_MAX_10_EXP	4932	Max. decimal exponent
LDBL_MAX_EXP	16384	Max. binary exponent
LDBL_MIN	3.3621031431120935063e-4932L	Min. positive value
LDBL_MIN_10_EXP	(-4931)	Min. decimal exponent
LDBL_MIN_EXP	(-16381)	Min. binary exponent
_LDBL_RADIX	2	Exponent radix
_LDBL_ROUNDS	1	Addition rounding: near

Type ptrdiff_t

Include <stddef.h>

This is the type of result of subtracting one pointer from another.

Type sig_atomic_t

Include <signal.h>

This is a type of signal variable, whose value can be set even in the presence of interrupts.

Type fpos_t

Include <stdio.h>

Context fgetpos, fsetpos

This is the type for uniquely specifying a position within a file.

Type FILE

Include <stdio.h>

This is the file type used to specify a file when using stream functions.

Constant FILENAME_MAX

Include <stdio.h>

This is the maximum permissible length for filename.

_MAX_PATH

Constant FOPEN_MAX, _SYS_OPEN

Include <stdio.h>

This is the maximum number of files that can be opened simultaneously. FOPEN_MAX is the ANSI-compatible name. _SYS_OPEN is maintained for compatibility.

Constant TMP_MAX, L_tmpnam

Include <stdio.h>

TMP_MAX is the maximum number of unique filenames that the tmpnam function can generate.
L_tmpnam is the length of temporary filenames generated by tmpnam.

Constant EXIT_SUCCESS, EXIT_FAILURE

Include <stdlib.h>

Context exit

These are arguments for the exit function.

Constant _DOS_MODE, _WIN_MODE

Include <stdlib.h>

These are the values for the _osmode variable.

Constant _REAL_MODE, _PROT_MODE

Include <stdlib.h>

These are the values for the _cpumode variable.

Constant CLOCKS_PER_SEC, CLK_TCK

Include <time.h>

Context clock

The time in seconds is the value returned by the clock function, divided by CLOCKS_PER_SEC. CLK_TCK is equivalent, but considered obsolete.

Constant _HEAP_MAXREQ

Include <malloc.h>

Context malloc, calloc

MS-DOS Time/Date Formats

The file time and the date are stored individually, using unsigned integers as bit fields. File time and date are packed as follows:

Time

Bit Position:	0 1 2 3 4	5 6 7 8 9 A	B C D E F
Length:	5	6	5
Contents:	hours	minutes	2-second increments
Value Range:	0-23	0-59	0-29 in two-second intervals

Date

Bit Position:	0 1 2 3 4 5 6	7 8 9 A	B C D E F
Length:	7	4	5
Contents:	year	month	day
Value Range:	0-119 (relative to 1980)	1-12	1-31

Example

The following code sample extracts the components of a date from a variable wr_date containing a date packed in the format described above. You can use similar methods to extract the time from a variable containing a packed time.

```
#ifdef __cplusplus
    _find_t ft;
#else
    struct _find_t ft;
#endif
    int nDay = ft.wr_date & 0x1f;
    int nMonth = (ft.wr_date >> 5) & 0x0f;
    int nYear = (ft.wr_date >> 9) + 80;
```

Constant HUGE_VAL, _LHUGE_VAL

Include <math.h>

HUGE_VAL and _LHUGE_VAL are the largest representable double or long double values. These values are returned by many run-time math functions when an error occurs. For some functions, -HUGE_VAL is returned.

Type jmp_buf

Include <setjmp.h>

This is the array type used by the setjmp and longjmp functions to save and restore the program environment.

Constant MB_CUR_MAX

Include <stdlib.h>

Context ANSI multibyte- and wide-character conversion functions.

The value of MB_CUR_MAX is the maximum number of bytes in a multibyte character for the current locale.

mblen
mbtowc
mbstowcs
wchar_t
wcstombs
wctomb
MB_LEN_MAX

Type wchar_t

Include <stddef.h>, <stdlib.h>

This is the integral type that can represent all characters of any national character set.

mblen
mbtowc
mbstowcs
wcstombs
wctomb
MB_CUR_MAX
MB_LEN_MAX

Virtual Memory Constants

Constant _VM_ALLDOS, _VM_ALLSWAP, _VM_CLEAN, _VM_DIRTY, _VM_DISK, _VM_EMS, _VM_NULL, _VM_XMS

Include <vmemory.h>

Context virtual memory functions.

Use these manifest constants as arguments to the virtual memory functions.

Value	Meaning
_VM_NULL	Insufficient memory is available, or requested block size is too large
_VM_DIRTY	Write contents of block to auxiliary memory when swapping occurs
_VM_CLEAN	Discard contents of block when swapping occurs
_VM_ALLDOS	Use all available MS-DOS memory
_VM_EMS	Use expanded memory
_VM_XMS	Use extended memory
_VM_DISK	Use disk space
_VM_ALLSWAP	(_VM_EMS | _VM_XMS | _VM_DISK)

QuickWin Constants

Constant _WINBUFINF, _WINBUFDEF, _WINSIZEMIN, _WINSIZEMAX, _WINSIZERESTORE, _WINSIZECHAR, _WINMAXREQ, _WINCURRREQ, _WINPERSIST, _WINNOPERSIST, _WINFRAMEHAND, _WINSTATBAR, _WINTILE, _WINCASCADE, _WINARRANGE, _WINEXITPROMPT, _WINEXITNOPERSIST, _WINEXITPERSIST

Include <io.h>

Context QuickWin functions.

Use these manifest constants as arguments to the QuickWin functions.

Constant	Description
_WINBUFINF	Value for windows screen buffer size
_WINBUFDEF	Value for windows screen buffer size
_WINSIZEMIN	Size/move setting
_WINSIZEMAX	Size/move setting
_WINSIZERESTORE	Size/move setting
_WINSIZECHAR	Size/move setting
_WINMAXREQ	Size/move query type
_WINCURRREQ	Size/move query type

_WINPERSIST	Value for closing window
_WINNOPERSIST	Value for closing window
_WINFRAMEHAND	Pseudo file handle for frame window
_WINSTATBAR	Menu item
_WINTILE	Menu item
_WINCASCADE	Menu item
_WINARRANGE	Menu item
_WINEXITPROMPT	QuickWin exit option
_WINEXITNOPERSIST	QuickWin exit option
_WINEXITPERSIST	QuickWin exit option

Predefined Macros

The compiler automatically defines macros, or identifiers, useful in writing portable programs. You can use these macros to conditionally compile sections of code. These macros are always defined unless otherwise stated.

Microsoft Specific

Macro	Specifies
_CHAR_UNSIGNED	Default char type is unsigned. Macro defined when /J is specified.
__cplusplus	Macro defined when compiling a C++ program. (This macros is not defined for the C compiler.)
_DLL	Code for run-time library as a dynamic-link library. Defined when /MD is specified.
_FAST	Fast-Compile. Macro defined when /f is specified. Supersedes _QC, which is still supported but not recommended. Using /Od causes CL to compile your program with /f. The /f option compiles source files without any default optimizations.
M_I286, _M_I286	80286 processor. Macro defined when /G1 or /G2 is specified.
M_I386, _M_I386	80386 processor. Macro defined when /G3 is specified.
M_I8086, _M_I8086	8088 or 8086 processor;

	default or defined when /G0 is specified.
M_I86, _M_I86	Always defined. Identifies target machine as a member of the 8086 family.
M_I86*m*M, _M_I86*m*M	Always defined. Member of the I86 processor family. Memory model type: *m* = T Tiny S Small (default) C Compact model M Medium model L Large model H Huge model Macros defined by /AT, /AS, /AC, /AM, /AL, and /AH, respectively.
_MSC_VER	Microsoft C version; currently defined as 800.
MSDOS, _MSDOS	Always defined. Identifies target operating system as MS-DOS.
__STDC__	Full conformance with the ANSI C standard. Defined as the integer constant 1 only if the /Za command-line option is given; otherwise is undefined.
_PCODE	Defined for sections of code that are compiled as p-code. Macro defined when /Oq is enabled.
_QC	Supported for compatibility with Microsoft C version 6.0. The _FAST macro (or /f) is the default and is the recommended alternative.
_WINDLL	Windows protected-mode dynamic-link library is selected with /GD.
_WINDOWS	Windows protected-mode is selected with /GA, /Gw, /GW, /Mq, or /GD.

Predefined Time and File Macros

_clearscreen Constants

Constant _GCLEARSCREEN, _GVIEWPORT, _GWINDOW

Include <graph.h>

Context _clearscreen

These constants specify the area of the screen to be erased and filled with the current background color.

The constants and their corresponding actions are as follows:

Constant	Area Cleared and Filled
_GCLEARSCREEN	Entire screen
_GVIEWPORT	Current viewport
_GWINDOW	Current text window

_displaycursor Constants

Constant _GCURSORON, _GCURSOROFF

Include <graph.h>

Context _displaycursor

These constants specify the text cursor condition.

Control Argument for Filled Figures

Constant _GBORDER, _GFILLINTERIOR

Include <graph.h>

Context _ellipse, _rectangle, _polygon, _pie

These constants specify whether the figure is to be filled.

The constants and their corresponding actions are as follows:

Constant	Action
_GBORDER	Do not fill the figure
_GFILLINTERIOR	Fill the figure, using a scanfill algorithm, with the current color and fill mask

Constants for Screen Mode

Constants _DEFAULTMODE, _MAXRESMODE, _MAXCOLORMODE, _TEXTBW40, _TEXTC40, _TEXTBW80, _TEXTC80, _MRES4COLOR, _MRESNOCOLOR, _HRESBW, _TEXTMONO, _HERCMONO, _MRES16COLOR, _HRES16COLOR, _ERESNOCOLOR, _ERESCOLOR, _VRES2COLOR, _VRES16COLOR, _MRES256COLOR, _ORESCOLOR

Context _setvideomode, _setvideomoderows

Include <graph.h>

These constants specify the video screen mode for all graphics operations. See VESA Constants for Screen Mode, _MAXRESMODE & _MAXCOLORMODE Constants for Screen Mode,and Hercules Support for more information.

The *mode* argument can be one of the following manifest constants (defined in GRAPH.H).

Mode	Type	Size	Colors	Adapter
_DEFAULTMODE	Mode existing at startup			
_MAXRESMODE	Highest resolution in graphics mode			
_MAXCOLORMODE	Maximum colors in graphics mode			
_TEXTBW40	BW/T	40 columns	32	CGA
_TEXTC40	C/T	40 columns	32	CGA
_TEXTBW80	BW/T	80 columns	32	CGA
_TEXTC80	C/T	80 columns	32	CGA
_MRES4COLOR	C/G	320 x 200	4	CGA
_MRESNOCOLOR	BW/G	320 x 200	4	CGA
_HRESBW	BW/G	640 x 200	2	CGA
_TEXTMONO	M/T	80 columns	32	MDPA
_HERCMONO *	M/G/H	720 x 348	2	HGC

_MRES16COLOR	C/G	320 x 200	16	EGA
_HRES16COLOR	C/G	640 x 200	16	EGA
_ERESNOCOLOR	M/G	640 x 350	4	EGA
_ERESCOLOR	C/G	640 x 350	16/4	EGA
_VRES2COLOR	C/G	640 x 480	2	VGA
_VRES16COLOR	C/G	640 x 480	16	VGA
_MRES256COLOR	C/G	320 x 200	256	VGA
_ORESCOLOR	C/G	640 x 400	1 of 16	OGA

Type M indicates monochrome, BW indicates monochrome, C indicates color output, T indicates text, H indicates Hercules graphics, and G indicates graphics generation.

Size For text modes, size is given in characters (number of columns). For graphics modes, size is given in pixels (horizontal x vertical).

Colors For monochrome displays, the number of colors is the number of attributes or shades of gray.

Adapter Adapters are the IBM (and compatible) Monochrome Adapter (MDPA), Color Graphics Adapter (CGA), Enhanced Graphics Adapter (EGA), Video Graphics Array (VGA), Hercules-compatible adapter, and Olivetti-compatible adapter.

* In _HERCMONO mode, the text dimensions are 80 columns by 25 rows, with a 9 by 14 character box. The bottom two scan lines of row 25 are not visible.

This table describes only standard hardware; however, display hardware that is strictly compatible with IBM, Hercules, or Olivetti hardware should also work as described.

VESA Constants for Screen Mode

Constant _ORES256COLOR, _VRES256COLOR, _SRES16COLOR, _SRES256COLOR, _XRES16COLOR, _XRES256COLOR, _ZRES16COLOR, _ZRES256COLOR

Context _setvideomode, _setvideomoderows

Include <graph.h>

These constants specify the VESA video screen mode for all graphics operations. VESA is an acronym for Video Electronics Standards Association. See Constants for Screen Mode, _MAXRESMODE & _MAXCOLORMODE Constants for Screen Mode, and Hercules Support for more information.

The *mode* argument to the_setvideomode function can be one of the following manifest constants (defined in GRAPH.H). These constants support screen modes specified by VESA. Other nonstandard Super VGA modes may also be supported. Note that some, or all, of these manifest constants may be supported by graphics cards that support the VESA Super Video standard VS891001. Other modes may also be supported; a TSR driver may be required.

Mode	VESA No.	Type *	Size	Colors	Adapter
_ORES256COLOR	0x0100	C/G	640 x 400	256	SVGA
_VRES256COLOR	0x0101	C/G	640 x 480	256	SVGA
_SRES16COLOR **	0x0102	C/G	800 x 600	16	SVGA
_SRES256COLOR **	0x0103	C/G	800 x 600	256	SVGA
_XRES16COLOR ***	0x0104	C/G	1024 x 768	16	SVGA
_XRES256COLOR ***	0x0105	C/G	1024 x 768	256	SVGA
_ZRES16COLOR ****	0x0106	C/G	1280 x 1024	16	SVGA
_ZRES256COLOR ****	0x0107	C/G	1280 x 1024	256	SVGA

* C indicates color output and G indicates graphics generation.

** Requires NEC MultiSync 3D or equivalent or better.

*** Requires NEC MultiSync 4D or equivalent or better.

**** Requires NEC MultiSync 5D or equivalent or better.

Warning DO NOT attempt to set _SRES16COLOR, _SRES256COLOR, _XRES16COLOR, _XRES256COLOR, _ZRES16COLOR, or _ZRES256COLOR without ensuring that your monitor can safely handle that resolution. Otherwise, you may risk damaging your display monitor! Consult your owner's manual for details. The _MAXRESMODE and _MAXCOLORMODE constants never select these constants.

MAXRESMODE and _MAXCOLORMODEConstants for Screen Mode

Constant _MAXRESMODE, _MAXCOLORMODE

Context _setvideomode, _setvideomoderows

Include <graph.h>

The two special modes for the _setvideomode function, _MAXRESMODE and _MAXCOLORMODE, select the highest resolution or greatest number of colors available with the current hardware, respectively. These two modes fail for adapters that do not support graphics modes. They never select _SRES16COLOR, _SRES256COLOR, _XRES16COLOR, _XRES256COLOR, _ZRES16COLOR, or _ZRES256COLOR mode. See Constants for Screen Mode, VESA Constants for Screen Mode, and Hercules Support for more information.

The following table lists the videomode selected for different adapter and monitor combinations when _MAXRESMODE or _MAXCOLORMODE is specified:

Adapter/Monitor	_MAXRESMODE	_MAXCOLORMODE
MDPA	fail	fail
HGC	_HERCMONO	_HERCMONO
CGA color	_HRESBW	_MRES4COLOR
CGA noncolor	_HRESBW	_MRESNOCOLOR
OCGA	_ORESCOLOR	_MRES4COLOR
OEGA color	_ORESCOLOR	_ERESCOLOR
EGA color 256k	_HRES16COLOR	_HRES16COLOR
EGA color 64k	_HRES16COLOR	_HRES16COLOR
EGA ecd 256k	_ERESCOLOR	_ERESCOLOR
EGA ecd 64k	_ERESCOLOR	_HRES16COLOR
EGA mono	_ERESNOCOLOR	_ERESNOCOLOR
MCGA	_VRES2COLOR	_MRES256COLOR
VGA	_VRES16COLOR	_MRES256COLOR
OVGA	_VRES16COLOR	_MRES256COLOR
SVGA	_VRES256COLOR	_VRES256COLOR*

* If _VRES256COLOR is supported by the adapter/monitor combination. If not, _MAXCOLORMODE will be either _ORES256COLOR (if supported) or _MRES256COLOR and _MAXRESMODE will be _VRES16COLOR.

Note that a color monitor is assumed for CGA adapters if the startup text mode was TEXTC80 or TEXTC40. If the initial text mode is TEXTBW80 or TEXTBW40, a noncolor CGA monitor is assumed.

Constant _MAXTEXTROWS

Include <graph.h>

Context _settextrows, _setvideomoderows

These constants specify the maximum number of rows available. In text modes, the maximum is 50 for VGA, 43 for EGA, and 25 for others. In graphics modes that support 30 or 60 rows, _MAXTEXTROWS specifies 60 rows. In SVGA modes, _MAXTEXTROWS specifies the vertical resolution (as returned in a_videoconfig struct by the _getvideoconfig function) divided by 8.

Video Adapter Constants

Constant _CGA, _EGA, _VGA, _MCGA, _MDPA, _HGC, _OCGA, _OEGA, _OVGA, _SVGA

Include <graph.h>

Context _getvideoconfig

The video adapter constants are the possible values given by adapter element of the _videoconfig structure. This value indicates the currently active video adapter.

The video adapter constants and their meanings are listed below:

Adapter Constant	Meaning
_CGA	Color Graphics Adapter
_EGA	Enhanced Graphics Adapter
_VGA	Video Graphics Array
_MCGA	Multicolor Graphics Array
_MDPA	Monochrome Display Adapter
_HGC	Hercules Graphics Card
_OCGA	Olivetti (AT&T) Color Graphics Adapter
_OEGA	Olivetti (AT&T) Enhanced Graphics Adapter
_OVGA	Olivetti (AT&T) Video Graphics Array
_SVGA	Super VGA with VESA BIOS support

Monitor Display Constants

Constant _MONO, _COLOR, _ENHCOLOR, _ANALOG, _ANALOGMONO, _ANALOGCOLOR

Include <graph.h>

Context _getvideoconfig

The monitor display constants are the possible values given by monitor element of _videoconfig structure. This value indicates the currently active monitor.

The monitor display constants and their meanings are listed below:

Monitor Constant	Meaning
_MONO	Monochrome monitor
_COLOR	Color (or enhanced monitor emulating color)
_ENHCOLOR	Enhanced color
_ANALOG	Analog monochrome and color modes
_ANALOGMONO	Analog monochrome only

_ANALOGCOLOR	Analog color only

Chart Type Constants

Constant _PG_BARCHART, _PG_COLUMNCHART, _PG_LINECHART, _PG_SCATTERCHART, _PG_PIECHART, _PG_PLAINBARS, _PG_STACKEDBARS, _PG_POINTANDLINE, _PG_POINTONLY, _PG_PERCENT, _PG_NOPERCENT

Include <graph.h>

Context _pg_defaultchart

These constants specify the type and style of the default chart for the presentation-graphics routines.

The *charttype* argument of_pg_defaultchart specifies one of five types of charts:

Chart-Type Constant	Meaning
_PG_BARCHART	Bar chart
_PG_COLUMNCHART	Column chart
_PG_LINECHART	Line chart
_PG_SCATTERCHART	Scatter chart
_PG_PIECHART	Pie chart

The *chartstyle* argument specifies the style of the chart. Each of the five types of charts can appear in two different chart styles, specified by the constants below:

Chart Type	Chart Styles Available
Bar	_PG_PLAINBARS, _PG_STACKEDBARS
Column	_PG_PLAINBARS, _PG_STACKEDBARS
Line	_PG_POINTANDLINE, _PG_POINTONLY
Scatter	_PG_POINTANDLINE, _PG_POINTONLY
Pie	_PG_PERCENT, _PG_NOPERCENT

For pie charts in the _PG_PERCENT format, percentages are printed next to each slice. For bar and column charts, the styles are applicable only when more than one series appears on the same chart. The _PG_PLAINBARS style arranges the bars or columns for the different series side by side, showing relative heights or lengths. The _PG_STACKEDBARS style emphasizes relative sizes between bars and columns.

Image Constants

Constant _GAND, _GOR, _GPRESET, _GPSET, _GXOR

Include <graph.h>

Context _putimage, _putimage_w

These constants specify the interaction between the stored image and the image that is already on the screen.

The image constants are described below:

_GAND
Transfers the image over an existing image on the screen. The resulting image is the logical-AND product of the two images: points that had the same color in both the existing image and the new one will remain the same color, while points that have different colors are joined by logical-AND.

_GOR
Superimposes the image onto an existing image. The new image does not erase the previous screen contents.

_GPRESET
Transfers the data point-by-point onto the screen. Each point has the inverse of the color attribute it had when taken from the screen by _getimage, producing a negative image.

_GPSET
Transfers the data point-by-point onto the screen. Each point has the exact color attribute it had when taken from the screen by _getimage.

_GXOR
Causes the points on the screen to be inverted where a point exists in the image buffer. This behavior is exactly like that of the cursor: when an image is put against a complex background twice, the background is restored unchanged. This allows you to move an object around without erasing the background. The _GXOR constant is a special mode often used for animation.

Color Constants

Constant _BLACK, _BLUE, _GREEN, _CYAN, _RED, _MAGENTA, _BROWN, _WHITE, _GRAY, _LIGHTBLUE, _LIGHTGREEN, _LIGHTCYAN, _LIGHTRED, _LIGHTMAGENTA, _YELLOW, _BRIGHTWHITE

Include <graph.h>

Context _remapallpalette, _remappalette, _setbkcolor

These constants specify the color value for palettes and background color.

Note _LIGHTYELLOW is equivalent to _YELLOW, but is considered obsolete.

Text-Wrapping Constants

Constant _GWRAPON, _GWRAPOFF

Include <graph.h>

Context _wrapon

These constants specify the text wrapping of text output with the _outtext and _outmem functions.

The text-wrapping constants and their corresponding actions are as follows:

Constant	Action
_GWRAPON	Wraps lines at window border
_GWRAPOFF	Truncates line at window border

Chart Title Constants

Constant _PG_LEFT, _PG_CENTER, _PG_RIGHT, _PG_BOTTOM, _PG_OVERLAY

Include <pgchart.h>

Context Justify member of _titletype structure. Place member of _legendtype structure.

_PG_LEFT, _PG_CENTER, and _PG_RIGHT specify how the title will be justified within the chart window.

_PG_RIGHT, _PG_BOTTOM, and _PG_OVERLAY specify how the legend will be placed within the data window.

Chart Axis Constants for Scales

Constant _PG_LINEARAXIS, _PG_LOGAXIS

Include <pgchart.h>

Context range type member of _axistype structure

These constants specify whether the scale of the axis will be linear or logarithmic.

Chart Axis Constants for Tick Marks

Constant _PG_DECFORMAT, _PG_EXPFORMAT

Include <pgchart.h>

Context ticformat member of _axistype structure

These constants specify whether the tick mark labels will be in decimal or exponential format.

_titletype Structure

Include <pgchart.h>

Context _axistype and _chartenv types used by _pg_chart and other chart functions.

Structure:

```
typedef struct
{
   char title[_PG_TITLELEN];      // Title
   text short titlecolor;         // Internal palette color for title
   text short justify;            // _PG_LEFT, _PG_CENTER, _PG_RIGHT
} _titletype;
```

_axistype Structure

Include <pgchart.h>

Context _chartenv type used by _pg_chart and other chart functions.

Structure:

```
typedef struct
{
   short grid;          // TRUE=grid lines drawn; FALSE=no lines
   short gridstyle;     // Style number from style pool for grid line
   titletype axistitle; // Title definition for axis
   short axiscolor;     // Color for axis
   short labeled;       // TRUE=tic marks and titles drawn
   short rangetype;     // _PG_LINEARAXIS, _PG_LOGAXIS
   float logbase;       // Base used if log axis
   short autoscale;     // TRUE=next 7 values calculated by system
   float scalemin;      // Minimum value of scale
   float scalemax;      // Maximum value of scale
   float scalefactor;   // Scale factor for data on this axis
   titletype scaletitle;// Title definition for scaling factor
   float ticinterval;   // Distance between tic marks (world coord.)
   short ticformat;     // _PG_EXPFORMAT or _PG_DECFORMAT for tic labels
   short ticdecimals;   // Number of decimals for tic labels (max=9)
} _axistype;
```

_windowtype Structure

Include <pgchart.h>

Context _legendtype and _chartenv types used by _pg_chart and other chart functions.

Structure:

```
typedef struct
{
   short x1;           // Left edge of window in pixels
   short y1;           // Top edge of window in pixels
   short x2;           // Right edge of window in pixels
   short y2;           // Bottom edge of window in pixels
   short border;       // TRUE for border, FALSE otherwise
   short background;   // Internal palette color for window background
   short borderstyle;  // Style bytes for window border
   short bordercolor;  // Internal palette color for window border
} _windowtype;
```

_legendtype Structure

Include <pgchart.h>

Context _chartenv type used by _pg_chart and other chart functions.

Structure:

```
typedef struct
{
   short legend;              // TRUE=draw legend; FALSE=no legend
   short place;               // _PG_RIGHT, _PG_BOTTOM, _PG_OVERLAY
   short textcolor;           // Internal palette color for text
   short autosize;            // TRUE=system calculates size
   windowtype legendwindow;   // Window definition for legend
} _legendtype;
```

_chartenv Structure

Include <pgchart.h>

Context _pg_chart and other chart functions.

Structure:

```
typedef struct
{
   short charttype;         // _PG_BARCHART, _PG_COLUMNCHART, _PG_LINECHART,
                           //   _PG_SCATTERCHART, _PG_PIECHART
   short chartstyle;        // Style for selected chart type
   _windowtype chartwindow; // Window definition for overall chart
   _windowtype datawindow;  // Window definition for data part of chart
   _titletype maintitle;    // Main chart title
   _titletype subtitle;     // Chart subtitle
   _axistype xaxis;         // Definition for X axis
   _axistype yaxis;         // Definition for Y axis
   _legendtype legend;      // Definition for legend
} _chartenv;
```

_paletteentry Structure

Include <pgchart.h>

Context _pg_getpalette, _pg_setpalette

Structure:

```
typedef struct
{
   unsigned short      color;
   unsigned short      style;
   fillmap             fill;
   char                plotchar;
} _paletteentry;
```

_videoconfig Structure

Include <graph.h>

Context _getvideoconfig

Structure:

```
struct _videoconfig
{
   short numxpixels;    // Number of pixels on X axis
   short numypixels;    // Number of pixels on Y axis
   short numtextcols;   // Number of text columns available
   short numtextrows;   // Number of text rows available
   short numcolors;     // Number of colors
   short bitsperpixel;  // Number of bits per pixel
   short numvideopages; // Number of available video pages
   short mode;          // Current video mode
   short adapter;       // Active display adapter
   short monitor;       // Active display monitor
   short memory;        // Adapter video memory in K bytes
};
```

_xycoord Structure

Include <graph.h>

Context _getcurrentposition, _moveto, _polygon, _getarcinfo, _setvieworg, _getviewcoord, _getphyscoord, ..._xy functions

Structure:

```
struct _xycoord           // Zero-based x and y
{
   short xcoord;
   short ycoord;
};
```

_rccoord Structure

Include: <graph.h>

Context _settextposition, _gettextposition

Structure:

```
struct _rccoord           // Ones-based row and column
{
   short row;
   short col;
};
```

_wxycoord Structure

Include <graph.h>

Context _getwindowcoord, _getviewcoord_wxy, _moveto_w,_getcurrentposition_w, ..._wxy functions

Structure:

```
struct _wxycoord
{
   double wx;          // Window x coordinate
   double wy;          // Window y coordinate
};
```

_fontinfo Structure

Include <graph.h>

Context _getfontinfo

Structure:

```
struct _fontinfo
{
   int type;             // b0 set=vector, clear=bit map
   int ascent;           // Pix dist from top to baseline
   int pixwidth;         // Character width in pixels, 0=prop
   int pixheight;        // Character height in pixels
   int avgwidth;         // Average character width in pixels
   char filename[81];    // Filename including path
   char facename[32];    // Font name
};
```

Text-Scrolling Constants

Constant _GSCROLLUP, _GSCROLLDOWN

Include <graph.h>

Context _scrolltextwindow

These constants specify the direction in which to scroll text.

The text-scrolling constants and their corresponding actions are as follows:

Constant	Action
_GSCROLLUP	Scroll up one line
_GSCROLLDOWN	Scroll down one line

_ERESNOCOLOR Color Values

Constant _MODEFOFF, _MODEFOFFTOON, _MODEFOFFTOHI, _MODEFONTOOFF, _MODEFON, _MODEFONTOHI, _MODEFHITOOFF, _MODEFHITOON, _MODEFHI

Include <graph.h>

Context _remapallpalette, _remappalette, _setbkcolor

These constants specify the colors available in _ERESNOCOLOR mode.

_TEXTMONO Color Values

Constant _MODE7OFF, _MODE7ON, _MODE7HI

Include <graph.h>

Context _remapallpalette, _remappalette, _setbkcolor

These constants specify the colors available in _TEXTMONO mode.

pgchart Error Codes

Constant: _PG_TOOFEWSERIES, _PG_TOOSMALLN, _PG_NOMEMORY, _PG_BADDATAWINDOW, _PG_BADLEGENDWINDOW, _PG_BADCHARTTYPE, _PG_BADSCREENMODE, _PG_NOTINITIALIZED, _PG_BADCHARTWINDOW, _PG_BADSCALEFACTOR, _PG_BADLOGBASE, _PG_BADCHARTSTYLE

Include <pgchart.h>

Context _pg_initchart and other _pg... functions

These constants are the error codes returned by the presentation-graphics functions. Error codes greater than 100 will terminate the chart routine; others will cause default values to be used.

The pgchart error codes and their corresponding values and meanings are listed below:

Constant	Value	Meaning
_PG_TOOFEWSERIES	110	Number of series <=0
_PG_TOOSMALLN	109	Number of data points <=0
_PG_NOMEMORY	108	Not enough memory
_PG_BADDATAWINDOW	107	Data window invalid
_PG_BADLEGENDWINDOW	105	Legend window invalid
_PG_BADCHARTTYPE	104	Chart type invalid
_PG_BADSCREENMODE	103	Not in graphics mode
_PG_NOTINITIALIZED	102	_pg_initchart has not been called
_PG_BADCHARTWINDOW	7	x1=x2 or y1=y2 in chart window
_PG_BADSCALEFACTOR	6	Scale factor=0
_PG_BADLOGBASE	5	Log base <=0
_PG_BADCHARTSTYLE	4	Chart style invalid

The functions that can produce these error codes are listed below.

Function	Possible Error Codes
_pg_analyzechart	_PG_NOTINITIALIZED, _PG_BADCHARTSTYLE,

	_PG_BADCHARTTYPE, _PG_BADLEGENDWINDOW, _PG_BADCHARTWINDOW, _PG_BADDATAWINDOW, _PG_NOMEMORY, _PG_BADLOGBASE, _PG_BADSCALEFACTOR, _PG_TOOFEWSERIES _PG_TOOSMALLN
_pg_analyzechartms	(same as _pg_analyzechart)
_pg_analyzepie	(same as_pg_analyzechart; does not return _PG_TOOSMALLN)
_pg_analyzescatter	(same as _pg_analyzechart)
_pg_analyzescatterms	(same as _pg_analyzechart)
_pg_chart	(same as _pg_analyzechart)
_pg_chartms	(same as _pg_analyzechart)
_pg_chartpie	(same as _pg_analyzechart)
_pg_chartscatter	(same as _pg_analyzechart)
_pg_chartscatterms	(same as_pg_analyzechart)
_pg_defaultchart	_PG_BADCHARTTYPE
_pg_hlabelchart	_PG_NOTINITIALIZED, _PG_BADCHARTWINDOW
_pg_vlabelchart	_PG_NOTINITIALIZED, _PG_BADCHARTWINDOW
_pg_initchart	_PG_BADSCREENMODE
_pg_getchardef	_PG_NOTINITIALIZED
_pg_setchardef	_PG_NOTINITIALIZED
_pg_getpalette	_PG_BADSCREENMODE
_pg_setpalette	_PG_BADSCREENMODE
_pg_resetpalette	_PG_BADSCREENMODE

Hercules Support

In _HERCMONO mode, only monochrome (two-color) text and graphics are supported. The screen resolution is720 x 348 pixels. The text dimensions are 80 columns by 25 rows, with a 9 x 14 character box. The bottom two scan lines of the twenty-fifth row are not visible.

You must install the Hercules driver MSHERC.COM before running your program.Type MSHERC to load the driver. This can be automated by adding a line to your AUTOEXEC.BAT file.

If you have both a Hercules monochrome card and a color video card, you should install MSHERC.COM with the /H (/HALF) option. The /H option causes the driver to use one instead of two graphics pages. This prevents the two video cards from attempting to use the same memory. You do not need to use the /H option if you have only a Hercules card. See your Hercules hardware manuals for more details on compatibility.

To use a mouse, you must follow special instructions for Hercules cards in the book *Microsoft Mouse Programmer's Reference Guide*. (This book is sold separately; it is not supplied with either Microsoft C++ or the mouse package.

const

Keyword const

Syntax const *declaration*
member-function const

When it modifies a data declaration, the const keyword specifies that the object or variable is not modifiable. When it follows a member function's parameter list, the const keyword specifies that the function doesn't modify the object for which it is invoked.

See Constant Values and Constant Member Functions for more information.

volatile
define

Constant Values

In both C and C++, the const keyword specifies that a variable's value is constant and tells the compiler to prevent the programmer from modifying it. For example:

```
const int i = 5;

i = 10; // Error
i++;    // Error
```

In C++, you can use the const keyword instead of the #define preprocessor directive to define constant values. Values defined with const are subject to type checking, and can be used in place of constant expressions. For example, in C++ you can specify the size of an array with a const variable as follows:

```
const int maxarray = 255;
char store_char[maxarray];  // Legal in C++; illegal in C
```

In C, constant values default to external linkage, so they can appear only in source files. In C++, constant values default to internal linkage, which allows them to appear in header files.

The const keyword can also be used in pointer declarations. For example:

```
char *const aptr = mybuf;  // Constant pointer

*aptr = 'a';       // Legal
aptr = yourbuf;    // Error
```

A pointer to a variable declared as const can only be assigned to a pointer that is also declared as const.

```
const char *bptr = mybuf;  // Pointer to constant data

*bptr = 'a';       // Error
bptr = yourbuf;    // Legal
```

You can use pointers to constant data as function parameters in order to prevent the function from modifying a parameter passed through a pointer.

See #define for more information.

Constant Member Functions

Declaring a member function with the const keyword specifies that the function is a "read-only" function that does not modify the object for which it is called.

Declaring Constant Member Functions

To declare a constant member function, place the const keyword after the closing parenthesis of the argument list. The const keyword is required in both the declaration and the definition. A constant member function cannot modify any data members or call any member functions that aren't constant.

```
class Date
{
public:
   Date( int mn, int dy, int yr );
   int getMonth() const;       // A read-only function
   void setMonth( int mn );    // A write function;
                               //    cannot be const
_private:
   int month;
};

int Date::getMonth() const
{
   return month;        // Doesn't modify anything
}

_void Date::setMonth( int mn )
{
   month = mn;          // Modifies data member
}
```

Declaring Constant Objects

To declare a constant object, place the const keyword at the start of an object declaration:

```
const Date birthday( 3, 4, 1985 );
```

You can only call constant member functions for a constant object. This ensures that the object is never modified.

```
birthday.getMonth();    // Okay
birthday.setMonth( 4 ); // Error
```

You can call either constant or nonconstant member functions for a nonconstant object. You can also overload a member function using the const keyword; this allows a different version of the function to be called for constant and nonconstant objects.

You cannot declare constructors or destructors with the const keyword.

enum

Keyword enum

Syntax enum [*tag*] {*enum-list*} [*declarator*];

enum *tag declarator*;

 More information on *declarator*

The enum keyword specifies an enumerated type.

An enumerated type is a user-defined type consisting of a set of named constants called enumerators. By default, the first enumerator has a value of 0, and each successive enumerator is one larger than the value of the previous one, unless you explicitly specify a value for a particular enumerator. Enumerators do not have to have unique values. The name of each enumerator is treated as a constant and must be unique within the scope where the enum is defined.

Example

```
enum Days                       // Declare enum type Days
{
   saturday,                   // saturday = 0 by default
   sunday = 0,                 // sunday = 0 as well
   monday,                     // monday = 1
   tuesday,                    // tuesday = 2
   wednesday,                  // etc.
   thursday,
   friday
} today;                        // Variable today has type Days

int  tuesday;                   // Error, redefinition of tuesday
```

In C, you can use the enum keyword and the tag to declare variables of the enumerated type. In C++, you can use the tag alone. For example:

```
enum Days yesterday;     // Legal in C and C++
Days tomorrow;           // Legal in C++ only

yesterday = monday;
```

An enumerated type is an integral type. An enumerator can be promoted to an integer value. However, converting an integer to an enumerator requires an explicit cast, and the results are not defined. For example:

```
int i = tuesday;           // Legal; i = 2
yesterday = 0;             // Error; no conversion
yesterday = (Days)0;       // Legal, but results undefined
```

In C++, enumerators defined within a class are accessible only to member functions of that class unless qualified with the class name (for example, class_name::enumerator). You can use the same syntax for explicit access to the type name (class_name::tag).

class
struct

__export

Keyword __export

Syntax __export *declarator*


More information on *declarator*

The __export keyword specifies that a symbol is exported from a dynamic-link library (DLL). For compatibility with previous versions, _export is a synonym for __export.

Any of the following can be declared as __export:

- Data
- Functions
- Member functions
- Constructors
- Destructors
- Operators

The main use for __export is to export symbols that reside in a dynamic-link library. You can also export call-back functions for Microsoft Windows.

When you use the __export keyword, the linker assumes the exported symbol has the following attributes:

- No input/output (I/O) privileges
- Shared data
- Not resident
- No alias name

If you want to alter these attributes, you must use a module-definition (.DEF) file with an EXPORTS statement when building a DLL. See EXPORTS Statement for more information.

The __export keyword also causes the compiler to enter the size, in words, of the function's parameters into the export record of the object module. This size information corresponds to the *pwords* field of an EXPORTS statement that is in a .DEF file. You cannot override the size information in the export record with an EXPORTS entry in the .DEF file.

If you have an EXPORTS entry for a function, the *pwords* field in the .DEF file should be set either to 0 (which tells the linker to use the value given by the compiler) or to the same value given by the compiler. The *pwords* field is ignored unless you also request I/O privilege.

You can create an import library for a DLL in the following manner:

1. Provide a .DEF file entry for every symbol that you want to export.
2. Use the IMPLIB utility. It creates a file with a .LIB extension for use by the linker. Specify the .LIB file on the LINK line with the other libraries.

The following statement declares funcsample as a far Pascal function that takes a single argument of any pointer type and does not return a value. The presence of __export causes the function to be exported.

```
void __export __far __pascal funcsample( void *s );
```

You can also use the export keyword with the /GA and the /GD compiler options to generate efficient prolog/epilog code. For protected-mode applications, use the /GA option to generate the correct prolog/epilog code for all far functions explicitly marked as __export.

For protected-mode DLLs, use the /GD option to generate the correct prolog/epilog code and to create a linker EXPDEF record for all far functions explicitly marked as __export.

EXPORTS Statement
Module-Definition Files

extern

Keyword extern

Syntax extern *declarator*

extern *string-literal declarator*

extern *string-literal* { *declarator-list* }

More information on *declarator*

The extern keyword declares a variable or function and specifies that it has external linkage (its name is visible from files other than the one in which it's defined). When modifying a variable, extern specifies that the variable has static duration (it is allocated when the program begins and deallocated when the program ends). The variable or function may be defined in another source file, or later in the same file. In C++, when used with a string, extern specifies that the linkage conventions of another language are being used for the declarator(s). See Declarations and Definitions for more information.

Declarations of variables and functions at file scope are external by default.

In C++, *string-literal* is the name of a language. The language specifier "C++" is the default. "C" is the only other language specifier currently supported by Microsoft C/C++. This allows you to use functions or variables defined in a C module.

Example

```
extern "C" int printf( const char *, ... );

extern "C"
{
   int getchar( void );
   int putchar( int );
}
```

All of the standard include files use the extern "C" syntax to allow the run-time library functions to be used in C++ programs.

auto
register
static
const
volatile

Declarations and Definitions

A declaration specifies certain attributes of an identifier but does not allocate storage. The extern keyword tells the compiler to defer allocation. An initializer is not allowed in a declaration. For example:

```
extern int y;          // Names a variable. Defers storage allocation.
extern const start;    // Names a constant. Defers storage allocation.
struct person;         // Names a structure type. The members
                       //    are not defined and the size of
                       //    the struct is not known.
int sum( int, int );   // Names a function and specifies return
                       //    type and number and type of
                       //    parameters. Parameter names are
                       //    recommended but not required.
```

For a variable, a definition provides both a name and a type that allow the compiler to allocate a section of memory that is large enough to accommodate the type. An initializer is allowed in a definition. For example:

```
int x;                 // Allocates an integer.
int y = 0;             // Allocates and initializes an integer.
const start = 1;       // Allocates and initializes a constant.
struct person          // Defines a structure type and allocates
{                      //    a variable of that type.
   long phone;
   char *name;
} customer;
```

For a function, a definition provides everything that a declaration provides, plus a function body. For example:

```
int sum( int a, int b )
{
return a + b;
}
```

__huge

Keyword __huge

Syntax type __huge *declarator*

class __huge *class-name*

class __huge *class-name func*( )

type *member-func*( ) __huge

More information on *declarator*

The __huge keyword specifies that a data object can reside anywhere in memory and is not assumed to reside in the current data segment. Individual data items can exceed 64K in size. Data is referenced with a 32-bit address, and pointers declared as __huge are 32-bit values.

The __huge keyword can be used to modify data objects, pointers, classes, the this pointer of member functions, and the addressing mode of objects returned by functions.

If the /Ze compiler option is used, both huge and _huge are synonyms for __huge. If /Za is used, only __huge is accepted.

Note Do not use __huge in 32-bit programs.

Data Objects or Pointers

The compiler can put a huge data object anywhere in memory. A huge object can be larger than 64K.

A huge pointer is a 32-bit value and can address data in any segment. For example:

```
char __huge a[100000];       // A huge array
char __huge *hp;             // A huge pointer
```

Classes

Objects of a huge class can reside anywhere in memory and can be larger than 64K in size, unless an overriding keyword appears in the individual declaration. Structure or union types also can be declared __huge. For example:

```
class __huge Node
{
   // ...
};

Node my_node;              // Huge by default
Node __near your_node;     // Explicitly declared __near
```

The this Pointer of Member Functions

You can overload a member function to operate on huge objects if objects of the class are not huge by default. Within the function, the this pointer is a huge pointer. For example:

```
class Node
{
public:
   void print() __huge;    // Called for huge objects
   void print();           // Called for default objects
private:
   // ...
};
```

The Addressing Mode of Return Objects

You can specify that a function returns a huge object. This is useful if you call a member function for the temporary object returned by the function. For example:

```
class __huge Node make_node(); // Function returns a huge Node

make_node().print();     // Call huge print() for
                         //    temporary object
```

__based
__far
__near
__huge
Memory Models

__far

Keyword __far

Syntax type __far *declarator*

class __far *class-name*

class __far *class-name func*( )

type *member-func*( ) __far

More information on *declarator*

The __far keyword specifies that a data object can reside anywhere in memory and is not assumed to reside in the default data segment. It specifies that a function can be called by other functions anywhere in memory, and not only by those in the same code segment. Functions and data are referenced with 32-bit addresses, and pointers declared as __far are 32-bit values.

The __far keyword can be used to modify data objects, pointers, functions, classes, the this pointer of member functions, and the addressing mode of objects returned by functions.

If the /Ze compiler option is used, both far and _far are synonyms for __far. If /Za is used, only __far is accepted.

Data Objects or Pointers

A far data object can reside anywhere in memory.

A far pointer is a 32-bit address value that provides access to data in any segment. For example:

```
char __far a;                 // Far character
char __far *fp;               // Far pointer
```

The Calling Convention of Functions

A far function can be called by functions in any code segment. A pointer to a far function is a 32-bit value. For example:

```
char __far redraw();
```

Classes

Objects of a far class can reside anywhere in memory, unless an overriding keyword appears in the individual declaration. Structure or union types also can be declared as __far. For example:

```
class __far Node
{
   // ...
};

Node my_node;              // Far by default
Node __near your_node;     // Explicitly declared __near
```

Member Functions

You can overload a member function to operate on far objects if objects of the class are not far by default. Within the function, the this pointer is a far pointer. For example:

```
class Node
{
public:
   void print() __far;     // Called for far objects
   void print();           // Called for default objects
private:
   // ...
};
```

The Addressing Mode of Return Objects

You can specify that a function return a far object. This is useful if you call a member function for the temporary object returned by the function. For example:

```
class __far Node make_node(); // Function returns a far Node

make_node().print();     // Call far print() for temporary
                         //     object
```

__near

Keyword __near

Syntax type __near *declarator*

class __near *class-name*

class __near *class-name func*( )

type *member-func*( ) __near

More information on *declarator*

The __near keyword specifies that a data object resides in the default data segment. It specifies that a function can be called only by functions in the same code segment. Functions and data are referenced with 16-bit addresses, and pointers declared as __near are 16-bit values.

The __near keyword can be used to modify data objects, pointers, functions, classes, the this pointer of member functions, and the addressing mode of objects returned by functions.

If the /Ze compiler option is used, both near and _near are synonyms for __near. If /Za is used, only __near is accepted.

Data Objects or Pointers

A near data object resides in the default data segment.

A near pointer is a 16-bit value and can address locations within the default segment, but not locations outside of it. For example:

```
char __near a;              // Char in default data segment
char __near *np;            // Near pointer
```

The Calling Convention of Functions

A near function can be called only by other functions in the same code segment. A pointer to a near function is a 16-bit value. For example:

```
char __near redraw();
```

You can control the placement of functions within segments. See __based for more information.

Classes

Objects of a near class reside in the default data segment unless an overriding keyword appears in the individual declaration. Structure or union types can also be declared as __near. For example:

```
class __near Node
{
   // ...
};

Node my_node;              // Near by default
Node __far your_node;      // Explicitly declared __far
```

Member Functions

You can overload a member function to operate on near objects if objects of the class are not near by default. Within the function, the this pointer is a near pointer. For example:

```
class Node
{
public:
   void print() __near;      // Called for near objects
   void print();             // Called for default objects
private:
   // ...
};
```

The Addressing Mode of Return Objects

You can specify that a function return a near object. This is useful if you call a member function for the temporary object returned by the function. For example:

```
class __near Node make_node(); // Function returns a near Node

make_node().print();     // Call near print() for temporary __//    object
```

__interrupt

Keyword __interrupt

Syntax __interrupt *declarator*

More information on *declarator*

The __interrupt keyword specifies that a function is an interrupt handler. The compiler generates entry and exit sequences for the interrupt handler, including saving and restoring all registers and executing an IRET instruction to return.

The __interrupt keyword is used in C to label a function as an interrupt handler. It can also be applied to static member functions in C++. An interrupt handler cannot be declared inline.

An interrupt handler must be a far function. If you are compiling with the small (default) or compact memory model, you must declare your interrupt handler with the __far attribute.

An interrupt handler must use the C calling convention. If you use the /Gc or /Gr compiler options, you must declare your interrupt handler with the __cdecl attribute.

You cannot declare an interrupt handler with either the __saveregs attribute or the __fastcall calling convention.

If the /Ze compiler option is used, both interrupt and _interrupt are synonyms for __interrupt. If /Za is used, only __interrupt is accepted.

See Interrupt Handler Parameters, Transferring Interrupt Control, and Special Considerations.

_chain_intr
_dos_getvect
_dos_keep
_dos_setvect

Interrupt Handler Parameters

On entry to an interrupt handler, the DS register is initialized to the near data segment, allowing your interrupt handler to access global variables.

In addition, all registers (except SS) are saved on the stack. You can access these registers within the function if you declare a function parameter list containing a formal parameter for each saved register. For example:

```
_void __interrupt __far int_handler( unsigned _es, unsigned _ds,
                                     unsigned _di, unsigned _si,
                                     unsigned _bp, unsigned _sp,
                                     unsigned _bx, unsigned _dx,
                                     unsigned _cx, unsigned _ax,
                                     unsigned _ip, unsigned _cs,
                                     unsigned flags )
{
. . .
}
```

You can omit parameters from the end of the list in your declaration, but you cannot omit parameters from the beginning of the list. For example, if your handler needs to use only DI and SI, you must still provide the ES and DS arguments, but not necessarily BX, DX, or those that follow.

The compiler always saves and restores registers in the same fixed order. Thus, no matter what names you use in the formal parameter list, the first parameter in the list refers to ES, the second refers to DS, and so on.

Do not give your parameters actual register names because they can conflict with the inline assembler. To prevent conflict and retain the documentation of the register names, use an underscore when naming your parameters (for example, _ax, _bx).

Passing Additional Arguments

You can pass additional arguments if your interrupt handler is to be called directly from C or C++ rather than by an INT instruction. To do this, you must declare all register parameters and then declare your parameter at the end of the list.

Changing Parameters

If you change any parameters of an interrupt handler during function execution, the corresponding register contains the changed value when the function returns. The code below causes DI to contain -1 when int_handler returns.

```
void __interrupt __far int_handler( unsigned _es, unsigned _ds,
                                    unsigned _di, unsigned _si)
{
   _di = -1;
}
```

Do not modify the values of IP or CS in an interrupt handler. If you need to modify a flag, use the bitwise-OR operator (|) so that the other flags are not changed.

Transferring Interrupt Control

An interrupt handler can transfer control to a second interrupt routine in either of two ways:

- Call the interrupt routine (after casting it to an interrupt handler if necessary) if you need to do further processing after the second interrupt routine finishes. For example:

```
void __interrupt __far new_int()
{
   // Initial processing here
   . . .
   (*old_int)();
   . . .
   // Final processing here
}
```

- Call _chain_intr with the interrupt routine as an argument. Do this if your routine is finished and you want the second interrupt routine to terminate the interrupt call. For example:

```
void __interrupt __far new_int()
{
   . . .
   // Initial processing here
   . . .
   // This is never executed
   _chain_intr( old_int );
}
```

An interrupt handler should avoid calling the standard library functions, especially functions that rely on either INT 21H calls or BIOS calls. Functions that rely on INT 21H calls include I/O functions and _dosxxx functions. Functions that rely on the BIOS include graphics functions and _biosxxx functions.

It may be safe to use functions that do not rely on INT 21H or BIOS, such as string-handling functions. Before calling a standard library function in an interrupt handler, be sure that you are familiar with the library function and what it does.

Special Considerations in Interrupt Handlers

When an interrupt handler is called by an INT instruction, the interrupt enable flag is cleared. This means that no further interrupts (including keyboard, time-of-day, and other crucial interrupts) are processed until your function returns. If your function needs to perform significant processing, you should use the enable function to set the interrupt flag so that other interrupts can be handled.

Interrupt handlers are special cases of C functions, since they are potentially reentrant. When designing an interrupt-handling function in C, consider the following guidelines:

- If your function does not use the enable function to set the interrupt flag, it is not reentrant: important interrupts may be ignored.
- If your function does use the enable function to set the interrupt flag, another interrupt may take place. Make sure that your handler takes this into account.

__segment

Keyword __segment

Syntax __segment *declarator*

More information on *declarator*

The __segment keyword is a data type representing a 16-bit segment address. For compatibility with previous versions, _segment is a synonym for __segment.

The __segment keyword can be used to provide a segment base for based pointers, which are limited to offset values. A based pointer is combined with the segment to produce a full address. You also can use __segment to specify where data is allocated.

You can declare variables of type __segment:

```
__segment myseg;
```

Pointers based on segment variables can be used for dynamic allocation of based objects. See the example program HEAPBASE.C for more information.

Segment variables can be explicitly combined with pointers declared as __based(void) by using the ":>" operator.

You also can use the __segment keyword in cast expressions. The segment portion of a pointer's value is extracted by the following cast:

(__segment)*ptr*

If *ptr* is a near pointer, the expression evaluates to the contents of the DS register. If *ptr* is a far pointer, the expression evaluates to the segment portion of *ptr*.

The segment in which a variable resides can also be found with a similar cast expression:

(__segment)&*var*

Example

```
char *chp;
int i;

// based pointer having same segment value as chp
char __based( (__segment)chp ) *b_p;

// based pointer using i's segment as its segment value
double __based( (__segment)&i ) *b_a;
```

__based
__self

__segname

Keyword __segname

Syntax __segname("*segment-name*")

The __segname specifies the name of a segment. For compatibility with previous versions, _segname is a synonym for __segname.

You can declare a based variable by giving it a segment constant as a base. Microsoft C/C++ predefines four segments:

Segment	Description
_CODE	Default code segment
_CONST	Constant segment for strings such as "This is a constant string"
_DATA	Default data segment
_STACK	Stack segment

The __segname keyword marks the name of a segment. It is always followed by parentheses and a string, as in the example below:

```
// Compile in small model, DOS only
#include <stdio.h>
#include <malloc.h>

char __based( __segname( "_CODE" ) ) mystring[] =
   "Code-based string.\n";

// Code-based integer:
int __based( __segname( "_CODE" ) ) ib = 12345;
void main()
{
   printf( "%Fs %d", (char __far *)mystring, ib );
}
```

The mystring variable is declared as an array of characters based in the code segment. The ib variable is an integer (not a pointer) that is also based in the code segment.

Note that the small model version of printf would treat mystring as a near pointer. The F in the format specifier %Fs forces the function to treat it as a far pointer, and the cast to (char __far*) coerces the address to four bytes.

You also can name your own segments. The declaration of *mystring* might look like this:

```
char __based( __segname( "MYSEGMENT" ) ) mystr[] = "Based string";
```

In the example above, the compiler creates a new segment called MYSEGMENT and places the string there.

__based
__segment
__self

static

Keyword static

Syntax static *declarator*

More information on *declarator*

When modifying a variable, the static keyword specifies that the variable has static duration (it is allocated when the program begins and deallocated when the program ends) and initializes it to 0 unless another value is specified. When modifying a variable or function at file scope, the static keyword specifies that the variable or function has internal linkage (its name is not visible from outside the file in which it is declared).

In C++, when modifying a data member in a class declaration, the static keyword specifies that one copy of the member is shared by all the instances of the class. When modifying a member function in a class declaration, the static keyword specifies that the function accesses only static members.

Example:

```
static int i;         // Variable accessible only from this file

static void func();   // Function accessible only from this file

int max_so_far( int curr )
{
   static int biggest;    // Variable whose value is retained
                          //    through multiple invocations of
                          //    the function

   if( curr > biggest )
      biggest = curr;

   return biggest;
}

// C++ only

class SavingsAccount
{
public:
   static void setInterest( float newValue )  // Member function
      { currentRate = newValue; }             //    that accesses
                                              //    only static
                                              //    members
private:
   char name[30];
   float total;
   static float currentRate;    // One copy of this member is
                                //    shared among all instances
                                //    of SavingsAccount
};

// Static data members must be initialized at file scope, even
//    if private.
float SavingsAccount::currentRate = 0.00154;
```

auto
extern
register

struct

Keyword struct

Syntax struct [*tag*] { *member-list* } [*declarators*];

[struct] *tag declarators*;

More information on *declarators*

The struct keyword defines a structure type and/or a variable of a structure type. See Anonymous Structures and Unsized Array in a Structure for more information.

A structure type is a user-defined composite type. It is composed of "fields" or "members" that can have different types.

In C++, a structure is the same as a class except that its members are public by default.

Declaring a Structure

Use the struct keyword with a *member-list* enclosed by braces to declare a structure type. The *tag* identifies the type.

For both C and C++, a data item in the *member-list* can be any valid data declaration, including another structure, or it can be a bit field in the following form:

int-type-specifier [*identifier*] : *constant-expression*

The *int-type-specifier* must be an integral type. The *constant-expression* specifies the number of bits in the field. Unnamed bit fields can be used for alignment. If an unnamed field has width 0, the next field is aligned on the current packing boundary as defined by /Zp or the pack pragma.

A structure type cannot contain itself as a member. It can contain a pointer to itself.

Using A Structure

In C, you must use the following syntax to declare a variable as a structure:

struct *tag declarator*;

You have the option of declaring variables when the structure type is defined by placing one or more comma-separated variable names between the closing brace and the semicolon.

In C++, the struct keyword is unnecessary once the type has been defined. This allows you to declare variables with this syntax:

tag declarator;

As with C, you have the option of declaring variables in the declaration of the struct. For example:

```
struct PERSON           // Declare PERSON struct type
{
   int   age;           // Members of several types
   long  ss;
   float weight;
   char  name[25];
} family_member;        // Define object of type PERSON

   struct PERSON sister;   // C declaration of a structure
                           //     variable
   PERSON brother;         // C++ declaration of a
                           //     structure
```

```
   sister.age  = 13;       // Assign values to members
   brother.age =  7;
```

Structure variables can be initialized. The initialization for each variable must be enclosed in braces. For example:

```
struct POINT            // Declare POINT structure
{
   int x;               // Define members x and y
   int y;
} here = { 20, 40 };    // Variable here has
                        //     values x = 20, y = 40
struct POINT there;     // Variable there has POINT type
struct CELL             // Declare CELL bit field
{
   unsigned character  : 8;  // 00000000 ????????
   unsigned foreground : 3;  // 00000??? 00000000
   unsigned intensity  : 1;  // 0000?000 00000000
   unsigned background : 3;  // 0???0000 00000000
   unsigned blink      : 1;  // ?0000000 00000000
} screen[25][80];       // Array of bit fields
```

class
union
enum
Types

Anonymous Structures

A Microsoft C extension allows you to declare a structure variable within another structure without giving it a name. These nested structures are called anonymous structures. C++ does not allow anonymous structures.

You can access the members of an anonymous structure as if they were members in the containing structure. For example:

```
struct phone
{
   int  areacode;
   long number;
};

struct person
{
   char   name[30];
   char   sex;
   int    age;
   int    weight;
   struct phone;    // Anonymous structure; no name needed
} Jim;

Jim.number = 1234567;
```

Unsized Arrays in Structures

A Microsoft extension allows the last member of a C or C++ structure or class to be a variable-sized array. These are called unsized arrays. The unsized array at the end of the structure allows you to append a variable-sized string or other array, thus avoiding the run-time execution cost of a pointer dereference.

For example, you can declare the following:

```
struct PERSON
{
   unsigned number;
   char     name[];    // Unsized array
};
```

If you apply the sizeof operator to this structure, the ending array size is considered to be 0. The size of this array is 2 bytes, which is the size of the unsigned member. To get the true size of a variable of type PERSON, you would need to obtain the array size separately.

The size of the structure is added to the size of the array to get the total size to be allocated. After allocation, the array is copied to the array member of the structure, as shown below:

```
struct PERSON *ptr;
char who[40];

printf( "Enter name: " );
gets( who );

// Allocate space for structure, name, and terminating null
ptr = malloc( sizeof( struct PERSON ) + strlen( who ) + 1 );

// Copy the string to the name member
strcpy( ptr->name, who );
```

Even after the structure variable's array is initialized, the sizeof operator returns the size of the variable without the array.

Structures with unsized arrays can be initialized, but arrays of such structures cannot be initialized. For example:

```
struct PERSON me  = { 6, "Me" };        // Legal
struct PERSON you = { 7, "You" };

struct PERSON us[2] = { { 8, "Them" },  // Error
                        { 9, "We" } };
```

An array of characters initialized with a string literal gets space for the terminating null; an array initialized with individual characters (for example, {'a', 'b', 'c'}) does not.

A structure with an unsized array can appear in other structures, as long as each is the last member declared in its enclosing structure. Classes or structures with unsized arrays cannot have direct or indirect virtual bases.

typedef

Keyword typedef

Syntax typedef *type-declaration synonym*;

The typedef keyword defines a synonym for the specified *type-declaration*. The identifier in the *type-declaration* becomes another name for the type, instead of naming an instance of the type.

Example:

```
typedef unsigned long ulong;

ulong ul;     // Equivalent to "unsigned long ul;"

typedef struct mystructtag
{
   int   i;
   float f;
   char  c;
} mystruct;

mystruct ms;   // Equivalent to "struct mystructtag ms;"

typedef int (*funcptr)();  // funcptr is synonym for "pointer
                           //    to function returning int"

funcptr table[10];   // Equivalent to "int (*table[10])();"
```

class
struct
enum
union

union

Keyword union

Syntax union [*tag*] {*member-list*} [*declarators*];

[union] *tag declarators*;

More information on *declarators*

The union keyword declares a union type and/or a variable of a union type. See Anonymous Unions for more information.

A union is a user-defined data type that can hold values of different types at different times. It is similar to a structure except that all of its members start at the same location in memory. A union variable can contain only one of its members at a time. The size of the union is at least the size of the largest member.

A C++ union is a limited form of the class type. It can contain access specifiers (public, protected, private), member data, and member functions, including constructors, and destructors. It cannot contain virtual functions or static data members. It cannot be used as a base class, nor can it have base classes. Default access of members in a union is public.

A C union type can contain only data members.

Declaring a Union

Begin the declaration of a union with the union keyword, and enclose the member list in curly braces:

```
union UNKNOWN    // Declare union type
{
   char   ch;
   int    i;
   long   l;
   float  f;
   double d;
} var1;          // Optional declaration of union variable
```

In C, you must use the union keyword to declare a union variable. In C++, the union keyword is unnecessary:

```
union UNKNOWN var2;   // C declaration of a union variable
UNKNOWN var3;         // C++ declaration of a union variable
```

Using a Union

A variable of a union type can hold one value of any type declared in the union. Use the dot operator (.) to access a member of a union:

```
var1.i = 6;           // Use variable as integer
var2.d = 5.327;       // Use variable as double
```

class
struct
Types

Anonymous Union

In C++, an anonymous union is a union without a tag or declarators; it declares an unnamed object. An anonymous union cannot have member functions or private or protected members.

A global anonymous union must be static. A local anonymous union must be either static or automatic, not external.

In C, an anonymous union can have a tag; it cannot have declarators.

The name of each member must be unique within the scope where the union is declared. You can directly access the members of an anonymous union as follows:

```
static union       // Global anonymous unions must be static
{
   int i;
   float f;
   union
   {
      char c;
      unsigned char uc;
   };
};

void my_func()
{
   i = 1;
   f = 3.14;
   c = 'c';
   uc = 'u';
};
```

voic

Keyword void

Syntax void *declarator*

More information on *declarator*

When used as a function return type, the void keyword specifies that the function does not return a value. When used for a function's parameter list, void specifies that the function takes no parameters. When used in the declaration of a pointer, void specifies that the pointer is a "universal" pointer.

If a pointer's type is void *, the pointer can point to any variable that is not declared with the const or volatile keywords. A void pointer cannot be dereferenced unless it is cast to another type. A void pointer can be converted into any other type of data pointer.

A void pointer can point to a function, but not to a class member in C++.

You cannot declare a variable of type void.

Example

```
void vobject;              // Error
void *pv;                  // Okay
int  *pint; int i;

void main()                // main has no return value
{
pv = &i;
pint = (int *)pv;     // Cast optional in C
                      //     required in C++
}
```

template

Keyword template

This product does not support templates**.**

try

Keyword try

This product does not support try**.**

Inheritance Keywords

Keywords __single_inheritance, __multiple_inheritance, __virtual_inheritance

Syntax class [__single_inheritance] *class-name*;

class [__multiple_inheritance] *class-name*;

class [__virtual_inheritance] *class-name*;

class-name
The name of the class being declared.

If you need to declare a pointer to a member of a class prior to defining the class, you can specify the inheritance used in the class declaration using the __single_inheritance, __multiple_inheritance, or __virtual_inheritance keywords described here, thus allowing control of the code generated on a per-class basis. Or you can use either the /vmg command-line option or the related pointers_to_members pragma.

Note If you always declare a pointer to a member of a class after defining the class, you dont need to use any of these options.

Declaring a pointer to a member of a class prior to the class definition impacts the size and speed of the resulting executable file. The number of bytes required to represent a pointer to a member of a class and the code required to interpret the representation may depend on whether the class is defined with no, single, multiple, or virtual inheritance.

In general, the more complex the inheritance used by a class, the greater the number of bytes required to represent a pointer to a member of the class and the larger the code required to interpret the pointer.

As shown in this example,

```
class __single_inheritance S;
int S::p;
```

regardless of command-line options or pragmas, pointers to members of class `S` will use the smallest possible representation.

Note The same forward declaration of a class pointer-to-member representation should occur in every translation unit that declares pointers to members of that class, and the declaration should occur before the pointers to members are declared.

class

Keyword class

Syntax class [*models*] [*tag* [**:** *base-list*]]
{
member-list
} [*declarators*]**;**
[*class*] *tag declarators***;**

The class keyword declares a class type or defines an object of a class type.

The elements of a class definition are as follows:

models
Specifies the default memory model for objects of the class type: __near, __far, or __huge.

tag
Names the class type. The tag becomes a reserved word within the scope of the class.

base-list
Specifies the class(es) from which the class is derived (its base classes). Each base class's name can be preceded by an access specifier (public, protected, private) and the virtual keyword. See virtual Access Specifiers for more information.

member-list
Declares members or friends of the class. Members can include data, functions, nested classes, enums, bit fields, and type names. Friends can include functions or classes. Explicit data initialization is not allowed. A class type cannot contain itself as a nonstatic member. It can contain a pointer or a reference to itself. See virtual Access Specifiers for more information.

declarators
Declares one or more objects of the class type.

struct
union
public
private
protected
__multiple_inheritance
__single_inheritance
__virtual_inheritance
Memory Models

virtual

Keyword virtual

Syntax virtual *member-function-declarator*
virtual [*access-specifier*] *base-class-name*

The virtual keyword declares a virtual function or a virtual base class.

See Virtual Function, Virtual Base Class, and Pure Virtual Functions and Abstract Base Classes for more information.

The elements of a virtual declaration are as follows:

member-function-declarator
Declares a member function.

member-function-declarator
Defines the level of access to the __base class. Can appear before or after the virtual keyword.

base-class-name
Identifies a previously declared class type.

class
private
public
protected

Virtual Function

A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

For example:

```
class WageEmployee
{
 public:
   virtual float computePay();
};

class SalesPerson : public WageEmployee
{
public:
   float computePay();
};
```

You can execute different versions of computePay( ) depending on the type of object you're calling it for:

```
WageEmployee aWorker;
SalesPerson aSeller;
WageEmployee *wagePtr;

wagePtr = &aWorker;
wagePtr->computePay();   // call WageEmployee::computePay
wagePtr = &aSeller;
wagePtr->computePay();   // call SalesPerson::computePay
```

The virtual keyword is needed only in the base class's declaration of the function; any subsequent declarations in derived classes are virtual by default.

A derived class's version of a virtual function must have the same parameter list and return type as those of the base class. If these are different, the function is not considered a redefinition of the virtual function. A redefined virtual function cannot differ from the original only by return type.

Pure Virtual Functions and Abstract Classes

An abstract class is a class that contains at least one pure virtual function. Specify a virtual function as pure by placing = 0 at the end of its declaration. You don't have to supply a definition for a pure virtual function.

You cannot declare an instance of an abstract base class; you can use it only as a base class when declaring other classes.

In the following program, draw( ) is a pure virtual function defined in the abstract class Shape. You cannot declare Shape objects. Shape acts as a base class for Rectangle and Circle. Rectangle and Circle provide definitions for draw( ), so you can declare instances of those classes and call draw( ) for them.

```
#include <iostream.h>

class Shape
{
public:
   virtual void draw() = 0;
};

class Rectangle: public Shape
{
public:
   void draw();
};

class Circle : public Shape
{
public:
   void draw();
};
```

Virtual Base Class

Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritance. Consider the following example:

```
class Employee
{
private:
   char name[30];
};

class SalesPerson : public Employee
{
};

class Manager : public Employee
{
};

class SalesManager : public SalesPerson, public Manager
{
};
```

In this hierarchy, the Employee class acts as an indirect base class to SalesManager twice. The following diagram describes a SalesManager object:

Each SalesManager object contains two copies of the data members defined in Employee. This duplication wastes space and requires you to specify which copy of Employee's members you want whenever you access them.

The use of the virtual keyword eliminates these problems. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use it as a virtual base. For example:

```
class Employee
{
private:
   char name[30];
};

class SalesPerson : virtual public Employee
{
};

class Manager : virtual public Employee
{
};

class SalesManager : public SalesPerson, public Manager
{
};
```

A SalesManager object can now be described by the following diagram:

The SalesPerson and Manager base classes share the same copy of Employee's data members. As a result, each SalesManager object contains just one copy of Employee's data members, and references to Employee's members are unambiguous. Notice that the virtual keyword appears in the base lists of the SalesPerson and Manager classes, not SalesManager.

new

Keyword new

Syntax [**::**] new [*model*] [*placement*] *type-name* [*initializer*]

[**::**] new [*model*] [*placement*] (*type-name*) [*initializer*]

The new keyword allocates memory for an object of *type-name* from the free store and returns a suitably typed, nonzero pointer to an object. If unsuccessful, new returns zero.

Use the delete operator to deallocate the memory allocated with the new operator.

The following describes the elements of new:

placement
Provides a way of passing additional arguments if you overload new. See Customizing the new Operator for more information.

type-name
Specifies type to be allocated. If the type specification is complicated, it can be surrounded by parentheses to force the order of binding.

model
Specifies the addressing mode of the returned pointer. Can be __near, __far, __huge, or __based(*segment-expression*). _When you specify one of these pointer types to new, you must use the same version of the delete operator.

initializer
Provides a value for the initialized object. Initializers cannot be specified for arrays. The new operator will create arrays of objects only if the class has a default constructor.

This statement allocates an integer.

```
int *pi = new int;
```

This statement allocates a character and initializes it.

```
char *pc = new char( 'a' );
```

This statement allocates an instance of Date using the class's three-argument constructor, and returns a far pointer to it.

```
Date *pmc = new __far Date( 3, 12, 1985 );
```

This statement allocates an array of characters.

```
char *pstr = new char[sizeof( str )];
```

This statement allocates a two-dimensional array of characters of size dim * 10. When allocating a multi-dimensional array, all dimensions except the first must be constant expressions.

```
char (*pchar)[10] = new char[dim][10];
```

This statement allocates an array of seven pointers to functions that return integers.

```
int (**p) () = new (int (*[7]) ());
```

delete
_set_new_handler
__near
__far
__huge
__based
Memory Models
Customizing the new Operator

Customizing the new Operator

You can write your own version of the new operator to implement a custom memory-allocation scheme. You must use one of the following prototypes:

```
void __near *operator new( size_t size );
void __far  *operator new( size_t size );
void __huge *operator new( unsigned long elems, size_t size );
void __based(void) *operator new( __segment seg, size_t size );
```

A new operator that returns a near or a far pointer has a parameter of type size_t, which is the amount of space to be allocated. The compiler sets the value of this parameter when the new operator is called.

A new operator that returns a huge pointer acts like an array allocator and has two parameters. The first parameter is the number of elements to allocate, and the second parameter is the size of one element. If the total size of the array is greater than 128K, the size of each element must be a power of 2.

A new operator that returns a based pointer has two parameters. The first is a segment variable, which is the segment specified using the __based keyword in the allocation expression. The second parameter is the amount of space to be allocated.

Passing Additional Arguments

You can pass additional arguments to your customized new operator; these are called "placement" arguments. These arguments appear last when you declare the argument list of the new operator's prototype, but they appear before the type name and in parentheses when you call new. For example, if you want a near new operator that takes an integer placement argument, use the following prototype:

```
void __near *operator new( size_t size, int parm );
```

To call this operator and specify a value for the placement argument, you can use the following syntax:

```
T __near *pT = new __near (25) T;
```

The placement argument receives the value 25.

delete

Keyword delete

Syntax [**::**] delete *pointer*

[**::**] delete [ ] *pointer*

The delete keyword deallocates a block of memory. The argument *pointer* must point to a block of memory previously allocated by the new operator. If *pointer* points to an array, place empty brackets before *pointer*.

The run-time library provides four versions of the delete operator to deallocate pointers that are near, far, huge, or based. The version that is selected depends on the addressing mode of *pointer* (for example, if it's a near pointer, the near delete operator is called).

If the addressing mode of *pointer* doesn't reflect the version of the new operator that was used to allocate the memory, the incorrect version of the delete operator is called. For example:

```
Node __far *fpN;
fpN = new Node __near;  // convert near to far
delete fpn;             // far delete invoked for near object
```

Here, the compiler chooses the inappropriate delete operator for the pointer, which results in a run-time error. To prevent this problem, explicitly cast the pointer to the desired addressing mode:

```
delete (Node __near *)fpN;
```

Customizing the delete Operator

You can write your own version of the delete operator to implement a customized memory-allocation scheme. The delete operator you define must use one of the following prototypes:

```
void operator delete( void __near *ptr );
void operator delete( void __far *ptr );
void operator delete( void __huge *ptr );
void operator delete( __segment seg, void __based(void) *ptr );
```

You can also define class-specific versions of any of these forms of delete. When defining a class-specific version, you can specify an optional argument of type size_t. If present, the argument is automatically set to the size of the object being deleted. You cannot specify this argument when redefining the global version of delete.

this

Keyword this

Syntax this

The this pointer is a pointer accessible only within the member functions of a class, struct, or union type. It points to the object for which the member function is called. Static member functions do not have a this pointer.

When a nonstatic member function is called for an object, the address of the object is passed as a hidden argument to the function. For example, the following function call

```
myDate.setMonth( 3 );
```

can be interpreted this way:

```
setMonth( &myDate, 3 );
```

The object's address is available from within the member function as the this pointer. It is legal, though unnecessary, to use the this pointer when referring to members of the class. For example:

```
void Date::setMonth( int mn )
{
   month = mn;             // These three statements
   this->month = mn;       //     are equivalent
   (*this).month = mn;
}
```

The expression (*this) is commonly used to return the current object from a member function.

Note Modifying the this pointer is illegal in the latest version of C++.

friend

Keyword friend

Syntax friend *class-name*;

friend *function-declarator*;

The friend keyword allows a function or class to gain access to the private and protected members a class. See Friend Classes and Friend Functions for more information.

class
public
private
protected

Friend Functions

A friend function is a function that is not a member of a class but has access to the class's private and protected members.

A friend function is declared by the class that is granting access. For example:

```
class Complex
{
public:
   Complex( float re, float im );
   friend Complex operator+( Complex first, Complex second );
private:
   float real, imag;
};
```

The friend declaration can be placed anywhere in the class declaration. It is not affected by the access control keywords.

A friend function is defined as a nonmember function. For example:

```
Complex operator+( Complex first, Complex second )
{
   return Complex( first.real + second.real,
                   first.imag + second.imag );
}
```

In this example, the friend function operator+ has access to the private data members of the Complex objects it receives as parameters.

Notice that the friend keyword does not appear in the function definition.

Friend Classes

A friend class is a class all of whose member functions are friend functions of a class, i.e., whose member functions have access to the other class's private and protected members.

For example:

```
class YourClass
{
friend class YourOtherClass;  // Declare a friend class
private:
   int topSecret;
};

class YourOtherClass
{
public:
   void change( YourClass yc );
};

void YourOtherClass::change( YourClass yc )
{
   yc.topSecret++;    // Can access private data
}
```

Friendship is not mutual unless explicitly specified as such. In the above example, member functions of YourClass cannot access the private members of YourOtherClass.

Friendship is not inherited, meaning that classes derived from YourOtherClass cannot access YourClass's private members; nor is it transitive, so classes that are friends of YourOtherClass cannot access YourClass's private members.

inline, __inline

Keyword inline, __inline

Syntax inline *function_declarator*;

The inline and __inline keywords allow the compiler to insert a copy of the function body in each place the function is called. This substitution occurs at the discretion of the compiler. The inline keyword is available only in C++. The __inline keyword is available in both C and C++. For compatibility with previous versions, _inline is a synonym for __inline.

Using inline functions can make your program faster because they eliminate the overhead associated with function calls. Functions expanded inline are subject to code optimizations not available to normal functions. For example:

```
inline int max( int a , int b )
{
   if( a > b ) return a;
   return b;
}
```

A class's member functions can be declared inline either by using the inline keyword or by placing the function definition within the class definition. For example:

```
class MyClass
{
public:
   void print() { cout << i << ''; }  // Implicitly inline
private:
   int i;
};
```

operator

Keyword operator

Syntax type operator *operator-symbol* (*parameter-list*)

The operator keyword declares a function specifying what *operator-symbol* means when applied to instances of a class. This gives the operator more than one meaning, or "overloads" it. The compiler distinguishes between the different meanings of an operator by examining the types of its operands.

Rules of Operator Overloading

- You can overload the following operators:

+	**-**	*****	**/**	**%**	**^**
!	**=**	**<**	**>**	**+=**	**-=**
^=	**&=**	**|=**	**<<**	**>>**	**<<=**
<=	**>=**	**&&**	**||**	**++**	**--**
( )	[ ]	new	delete	**&**	**|**
~	***=**	**/=**	**%=**	**>>=**	**==**
!=	,	**->**	**->***		

- If an operator can be used as either a unary or a binary operator, you can overload each use separately.
- You can overload an operator using either a nonstatic member function or a global function that's a friend of a class. A global function must have at least one parameter that is of class type or a reference to class type.
- If a unary operator is overloaded using a member function, it takes no arguments. If it is overloaded using a global function, it takes one argument.
- If a binary operator is overloaded using a member function, it takes one argument. If it is overloaded using a global function, it takes two arguments.

Restrictions on Operator Overloading

- You cannot define new operators, such as **.
- You cannot change the precedence or grouping of an operator, nor can you change the numbers of operands it accepts.
- You cannot redefine the meaning of an operator when applied to built-in data types.
- Overloaded operators cannot take default arguments.
- You cannot overload any preprocessor symbol, nor can you overload the following operators:

.	**.***	**::**	**?:**	**:>**

- The assignment operator has some additional restrictions. It can be overloaded only as a nonstatic member function, not as a friend function. It is the only operator that cannot be inherited; a derived class cannot use a base class's assignment operator.

Example

The following stack class overloads the + operator to add two complex numbers and return the result.

```
class Complex
{
public:
   Complex( float re, float im );
   Complex operator+( Complex &other );
   friend Complex operator+( int first, Complex &second );
private:
   float real, imag;
};
```

```
// Operator overloaded using a member function Complex Complex::operator+( Complex
&other ) { return Complex( real + other.re, imag + other.im ); };

// Operator overloaded using a friend function
Complex operator+( int first, Complex &second )
{
return Complex( first + second.re, second.im );
}
```

public

Keyword public

Syntax public**:** [*member-list*]

public *base-class*

When preceding a list of class members**,** the public keyword specifies that those members are accessible from any function**.** This applies to all members declared up to the next access specifier or the end of the class**.**

When preceding the name of a base class**,** the public keyword specifies that the public and protected members of the base class are public and protected members**,** respectively**,** of the derived class**.**

Default access of members in a class is private**.** Default access of members in a structure or union is public**.**

Default access of a base class is private for classes and public for structures**.** Unions cannot have base classes**.**

Example

```
class BaseClass
{
public:
   int pubFunc();
};

class DerivedClass : public BaseClass
{
};

void main()
{
   BaseClass aBase;
   DerivedClass aDerived;

   aBase.pubFunc();        // pubFunc() is accessible
                           //    from any function
   aDerived.pubFunc();     // pubFunc() is still public in
                           //    derived class
}
```

private
protected
friend
Table of Access Privileges

private

Keyword private

Syntax private**:** [*member-list*]
private *base-class*

When preceding a list of class members**,** the private keyword specifies that those members are accessible only from member functions and friends of the class**.** This applies to all members declared up to the next access specifier or the end of the class**.**

When preceding the name of a base class**,** the private keyword specifies that the public and protected members of the base class are private members of the derived class**.**

Default access of members in a class is private**.** Default access of members in a structure or union is public**.**

Default access of a base class is private for classes and public for structures**.** Unions cannot have base classes**.**

Example

```
class BaseClass
{ public:
   // privMem accessible from member function
   int pubFunc() { return privMem; }
private:
   void privMem;
};

class DerivedClass : public BaseClass
{
public:
   void usePrivate( int i )
      { privMem = i; }   // Error: privMem not accessible
                         //    from derived class
};

class DerivedClass2 : private BaseClass
{
public:
   // pubFunc() accessible from derived class
   int usePublic() { return pubFunc(); }
};

void main()
{
   BaseClass aBase;
   DerivedClass aDerived;
   DerivedClass2 aDerived2;

   aBase.privMem = 1;      // Error: privMem not accessible
   aDerived.privMem = 1;   // Error: privMem not accessible
                           //    in derived class
   aDerived2.pubFunc();    // Error: pubFunc() is private in
                           //    derived class
}
```

protected

Keyword protected

Syntax protected: [*member-list*]

proctected *base-class*

When preceding a list of class members, the protected keyword specifies that those members are accessible only from member functions and friends of the class and its derived classes. This applies to all members declared up to the next access specifier or the end of the class.

When preceding the name of a base class, the protected keyword specifies that the public and protected members of the base class are protected members of the derived class.

Default access of members in a class is private. Default access of members in a structure or union is public.

Default access of a base class is private for classes and public for structures. Unions cannot have base classes.

Example

```
class BaseClass
{
protected:
   int protectFunc();
};

class DerivedClass : public BaseClass
{
public:
   int useProtect()
      { protectFunc(); }    // protectFunc accessible
                            //    from derived class };

void main()
{
BaseClass aBase;
DerivedClass aDerived;

   aBase.protectFunc();     // Error: protectFunc not
                            //    accessible
   aDerived.protectFunc();  // Error: protectFunc not
                            //    accessible in derived class
}
```

public
private
friend
Table of Access Privileges

Table of Access Privileges

Access in Base Class	Base Class Inherited as	Access in Derived Class
Public	Public	Public
Protected		Protected
Private		No access*
Public	Protected	Protected
Protected		Protected
Private		No access*
Public	Private	Private
Protected		Private
Private		No access*

* Unless friend declarations within the base class explicitly grant access

__asm

Keyword __asm

Syntax __asm *assembly language instruction*

```
__asm
   {
      assembly language instructions
   }
```

If used without braces, the __asm keyword means that the rest of the line is an assembly-language statement. If used with braces, it means that each line between the braces is an assembly-language statement. For compatibility with previous versions, _asm is a synonym for __asm.

Note Microsoft C++ does not support the AT&T C++ asm keyword.

The language recognized by the inline assembler is a subset of that recognized by the Microsoft Macro Assembler (MASM). The default instruction set is the 8086/8087, or the 80286/80287 if the /G2 option is given.

The following MASM concepts are not recognized by the inline assembler:

- Data directives and operators (DB, DW, DUP, RECORD, STRUCT, etc.)
- Macros, equates, and related directives and operators
- Segment directives and names
- Calls and jumps to far labels

An __asm block can use the following C-language elements:

- Symbols, including labels, variables, and function names
- Constants, including symbolic constants and enum members
- Macros and preprocessor directives
- Type or typedef names wherever a MASM type is legal
- C comments (starting with // or enclosed in /* */)
- C constants (0xff is the same as 0FFh)

__asm Keyword Statement Format

Since the __asm keyword is a statement separator, you can put assembly instructions on the same line. The following statements are equivalent:

```
__asm                   // __asm block
{
   mov ax, 01h
   int 10h
}

__asm mov ax, 01h       // Separate __asm lines
__asm int 10h

// Multiple __asm statements on a line
__asm mov ax, 01h   __asm int 10h
```

Assembly Language Operators and Directives

The following conditions apply to assembly language operators:

- Segment overrides must use a segment register (es:[bx]).
- The SEG and OFFSET operators can be used with C variable names (SEG i or OFFSET i).
- The LENGTH, SIZE, and TYPE operators can be used with C arrays.
- Indexes within brackets ([ ]) are unscaled.
- The dollar symbol ($) can be used as the current location counter.

The inline assembler recognizes only the EVEN and ALIGN MASM directives.

Using C Symbols with the Inline Assembler

An __asm block can refer to any C symbol in the scope where the block appears. This includes arguments, functions, and local, static local, and global variables. Each assembly-language statement can contain only one C symbol (except in LENGTH, TYPE, and SIZE expressions). Functions referenced in an __asm block must be declared (prototyped) earlier in the program.

If a structure or union member has a unique name, an __asm block can refer to it by using only the member name (mov ax, [bx].unique). If the member name is not unique, you must place a variable or typedef name before the period operator (mov ax, [bx]var.copy).

You can define functions in C and implement them with the inline assembler, as shown below:

```
int power2( int num, int power )
{
   __asm
   {
      __mov ax, num    ; Get first argument
      __mov cx, power  ; Get second argument
      __shl ax, cl     ; AX = AX * ( 2 to the power of CL )
   }
}  // Return with result in AX
```

This assumes the __cdecl or __pascal calling convention. To avoid register conflicts, you should not use the __fastcall calling convention for functions with __asm blocks.

Preserving Registers with the Inline Assembler

An __asm block inherits whatever register values happen to result from the normal flow of control. Within a function, you do not need to preserve the AX, BX, CX, DX, or ES registers. You should preserve DI, SI, DS, SS, SP, and BP.

If your function changes the direction flag using the STD or CLD instructions, you should restore the flag to its original value.

If the return value is a char, int, or near pointer, use the AX register. If the return value is a long or a far pointer, place the high word in DX and the low word in AX. To return a value longer than that, store the value in memory and return a pointer to it.

Defining __asm Blocks as C Macros

To use macros to insert assembly code into C or C++ code, follow these rules:

1. Enclose the __asm block in braces.
2. Put the __asm keyword in front of each assembly instruction.
3. Use enclosing C comments (/* comment */); if you use C++ single-line comments (//), the compiler ignores the rest of the macro.
4. Use the backslash (\) to join the statements into a single line. For example:

```
#define BEEP __asm       \
/* Beep sound */         \
{                        \
   __asm mov ah, 2       \
   __asm mov dl, 7       \
   __asm int 21h         \
}
```

This macro expands to a single line:

```
__asm { __asm mov ah, 2  __asm mov dl, 7 __asm int 21h }
```

The Inline Assembler and Compiler Options

The presence of an __asm block in a function disables automatic register variable storage**.** The presence of inline assembly code in a function inhibits the following optimizations for the entire function**:**

- Removal of invariant code from loops (/Ol)
- Global register allocation (/Oe)
- Global optimizations and common subexpressions (/Og)

__based

Keyword __based

Syntax type __based(*base*) *declarator*

More information on *declarator*

The __based keyword specifies that a pointer is a 16-bit value interpreted as an offset from *base*; or that a data object resides in the segment given by *base* ; or that a function resides in the segment given by *base*. For compatibility with previous versions, _based is a synonym for __based.

The __based keyword supersedes the alloc_text pragma for specifying the location of a function. The alloc_text pragma is still supported for compatibility reasons.

Based pointers can address any location in memory but are only two bytes in size because they contain only the offset portion of an address. The segment portion of the address is stored separately and is combined with the offset when needed. Multiple based pointers can share the same segment value, so they require less memory than far pointers. See Pointers Based on a Fixed Segment, Pointers Based on a Nonfixed Segment, Pointers Based on the __self Keyword, and Pointers Based on Void for more information.

You also can use the __based keyword to specify the segment in which data or functions reside.

See Data Based in a Segment and Functions Based in a Segment for more information.

_bfreeseg
_bheapseg
_bmalloc
__far
__huge
__near
__segment
__segname
__self
void
Memory models

Pointers Based on a Fixed Segment

Pointers based on a fixed segment are restricted to accessing one segment of memory. By making assignments to the based pointer, you change only the offset portion of the address. For compatibility with previous versions, _segname is a synonym for __segname.

Pointers Based on a Named Segment

Form of *base*:

__segname(*string-literal*)

The *string-literal* can be the name of one of four predefined segments ("_CODE", "_CONST", "_DATA" or "_STACK"), or it can be the name of a new segment you define. A based pointer declared this way can address locations only in the specified segment. For example:

```
// pointer into current code segment
char __based(__segname("_CODE")) *bp;
```

Pointers Based on the Segment of a Variable

Form of *base*:

(__segment)&*var*

A based pointer declared this way uses the segment of the address of *var* as its base. It can address locations only in the same segment as *var*.

For example:

```
int i;
// pointer using i's segment as its base
char __based((__segment)&i) *bp;
```

Pointers Based on a Nonfixed Segment

Pointers based on a nonfixed segment have access to locations in any segment simply by changing the value of the base. Changing a single segment value causes all pointers based on that segment to address new locations. You can also make assignments to the based pointers themselves to change their offset values. For compatibility with previous versions, _segment is a synonym for __segment.

Pointers Based on the Segment of Another Pointer

Form of *base*:

(__segment)*ptr*

A based pointer declared this way uses the segment portion of *ptr* as its base. If *ptr* is a near pointer, the declared pointer uses the DS register as its base. If *ptr* is a far pointer, the declared pointer uses the segment value of *ptr* as its base. Changing the segment value of *ptr* causes the based pointer to address a new location. For example:

```
char __far *far_ptr;
char __based((__segment)far_ptr) *bp;  // *bp takes the same
                                       // segment as far_ptr
```

Pointers Based on a Segment Variable

Form of *base*:

segvar

A based pointer declared this way uses *segvar* as its base. Assigning a new value to *segvar* causes the based pointer to address a different location. This type of based pointer can be used for dynamic allocation of based objects. For example:

```
__segment seg1;
char __based(seg1) *bp;  // pointer using seg1 as its base
```

See the Dynamic Allocation Example for more information.

Pointers Based on Another Pointer

Form of *base*:

ptr

A based pointer declared this way acts as an offset from *ptr*. Assigning a new value to *ptr* causes the based pointer to address a different location. The 32-bit compiler recognizes this form of based addressing. For example*:

```
char *cp;
char __based(cp) *bp;    // pointer using ip as its base

cp = (char *)malloc(100);
bp = 2; *bp = 'a';               // equivalent to cp[2] = 'a';
```

Pointers Based on the __self Keyword

Form of *base*:

(__segment)__self

A based pointer declared this way can address locations in only the segment in which the pointer itself is stored. Such pointers can save space in a linked list or tree if the entire data structure fits in a single segment. For example:

```
typedef struct tree TREE;

struct tree
{
int name;
TREE __based( (__segment)__self ) *left;
TREE __based( (__segment)__self ) *right;
};
```

For compatibility with previous versions, _segname is a synonym for __segname; _self is a synonym for __self.

Pointers Based on void

Form of *base*:

void

A based pointer declared this way acts as an offset into any segment. It can be combined with any segment value using the :> operator. For example:

```
__segment segvar = 0xB800;   // specify a segment value

int __based(void) *bp = 4;   // offset of 4

// set contents of location
0xB804 *(segvar:>bp) = (0x07 << 8) + 'A';
```

Data Based in a Segment

Data declared with __based is allocated in the segment specified.

Data Stored in a Named Segment

Form of *base*:

__segname(*string literal*)

The *string literal* can be the name of a predefined segment ("_CODE", "_CONST", or "_DATA", but not "_STACK"), or it can be the name of a new segment you define. For example:

```
// string stored in current code segment
char __based(__segname("_CODE")) cs[] = "code-based string";
```

For compatibility with previous versions, _segname is a synonym for __segname.

External Data Based on a Segment Variable

Form of *base*:

segvar

Data declared this way resides in a location determined at run time. You can relocate a segment in memory, set *segvar* to the new location of that segment, and access a variable stored in that segment without using pointers. Because no space is reserved for the variable at compile time, the variable must be declared as extern. For example:

```
__segment seg1;
extern char __based(seg1) c;
```

Data Based on the Address of Another Variable

Form of *base*:

&*var*

The *var* specified must be based on a named segment. This declaration places both variables in the same segment. For example:

```
// int stored in segment MYSEGMENT
int __based(__segname("MYSEGMENT")) myvar1;

int __based( (__segment)&myvar1 ) myvar2;
```

Functions Based in a Segment

Form of *base*:

__segname(*string literal*)

A function declared as __based resides in the code segment named by *string literal*. You can use the __near or __far keywords with the __based keyword when declaring a function. In programs that use overlays, you can reduce swapping by using __based to group together functions that frequently call one another. You also can use __based to ensure that near functions reside in the same segment as the functions that call them. For example:

```
// FILE 1 - compiled under large model
void __based(__segname("MYSEG")) farfunc()  // far by default
{
   nearfunc();
}

// FILE 2 - compiled under large model
void __near __based(__segname("MYSEG")) nearfunc()
{
   // ...
}
```

If these two functions were not declared as based, this program would suffer a runtime error. Since both functions are based in the MYSEG segment, the program executes correctly.

For compatibility with previous versions, _segname is a synonym for __segname.

Using the Commit-to-Disk Feature with Existing Code

By default, calls to the fflush or _flushall library functions write data to buffers maintained by the operating system; the operating system determines the optimal time to actually write the data to disk. The commit-to-disk feature of the run-time library lets you ensure that critical data is written directly to disk rather than to the operating system's buffers. You can give this capability to an existing program without rewriting it by linking its object files with COMMODE.OBJ. COMMODE.OBJ is provided in the compiler's LIB directory. For example:

```
link proj.obj commode.obj /noe,,,slibcer.lib;
```

In the resulting executable file, calls to the fflush function write the contents of the buffer directly to disk, and calls to the _flushall function write the contents of all buffers to disk. These two functions are the only ones affected by COMMODE.OBJ.

fflush
_flushall
_fdopen
fopen
_commit
_dos_commit

native_caller

Pragma native_caller

Syntax #pragma native_caller([{ on | off }])

The native_caller pragma controls the removal of native-code entry points from within source code**.**

If you have p-code functions that are called only by other p-code functions**,** you can omit those entry points and save those bytes by using the /Gn compiler option or on a function-by-function basis with this pragma**.**

/Gn

alloc_text, same_seg

Pragma alloc_text, same_seg

Syntax #pragma alloc_text(*textsegment*, *function1*, ...)

#pragma same_seg(*variable1*, ...)

The alloc_text pragma specifies the name of the segment where the specified function definitions are to reside.

The alloc_text pragma does not handle C++ member functions or overloaded functions. Use of the __based keyword is preferred when it is necessary to specify the segment in which a function resides. The alloc_text pragma is supported for compatibility with previous versions of Microsoft C.

Note The same_seg pragma is no longer supported. In previous versions of Microsoft C, the same_seg pragma instructed the compiler to assume that the specified external variables were allocated in the same data segment. Use of the __based keyword is preferred when it is necessary to specify the segment in which external variables reside.

The alloc_text pragma gives you source-level control over the segment in which particular functions are allocated.

If you use overlays or swapping techniques to handle large programs, alloc_text allows you to tune the contents of their text segments for maximum efficiency. The alloc_text pragma must appear after the declarations of any of the specified functions and before the definitions of these functions.

Functions referenced in an alloc_text pragma should be defined in the same module as the pragma. If this is not done and an undefined function is later compiled into a different text segment, the error may or may not be caught. Although the program will usually run correctly, the function will not be allocated in the intended segments.

Other limitations on alloc_text are as follows:

- It cannot be used inside a function.
- It must be used after the function has been declared, but before the function has been defined.

Functions given in an alloc_text pragma can be implicitly near (because the small or compact memory model is used) or explicitly near (declared with the near keyword), provided they are called only by functions in the same segment.

The following example shows code that may result in these problems:

```
// Module 1
#pragma  alloc_text( SEG1, sample, example )
int sample( int i )
{
   i = example( 10 );
}


// Module 2
int example( int i )
{
   return i * i;
}
```

You can prevent problems in the preceding code by placing the definitions of the functions sample and example in the same module.

__based
check_stack

comment

Pragma comment

Syntax #pragma comment(*comment-type* [, *commentstring*])

The comment pragma allows you to place a comment record in an object file or executable file.

The *comment-type* is one of four predefined identifiers, described below, that specify the type of comment record. The optional *commentstring* is a string literal that provides additional information for some comment types. Because *commentstring* is a string literal, it obeys all the rules for string literals with respect to escape characters, embedded quotation marks ("), and concatenation.

The following comment records are allowed:

compiler
: Places the name and version number of the compiler in the object file. This comment record is ignored by the linker. If you supply a *commentstring* parameter for this record type, the compiler generates a warning error message.

exestr
: Places *commentstring* in the object file. At link time, this string is placed in the executable file. The string is not loaded into memory when the executable file is loaded; however, it can be found with a program that finds printable strings in files. One use for this comment-record type is to embed a version number or similar information in an executable file.

lib
: Places a library-search record in the object file. This comment type must be accompanied by a *commentstring* parameter containing the name (and possibly the path) of the library for which you want the linker to search. Since the library name precedes the default library-search records in the object file, the linker searches for this library just as if you had named it on the command line. You can place multiple library-search records in the same source file; each record appears in the object file in the same order in which it is encountered in the source file.

user
: Places a general comment in the object file. The *commentstring* parameter contains the text of the comment. This comment record is ignored by the linker.

The following pragma causes the linker to search for the EMAPI.LIB library. The linker searches first in the current working directory and then in the path specified in the LIB environment variable.

```
#pragma comment( lib, "emapi" )
```

The following pragma causes the compiler to place the name and version number of the compiler in the object file:

```
#pragma comment( compiler )
```

Note For comments that take a *commentstring* parameter, you can use a macro in any place where you would use a string literal, provided that the macro expands to a string literal. You can also concatenate any combination of string literals and macros that expand to string literals. For example, the following statement is acceptable:

```
#pragma comment( user, "Compiled on " __DATE__ " at " __TIME__ )
```

data_seg

Pragma data_seg

Syntax #pragma data_seg (["*seg_name*" [, " *seg_class*"])

The data_seg pragma specifies a segment where data is to be allocated, allowing the use of based allocation without rewriting code. Using #pragma data_seg is equivalent to using __based when __based is used for allocation. The *seg_class* parameter allows you to assign a segment class to a segment name. This pragma applies only to initialized data and does not affect tentative definitions. Using #pragma data_seg without a *seg_name* string resets allocation to whatever it was when compilation began.

Note The data_seg pragma is not equivalent to the data_seg pragma supported by earlier versions of the compiler.

__based
__loadds
code_seg
Naming Segments

code_seg

Pragma code_seg

Syntax #pragma code_seg (["*seg_name*" [, "*seg_class*"])

The code_seg pragma specifies a segment where functions are to be allocated, allowing the use of based allocation without rewriting code. Using #pragma code_seg is equivalent to using __based when __based is used for allocation. You can specify the class for the segment by giving the *seg_class* as a string. Using #pragma code_seg without a *seg_name* string resets allocation to whatever it was when compilation began.

data_seg

warning

Pragma warning

Syntax #pragma warning(*warning-specifier*[**;** *warning-specifier*])

The warning pragma controls how the compiler treats the specified warning messages**.** The *warning-specifier* has the syntax *warning-type*:*warning-number-list***.** The possible values for *warning-type* are as follows**:**

Option	Meaning
once	Display the specified warning[s] only once**.**
1**,** 2**,** 3**,** 4	Force the specified warning[s] to the specified warning level**.**
default	Use the standard warning level for the warning[s]**.**
disable	Disable the specified warning[s]**.**
error	Report the warning[s] as error[s]**.**

The *warning-number-list* can contain any warning numbers**.** Multiple options can be specified in the same pragma directive as follows**:**

```
#pragma warning( disable : 4507 34; once : 4385; error : 164 )
```

This is functionally equivalent to**:**

```
#pragma warning ( disable : 4507 34 )  // Disable warning messages
                                           4507 and 34.
#pragma warning ( once : 4385 )        // Issue warning 4385
                                           only once.
#pragma warning ( error : 164 )        // Report warning 164
                                           as an error.
```

/W

init_seg

Pragma init_seg

Syntax #pragma init_seg({ compiler | lib | user | "*seg-name*" })

The init_seg pragma specifies a keyword or segment that affects the order in which startup code is executed. Because initialization of global static objects can involve executing code, you must specify a keyword that defines when the objects are to be constructed. It is particularly important to use the init_seg pragma in DLLs or libraries requiring initialization.

The options to the init_seg pragma are:

compiler
: Reserved for Microsoft C run-time library initialization. Objects in this group are constructed first.

lib
: Available for third-party class-library vendors' initializations. Objects in this group are constructed after those marked as compiler but before any others.

user
: Available to any user. Objects in this group are constructed last.

seg-name
: Allows explicit specification of the initialization segment. Objects in a user-specified *seg-name* are not implicitly constructed; however, their addresses are placed in the segment named by *seg-name*.

If you need to defer initialization (for example, in a DLL), you may choose to specify the segment name explicitly. You must then call the constructors for each static object. For an example of how these initializations are done, see the file CRT0DAT.ASM in the STARTUPdirectory.

linesize

Pragma linesize

Syntax #pragma linesize([*characters*])

The linesize pragma specifies the number of characters per line in a source listing (created with the /Fs option).

The optional parameter *characters* is an integer constant in the range 79-132 that specifies the number of characters that you want each line of the source listing to have. If *characters* is absent, the compiler uses the value specified in the /Sl option or, if that option is absent, the default value of 79 characters per line. Note that linesize takes effect in the line after that in which the pragma itself appears.

The following pragma causes the source listing to have 132 characters per line:

```
#pragma linesize( 132 )
```

page
pagesize
skip
title
subtitle

message

Pragma message

Syntax #pragma message(*messagestring*)

The message pragma sends a message to the standard output without terminating the compilation.

The *messagestring* parameter is a string literal containing the message that you want to send to the standard output device.

This pragma does not cause termination of the compilation. A typical use of the message pragma is to display informational messages at compile time.

The following code fragment uses the message pragma to display a message during compilation:

```
#if M_I86MM #pragma message( "Medium memory model" ) #endif
```

The *messagestring* parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination. For example, the following statements display the name of the file being compiled and the date and time when the file was last modified:

```
#pragma message( "Compiling " __FILE__ )
#pragma message( "Last modified on " __TIMESTAMP__ )
```

optimize

Pragma optimize

Syntax #pragma optimize("[*optimization-switch-list*]", {off | on})

The optimize pragma specifies optimizations to be performed. It must appear outside of a function.

The *optimization-switch-list* may be zero or more of the following: a, w, t, s, c, g, e, l, p, z, n, r, o, q, f, and v. These are the same letters used with the /O compiler options. For example,

```
#pragma optimize( "lge", off )
```

Two special forms are supported. The form below turns off all optimization:

```
#pragma optimize( "", off )
```

The form below restores all optimization switch settings to their original (or default) settings:

```
#pragma optimize( "", on )
```

The technique of passing an empty string ("") to the optimize pragma to turn all optimizations off and then on for selected functions works for all optimizations except p-code. The work-around is as follows:

```
#pragma optimize( "", off  )   // Turns off all optimizations
                               //   except p-code
#pragma optimize( "q", off )  // Turns off p-code
voic myfunc( void )
{
}
#pragma optimize( "", on   )   // Turns on all optimizations
                               //   except p-code
#pragma optimize( "q", on  )  // Turns on p-code
```

Optimization Options

page

Pragma page

Syntax #pragma page([*pages*])

The page pragma skips the specified number of pages in the source listing**.**

The page pragma generates a formfeed (page eject) in the source listing (created with /Fs) at the place where the pragma appears**.**

The optional *pages* parameter is an integer constant in the range 1-127 that specifies the number of pages to eject**.** If *pages* is absent**,** the pragma uses a default value of 1**,** in which case the next line in the source file appears at the top of the next listing page**.**

linesize
pagesize
skip
title
subtitle

pagesize

Pragma pagesize

Syntax #pragma pagesize([*numlines*])

The pagesize pragma sets the number of lines per page in the source listing.

The optional *numlines* parameter is an integer constant in the range 15-255 that specifies the number of lines that you want each page of the source listing to have. If *numlines* is absent, the pragma sets the page size to the number of lines specified in the /Sp option or, if that option is absent, to a default value of 63 lines.

The following statement causes the source listing to have 66 lines per page:

```
#pragma pagesize( 66 )
```

linesize
page
skip
title
subtitle

skip

Pragma skip

Syntax #pragma skip([*lines*])

The skip pragma skips the specified number of lines in the source listing.

The skip pragma generates a newline character (CR/LF) in the source listing at the point where the pragma appears.

The optional *lines* parameter is an integer constant in the range 1-127 that specifies the number of lines to skip. If this parameter is absent, the skip pragma defaults to one line.

The following statement places one blank line in the source listing:

```
#pragma skip()
```

linesize
pagesize
page
title
subtitle

title, subtitle

Pragma title, subtitle

Syntax #pragma title("*titlename*")

#pragma subtitle("*subtitlename*")

These pragmas specify a title or subtitle for the source listing.

The parameter is a string literal containing the title or subtitle for subsequent pages in the source listing. The title appears in the upper left corner of each page of the listing. The subtitle appears below the title on each page.

If you supply a null string ("") as the *titlename* parameter, title removes any title that was previously set. The *titlename* parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination. Likewise, these are true for the subtitle pragma.

The following statements set a title and subtitle:

```
#pragma title( "File I/O Module" )
#pragma subtitle( "Error handler" )
```

linesize
pagesize
skip
page

hdrstop

Pragma hdrstop

Syntax #pragma hdrstop[("*filename*")]

If a C or a C++ file contains a hdrstop pragma, the compiler saves the state of the compilation up to the location of the pragma. The compiled state of any code that follows the pragma is not saved.

The hdrstop pragma cannot occur inside a header file. It must occur in the source file at the file level; that is, it cannot occur within any data or function declaration or definition.

Use *filename* to name the precompiled header file in which the compiled state is saved. The *filename* must be a string (enclose it in quotation marks) and it must be enclosed in parentheses. A space between hdrstop and *filename* is optional. Note that you can also use the /Fp option to name the precompiled header file. If *filename* is not specified, the resulting filename is given the base name of the source file with a .PCH extension.

If no hdrstop pragma is specified, the compiler saves the state of the compilation through the end of the source file. As in the previous case, the name of the .PCH file is derived from the base name of the source file with a .PCH extension.

The hdrstop pragma is ignored unless either /Yu or /Yc is specified without a *filename*.

intrinsic, function

Pragma intrinsic, function

Syntax #pragma intrinsic(*function1* [, *function2*, ...])

#pragma function(*function1* [, *function2*, ...])

These pragmas specify that calls to the specified functions will be intrinsic or normal.

The intrinsic pragma option tells the compiler to generate intrinsic functions instead of function calls for specified functions. Alternatively, you can use the /Oi option to make intrinsic the default form for functions that have intrinsic forms. In this case, you can use the function pragma to override /Oi for specified functions.

Intrinsic functions may be inline functions, or may use special argument-passing conventions. In some cases, they may do nothing. Programs that use intrinsic functions are faster because they do not include the overhead associated with function calls. However, they may be larger because of the additional code that is generated.

The following functions have intrinsic forms:

_disable	_outpw
_enable	_rotl
_fmemcmp	_rotr
_fmemcpy	_strset
_fmemset	abs
_fstrcat	fabs
_fstrcmp	labs
_fstrcpy	memcmp
_fstrlen	memcpy
_fstrset	memset
_inp	strcat
_inpw	strcmp
_lrotl	strcpy
_lrotr	strlen
_outp	

Intrinsic versions of the memset, memcpy, and memcmp functions in compact- and large-model programs cannot handle huge arrays or huge pointers. To use huge arrays or huge pointers with these functions, you must compile your program with the huge memory model.

When the intrinsic pragma is used with the following functions, the function argument calling convention is changed to pass the arguments on the floating-point chip:

acos	acosl
asin	asinl
atan	atanl
atan2	atan2l
ceil	ceill
cos	cosl
cosh	coshl

exp	expl
floor	floorl
fmod	fmodl
log	logl
log10	log10l
pow	powl
sin	sinl
sinh	sinhl
sqrt	sqrtl
tan	tanl
tanh	tanhl

The intrinsic pragma affects a specified function beginning where the pragma appears**.** The effect continues to the end of the source file or to the appearance of a function pragma specifying that function**.**

loop_opt

Pragma loop_opt

Syntax #pragma loop_opt([{off | on}])

The loop_opt pragma turns loop optimization on or off for selected functions. When you want to turn off loop optimization, put the following line before the code on which you do not want to perform loop optimization:

```
#pragma loop_opt( off )
```

Note that the preceding line disables loop optimization for all code that follows it in the source file, not just the routines on the same line. To reinstate loop optimization, insert the following line:

```
#pragma loop_opt( on )
```

If no argument is given to the loop_opt pragma, loop optimization reverts to the behavior specified as a compiler option: enabled if the /Ox or /Ol option is in effect, and disabled otherwise. The interaction of the loop_opt pragma with the /Ol and /Ox options is explained in greater detail below:

Syntax	Compiled with /Ox or /Ol?	Action
#pragma loop_opt( )	No	Turns off optimization for loops that follow
#pragma loop_opt( )	Yes	Turns on optimization for loops that follow
#pragma loop_opt(on)	Yes or no	Turns on optimization for loops that follow
#pragma loop_opt(off)	Yes or no	Turns off optimization for loops that follow

check_pointer

Pragma check_pointer

Syntax #pragma check_pointer ([{ on | off }])

The check_pointer pragma instructs the compiler to turn off pointer checking if "off" is specified, or to turn on pointer checking if "on" is specified.

Syntax	Compiled w/ Pointer Check?	Action
#pragma check_pointer( )	Yes	Turns off pointer checking for

			pointers that follow
	#pragma check_pointer( )	No	Turns on pointer checking for pointers that follow
	#pragma check_pointer(on)	Yes or no	Turns on pointer checking for pointers that follow
	#pragma check_pointer(off)	Yes or no	Turns off pointer checking for pointers that follow

check_stack

check_stack

Pragma check_stack

Syntax #pragma check_stack ([{ on | off }])

The check_stack pragma instructs the compiler to turn off stack probes if "off" is specified, or to turn on stack probes if "on" is specified. If no argument is given, stack probes are treated according to the default (on, unless /Gs was used).

You can reduce the size of a program and speed up execution slightly by removing stack probes. You can do this with either the /Gs option or the check_stack pragma.

A "stack probe" is a short routine called on entry to a function to verify that there is enough room in the program stack to allocate local variables required by the function. The stack probe routine is called at every function entry point. Ordinarily, the stack probe routine generates a stack overflow message when it determines that the required stack space is not available. When stack checking is turned off, the stack probe routine is not called, and stack overflow can occur without being diagnosed (i.e., no error message is generated).

Use the /Gs option when you want to turn off stack checking for an entire module (if you know that the program does not exceed the available stack space). For example, stack probes may not be needed for programs that make very few function calls, or that have only modest local variable requirements. In the absence of the /Gs option, stack checking is on.

Use the check_stack pragma when you want to turn stack checking on or off for selected routines only, leaving the default (as determined by the presence or absence of the /Gs option) for the rest. When you want to turn off stack checking, put the following line before the definition of the function that you do not want to check:

```
#pragma check_stack( off )
```

Note that the preceding line disables stack checking for all routines that follow it in the source file, not just the routines on the same line. To reinstate stack checking, insert the following line:

```
#pragma check_stack( on )
```

For earlier versions of Microsoft C, the check_stack pragma had a different format: check_stack+ to enable stack checking and check_stack- to disable stack checking. Although the Microsoft C compiler still accepts this format, its use is discouraged, since it may not be supported in future versions.

If no argument is given for the check_stack pragma, stack checking reverts to the behavior specified on the command line: disabled if the /Gs option is given, or enabled otherwise. The interaction of the check_stack pragma with the /Gs option is explained in greater detail below:

Syntax	Compiled with /Ox or /Ol?	Action
#pragma check_stack( )	Yes	Turns off stack checking for routines that follow
#pragma check_stack( )	No	Turns on stack checking for routines that follow

#pragma check_stack(on)	Yes or no	Turns on stack checking for routines that follow
#pragma check_stack(off)	Yes or no	Turns off stack checking for routines that follow

The check_stack(off) and /Gs options should be used with great care**.** Although these options can make a program smaller and faster**,** they may mean that the program will not be able to detect certain execution errors**.**

check_pointer

pack

Pragma pack

Syntax #pragma pack([{1 | 2 | 4 | 8 | 16 }])

The pack pragma specifies packing alignment for structure types.

When storage is allocated for structures, structure members are ordinarily stored as follows:

- Items of type char or unsigned char, or arrays containing items of these types, are byte aligned.
- Structures are word aligned; structures of odd size are padded to an even number of bytes.
- All other types of structure members are word aligned.

To conserve space, or to conform to existing data structures, you may want to store structures more or less compactly. The /Zp compiler option or the pack pragma controls how structure data is "packed" into memory.

Use the /Zp option to specify the same packing for all structures in a module. When you give the /Zp[*n*] option, where *n* is 1, 2, 4, 8, or 16, each structure member after the first is stored on *n*-byte boundaries, depending on the option you choose. If you use the /Zp option without an argument, structure members are packed on 1-byte boundaries.

On some processors, the /Zp option can result in slower program execution because of the time required to unpack structure members when they are accessed. For example, on an 8086 processor, this option can reduce efficiency if members with int or long type are packed in such a way that they begin on odd-byte boundaries.

Use the pack pragma to specify packing other than the packing specified on the command line for particular structures. Give the pack(*n*) pragma, where *n* is 1, 2, 4, 8, or 16, before structures that you want to pack differently. To reinstate the packing given on the command line, give the pack( ) pragma with no arguments.

Syntax	Compiled with /Zp Option?	Action
#pragma pack( )	Yes	Reverts to packing specified by the compiler for structures that follow
#pragma pack( )	No	Reverts to default packing for structures that follow
#pragma pack(n)	Yes or no	Packs the following structures to the given byte boundary until changed or disabled

pointers_to_members

Pragma pointers_to_members

Syntax #pragma pointers_to_members(*pointer-declaration*, [*most-general-representation*])

The pointers_to_members pragma specifies whether a pointer to a class member can be declared before its associated class definition and controls the pointer size and the code required to interpret the pointer. You can place a pointers_to_members pragma in your source file as an alternative to using /vm*x* options on the command line.

The *pointer-declaration* argument specifies whether you have declared a pointer to a member before or after the associated function definition. The *pointer-declaration* argument is one of the following two symbols.

full_generality
: Generates safe, sometimes nonoptimal code. You use full_generality if any pointer to a member is declared after the associated class definition. This argument always uses the pointer representation specified by the *most-general-representation* argument. See information above for the size of each pointer type. Equivalent to /vmg.

best_case
: Generates safe, optimal code using best-case representation for all pointers to members. Requires defining the class before declaring a pointer to a member of the class. The default is best_case.

The *most-general-representation* argument specifies the smallest pointer representation that the compiler can safely use to reference any pointer to a member of a class in a translation unit. The argument can be one of the following.

single_inheritance
: The most general representation is single-inheritance, pointer to a member function. Causes an error if the inheritance model of a class definition for which a pointer to a member is declared is ever either multiple or virtual.

multiple_inheritance
: The most general representation is multiple-inheritance, pointer to a member function. Causes an error if the inheritance model of a class definition for which a pointer to a member is declared is virtual.

virtual_inheritance
: The most general representation is virtual-inheritance, pointer to a member function. Never causes an error. Default when #pragma pointers_to_members (full-generality) is used.

vtordisp

Pragma vtordisp

Syntax #pragma vtordisp(on | off)

The vtordisp pragma is applicable only to code that uses virtual bases. If a derived class overrides a virtual function that it inherits from a virtual base class, and a constructor or destructor for the derived class calls that function using a pointer to the virtual base class, the compiler may introduce additional hidden "vtordisp" fields into classes with virtual bases.

The vtordisp pragma enables the addition of the hidden vtordisp construction/destruction displacement member. This pragma affects the layout of classes which follow the pragma. The /vd0 and /vd1 options specify the same behavior for complete modules. Specifying off suppresses the hidden vtordisp members. Specifying on, the default, enables them where they are necessary. Turn off vtordisps only if there is no possibility that the class's constructors and destructors do not call virtual functions virtually.

setlocale

Pragma setlocale

Syntax #pragma setlocale(*locale_string*)

The setlocale pragma defines the locale (country and language) to be used when translating wide character constants and string literals.

Since the algorithm for converting multibyte characters to wide characters may vary by locale or the compilation may take place in a different locale from where an executable will be run, this pragma provides a way to specify the target locale at compiler time. This guarantees that the wide-character strings will be stored in the correct format. The *locale_string* currently supported for 16-bit targets is "C". The "C" locale maps each character in the string to its value as a wchar_t (unsigned short).

Reducing Text-Only Programs

You can reduce the executable-file size by about 8K or 10K for a program that uses only the text-mode functions from GRAPHICS.LIB.

To do so, link your program with TXTONLY.OBJ. If you explicitly name GRAPHICS.LIB in the compile or link command, place it after TXTONLY.OBJ on the command line. Use the /NOE option to avoid multiple symbol definitions. For example:

```
CL TEST.C NOGRAPH /LINK [ GRAPHICS.LIB ] /NOE
```

A program built with TXTONLY.OBJ cannot use graphics modes or graphics functions. It cannot try to change the palette. If the program tries to enter a graphics mode, _setvideomode will return an error.

offsetof

Macro offsetof

Syntax size_t offsetof(*struct-name*, *member-name*);

Include <stddef.h>

The macro offsetof takes a structure name and a member name as arguments. It returns the offset in bytes of the specified member from the beginning of the structure.

Types can be specified using the struct keyword.

Note offsetof is not a function and cannot be described using a C prototype.

Return Value This macro returns the offset of *member-name* from the beginning of the structure. It is undefined for bit fields.

Stringizing Operator

Operator #

Syntax #*parameter*

The stringizing operator is used only with the arguments of macros.

If a # precedes a formal parameter in the definition of the macro, the actual argument is enclosed in double quotation marks and treated as a string when the macro is expanded. The resulting string is concatenated with adjacent strings. For example,

```
#define debug( x ) printf( #x " = %d", x )
```

causes the statement

```
debug( width );
```

to be expanded into

```
printf( "width = %d", width );
```

White space between tokens in the actual argument is ignored. If the argument contains characters that normally must be preceded by a backslash (when appearing in a string (such as " or \), the backslash is automatically inserted.

Charizing Operator

Operator #@

Syntax #@*parameter*

The charizing operator is used only with the arguments of macros.

If a #@ precedes a formal parameter in the definition of the macro, the actual argument is enclosed in single quotation marks and treated as a character when the macro is expanded. For example,

```
#define makechar( x )  #@x
```

causes the statement

```
a = makechar( b );
```

to be expanded into

```
a = 'b';
```

causes the statement

```
printvar( 7 );
```

to be expanded into

```
printf( "%d", var7 );
```

The single-quotation character cannot be used with the charizing operator.

Token-Pasting Operator

Operator ##

Syntax *token##parameter*

parameter##token

The token-pasting operator is used only with macros.

If ## precedes or follows a formal parameter in the definition of a macro, the actual argument is concatenated with the token on the other side of the ## when the macro is expanded. For example,

```
#define printvar(x) printf("%d", var##x)
```

causes the statement

```
printvar(7);
```

to be expanded into

```
printf("%d",var7);
```

defined

Operator defined

Syntax defined(*identifier*)

!defined(*identifier*)

The defined operator is used with the #if directive to test whether *identifier* is currently defined. It returns true (nonzero) if *identifier* is currently defined and returns false (0) if it is not.

The logical NOT operator (!) can be used to reverse the logic of the defined operator. For example,

```
#if defined( __cplusplus )
#if defined( __cplusplus ) && !defined( M_I8086 )
```

#define
#if
#ifdef
#ifndef
#undef

define

Directive #define

Syntax #define *identifier substitution-text*

#define *identifier*([*parameter-list*]) *substitution-text*

This directive replaces all subsequent cases of *identifier* with the *substitution-text*.

The *substitution-text* can consist of one or more constants, keywords, or statements. When the identifier is replaced by a constant expression, it is a manifest constant. When the identifier is replaced by an expression containing parameters, it is a macro.

If *substitution-text* is more than one line, it can be continued onto successive lines by placing a backslash (before the end of each line. Enclosing *substitution-text* in parentheses ensures proper evaluation if the text is an expression or has a leading minus sign. The *substitution-text* can also be empty; this removes occurrences of the identifier from the file.

In the following example, a simple manifest constant is created by assigning a numeric value to the symbol PI:

```
#define  PI  3.14159265
```

If a *parameter-list* appears after the identifier, each occurrence of *identifier*(*actual-parameter-list*) is replaced by a version of the *substitution-text* that has actual arguments substituted for the parameters. There must be an equal number of actual arguments and parameters.

The optional *parameter-list* consists of one or more parameter names, separated by commas and enclosed by parentheses. No space can separate the identifier and the opening parenthesis. The parameter names appear in the *substitution-text* to mark the places where actual values will be substituted.

In the following example, the macro SQUARE is defined to multiply its argument by itself:

```
#define  SQUARE( arg ) ((arg) * (arg))
```

In C++, the const keyword and inline functions replace most uses of the #define directive.

See Stringizing Operator, Charizing Operator, and Token-Pasting Operator for more information.

defined
#if
#ifdef
#ifndef
#undef
const
auto_inline

auto_inline

Pragma auto_inline

Syntax #pragma auto_inline([on | off])

The auto_inline pragma inhibits the preprocessor from expanding a function when the /Ob2 option is in effect. To use it, place the pragma before and after a function definition:

```
#pragma auto_inline( off )
void not_expanded( void )
{
   /* ... */
}
#pragma auto_inline( on )
```

The auto_inline pragma cannot appear in the body of a function, nor can it be used with the invocation of a function. It will not affect a function specifically marked with the inline or __inline keywords.

If no value is supplied in parentheses, "on" is the default.

/Ob
inline_recursion
inline_depth

inline_depth, inline_recursion

Pragma inline_depth, inline_recursion

Syntax #pragma inline_depth([0**...** 255])

#pragma inline_recursion([on | off])

The inline_depth pragma controls the number of times inline expansion can occur by controlling the number of times that a series of function calls can be expanded (from 0 to 255 times). The inline_recursion pragma controls the inline expansion of directly or mutually recursive function calls.

Use these pragmas to control functions marked with the inline or __inline keywords, or functions that the compiler automatically expands under the /Ob2 option. Both pragmas require an /Ob command-line option setting of either 1 or 2.

The inline_depth pragma controls the number of times a series of function calls can be expanded. For example, if the inline depth is four, and if A calls B and B then calls C, all three calls will be expanded inline. However, if the closest inline expansion is two, only A and B are expanded, and C remains as a function call. By default, the value of inline_depth is eight.

The inline_recursion pragma controls how recursive functions are expanded. If inline_recursion is off, and if an inline function calls itself (either directly or indirectly), the function is expanded only once. If inline_recursion is on, the function is expanded multiple times until the value of inline_depth is reached or capacity limits are reached. By default, inline_recursion is off. This pragma takes effect at the first function call after the pragma is seen and does not affect the definition of the function.

The inline depth can be decreased during expansion but not increased. If the inline depth is six and during expansion the preprocessor encounters an inline_depth pragma with a value of eight, the depth remains six.

An inline depth of 0 inhibits inline expansion; an inline depth of 255 places no limit on inline expansion.

If either pragma is used without specifying a value, the default value is used.

/Ob
auto_inline

error

Directive #error

Syntax #error *message*

The #error directive causes the compiler to display *message* on stderr and return a nonzero code when the compiler terminates.

After the compiler encounters an #error directive, it scans the rest of the program for syntax errors but does not produce an object file. For example:

```
#if !defined( _CHAR_UNSIGNED )
   #error /J option required.
   #endif
```

#pragma message

#if, #elif, #else, #endif

Directive #if, #elif, #else, #endif

Syntax

#if *test expression*
[*text-block*]
[#elif *test expression*
text-block]
. . .
[#else
text-block]
#endif

This set of directives evaluates each test expression associated with an #if or #elif directive until a true (nonzero) expression is found, then processes the *text-block* associated with that test expression.

If there is an #else clause, the *text-block* associated with it is processed only if no test expression in the #if or #elif clauses has a nonzero value.

The *test expression* can be any expression that evaluates to a constant, and can contain logical operators and the defined operator. It cannot use the sizeof operator, type casts, or the float or enum types. The *text-block* can contain C code or compiler directives.

#ifdef
defined
#define
#undef

Identifiers

A C or C++ identifier must begin with an alphabetic character, either uppercase or lowercase, or an underscore (_). After the first character, an identifier can contain any combination of letters and digits. A single identifier can be up to 247 characters in length.

Both C and C++ are case sensitive; they treat "filename" and "FileName" as different identifiers.

#ifdef, #ifndef

Directive #ifdef
#ifndef

Syntax #ifdef *identifier*
#ifndef *identifier*

These directives test whether *identifier* is currently defined.

The #ifdef directive returns true (nonzero) if it is currently defined and returns false (0) if it is not. The #ifndef directive returns the opposite. The #if directive combined with the defined operator is preferred for all new programs because they can be used in expressions that check multiple conditions on the same line. For example:

```
#ifdef __cplusplus
#ifdef M_I8086
#pragma message ("C++ and 8086/88")
#endif #endif

#if defined( __cplusplus ) && defined( M_I8086 )
#pragma message ("C++ and 8086/88")
#endif
```

#if
defined
#undef
#define
identifier

#include

Directive #include

Syntax #include "*path-spec*"

#include <*path-spec*>

The #include directive inserts the contents of the file specified by *path-spec* into the current file.

If the path-spec contains the complete drive and path specification, that file is included without searching any directories. If *path-spec* is enclosed in double quotation marks, the directory of the file containing the #include directive is searched.

If the current file is also an include file, the directory of the parent file is searched. This search continues recursively through all the nested include files until the original source file's directory is searched.

If the file is still not found or if the path-spec is enclosed in angle brackets, the next directories searched are the ones specified using the /I command-line option. After those have been searched, the compiler searches the directories specified in the INCLUDE environment variable.

#line

Directive #line

Syntax #line *constant* ["*filename*"]

The #lind directive changes the compiler's internally stored line number to *constant*, and changes the internal filename to *filename*.

The current line number and filename are available through the predefined identifiers __LINE__ and __FILE__.

__FILE__
__LINE__

#undef

Directive #undef

Syntax #undef *identifier*

The #undef directive removes the current definition of *identifier*, which must have been previously defined with the #define directive.

#define
defined
#if
#ifdef
#ifndef

Predefined Time and File Macros

Macros __DATE__, __TIME__, __TIMESTAMP__, __FILE__, __LINE__

The following predefined macros provide information about the state of the compilation and the source file:

Macro	Description
__TIME__	The most recent compilation time of the current source file. The time is a string literal of the form: `hh:mm:ss`
__DATE__	The compilation date of the current source file. The date is a string literal of the form: `Mmm dd yyyy` The month name `Mmm` is the same as for dates generated by the library function asctime declared in TIME.H.
__TIMESTAMP__	The date and time of the last modification of the current source file, expressed as a string literal in the form: `Ddd Mmm Date hh:mm:ss yyyy` where `Ddd` is the abbreviated day of the week (Wed) and `Date` is an integer from 1 to 31.
__FILE__	The current source filename. __FILE__ expands to a string surrounded by double quotation marks.
__LINE__	The line number in the current source file. The line number is a decimal integer constant. It can be altered with a #line directive.

predefined macros

auto

Keyword auto

Syntax auto *declarator*;

More information on *declarator*

The auto keyword is the storage-class specifier indicating that the variable has local (automatic) extent. It is the default storage-class specifier for block-scoped variable declarations.

extern
register
static

More information on *declarator*

Syntax *declarator* **:**
[*pointer direct-declarator*]

direct-declarator **:**
identifier
(*declarator*)
direct-declarator [*constant-expression*]
direct-declarator (*parameter-type-list*)
direct-declarator ([*identifier-list*])

A declarator is the part of a declaration that specifies the name that is to be introduced into the program. It can include modifiers such as * (pointer-to) and any of the Microsoft calling-convention and memory-model keywords.

You use declarators to declare arrays of values, pointers to values, and functions returning values of a specified type. Declarators appear in pointer, array, and function declarations.

Each declarator declares at least one identifier. A declarator must include a type specifier to be a complete declaration. The type specifier gives the type of the elements of an array type, the type of object addressed by a pointer type, or the return type of a function.

break

Keyword break

Syntax break;

The break keyword terminates the smallest enclosing do, for, switch, or while statement in which it appears.

switch
for
while
continue

__cdecl

Keyword __cdecl

Syntax __cdecl *declarator*

More information on *declarator*

The __cdecl keyword instructs the compiler to use the C naming and calling conventions for the variable or function.

In the C naming convention, the compiler adds an underscore to the beginning of a name and maintains case-sensitivity. In the C calling convention, function arguments are pushed from right to left.

Place the __cdecl modifier before a variable or a function name; it can appear before or after the __near and __far modifiers. Because the C naming and calling conventions are the default, the only time you need to use __cdecl is when you've specified the /Gc (Pascal/FORTRAN) or the /Gr (fastcall) compiler options. For compatibility with previous versions, both cdecl and _cdecl are synonyms for __cdecl.

__fastcall
__fortran
__pascal

continue

Keyword continue

Syntax continue**;**

The continue keyword passes control to the next iteration of the smallest enclosing do, for, or while statement in which it appears.

for
while
break

do

Keyword do

Syntax do
statement
while(*expression*)**;**

The do keyword executes *statement* repeatedly until *expression* becomes false (0)**.**

while
break
continue

__emit

Keyword __emit

Syntax __asm __emit *byte*

The __emit keyword defines a single immediate *byte* at the current location.

The __emit pseudoinstruction is similar to the DB directive of the Microsoft Macro Assembler (MASM). It allows definition of a single immediate *byte* at the current location in the current code segment. However, __emit can define only one byte at a time, and it can only define bytes in the code (_TEXT) segment. It uses the same syntax as the INT instruction.

One use for __emit is to define 80386-specific instructions, which the inline assembler does not support. The following fragment defines the 80386 CWDE instruction:

```
// Macro for cwde instruction assumes 16-bit mode
#define cwde __asm __emit 0x66 __asm __emit 0x98
 . . .
__asm
{
    . . .          ; Assembly instructions
    cwde           ; Macro to generate CWDE
    . . .          ; More instructions
}
```

__asm

__fastcall

Keyword __fastcall

Syntax __fastcall *declarator*


More information on *declarator*

The __fastcall keyword specifies that the function uses a calling convention that passes arguments in registers rather than on the stack, resulting in faster code. For compatibility with previous versions, _fastcall is a synonym for __fastcall.

The choice of registers depends on the type of arguments:

Type	Register Candidates
char / unsigned char	AL, DL, BL
int / unsigned int	AX, DX, BX
long / unsigned long	DX:AX
near pointer	BX, AX, DX
far or huge pointer	passed on the stack

Arguments are allocated to suitable registers if available, and are pushed onto the stack otherwise. Structs, unions, and all floating-point types are always pushed onto the stack.

Return values of four bytes or smaller, including structs and unions, are placed in the registers as follows:

Size	Register
1 byte	AL
2 bytes	AX
4 bytes	DX:AX

Floating-point values are returned on the floating-point stack. To return structs or unions larger than four bytes, the calling program pushes a hidden last parameter, which is a near pointer to a buffer in which the value is to be returned. A far pointer to SS:*hiddenparam* must be returned in DX:AX.

The __fastcall calling convention cannot be used with functions having variable-length parameter lists or with functions having any of the following attributes: __cdecl, __export, __fortran, __interrupt, __pascal, __saveregs.

__cdecl
__pascal
__fortran
__interrupt

for

Keyword for

Syntax for([*init-expr*]; [*cond-expr*]; [*loop-expr*])
statement

The for keyword executes *statement* repeatedly.

First, the initialization (*init-expr*) is evaluated. Then, while the conditional expression (*cond-expr*) evaluates to a nonzero value, *statement* is executed and the loop expression (*loop-expr*) is evaluated. When *cond-expr* becomes 0, control passes to the statement following the for loop.

while
break
continue

__pascal, __fortran

Keyword __pascal, __fortran

Syntax __pascal *declarator*

__fortran *declarator*

More information on *declarator*

These modifiers instruct the compiler to use the Pascal/FORTRAN naming and calling conventions for the variable or function.

In the Pascal/FORTRAN naming convention, names are converted into uppercase. In the Pascal/FORTRAN calling convention, function arguments are pushed from left to right. The __fortran and __pascal modifiers are synonyms.

Place these modifiers before the variable or function name; they can appear before or after the __near and __far modifiers. For compatibility with previous versions, pascal and _pascal are synonyms for __pascal, and fortran and _fortran are synonyms for __fortran.

__cdecl
__fastcall
__interrupt

goto

Keyword goto

Syntax goto *name*;
. . .
name: *statement*

The goto keyword transfers control directly to the *statement* specified by the label *name*.

if
return
switch

if, else

Keywords if, else

Syntax if(*expression*)
statement1
[else
statement2]

The if keyword executes *statement1* if *expression* is true (nonzero); if else is present and *expression* is false (zero), it executes *statement2*. After executing *statement1* or *statement2*, control passes to the next statement.

switch

__loadds

Keyword __loadds

Syntax __loadds *declarator*

More information on *declarator*

The __loadds keyword loads the data-segment register (DS) with a segment value upon entry to a function. For compatibility with previous versions, _loadds is a synonym for __loadds.

The __loadds keyword causes the data-segment (DS) register to be loaded with a specified segment value upon entering the specified function. The previous DS value is restored when the function terminates.

Functions declared with the __loadds keyword cause the most recently specified data segment to be loaded into the DS register. The compiler uses the segment name specified by the /ND (name-data-segment) option or, if no segment has been specified, the default group DGROUP. Note that this function modifier has the same effect as the /Au option, but on a function-by-function basis.

Example

The main file contains a declaration of funcsample, a far function taking a single argument of any pointer type and returning no value. The function loads a new data segment upon entry.

```
void __far __loadds funcsample( void *s );
main()
{
   char s[11];
   // Call the sample __loadds function
   funcsample( (void *)s ); }
```

The second file defines the function, and is compiled with the option /ND MY_DATA.

```
// Define the function that will load DS from MY_DATA
void __far __loadds funcsample( void *s )
{
 . . .
}
```

__saveregs
__export
__interrupt
Naming Segments

register

Keyword register

Syntax register *declarator*

More information on *declarator*

The register keyword specifies that the variable is to be stored in a machine register, if possible.

auto
extern
static

return

Keyword return

Syntax return [*expression*]**;**

The return keyword terminates execution of the function in which it appears and returns control (and the value of *expression* if given) to the calling function**.**

__saveregs

Keyword __saveregs

Syntax __saveregs *declarator*

More information on *declarator*

The __saveregs keyword saves and restores CPU registers when entering and exiting a function, respectively. For compatibility with previous versions, _saveregs is a synonym for __saveregs.

The __saveregs keyword is useful in any case in which it is not certain what the register conventions of the caller might be. For instance, __saveregs could be used for a general-purpose function that will reside in a dynamic-link library. Because a function in a dynamic-link library might be called from any language, you may choose not to assume Microsoft C calling conventions in some cases.

The __saveregs keyword causes the compiler to generate code that saves and restores all CPU registers when entering and exiting the specified function. Note that __saveregs does not restore registers used for a return value (the AX register, or AX and DX).

It is not possible to declare a function with both the __saveregs and the __interrupt attributes.

The following statement declares *funcptr* as a far pointer to a function with no arguments, returning a char pointer. The presence of __saveregs tells the compiler that the function called through *funcptr* saves and restores register contents. In this example, the __loadds keyword also tells the compiler that the target function loads its own data segment.

```
char *(__far __saveregs __loadds *funcptr)( void );
```

__loadds
__interrupt
__export

__self

Keyword __self

Syntax __self

The __self keyword is a base expression indicating the segment in which the variable itself resides. For compatibility with previous versions, _self is a synonym for __self

The __self keyword can be used to cast to a segment value, as in the example below:

```
typedef struct Tree TREE;


struct Tree
{
   int name; TREE __based( (__segment)__self ) *left;
   TREE __based( (__segment)__self ) *right;
};
```

The pointers within the Tree structure type are self-based, meaning that for a given structure variable, they point within the segment in which the structure variable is located.

__based
__segment
__segname

sizeof

Keyword sizeof

Syntax sizeof *expression*

The sizeof keyword gives the amount of storage, in bytes, associated with a variable or a type (including aggregate types).

The *expression* is either an identifier or a type-cast expression (a type specifier enclosed in parentheses).

When applied to a structure type or variable, sizeof returns the actual size, which may include padding bytes inserted for alignment. When applied to a statically-dimensioned array, sizeof returns the size of the entire array. The sizeof operator cannot return the size of dynamically-allocated arrays or external arrays. For example:

```
int  i = sizeof( int );
struct POS                              // sizeof( struct POS )
{                                       //   may not be two
   char row;                            //   because of alignment
   char col;
};
int  array[] = { 1, 2, 3, 4, 5 };       // sizeof( array ) is 10
                                        // sizeof( array[0] ) is 2
int  sizearr =                          // Count of items in array
   sizeof( array ) / sizeof( array[0] );
```

Operators

__stdcall

Keyword __stdcall

Syntax __stdcall *declarator*

More information on *declarator*

Note The __stdcall attribute is not valid for 16-bit targets.

The __stdcall keyword specifies that the arguments of the designated function are pushed right to left and that an underscore is prepended to the name. The __stdcall operator is functionally similar to __cdecl, except that no underscore is prepended to the function name. For compatibility with previous versions, _stdcall is a synonym for __stdcall.

If a prototype for the function exists and takes a fixed number of arguments, the callee pops the stack; otherwise, the caller pops the stack.

If a function is marked as __stdcall and has a variable number of arguments, it is implemented as __cdecl. If a Pascal function has a variable number of arguments, the compiler reports an error.

The calling convention specified in a prototype or a declaration overrides any convention specified by a command-line switch.

/Gc
/Gd

/Zc
__cdecl
__interrupt
__loadds
__pascal

switch, case, default

Keywords switch, case, default

Syntax

```
switch( expression )
{
    [case constant-expression:]
    . . .
        [statement]
    . . .
    [default:
        statement]
}
```

The switch and case keywords evaluate *expression* and execute any statement associated with *constant-expression* whose value matches the initial expression.

If there is no match with a constant expression, the statement associated with the default keyword is executed. If the default keyword is not used, control passes to the statement following the switch block.

if
break

volatile

Keyword volatile

Syntax volatile *declarator*


More information on *declarator*

The volatile keyword is a type qualifier used to declare that an object can be modified in the program by something other than statements, such as the operating system or the hardware.

Objects declared as volatile are not used in optimizations because their value can change at any time. The system always reads the current value of a volatile object at the point it is requested, even if the previous instruction asked for a value from the same object. Also, the value of the object is written immediately on assignment.

One use of the volatile qualifier is to provide access to memory locations used by asynchronous processes such as interrupt handlers.

const

while

Keyword while

Syntax while(*expression*)
statement

The while keyword executes *statement* repeatedly until *expression* becomes 0**.**

do
for
break
continue

main

Keyword main, argc, argv, envp

Syntax
```
main( int argc, char *argv[ ], char *envp[ ] )
{
    program-statements
}
```

The main function is the name of the function that marks the beginning and end of program execution. A C or C++ program must have one function named main.

The main function can take the following three optional arguments, traditionally called argc, argv, and envp (in that order):

argc
: An integer specifying how many arguments are passed to the program from the command line. Since the program name is considered an argument, argc is at least one.

argv
: An array of null-terminated strings. It can be declared as an array of pointers to char (char *argv[ ]) or as a pointer to pointers to char (char **argv). The first string (argv[0]) is the program name, and each following string is an argument passed to the program from the command line. The last pointer (argv[argc]) is NULL.

envp
: A pointer to an array of environment strings. It can be declared as an array of pointers to char (char *envp[ ]) or as a pointer to pointers to char (char **envp). The end of the array is indicated by a NULL pointer.

See Expanding Wildcard Arguments, Parsing Command-Line Arguments, and Suppressing Command-Line Processing for more information.

WinMain
LibMain
_dos_findfirst
getenv
_putenv
_searchenv

Addition Operator

Operator +

The addition operator (+) causes its two operands to be added. Both operands can be either integral or floating types, or one operand can be a pointer and the other an integer.

When an integer is added to a pointer, the integer value (*i*) is converted by multiplying it by the size of the value that the pointer addresses. After conversion, the integer value represents *i* memory positions, where each position has the length specified by the pointer type. When the converted integer value is added to the pointer value, the result is a new pointer value representing the address *i* positions from the original address. The new pointer value addresses a value of the same type as the original pointer value and therefore is the same as array indexing. If the sum pointer points outside the array, except at the first location beyond the high end, the result is undefined.

logical-NOT
subtraction
bitwise complement

Subtraction Operator

Operator −

The subtraction operator (−) subtracts the second operand from the first. Both operands can be either integral or floating types, or one operand can be a pointer and the other an integer.

When two pointers are subtracted, the difference is converted to a signed integral value by dividing the difference by the size of a value of the type that the pointers address. The size of the integral value is defined by the type ptrdiff_t in the standard include file STDDEF.H. The result represents the number of memory positions of that type between the two addresses. The result is only guaranteed to be meaningful for two elements of the same array.

When an integer value is subtracted from a pointer value, the subtraction operator converts the integer value (*i*) by multiplying it by the size of the value that the pointer addresses. After conversion, the integer value represents *i* memory positions, where each position has the length specified by the pointer type. When the converted integer value is subtracted from the pointer value, the result is the memory address *i* positions before the original address. The new pointer points to a value of the type addressed by the original pointer value.

logical-NOT
addition
bitwise complement

Indirection Operator

Operator *

The indirection operator (*) accesses a value indirectly, through a pointer. The operand must be a pointer value. The result of the operation is the value addressed by the operand; that is, the value at the address to which its operand points. The type of the result is the type that the operand addresses.

If the operand points to a function, the result is a function designator. If it points to a storage location, the result is an l-value designating the storage location.

If the pointer value is invalid, the result is undefined. The following list includes some of the most common conditions that invalidate a pointer value.

- The pointer is a null pointer.
- The pointer specifies the address of a local item that is not visible at the time of the reference.
- The pointer specifies an address that is inappropriately aligned for the type of the object pointed to.
- The pointer specifies an address not used by the executing program.

Multiplication Operator

The multiplication operator (*) causes its two operands to be multiplied.

The multiplication operator can take integral- or floating-type operands; the types of the operands can be different.

Note Since the conversions performed by the multiplicative operators do not provide for overflow or underflow conditions, information may be lost if the result of a multiplicative operation cannot be represented in the type of the operands after conversion.

Division Operator

Operator /

The division operator (/) causes the first operand to be divided by the second. If two integer operands are divided and the result is not an integer, it is truncated according to the following rules:

The result of division by 0 is undefined according to the ANSI C standard and causes an error at compile- or runtime.

If both operands are positive or unsigned, the result is truncated toward 0.

If either operand is negative, whether the result of the operation is the largest integer less than or equal to the algebraic quotient or is the smallest integer greater than or equal to the algebraic quotient is implementation defined.

Modulus Operator

Operator %

The result of the remainder operator (%) is the remainder when the first operand is divided by the second. When the division is inexact, the result is determined by the following rules:

If the right operand is zero, the result is undefined.

If both operands are positive or unsigned, the result is positive.

If either operand is negative and the result is inexact, the result is implementation defined.

Relational and Equality Operators

Operators <, <=, >, >=, ==, !=

The binary relational and equality operators compare their first operand to their second operand to test the validity of the specified relationship. The result of a relational expression is 1 if the tested relationship is true and 0 if it is false. The type of the result is int.

Operator	Relationship Tested
<	First operand less than second operand
>	First operand greater than second operand
<=	First operand less than or equal to second operand
>=	First operand greater than or equal to second operand
==	First operand equal to second operand
!=	First operand not equal to second operand

Assignment Operators

Operator =, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=

Syntax *assignment-expression* **:**
conditional-expression
unary-expression assignment-operator assignment-expression

An assignment operation assigns the value of the right-hand operand to the storage location named by the left-hand operand. Therefore, the left-hand operand of an assignment operation must be a modifiable l-value. After the assignment, an assignment expression has the value of the left operand but is not an l-value. The *assignment-operator* is one of the following**:**

Operator	Operation Performed
=	Simple assignment
***=**	Multiplication assignment
/=	Division assignment
%=	Remainder assignment
+=	Addition assignment
-=	Subtraction assignment
<<=	Left-shift assignment
>>=	Right-shift assignment
&=	Bitwise-AND assignment
|=	Bitwise-inclusive-OR assignment
^=	Bitwise-exclusive-OR assignment

Postfix Increment and Decrement Operators

Operator ++, --

Syntax *postfix-expression* **:**

postfix-expression ++
postfix-expression −−

Operands of the postfix increment and decrement operators are scalar types that are modifiable l-values.

The result of the postfix increment or decrement operation is the value of the operand. After the result is obtained, the value of the operand is incremented (or decremented). The following code illustrates the postfix increment operator.

```
if( var++ > 0 )
    *p++ = *q++;
```

In this example, the variable `var` is compared to 0, then incremented. If `var` was positive before being incremented, the next statement is executed. First, the value of the object pointed to by `q` is assigned to the object pointed to by `p`. Then, `q` and `p` are incremented.

Address-of, Bitwise-AND, Reference

Operator &

Address-of Operator

The address-of operator (&) gives the address of its operand. The operand of the address-of operator can be either a function designator or an l-value that designates an object that is not a bit field and is not declared with the register storage-class specifier.

The result of the address operation is a pointer to the operand. The type addressed by the pointer is the type of the operand.

The address-of operator can only be applied to variables with fundamental, structure, or union types that are declared at the file-scope level, or to subscripted array references. In these expressions, a constant expression that does not include the address-of operator can be added to or subtracted from the address expression.

Bitwise-AND Operator

Syntax *AND-expression* :
equality-expression
AND-expression & *equality-expression*

The bitwise-AND operator (&) compares each bit of its first operand to the corresponding bit of its second operand. If both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

Converting to Reference Types (C++ Only)

In C++, any object whose address can be converted to a given pointer type can also be converted to the analogous reference type. For example, any object whose address can be converted to type char * can also be converted to type char &. No constructors or class conversion functions are called to make a conversion to a reference type.

bitwise-exclusive-OR
bitwise-inclusive-OR
shift operators
bitwise complement

Bitwise-exclusive-OR Operator

Operator ^

The bitwise-exclusive-OR (^) operator compares each bit of its first operand to the corresponding bit of its second operand. If one bit is 0 and the other bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

bitwise-AND
bitwise-inclusive-OR
shift operators
bitwise complement

Bitwise-inclusive-OR

Operator |

The bitwise-inclusive-OR (|) operator compares each bit of its first operand to the corresponding bit of its second operand. If either bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

bitwise-AND
bitwise-exclusive-OR
shift operators
bitwise complement

Bitwise Shift Operators: Left Shift and Right Shift

Operator <<, >>

Syntax *shift-expression* :
additive-expression
shift-expression **<<** *additive-expression*
shift-expression **>>** *additive-expression*

The shift operators shift their first operand left (**<<**) or right (**>>**) by the number of positions the second operand specifies.

Both operands must be integral values. These operators perform the usual arithmetic conversions; the type of the result is the type of the left operand after conversion.

For leftward shifts, the vacated right bits are set to 0. For rightward shifts, the vacated left bits are filled based on the type of the first operand after conversion. If the type is unsigned, they are set to 0. Otherwise, they are filled with copies of the sign bit.

The result of a shift operation is undefined if the second operand is negative, or if the right operand is greater than or equal to the width in bits of the promoted left operand.

bitwise-AND
bitwise-exclusive-OR
bitwise-inclusive-OR
bitwise complement

Bitwise Complement Operator

Operator ~

The bitwise-complement (or bitwise-NOT) operator produces the bitwise complement of its operand. The operand must be of integral type. This operator performs usual arithmetic conversions; the result has the type of the operand after conversion.

logical-NOT
addition
bitwise-AND
bitwise-exclusive-OR
bitwise-inclusive-OR
bitwise complement

Logical-AND Operator

Operators &&

Syntax *logical-AND-expression* :
inclusive-OR-expression
logical-AND-expression && *inclusive-OR-expression*

The logical operators, logical AND (&&) and logical OR (||), are used to combine multiple conditions formed using relational or equality expressions.

The logical AND operator returns the integral value 1 if both operands are nonzero; otherwise, it returns 0. Logical AND has left-to-right associativity.

The operands to the logical AND operator need not be of the same type, but they must be of integral or pointer type. The operands are commonly relational or equality expressions.

The first operand is completely evaluated and all side effects are completed before continuing evaluation of the logical AND expression.

The second operand is evaluated only if the first operand evaluates to true (nonzero). This short-circuit evaluation eliminates needless evaluation of the second operand when the logical AND expression has already been determined false. Short-circuit evaluation can be used to prevent null-pointer dereferencing, as shown in the following example:

```
char *pch = 0;
...
(pch) && (*pch = 'a');
```

If `pch` is null (0), the right side of the expression is never evaluated. Therefore, the assignment through a null pointer is impossible.

logical-OR
logical-NOT

Logical OR Operator

Operator ||

Syntax *logical-OR-expression* :
logical-AND-expression
logical-OR-expression **||** *logical-AND-expression*

The logical operators, logical AND (&&) and logical OR (**||**), are used to combine multiple conditions formed using relational or equality expressions.

The logical OR operator (**||**) returns the integral value 1 if either operand is nonzero; otherwise, it returns 0. Logical OR has left-to-right associativity.

The operands to the logical OR operator need not be of the same type, but they must be of integral or pointer type. The operands are commonly relational or equality expressions.

The first operand is completely evaluated and all side effects are completed before continuing evaluation of the logical OR expression.

The second operand is evaluated only if the first operand evaluates to false (0). This short-circuit evaluation eliminates needless evaluation of the second operand when the logical OR expression has already been determined true.

logical-AND
logical-NOT

Sequential-Evaluation Operator

Operator **,**

Syntax *expression* :
assignment-expression
expression **,** *assignment-expression*

The sequential-evaluation operator, also called the comma operator, evaluates its two operands sequentially from left to right.

The left operand of the sequential-evaluation operator is evaluated as a void expression. The result of the operation has the same value and type as the right operand. Each operand can be of any type. The sequential-evaluation operator does not perform type conversions between its operands, and it does not yield an l-value. There is a sequence point after the first operand, which means all side effects from the evaluation of the left operand are completed before beginning evaluation of the right operand.

The sequential-evaluation operator is typically used to evaluate two or more expressions in contexts where only one expression is allowed.

Commas can be used as separators in some contexts. However, you must be careful not to confuse the use of the comma as a separator with its use as an operator; the two uses are completely different.

Logical NOT Operator

Operator !

The logical-negation (logical-NOT) operator produces the value 0 if its operand is true (nonzero) and the value 1 if its operand is false (0). The result has int type. The operand must be an integral, floating, or pointer value.

The following examples illustrate the unary arithmetic operators (+, –, ~, and !):

```
short x = 987;
    x = -x;
```

In the example above, the new value of x is the negative of 987, or 987.

```
unsigned short y = 0xAAAA;
    y = ~y;
```

In this example, the new value assigned to y is the ones complement of the unsigned value 0xAAAA, or 0x5555.

```
if( !(x < y) )
```

If x is greater than or equal to y, the result of the expression is 1 (true). If x is less than y, the result is 0 (false).

Unary arithmetic operations on pointers are illegal.

logical-AND
logical-OR

Base Operator (Microsoft Specific)

Operator :>

The base operator (**:>**) is a Microsoft extension added to support based addressing. The base operator combines a memory segment address with an address that can be dereferenced with the indirection (*****) operator. It has the form

segvar **:>** *offset*

The *segvar* can be any expression that evaluates to a valid memory segment address. You can use a variable or expression of type __segment, or create your own segments using the __segname operator. The *offset* can be a pointer with the form

type __based(void) *

or any 16-bit expression.

Conditional Operator

Operator ? :

Syntax *conditional expression* :
logical-OR-expression
logical-OR-expression **?** *expression* **:** *conditional-expression*

Example

```
(val >= 0) val ? -val :
```

If the condition is true, the expression evaluates to val. If not, the expression equals −val.

Function Call Operator

Operator ( )

Syntax *postfix-expression* :
postfix-expression **(** [*argument-expression-list*] **)**

argument-expression-list :
assignment-expression
argument-expression-list **,** *assignment-expression*

A function call is an expression that includes the name of the function being called or the value of a function pointer and, optionally, the arguments being passed to the function.

The *postfix-expression* must evaluate to a function address (for example, a function identifier or the value of a function pointer), and *argument-expression-list* is a list of expressions (separated by commas) whose values (the arguments) are passed to the function. The *argument-expression-list* argument can be empty.

A function-call expression has the value and type of the functions return value. A function cannot return an object of array type. If the functions return type is void (that is, the function has been declared never to return a value), the function-call expression also has void type.

Array Element

Operator [ ]

A postfix expression followed by an expression in square brackets (**[ ]**) is a subscripted representation of an element of an array object. A subscript expression represents the value at the address that is *expression* positions beyond *postfix-expression* when expressed as

postfix-expression [*expression*]

Usually, the value represented by *postfix-expression* is a pointer value, such as an array identifier, and *expression* is an integral value. However, all that is required syntactically is that one of the expressions be of pointer type and the other be of integral type. Thus the integral value could be in the *postfix-expression* position and the pointer value could be in the brackets in the *expression*, or subscript, position.

A subscript expression can also have multiple subscripts, as follows:

expression1 [*expression2*] [*expression3*]...

Subscript expressions associate from left to right. The leftmost subscript expression, *expression1*[*expression2*], is evaluated first. The address that results from adding *expression1* and *expression2* forms a pointer expression; then *expression3* is added to this pointer expression to form a new pointer expression, and so on until the last subscript expression has been added. The indirection operator (*****) is applied after the last subscripted expression is evaluated, unless the final pointer value addresses an array type (see examples below).

Expressions with multiple subscripts refer to elements of multidimensional arrays. A multidimensional array is an array whose elements are arrays. For example, the first element of a three-dimensional array is an array with two dimensions.

Structure or Union Member

Operators **.** and **->**

Syntax *postfix-expression* **.** *identifier*

postfix-expression -> *identifier*

A member-selection expression refers to members of structures and unions. Such an expression has the value and type of the selected member.

This list describes the two forms of the member-selection expressions:

1. In the first form, *postfix-expression* represents a value of struct or union type, and *identifier* names a member of the specified structure or union. The value of the operation is that of *identifier* and is an l-value if *postfix-expression* is an l-value.
2. In the second form, *postfix-expression* represents a pointer to a structure or union, and *identifier* names a member of the specified structure or union. The value is that of *identifier* and is an l-value.

The two forms of member-selection expressions have similar effects.

In fact, an expression involving the member-selection operator (->) is a shorthand version of an expression using the period (**.**) if the expression before the period consists of the indirection operator (*****) applied to a pointer value. Therefore,

expression -> *identifier*

is equivalent to

(*expression**) .** *identifier*

when *expression* is a pointer value.

Type Cast

Operator (*type*)

Syntax *cast-expression* :
unary-expression
(*type-name*) *cast-expression*

A type cast provides a method for explicit conversion of the type of an object in a specific situation.

The compiler treats *cast-expression* as type *type-name* after a type cast has been made. Casts can be used to convert objects of any scalar type to or from any other scalar type. Explicit type casts are constrained by the same rules that determine the effects of implicit conversions. Additional restraints on casts may result from the actual sizes or representation of specific types.

Scope Resolution Operator

Operator ::

In C++, you can tell the compiler to use the global variable rather than the local variable by prefixing the variable with **::**, the scope resolution operator. For example:

```
// Scope resolution operator
#include <iostream.h>
int amount = 123; // A global variable
void main()
{
    int amount = 456;    //A local variable
    cout << ::amount;    ??Print the global variable
    cout << '\n';
    cout << amount;      //Print the local variable
```

The example has two variables named amount. The first is global and contains the value 123. The second is local to the main function. The two colons tell the compiler to use the global `amount` instead of the local one.

If you have nested local scopes, the scope resolution operator doesn't provide access to variables in the next outermost scope. It provides access to only the global variables.

Pointer to Member Operators

Operator **.*** and **->***

Syntax *pm-expression* :

cast-expression

pm-expression **.*** *cast-expression*

pm-expression **->*** *cast-expression*

The binary operator .* combines its first operand, which must be an object of class type, with its second operand, which must be a pointer-to-member type.

The binary operator **->*** combines its first operand, which must be a pointer to an object of class type, with its second operand, which must be a pointer-to-member type.

In an expression containing the .* operator, the first operand must be of the class type of the pointer to member specified in the second operand or of a type unambiguously derived from that class.

In an expression containing the **->*** operator, the first operand must be of the type "pointer to the class type" of the type specified in the second operand, or it must be of a type unambiguously derived from that class.

_REGS Union; _WORDREGS, _BYTEREGS Structures

Include <dos.h> or <bios.h>

Context _int86, _intdos

Union

```
union _REGS            // Union of either byte or word registers
   struct _WORDREGS x;
   struct _BYTEREGS h;
};
```

Structure

```
struct _WORDREGS       // Word registers and carry flag of 8088
{
   unsigned int ax;
   unsigned int bx;
   unsigned int cx;
   unsigned int dx;
   unsigned int si;
   unsigned int di;
   unsigned int cflag;
};
struct _BYTEREGS       // Byte registers of 8088
{
   unsigned char al, ah;
   unsigned char bl, bh;
   unsigned char cl, ch;
   unsigned char dl, dh;
};
```

_SREGS Structure

Include <dos.h> or <bios.h>

Context _int86x, _intdosx, _segread

Structure

```
struct _SREGS          // Segment registers of 8088
{
   unsigned int es;
   unsigned int cs;
   unsigned int ss;
   unsigned int ds;
};
```

_complex, _complexl Structures

Include <math.h>

Context _cabs, _cabsl

Structure:

```
struct _complex
{
   double x;           // Real component
   double y;           // Imaginary component
}

struct _complexl
{
   long double x;      // Real component
   long double y;      // Imaginary component
};
```

_diskfree_t Structure

Include <dos.h>

Context _dos_getdiskfree

Structure

```
struct _diskfree_t
{
   unsigned total_clusters;          // Clusters per drive
   unsigned avail_clusters;          // Available clusters
   unsigned sectors_per_cluster;     // Sectors per cluster
   unsigned bytes_per_sector;        // Bytes per sector
};
```

_diskinfo_t Structure

Include <bios.h>

Context _bios_disk

Structure

```
struct _diskinfo_t
{
   unsigned drive;      // Drive: 0-0x7f floppy, 0x80=0xff fixed disk
   unsigned head;       // Head number
   unsigned track;      // Track number
   unsigned sector;     // Start sector number
   unsigned nsectors;   // Number of sectors to read, write,
                        //   or compare
   void __far *buffer;  // Memory location to write to, read from,
                        //   or compare
};
```

div_t, ldiv_t Structures

Include <stdlib.h>

Context div, ldiv

Structure

```
typedef struct
{
   int quot;          // Quotient
   int rem;           // Remainder
} div_t;

typedef struct
{
   long quot;         // Quotient
   long rem;          // Remainder
}; ldiv_t
```

_DOSERROR Structure

Include <dos.h>

Context _dosexterr

Structure

```
struct _DOSERROR
{
   int exterror;            // Extended error code
   char class;              // Error class
   char action;             // Recommended recovery action
   char locus;              // Location of error
};
```

_dostime_t Structure

Include <dos.h>

Context _dos_gettime, _dos_settime

Structure

```
struct _dostime_t
{
                              // Current system time
   unsigned char hour;        // Hour: 0-23
   unsigned char minute;      // Minute: 0-59
   unsigned char second;      // Second: 0-59
   unsigned char second;      // 1/100 second: 0-99
};
```

_dosdate_t Structure

Include <dos.h>

Context _dos_getdate, _dos_setdate

Structure

```
struct _dosdate_t
{                            // Current date structure
   unsigned char day;        // Day of the month: 1-31
   unsigned char month;      // Month of the year: 1-12
   unsigned int year;        // Year: 0-119 relative to 1980
   unsigned char dayofweek;  // Day of the week: 0-6 (Sunday is 0)
};
```

_exception, _exceptionl Structures

Include <math.h>

Context _matherr, _matherrl

Structure

```
struct _exception
{
   int type;            // Exception type
   char name*;          // Name of function where error occurred
   double arg1;         // First argument to function
   double arg2;         // Second argument (if any) to function
   double retval;       // Value to be returned by function
};

struct _exceptionl
{
   int type;            // Exception type
   char *name;          // Name of function where error occurred
   long double arg1;    // First argument to function
   long double arg2;    // Second argument (if any) to function
   long double retval;  // Value to be returned by function
};
```

_find_t Structure

Include <dos.h>

Context _dos_findfirst

Structure

```
struct _find_t
{
                        // File information return structure
   char reserved[21];   // Reserved for use by DOS
   char attrib;         // Attribute for matched path: Attributes
   unsigned wr_time;    // Time of last file write:    Time Format
   unsigned wr_date;    // Date of last file write:    Date Format
   long size;           // Length of file in bytes
   char name[13];       // Null-terminated name of matched
                        //   file/directory, without the path
};
```

_HEAPINFO Structure

Include <malloc.h>

Context _heapwalk

Structure

```
typedef struct
{
   int __far *_pentry;       // Address of entry
   size_t _size;             // Size of entry
   int _useflag;             // Flag for whether entry is in use
} _HEAPINFO;
```

lconv Structure

Include <locale.h>

Context localeconv

Structure

```
struct lconv
{
   char *decimal_point;      // Nonmonetary decimal-point character
   char *thousands_sep;      // Nonmonetary digit-group separator
   char *grouping;           // Nonmonetary digit-group size
   char *int_curr_symbol;    // International currency symbol
   char *currency_symbol;    // Current locale currency symbol
   char *mon_decimal_point;  // Monetary decimal-point character
   char *mon_thousands_sep;  // Monetary digit-group separator
   char *mon_grouping;       // Monetary digit-group size
   char *positive_sign;      // Nonnegative monetary sign
   char *negative_sign;      // Negative monetary sign
   char int_frac_digits;     // International monetary fractional
                             //   digits
   char frac_digits;         // Monetary fractional digits
   char p_cs_precedes;       // Nonnegative currency-symbol placement
   char p_sep_by_space;      // Nonnegative currency-symbol separator
   char n_cs_precedes;       // Negative currency-symbol placement
   char n_sep_by_space;      // Negative currency-symbol separator
   char p_sign_posn;         // Sign placement in nonnegative monetary
                             //   value
   char n_sign_posn;         // Sign placement in negative monetary
                             //   value
};
```

Note The char * members of the struct are pointers to strings. Any of these (other than char *decimal_point) that equals "" is either of zero length or is not supported in the current locale. The char members of the struct are nonnegative numbers. Any of these that equals CHAR_MAX is not supported in the current locale.

The values for grouping and mon_grouping are interpreted according to the following rules:

Value	Interpretation
CHAR_MAX	No further grouping is to be performed.
0	The previous element is to be repeatedly used for the remainder of the digits.

n	The integer value n is the number of digits that makes up the current group. The next element is examined to determine the size of the next group of digits before the current group.

The values for int_curr_symbol are interpreted according to the following rules:

- The first three characters specify the alphabetic international currency symbol as defined in the ISO 4217 Codes for the Representation of Currency and Funds standard.
- The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

The values for p_cs_precedes and n_cs_precedes are interpreted according to the following rules (the n_cs_precedes rule is in parentheses):

Value	Interpretation
0	The currency symbol follows the value for a nonnegative (negative) formatted monetary value.
1	The currency symbol precedes the value for a nonnegative (negative) formatted monetary value.

The values for p_sep_by_space and n_sep_by_space are interpreted according to the following rules (the n_sep_by_space rule is in parentheses):

Value	Interpretation
0	The currency symbol is separated from the value by a space for a nonnegative (negative) formatted monetary value.
1	There is no space separation between the currency symbol and the value for a nonnegative (negative) formatted monetary value.

The values for p_sign_posn and n_sign_posn are interpreted according to the following rules:

Value	Interpretation
0	Parentheses surround the quantity and currency symbol.
1	The sign string precedes the quantity and currency symbol.
2	The sign string follows the quantity and currency symbol.
3	The sign string immediately precedes the currency symbol.
4	The sign string immediately follows the currency symbol.

_stat Structure

Include <sys\stat.h>

Context _stat, _fstat

Structure

```
struct _stat
{
   dev_t st_dev;             // Drive number or device handle
   ino_t st_ino;             // Not used in DOS
   unsigned short st_mode;   // Bit mask for file mode information
   short st_nlink;           // Always 1
   short st_uid;             // Not used in DOS
   short st_gid;             // Not used in DOS
   dev_t st_rdev;            // Same as st_dev
   off_t st_size;            // Size in bytes
   time_t st_atime;          // Time of last access on HPFS files
   time_t st_mtime;          // Time of last modification
   time_t st_ctime;          // Time of creation on HPFS files
};
```

Fields st_atime, st_mtime, and st_ctime always have the same value (time of last modification) in the DOS File Allocation Table (FAT) system.

_timeb Structure

Include <sys\timeb.h>

Context _ftime

Structure

```
struct _timeb
{
   time_t time;              // Time in seconds since
                             //    midnight, 1/1/70
   unsigned short millitm;   // Fraction in milliseconds
   short timezone;           // Time zone relative to UCT
   short dstflag;            // Daylight savings flag
};
```

tm Structure

Include <time.h>

Context asctime, localtime, gmtime, mktime, strftime

Structure

```
struct tm
{
   int tm_sec;    // Seconds after the minute - [0,59]
   int tm_min;    // Minutes after the hour - [0,59]
   int tm_hour;   // Hours since midnight - [0,23]
   int tm_mday;   // Day of the month - [1,31]
   int tm_mon;    // Months since January - [0,11]
   int tm_year;   // Years since 1900
   int tm_wday;   // Days since Sunday - [0,6]
   int tm_yday;   // Days since January 1 - [0,365]
   int tm_isdst;  // Daylight-saving-time flag
};
```

_utimbuf

Include <sys\utime.h>

Context _utime

Structure

```
struct _utimbuf
{
   time_t actime;     // Access time
   time_t modtime;    // Modification time
};
```

_wopeninfo Structure

Include <io.h>

Context _wopeninfo

Structure

```
struct _wopeninfo
{
   unsigned int _version;     // Windows version; use _QWINVER
   const char __far *_title;  // Window title, null-terminated string
   long _wbufsize;            // Screen buffer size
};
```

_wziseinfo Structure

Include <io.h>

Context _wsizeinfo

Structure

```
struct _wsizeinfo
{
   unsigned int _version;    // Windows version; use _QWINVER
   unsigned int _type;       // Kind of sizing for window
   unsigned int _x;          // Upper left window corner x coordinate
   unsigned int _y;          // Upper left window corner y coordinate
   unsigned int _h;          // Window height
   unsigned int _w;          // Window width
};
```

ANSI Character Set (Character Codes 0–255)

Dec	Hex	Char	Code†
0	00	€	NUL
1	01	€	SOH
2	02	€	STX
3	03	€	ETX
4	04	€	EOT
5	05	€	ENQ
6	06	€	ACK
7	07	€	BEL
8	08	€	BS
9	09	€	HT
10	0A	€	LF
11	0B	€	VT
12	0C	€	FF
13	0D	€	CR
14	0E	€	SO
15	0F	€	SI
16	10	€	SLE
17	11	€	CS1
18	12	€	DC2
19	13	€	DC3
20	14	€	DC4
21	15	€	NAK
22	16	€	SYN
23	17	€	ETB
24	18	€	CAN
25	19	€	EM
26	1A	€	SIB
27	1B	€	ESC
28	1C	€	FS
29	1D	€	GS
30	1E		RS
31	1F	€	US
32	20	(space)	
33	21	!	

34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E

70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i

106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	72	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	□
128	80	€
129	81	□
130*	82	‚
131*	83	*ƒ*
132*	84	„
133*	85	…
134*	86	†
135*	87	‡
136*	88	ˆ
137*	89	‰
138*	8A	Š
139*	8B	‹
140*	8C	Œ
141	8D	□

142	8E	Ž
143	8F	□
144	90	□
145	91	‘
146	92	’
147*	93	“
148*	94	”
149*	95	•
150*	96	–
151*	97	—
152*	98	˜
153*	99	™
154*	9A	š
155*	9B	›
156*	9C	œ
157	9D	□
158	9E	ž
159*	9F	Ÿ
160	A0	
161	A1	¡
162	A2	¢
163	A3	£
164	A4	¤
165	A5	¥
166	A6	¦
167	A7	§
168	A8	¨
169	A9	©
170	AA	ª
171	AB	«
172	AC	¬
173	AD	
174	AE	®
175	AF	¯
176	B0	°
177	B1	±

178	B2	²
179	B3	³
180	B4	´
181	B5	µ
182	B6	¶
183	B7	·
184	B8	¸
185	B9	¹
186	BA	º
187	BB	»
188	BC	¼
189	BD	½
190	BE	¾
191	BF	¿
192	C0	À
193	C1	Á
194	C2	Â
195	C3	Ã
196	C4	Ä
197	C5	Å
198	C6	Æ
199	C7	Ç
200	C8	È
201	C9	É
202	CA	Ê
203	CB	Ë
204	CC	Ì
205	CD	Í
206	CE	Î
207	CF	Ï
208	D0	Ð
209	D1	Ñ
210	D2	Ò
211	D3	Ó
212	D4	Ô
213	D5	Õ

214	D6	Ö
215	D7	×
216	D8	Ø
217	D9	Ù
218	DA	Ú
219	DB	Û
220	DC	Ü
221	DD	Ý
222	DE	Þ
223	DF	ß
224	E0	à
225	E1	á
226	E2	â
227	E3	ã
228	E4	ä
229	E5	å
230	E6	æ
231	E7	ç
232	E8	è
233	E9	é
234	EA	ê
235	EB	ë
236	EC	ì
237	ED	í
238	EE	î
239	EF	ï
240	F0	ð
241	F1	ñ
242	F2	ò
243	F3	ó
244	F4	ô
245	F5	õ
246	F6	ö
247	F7	÷
248	F8	ø
249	F9	ù
250	FA	ú

251	FB	û
252	FC	ü
253	FD	ý
254	FE	þ
255	FF	ÿ

□ Indicates that this character is not supported by Windows.

* Indicates that this character is available only in TrueType fonts.

† The "Code" column is meaningful only for characters 1–31.

For more information, see Appendix E in the *C++ Language Reference*.

Key Scan Codes

Key	Scan Code	
	Dec	**Hex**
ESC	1	01
1!	2	02
2@	3	03
3#	4	04
4$	5	05
5%	6	06
6^	7	07
7&	8	08
8*	9	09
9(	10	0A
0)	11	0B
-_	12	0C
=+	13	0D
BKSP	14	0E
TAB	15	0F
Q	16	10
W	17	11
E	18	12
R	19	13
T	20	14
Y	21	15
U	22	16
I	23	17
O	24	18
P	25	19
[{	26	1A
]}	27	1B
ENTER	28	1C
ENTER£	28	1C
L CTRL	29	1D
R CTRL	29	1D
A	30	1E
S	31	1F
D	32	20

F	33	21
G	34	22
H	35	23
J	36	24
K	37	25
L	38	26
;:	39	27
' "	40	28
`~	41	29
L SHIFT	42	2A
|	43	2B
Z	44	2C
X	45	2D
C	46	2E
V	47	2F
B	48	30
N	49	31
M	50	32
,<	51	33
.>	52	34
/?	53	35
GRAY /£	53	35
R SHIFT	54	36
*PRTSC	55	37
L ALT	56	38
R ALT£	56	38
SPACE	57	39
CAPS	58	3A
F1	59	3B
F2	60	3C
F3	61	3D
F4	62	3E
F5	63	3F
F6	64	40
F7	65	41
F8	66	42
F9	67	43

F10	68	44
F11£	87	57
F12£	88	58
NUM	69	45
SCROLL	70	46
HOME	71	47
HOME£	71	47
UP	72	48
UP£	72	48
PGUP	73	49
PGUP£	73	49
GRAY-	74	4A
LEFT	75	4B
LEFT£	75	4B
CENTER	76	4C
RIGHT	77	4D
RIGHT£	77	4D
GRAY+	78	4E
END	79	4F
END£	79	4F
DOWN	80	50
DOWN£	80	50
PGDN	81	51
PGDN£	81	51
INS	82	52
INS£	82	52
DEL	83	53
DEL£	83	53

£ These keys are only available on extended keyboards. Most are in the Cursor/Control cluster. If the raw scan code is read from the keyboard port (60H), it appears as two bytes (E0H) followed by the normal scan code. However, when the keypad ENTER and / keys are read through the BIOS interrupt 16H, only E0H is seen since the interrupt only gives one-byte scan codes.

For more information, see Appendix E in the *C++ Language Reference*.

class filebuf : public streambuf

Hierarchy

Members

The filebuf class is a derived class of streambuf that is specialized for buffered disk file I/O. The buffering is managed entirely within the Microsoft iostream Class Library. filebuf member functions call the run-time low-level I/O routines (the functions declared in IO.H) such as _sopen, _read, and _write.

The file stream classes, ofstream, ifstream, and fstream, use filebuf member functions to fetch and store characters. Some of these member functions are virtual functions defined for the streambuf class.

The reserve area, put area, and get area are introduced in the streambuf class description. The put area and the get area are always the same for filebuf objects. Also, the get pointer and put pointers are tied; when one moves so does the other.

#include <fstream.h>

ifstream
ofstream
streambuf
strstreambuf
stdiobuf

fileBuf Class Members _ All Public

	Member	Description
Construction/Destruction		
	filebuf	Constructs a filebuf object.
	~filebuf	Destroys a filebuf object.
Operations		
	open	Opens a file and attaches it to the filebuf object.
	close	Flushes any waiting output and closes the attached file.
	setmode	Sets the file's mode to binary or text.
	attach	Attaches the filebuf object to an open file.
Status/Information		
	fd	Returns the stream's file descriptor.
	is_open	Tests whether the file is open.

filebuf::attach

Syntax filebuf* attach(filedesc *fd*);

Hierarchy

Parameters

fd

A file descriptor as returned by a call to the run-time function _open or _sopen. filedesc is a typedef equivalent to int.

This function attaches this filebuf object to the open file specified by *fd*.

Return Value NULL when the stream is already attached to a file; otherwise the address of the **filebuf** object.

filebuf::close

Syntax: filebuf* close( );

This function flushes any waiting output, closes the file, and disconnects the file from the filebuf object. If an error occurs, the function returns NULL and leaves the filebuf object in a closed state. If there is no error, the function returns the address of the filebuf object and clears its error state.

filebuf::open

filebuf::fd

Syntax filedesc fd() const;

Returns the file descriptor associated with the filebuf object; filedesc is a typedef equivalent to int. Its value is supplied by the underlying file system. The function returns EOF if the object is not attached to a file.

filebuf::attach

filebuf::filebuf

Syntax filebuf( );

filebuf(filedesc *fd*);

filebuf(filedesc *fd*, char* *pr*, int *nLength*);

Parameters

fd

A file descriptor as returned by a call to the run-time function _sopen. filedesc is a typedef equivalent to int.

pr

Pointer to a previously allocated reserve area of length *nLength*.

nLength

The length (in bytes) of the reserve area.

The three filebuf constructors are described as follows:

Constructor	Description
filebuf()	Constructs a filebuf object without attaching it to a file.
filebuf(filedesc)	Constructs a filebuf object and attaches it to an open file.
filebuf(filedesc, char*, int)	Constructs a filebuf object, attaches it to an open file, and initializes it to use a specified reserve area.

filebuf::~filebuf

Syntax ~filebuf( );

Closes the attached file only if that file was opened by the open member function.

filebuf::is_open

Syntax int is_open( ) const;

Returns a nonzero value if this filebuf object is attached to an open disk file identified by a file descriptor; otherwise 0.

filebuf::open

filebuf::open

Syntax filebuf* open(const char* *szName*, int *nMode*, int *nProt* = filebuf::openprot);

Parameters

szName

The name of the file to be opened during construction.

nMode

An integer containing mode bits defined as ios enumerators that can be combined with the OR (**|**) operator. See the ofstream constructor for a list of the enumerators.

nProt

The file protection specification; defaults to the static integer filebuf::openprot, which is equivalent to the operating system default (filebuf::sh_compat for MS-DOS). The possible *nProt* values are as follows:

Value	Meaning
filebuf::sh_compat	Compatibility share mode (for MS-DOS only).
filebuf::sh_none	Exclusive mode--no sharing.
filebuf::sh_read	Read sharing allowed.
filebuf::sh_write	Write sharing allowed.

The filebuf::sh_read and filebuf::sh_write modes can be combined with the logical OR (|) operator.

Filebuf::open opens a disk file and attaches it with this filebuf object. If the file is already open, or if there is an error while opening the file, the function returns NULL; otherwise it returns the filebuf address.

filebuf::is_open
filebuf::close
filebuf::~filebuf

filebuf::setmode

Syntax int setmode(int *nMode* = filebuf::text);

Parameters

nMode

An integer that must be one of the static **filebuf** constants, as follows:

Value	Meaning
filebuf::text	Text mode (newline characters translated to and from carriage return-linefeed pairs under MS-DOS).
filebuf::binary	Binary mode (no translation).

This function sets the binary/text mode of the stream's filebuf object.

Return Value The previous mode if there is no error; otherwise 0.

ios binary manipulator
ios text manipulator

class fstream : public iostream

The fstream class is an iostream derivative specialized for combined disk file input and output. Its constructors automatically create and attach a filebuf buffer object.

The filebuf class documentation describes the get and put areas and their associated pointers. Although the filebuf object's get and put pointers are theoretically independent, the get area and the put area cannot both be active at the same time. When the streams mode changes from input to output, the get area is emptied and the put area is reinitialized. When the mode changes from output to input, the put area is flushed and the get area is reinitialized. Thus, either the get pointer or the put pointer is null at all times.

#include <fstream.h>

ifstream
ofstream
strstream
stdiostream
filebuf

fstream Class Members _ All Public

	Member	Description
Construction/Destruction		
	fstream	Constructs an fstream object.
	~fstream	Destroys an fstream object.
Operations		
	open	Opens a file and attaches it to the filebuf object and thus to the stream.
	close	Flushes any waiting output and closes the stream's file.
	setbuf	Attaches the specified reserve area to the stream's filebuf object.
	setmode	Sets the stream's mode to binary or text.
	attach	Attaches the stream (through the filebuf object) to an open file.
Status/Information		
	rdbuf	Gets the stream's filebuf object.
	fd	Returns the file descriptor associated with the stream.
	is_open	Tests whether the stream's file is open.

fstream::attach

Syntax void attach(filedesc *fd*);

Parameters

fd

A file descriptor as returned by a call to the run-time function _open or _sopen; filedesc is a typedef equivalent to int.

Attaches this stream to the open file specified by *fd*. The functions fails when the stream is already attached to a file. In that case, the function sets ios::failbit in the stream's error state.

filebuf::attach
fstream::fd

fstream::close

Syntax void close( );

Calls the close member function for the associated filebuf object. This function, in turn, flushes any waiting output, closes the file, and disconnects the file from the filebuf object. The filebuf object is not destroyed.

The stream's error state is cleared unless the call to filebuf::close fails.

filebuf::close
fstream::open
fstream::is_open

fstream::fd

Syntax filedesc fd( ) const;

Returns the file descriptor associated with the stream. filedesc is a typedef equivalent to int. Its value is supplied by the underlying file system.

filebuf::fd
fstream::attach

fstream::fstream

Syntax fstream( );

fstream(const char* *szName*, int *nMode*, int *nProt* = filebuf::openprot);

fstream(filedesc *fd*);

fstream(filedesc *fd*, char* *pch*, int *nLength*);

Parameters

szName

The name of the file to be opened during construction.

nMode

An integer that contains mode bits defined as ios enumerators that can be combined with the bitwise-OR (|) operator:

Value	Meaning
ios::app	The function performs a seek to the end of file. When new bytes are written to the file, they are always appended to the end, even if the position is moved with the ostream::seekp function.
ios::ate	The function performs a seek to the end of file. When the first new byte is written to the file, it is appended to the end, but when subsequent bytes are written, they are written to the current position.
ios::in	The file is opened for input. The original file (if it exists) will not be truncated.
ios::out	The file is opened for output.
ios::trunc	If the file already exists, its contents are discarded. This mode is implied if ios::out is specified and ios::ate, ios::app, and ios:in are not specified.
ios::nocreate	If the file does not already exist, the function fails.
ios::noreplace	If the file already exists, the function fails.
ios::binary	Opens the file in binary mode (the default is text mode).

Note that there is no ios::in or ios::out default mode for fstream objects. You must specify both modes if your fstream object must both read and write files.

nProt

The file protection specification; defaults to the static integer filebuf::openprot that is equivalent to the operating system default, which is filebuf::sh_compat under MS-DOS. The possible *nProt* values are as follows:

Value	Meaning
filebuf::sh_compat	Compatibility share mode (MS-DOS only).
filebuf::sh_none	Exclusive mode--no sharing.
filebuf::sh_read	Read sharing allowed.
filebuf::sh_write	Write sharing allowed.

The filebuf::sh_read and filebuf::sh_write modes can be combined with the logical OR (|) operator.

fd

A file descriptor as returned by a call to the run-time function _open or _sopen. filedesc is a typedef equivalent to int.

pch

Pointer to a previously allocated reserve area of length *nLength*. A NULL value (or *nLength* = 0) indicates that the stream will be unbuffered.

nLength

The length (in bytes) of the reserve area (0 = unbuffered).

The four fstream constructors are described as follows:

Constructor	Description
fstream()	Constructs an fstream object without opening a file.
fstream(const char*, int, int)	Contructs an fstream object, opening the specified file.
fstream(filedesc)	Constructs an fstream object that is attached to an open file.
fstream(filedesc, char*, int)	Constructs an fstream object that is associated with a filebuf object. The filebuf object is attached to an open file and to a specified reserve area.

All fstream constructors construct a filebuf object. The first three use an internally allocated reserve area, but the fourth uses a user-allocated area. The user-allocated area is not automatically released during destruction.

fstream::~fstream

Syntax ~fstream( );

Flushes the buffer, then destroys an fstream object, along with its associated filebuf object. The file is closed only if it was opened by the constructor or by the open member function.

The filebuf destructor releases the reserve buffer only if it was internally allocated.

fstream::is_open

Syntax int is_open( ) const;

Returns a nonzero value if this stream is attached to an open disk file identified by a file descriptor; otherwise 0.

filebuf::is_open
fstream::open
fstream::close

fstream::open

Syntax void open(const char* *szName*, int *nMode*, int *nProt* = filebuf::openprot);

Parameters

szName

The name of the file to be opened during construction.

nMode

An integer containing mode bits defined as ios enumerators that can be combined with the OR (|) operator. See the fstream constructor for a list of the enumerators. A valid mode must be specified (no default).

nProt

The file protection specification; defaults to the static integer filebuf::openprot. See the fstream constructor for a list of the other allowed values.

This function opens a disk file and attaches it to the stream's filebuf object. If the filebuf object is already attached to an open file, or if a filebuf call fails, the ios::failbit is set. If the file is not found, then the ios::failbit is set only if the ios::nocreate mode was used.

filebuf::open
fstream::fstream
fstream::close
fstream::is_open

fstream::rdbuf

Syntax filebuf* rdbuf( ) const;

Returns a pointer to the filebuf buffer object that is associated with this stream. (Note that this is not the character buffer; the filebuf object contains a pointer to the character area.)

fstream::setbuf

Syntax streambuf* setbuf(char* *pch*, int *nLength*);

Parameters

pch

A pointer to a previously allocated reserve area of length *nLength*. A NULL value indicates an unbuffered stream.

nLength

The length (in bytes) of the reserve area. A length of 0 indicates an unbuffered stream.

Attaches the specified reserve area to the stream's filebuf object. If the file is open and a buffer has already been allocated, the function returns NULL; otherwise it returns a pointer to the filebuf cast as a streambuf. The reserve area will not be released by the destructor.

fstream::setmode

Syntax int setmode(int *nMode* = filebuf::text);

Parameters

nMode

An integer that must be one of the static filebuf constants, as follows:

Value	Meaning
filebuf::text	Text mode (newline characters translated to and from carriage return\linefeed pairs).
filebuf::binary	Binary mode (no translation).

This function sets the binary/text mode of the stream's filebuf object. It may be called only after the file is opened.

Return Value The previous mode; -1 if the parameter is invalid, the file is not open, or the mode cannnot be changed.

ios binary manipulator
ios text manipulator

class ifstream : public istream

The ifstream class is an istream derivative specialized for disk file input. Its constructors automatically create and attach a filebuf buffer object.

The filebuf class documentation describes the get and put areas and their associated pointers. Only the get area and the get pointer are active for the ifstream class.

#include <fstream.h>

filebuf
streambuf
ofstream
fstream

ifstream Class Members _ All Public

	Member	Description
Construction/Destruction		
	ifstream	Constructs an ifstream object.
	~ifstream	Destroys an ifstream object.
Operations		
	open	Opens a file and attaches it to the filebuf object and thus to the stream.
	close	Closes the stream's file.
	setbuf	Associates the specified reserve area to the stream's filebuf object.
	setmode	Sets the stream's mode to binary or text.
	attach	Attaches the stream (through the filebuf object) to an open file.
Status/Information		
	rdbuf	Gets the stream's filebuf object.
	fd	Returns the file descriptor associated with the stream.
	is_open	Tests whether the stream's file is open.

ifstream::attach

Syntax void attach(filedesc *fd*);

Parameters

fd

A file descriptor as returned by a call to the run-time function _open or _sopen; filedesc is a typedef equivalent to int.

Attaches this stream to the open file specified by *fd*. The function fails when the stream is already attached to a file. In that case, the function sets ios::failbit in the stream's error state.

filebuf::attach
ifstream::fd

ifstream::close

Syntax void close( );

Calls the close member function for the associated filebuf object. This function, in turn, closes the file and disconnects the file from the filebuf object. The filebuf object is not destroyed.

The stream's error state is cleared unless the call to filebuf::close fails.

filebuf::close
ifstream::open
ifstream::is_open

ifstream::fd

Syntax filedesc fd( ) const;

Returns the file descriptor associated with the stream; filedesc is a typedef equivalent to int. Its value is supplied by the underlying file system.

filebuf::fd
ifstream::attach

ifstream::ifstream

Syntax ifstream( );

ifstream(const char* *szName*, int *nMode* = ios::in, int *nProt* = filebuf::openprot);

ifstream(filedesc *fd*);

ifstream(filedesc *fd*, char* *pch*, int *nLength*);

Parameters

szName

The name of the file to be opened during construction.

nMode

An integer that contains mode bits defined as ios enumerators that can be combined with the bitwise-OR (|) operator:

Value	Meaning
ios::in	The file is opened for input (default).
ios::nocreate	If the file does not already exist, the function fails.
ios::binary	Opens the file in binary mode (the default is text mode).

Note that the ios::nocreate flag is necessary if you intend to test for the file's existence (the usual case).

nProt

The file protection specification; defaults to the static integer filebuf::openprot that is equivalent to filebuf::sh_compat. The possible *nProt* values are as follows:

Value	Meaning
filebuf::sh_compat	Compatibility share mode.
filebuf::sh_none	Exclusive mode _ no sharing.
filebuf::sh_read	Read sharing allowed.
filebuf::sh_write	Write sharing allowed.

The filebuf::sh_read and filebuf::sh_write modes can be combined with the logical OR (|) operator.

fd

A file descriptor as returned by a call to the run-time function _open or _sopen; filedesc is a typedef equivalent to int.

pch

Pointer to a previously allocated reserve area of length *nLength*. A NULL value (or *nLength* = 0) indicates that the stream will be unbuffered.

nLength

The length (in bytes) of the reserve area (0 = unbuffered).

The four ifstream constructors are described as follows:

Constructor	Description
ifstream( )	Constructs an

	ifstream object without opening a file.
ifstream(const char*, int, int)	Contructs an ifstream object, opening the specified file.
ifstream(filedesc)	Constructs an ifstream object that is attached to an open file.
ifstream(filedesc, char*, int)	Constructs an ifstream object that is associated with a filebuf object. The filebuf object is attached to an open file and to a specified reserve area.

All ifstream constructors construct a filebuf object. The first three use an internally allocated reserve area, but the fourth uses a user-allocated area.

ifstream::~ifstream

Syntax ~ifstream( );

Destroys an ifstream object along with its associated filebuf object. The file is closed only if was opened by the constructor or by the open member function.

The filebuf destructor releases the reserve buffer only if it was internally allocated.

ifstream::is_open

Syntax int is_open( ) const;

Returns a nonzero value if this stream is attached to an open disk file identified by a file descriptor; otherwise 0.

filebuf::is_open
ifstream::open
ifstream::close

ifstream::open

Syntax void open(const char* *szName*, int *nMode* = ios::in, int *nProt* = filebuf::openprot);

Parameters

szName

The name of the file to be opened during construction.

nMode

An integer containing bits defined as ios enumerators that can be combined with the OR (|) operator. See the ifstream constructor for a list of the enumerators. The ios::in mode is implied.

nProt

The file protection specification; defaults to the static integer filebuf::openprot. See the ifstream constructor for a list of the other allowed values.

Opens a disk file and attaches it to the stream's filebuf object. If the filebuf object is already attached to an open file, or if a filebuf call fails, the ios::failbit is set. If the file is not found, then the ios::failbit is set only if the ios::nocreate mode was used.

filebuf::open
ifstream::ifstream
ifstream::close
ifstream::is_open

ifstream::rdbuf

Syntax filebuf* rdbuf( ) const;

Returns a pointer to the filebuf buffer object that is associated with this stream. (Note that this is not the character buffer; the filebuf object contains a pointer to the character area.)

ifstream::setbuf

Syntax streambuf* setbuf(char* *pch*, int *nLength*);

Parameters

pch

A pointer to a previously allocated reserve area of length *nLength*. A NULL value indicates an unbuffered stream.

nLength

The length (in bytes) of the reserve area. A length of 0 indicates an unbuffered stream.

Attaches the specified reserve area to the stream's filebuf object. If the file is open and a buffer has already been allocated, the function returns NULL; otherwise it returns a pointer to the filebuf, which is cast as a streambuf. The reserve area will not be released by the destructor.

ifstream::setmode

Syntax int setmode(int *nMode* = filebuf::text);

Parameters

nMode

An integer that must be one of the static filebuf constants, as follows:

Value	Meaning
filebuf::text	Text mode (newline characters translated to and from carriage return-linefeed pairs).
filebuf::binary	Binary mode (no translation).

This function sets the binary/text mode of the stream's filebuf object. It may be called only after the file is opened.

Return Value The previous mode; -1 if the parameter is invalid, the file is not open, or the mode cannnot be changed.

ios binary manipulator
ios text manipulator

class ios

As the iostream class hierarchy diagram shows, the ios class is the base class for all the input/output stream classes. While ios is not technically an abstract base class, you will not usually construct ios objects, nor will you derive classes directly from ios. Instead, you will use the derived classes istream and ostream or other derived classes.

Even though you will not use ios directly, you will be using many of the inherited member functions and data members described here. Remember that these inherited member function descriptions are not duplicated for derived classes.

#include <iostream.h>

istream
ostream

ios Class Members _ All Public

	Member	Description
Data Members (static)		
	basefield	Mask for obtaining the conversion base flags (dec, oct, or hex).
	adjustfield	Mask for obtaining the field padding flags (left, right, or internal).
	floatfield	Mask for obtaining the numeric format (scientific or fixed).
Construction/Destruction		
	ios	Constructor for use in derived classes.
	~ios	Virtual destructor.
Flag and Format Access Functions		
	flags	Sets or reads the stream's format flags.
	setf	Manipulates the stream's format flags.
	unsetf	Clears the stream's format flags.
	fill	Sets or reads the stream's fill character.
	precision	Sets or reads the stream's floating-point format display precision.
	width	Sets or reads the stream's output field width.
Status-Testing Functions		
	good	Indicates good stream status.

bad	Indicates a serious I/O error.
eof	Indicates end of file.
fail	Indicates a serious I/O error or a possibly recoverable I/O formatting error.
rdstate	Returns the stream's error flags.
clear	Sets or clears the stream's error flags.

User-Defined Format Flags

bitalloc	Provides a mask for an unused format bit in the stream's private flags variable (static function).
xalloc	Provides an index to an unused word in an array reserved for special-purpose stream state variables (static function).
iword	Converts the index provided by xalloc to a reference (valid only until the next xalloc).
pword	Converts the index provided by xalloc to a pointer (valid only until the next xalloc).

Other Functions

delbuf	Controls the connection of streambuf deletion with ios destruction.
rdbuf	Gets the stream's streambuf object.

	sync_with_stdio	Synchronizes the predefined objects cin, cout, cerr, and clog with the standard I/O system.
	tie	Ties a specified ostream to this stream.

Operators

	operator void*()	Converts a stream to a pointer that can be used only for error checking.
	operator !()	Returns a nonzero value if a stream I/O error has occurred.

ios Manipulators

	dec	Causes the interpretation of subsequent fields in decimal format (the default mode).
	hex	Causes the interpretation of subsequent fields in hexadecimal format.
	oct	Causes the interpretation of subsequent fields in octal format.
	binary	Sets the stream's mode to binary (stream must have an associated filebuf buffer).
	text	Sets the stream's mode to text--the default mode (stream must have an associated

	filebuf buffer).

Parameterized Manipulators (#include <iomanip.h> required)

setiosflags	Sets the stream's format flags.
resetiosflags	Resets the stream's format flags.
setfill	Sets the stream's fill character.
setprecision	Sets the stream's floating-point display precision.
setw	Sets the stream's field width (for the next field only).

ios Class Member Categories _ All Public Members

- Data Members (static)
- Construction/Destruction
- Flag and Format Access Functions
- Status-Testing Functions
- User-Defined Format Flags
- Other Functions
- Operators
- ios Manipulators
- Parameterized Manipulators

ios::bad

Syntax int bad( ) const;

Returns a nonzero value to indicate a serious I/O error. This condition corresponds to the badbit error state being set. Do not continue I/O operations on the stream in this situation.

ios::good
ios::fail
ios::rdstate

ios::bitalloc

Syntax static long bitalloc( );

The ios class currently defines 15 format flag bits accessible through flags and other member functions. These bits reside in a 32-bit private ios data member and are accessed through enumerators such as ios::left and ios::hex.

The bitalloc member function provides a mask for a previously unused bit in the data member. Once you obtain the mask, you can use it to set or test the corresponding custom flag bit in conjunction with the ios member functions and manipulators listed under "See Also."

ios::flags
ios::setf
ios::unsetf
ios setiosflags manipulator
ios resetiosflags manipulator

ios::clear

Syntax void clear(int *nState* = 0);

Parameters

nState

If 0, all error bits are cleared; otherwise bits are set according to the following masks (ios enumerators) that can be combined using the bitwise-OR (|) operator:

Value	Meaning
ios::goodbit	No error condition (no bits set)
ios::eofbit	End of file reached
ios::failbit	A possibly recoverable formatting or conversion error
ios::badbit	A severe I/O error

This function sets or clears the error-state flags. The rdstate function can be used to read the current error state.

ios::rdstate
ios::good
ios::bad
ios::eof

ios::delbuf

Syntax void delbuf(int *nDelFlag*);

int delbuf( ) const;

Parameters

nDelFlag

A nonzero value indicates that ~ios should delete the stream's attached streambuf object. A 0 value prevents deletion.

The first overloaded delbuf function assigns a value to the stream's buffer-deletion flag.

The second function returns the current value of the flag.

This function is public only because it is accessed by the Iostream_init class. Treat it as protected.

ios::rdbuf
ios::~ios

ios::eof

Syntax int eof( ) const;

This functions returns a nonzero value if end of file has been reached. This condition corresponds to the eofbit error flag being set.

ios::fail

Syntax int fail( ) const;

This function returns a nonzero value if any I/O error (not end of file) has occurred. This condition corresponds to either the badbit or failbit error flag being set. If a call to bad returns 0, you can assume that the error condition is nonfatal and that you can probably continue processing after you clear the flags.

ios::bad
ios::clear

ios::fill

Syntax char fill(char *cFill*);

char fill( ) const;

Parameters

cFill

The new fill character to be used as padding between fields.

The first overloaded function sets the stream's internal fill character variable to *cFill* and returns the previous value. The default fill character is a space.

The second fill function returns the stream's fill character.

ios setfill manipulator

ios::flags

Syntax long flags(long *lFlags*);

long flags( ) const;

Parameters

lFlags

The new format flag values for the stream. The values are specified by the following bit masks (**ios** enumerators) that can be combined using the bitwise-OR (|) operator:

Value	Meaning
ios::skipws	Skip white space on input.
ios::left	Left-align values; pad on the right with the fill character.
ios::right	Right-align values; pad on the left with the fill character (default alignment).
ios::internal	Add fill characters after any leading sign or base indication, but before the value.
ios::dec	Format numeric values as base 10 (decimal) (default radix).
ios::oct	Format numeric values as base 8 (octal).
ios::hex	Format numeric values as base 16 (hexadecimal).
ios::showbase	Display numeric constants in a format that can be read by the C++ compiler.
ios::showpoint	Show decimal point and trailing zeros for floating-point values.
ios::uppercase	Display uppercase A through F for hexadecimal values and E for scientific values.
ios::showpos	Show plus signs (+) for positive values.
ios::scientific	Display floating-point numbers in scientific format.
ios::fixed	Display floating-point numbers in fixed format.
ios::unitbuf	Cause ostream::osfx to flush the stream after each insertion. By default, cerr is unit buffered.
ios::stdio	Cause ostream::osfx to flush stdout and stderr after each insertion.

The first overloaded flags function sets the stream's internal flags variable to *lFlags* and returns the previous value.

The second function returns the stream's current flags.

ios::setf
ios::unsetf
ios setiosflags manipulator
ios resetiosflags manipulator
ios::adjustfield
ios::basefield
ios::floatfield

ios::good

Syntax int good( ) const;

This function returns a nonzero value if all error bits are clear. Note that the good member function is not simply the inverse of the bad function.

ios::bad
ios::fail
ios::rdstate

ios::init

Syntax Protected:
void init(streambuf* *psb*);

Parameters

psb
A pointer to an existing streambuf object.

This function associates an object of a streambuf-derived class with this stream and, if necessary, deletes a dynamically created stream buffer object that was previously associated. The init function is useful in derived classes in conjunction with the protected default istream, ostream, and iostream constructors. An ios-derived class constructor can thus construct and attach its own pretermined stream buffer object.

istream::istream
ostream::ostream
iostream::iostream

ios::ios

Syntax ios(streambuf* *psb*);

Parameters

psb
A pointer to an existing streambuf object.

This function is the constructor for ios. You will seldom need to invoke this constructor except in derived classes. Generally, you will be deriving classes not from ios but from istream, ostream, and iostream.

ios::~ios

Syntax virtual ~ios( );

This function is the virtual destructor for **ios**.

ios::iword

Syntax long& iword(int *nIndex*) const;

Parameters

nIndex

An index into a table of words that are associated with the ios object.

The xalloc member function provides the index into the table of special-purpose words. The pword function converts that index to a reference to a 32-bit word.

ios::xalloc
ios::pword

ios::precision

Syntax int precision(int *np*);

int precision( ) const;

Parameters

np

An integer that indicates the number of significant digits or significant decimal digits to be used for floating-point display.

The first overloaded precision function sets the stream's internal floating-point precision variable to *np* and returns the previous value. The default precision is six digits. If the display format is scientific or fixed, then the precision indicates the number of digits after the decimal point. If the format is automatic (neither floating point nor fixed), then the precision indicates the total number of significant digits.

The second function returns the stream's current precision value.

ios setprecision manipulator

ios::pword

Syntax void*& pword(int *nIndex*) const;

Parameters

nIndex

An index into a table of words that are associated with the ios object.

The xalloc member function provides the index into the table of special-purpose words. The pword function converts that index to a reference to a pointer to a 32-bit word.

ios::xalloc
ios::iword

ios::rdbuf

Syntax streambuf* rdbuf( ) const;

This function returns a pointer to the streambuf object that is associated with this stream. The rdbuf function is useful when you need to call streambuf member functions.

ios::rdstate

Syntax int rdstate( ) const;

This function returns the current error state as specified by the following masks (ios enumerators):

Value	Meaning
ios::goodbit	No error condition
ios::eofbit	End of file reached
ios::failbit	A possibly recoverable formatting or conversion error
ios::badbit	A severe I/O error or unknown state

The returned value can be tested against a mask with the AND (&) operator.

ios::clear

ios::setf

Syntax long setf(long *lFlags*);

long setf(long *lFlags*, long *lMask*);

Parameters

lFlags

Format flag bit values. See the flags member function for a list of format flags. These flags can be combined by using the bitwise-OR (|) operator.

lMask

Format flag bit mask.

The first overloaded setf function turns on only those format bits that are specified by 1s in *lFlags*. It returns a long that contains the previous value of all the flags.

The second function alters those format bits specified by 1s in *lMask*. The new values of those format bits are determined by the corresponding bits in *lFlags*. It returns a long that contains the previous value of all the flags.

ios::flags
ios::unsetf
ios setiosflags manipulator

ios::sync_with_stdio

Syntax static void sync_with_stdio( );

This function synchronizes the C++ streams with the standard I/O system. The first time this function is called, it resets the predefined streams (cin, cout, cerr, clog) to use a stdiobuf object rather than a filebuf object. After that, I/O using these streams can be freely mixed with I/O using stdin, stdout, and stderr. Some performance decrease will result because there is buffering both in the stream class and in the standard I/O file system.

After the call to sync_with_stdio, the ios::stdio bit is set for all affected predefined stream objects, and cout is set to unit buffered mode.

ios::tie

Syntax ostream* tie(ostream* *pos*);

ostream* tie( ) const;

Parameters

pos

A pointer to an ostream object.

The first overloaded tie function ties this stream to the specified ostream and returns the value of the previous tie pointer (NULL if this stream was not previously tied). A stream tie enables automatic flushing of the ostream in response to (1) a need for more characters or (2) the presence of characters to be consumed.

By default, cin is initially tied to cout so that attempts to get more characters from standard input may result in flushing standard output. In addition, cerr and clog are tied to cout by default.

The second function returns the value of the previous tie pointer (NULL if this stream was not previously tied).

ios::unsetf

Syntax long unsetf(long *lFlags*);

Parameters

lFlags

Format flag bit values. See the flags member function for a list of format flags.

This function clears the format flags specified by 1s in *lFlags*. It returns a long that contains the previous value of all the flags.

ios::flags
ios::setf
ios resetiosflags manipulator

ios::width

Syntax int width(int *nw*);

int width( ) const;

Parameters

nw

The minimum field width in characters.

The first overloaded width function sets the stream's internal field width variable to *nw*. When the width is 0 (the default), inserters insert only as many characters as necessary to represent the inserted value. When the width is not 0, the inserters pad the field with the stream's fill character, up to *nw*. If the unpadded representation of the field is larger than *nw*, the field is not truncated. Thus *nw* is a minimum field width.

The internal width value is reset to 0 after each insertion or extraction.

The second overloaded width function returns the current value of the stream's width variable.

ios setw manipulator

ios::xalloc

Syntax static int xalloc( );

This function provides extra ios object state variables without the need for class derivation. It does so by returning an index to an unused 32-bit word in an internal array. This index can be subsequently converted into a reference or pointer by using the iword or pword member functions.

Any call to xalloc invalidates values returned by previous calls to iword and pword.

ios::iword
ios::pword

ios::operator void* ()

Syntax operator void* ( ) const;

An operator that converts a stream to a pointer that can be compared to 0. The conversion returns 0 if either failbit or badbit is set in the stream's error state. See rdstate for a description of the error state masks. A nonzero pointer is not meant to be dereferenced.

ios::good
ios::fail

ios::operator ! ()

Syntax int operator !( ) const;

Returns a nonzero value if either failbit or badbit are set in the stream's error state. See rdstate for a description of the error state masks.

ios::good
ios::fail

ios::adjustfield

Syntax static const long adjustfield;

A mask that can be used to obtain the padding flag bits (left, right, or internal).

```
extern ostream os;
if( ( os.flags() & ios::adjustfield ) == ios::left ) .....
```

ios::flags

ios::basefield

Syntax static const long basefield;

A mask that can be used to obtain the current radix flag bits (dec, oct, or hex).

```
extern ostream os;
if( ( os.flags() & ios::basefield ) == ios::hex ) .....
```

ios::flags

ios::floatfield

Syntax static const long floatfield;

This function is a mask that can be used to obtain floating-point format flag bits (scientific or fixed).

```
extern ostream os;
if( ( os.flags() & ios::floatfield ) == ios::scientific ) .....
```

ios::flags

ios& binary

Syntax binary

Sets the stream's mode to binary. The default mode is text.

The stream must have an associated filebuf buffer.

ios text manipulator
ofstream::setmode
ifstream::setmode
filebuf::setmode

ios& dec

#include <iostream.h>

Syntax dec

This function sets the format conversion base to 10 (decimal).

ios hex manipulator
ios oct manipulator

ios& hex

Syntax hex

This function sets the format conversion base to 16 (hexadecimal).

ios dec manipulator
ios oct manipulator

ios& oct

Syntax oct

This function sets the format conversion base to 8 (octal).

ios dec manipulator
ios hex manipulator

resetiosflags

#include <iomanip.h>

Syntax SMANIP(long) resetiosflags(long *lFlags*);

Parameters

lFlags

Format flag bit values. See the flags member function for a list of format flags. These flags can be combined by using the OR (|) operator.

This parameterized manipulator clears only the specified format flags. This setting remains in effect until the next change.

setfill

#include <iomanip.h>

Syntax SMANIP(int) setfill(int *nFill*);

Parameters

nFill

The new fill character to be used as padding between fields.

This parameterized manipulator sets the stream's fill character. The default is a space. This setting remains in effect until the next change.

setiosflags

Syntax SMANIP(long) setiosflags(long *lFlags*);

Parameters

lFlags

Format flag bit values. See the flags member function for a list of format flags. These flags can be combined by using the OR (|) operator.

This parameterized manipulator sets only the specified format flags. This setting remains in effect until the next change.

setprecision

#include <iomanip.h>

Syntax SMANIP(int) setprecision(int *np*);

Parameters

np

An integer that indicates the number of significant digits or significant decimal digits to be used for floating-point display.

This parameterized manipulator sets the stream's internal floating-point precision variable to *np*. The default precision is six digits. If the display format is scientific or fixed, then the precision indicates the number of digits after the decimal point. If the format is automatic (neither floating point nor fixed), then the precision indicates the total number of significant digits.

This setting remains in effect until the next change.

setw

#include <iomanip.h>

Syntax SMANIP(int) setw(int *nw*);

Parameters

nw
The field width in characters.

This parameterized manipulator sets the stream's internal field width parameter. See the width member function for more information. This setting remains in effect only for the next insertion.

ios& text

Syntax text

This function sets the stream's mode to text (the default mode).

The stream must have an associated filebuf buffer.

ios binary manipulator
ofstream::setmode
ifstream::setmode
filebuf::setmode

class iostream : public istream, public ostream

The iostream class provides the basic capability for sequential and random-access I/O. It inherits functionality from both the istream and ostream classes.

The iostream class works in conjunction with classes derived from streambuf (for example, filebuf). In fact, most of the iostream "personality" comes from its attached streambuf class. You can use iostream objects for sequential disk I/O if you first construct an appropriate filebuf object. More often, you will use objects of classes fstream and strstream.

Derivation

For derivation suggestions, see the istream and ostream classes.

#include <iostream.h>

istream
ostream
fstream
strstream
stdiostream

iostream Class

	Member	Description
Public Members		
	iostream	Constructs an iostream object that is attached to an existing streambuf object.
	~iostream	Destroys an iostream object.
Protected Members		
	iostream	Acts as a void-argument iostream constructor.

iostream::iostream

Syntax

Public:
iostream(streambuf* *psb*);

Protected
iostream( );

Parameter

psb
A pointer to an existing streambuf object (or an object of a derived class).

ios::init

iostream::~iostream

Syntax virtual ~iostream( );

Virtual destructor for the iostream class.

class Iostream_init

The Iostream_init class is a static class that initializes the predefined stream objects cin, cout, cerr, and clog. A single object of this class is constructed "invisibly" in response to the reference of any of the predefined objects. The class is documented for completeness only. You will not normally construct objects of this class

#include <iostream.h>

Iostream_init Class Members _ All Public

Member	Description
Iostream_init	A constructor that initializes cin, cout, cerr, and clog.
~Iostream_init	The destructor for the Iostream_init class.

Iostream_init::Iostream_init

Syntax Iostream_init( );

Iostream_init constructor that initializes cin, cout, cerr, and clog. For internal use only.

Iostream_init::~Iostream_init

Syntax ~Iostream_init( );

Iostream_init destructor. For internal use only.

class istream : virtual public ios

The istream class provides the basic capability for sequential and random-access input. An istream object has a streambuf-derived object attached, and the two classes work together; the istream class does the formatting, and the streambuf class does the low-level buffered input.

You can use istream objects for sequential disk input if you first construct an appropriate filebuf object. More often, you will use the predefined stream object cin (which is actually an object of class istream_withassign), or you will use objects of classes ifstream (disk file streams) and istrstream (string streams).

Derivation

It is not always necessary to derive from istream in order to add functionality to a stream; consider deriving from streambuf instead, as illustrated in Chapter 1 of the *iostream Class Library Reference*. The ifstream and istrstream classes are examples of istream-derived classes that construct member objects of predetermined derived streambuf classes.

You can add manipulators without deriving a new class.

If you add new extraction operators for a derived istream class, then the rules of C++ dictate that you must reimplement all the base class extraction operators. See the "Derivation" section of the class ostream overview for an efficient reimplementation technique.

#include <iostream.h>

streambuf
ifstream
istrstream
istream_withassign

istream Class Members

	Member	Description
Construction/Destruction		
	istream	Constructs an istream object attached to an existing object of a streambuf-derived class.
	~istream	Destroys an istream object.
Prefix/Suffix Functions		
	ipfx	Check for error conditions prior to extraction operations (input prefix function).
	isfx	Called after extraction operations (input suffix function).
Input Functions		
	get	Extracts characters from the stream up to, but not including, delimiters.
	getline	Extracts characters from the stream (extracts and discards delimiters).
	read	Extracts data from the stream.
	ignore	Extracts and discards characters.
	peek	Returns a character without extracting it from the stream.
	gcount	Counts the characters extracted in the last unformatted operation.
	eatwhite	Extracts leading white space.
Other Functions		

	putmfc	Puts characters back to the stream.
	sync	Synchronizes the stream buffer with the external source of characters.
	seekg	Changes the stream's get pointer.
	tellg	Gets the value of the stream's get pointer.
Operators		
	operator >>	Extraction operator for various types.
Protected Members		
	istream	Constructs an istream object.
Manipulators		
	ws	Extracts leading white space.

istream Class Member Categories

- Construction/Destruction
- Prefix/Suffix Functions
- Input Functions
- Other Functions
- Operators
- Protected Members
- Manipulators

istream::eatwhite

Syntax void eatwhite( );

Extracts white space from the stream by advancing the get pointer past spaces and tabs.

istream ws manipulator

istream::gcount

Syntax int gcount( ) const;

Returns the number of characters extracted by the last unformatted input function. Formatted extraction operators may call unformatted input functions and thus reset this number.

istream::get
istream::getline
istream::ignore
istream::read

istream::get

Syntax int get( );

istream& get(char* *pch*, int *nCount*, char *delim* = '\n');

istream& get(unsigned char* *puch*, int *nCount*, char *delim* = '\n');

istream& get(signed char* *psch*, int *nCount*, char *delim* = '\n');

istream& get(char& *rch*);

istream& get(unsigned char& *ruch*);

istream& get(signed char& *rsch*);

istream& get(streambuf& *rsb*, char *delim* = '\n');

Parameters

pch, *puch*, *psch*
A pointer to a character array.

nCount
The maximum number of characters to store, including the terminating NULL.

delim
The delimiter character (defaults to newline).

rch, *ruch*, *rsch*
A reference to a character.

rsb
A reference to an object of a streambuf-derived class.

These functions extract data from an input stream as follows:

Variation	Description
get();	Extracts a single character from the stream and returns it.
get(char*, int, char);	Extracts characters from the stream until either *delim* is found, the limit *nCount* is reached, or the end of file is reached. The characters are stored in the array followed by a null terminator.
get(char&);	Extracts a single character from the stream and stores it as specified by the reference argument.
get(streambuf&, char);	Gets characters from the stream and stores them in a

	streambuf object until the delimiter is found or the end of the file is reached. The ios::failbit flag is set if the streambuf output operation fails.

In all cases, the delimiter is neither extracted from the stream nor returned by the function. The getline function, in contrast, extracts the delimiter but does not store it.

istream::getline
istream::read
istream::ignore
istream::gcount

istream::getline

Syntax istream& getline(char* *pch*, int *nCount*, char *delim* = '\n');
istream& getline(unsigned char* *puch*, int *nCount*, char *delim* = '\n');
istream& getline(signed char* *psch*, int *nCount*, char *delim* = '\n');

Parameters

pch, *puch*, *psch*
A pointer to a character array.

nCount
The maximum number of characters to store, including the terminating NULL.

delim
The delimiter character (defaults to newline).

This function extracts characters from the stream until either the delimiter *delim* is found, the limit *nCount*-1 is reached, or end of file is reached. The characters are stored in the specified array followed by a null terminator. If the delimiter is found, it is extracted but not stored.

The get function, in contrast, neither extracts nor stores the delimiter.

istream::get
istream::read

istream::ignore

Syntax istream& ignore(int *nCount* = 1, int *delim* = EOF);

Parameters

nCount

The maximum number of characters to extract.

delim

The delimiter character (defaults to EOF).

Extracts and discards up to *nCount* characters. Extraction stops if the delimiter *delim* is extracted or the end of file is reached. If *delim* = EOF (the default), then only the end of file condition causes termination. The delimiter character is extracted.

istream::ipfx

Syntax int ipfx(int *need* = 0);

Parameter

need

Zero if called from formatted input functions; otherwise the minimum number of characters needed.

This input prefix function is called by input functions prior to extracting data from the stream. Formatted input functions call ipfx(0), while unformatted input functions usually call ipfx(1).

Any ios object tied to this stream is flushed if *need* = 0 or if there are fewer than *need* characters in the input buffer. Also, ipfx extracts leading white space if ios::skipws is set.

Return Value A nonzero return value if the operation was successful; 0 if the stream's error state is nonzero, in which case the function does nothing.

istream::isfx

istream::isfx

Syntax void isfx( );

This input suffix function is called at the end of every extraction operation.

istream::istream

Syntax Public:
istream(streambuf* *psb*);

Protected:
istream( );

Parameters

psb
A pointer to an existing object of a streambuf-derived class.

Constructs an object of type istream.

ios::init

istream::~istream

Syntax virtual ~istream( );

Virtual destructor for the istream class.

istream::peek

Syntax int peek( );

Returns the next character without extracting it from the stream. Returns EOF if the stream is at end of file or if the ipfx function indicates an error.

istream::putmfc

Syntax istream& putmfc(char *ch*);

Parameter

ch

The character to put back; must be the character previously extracted.

Puts a character back into the input stream. The putmfc function may fail and set the error state. If *ch* does not match the character that was previously extracted, then the result is undefined.

istream::read

Syntax istream& read(char* *pch*, int *nCount*);

istream& read(unsigned char* *puch*, int *nCount*);

istream& read(signed char* *psch*, int *nCount*);

Parameters

pch, *puch*, *psch*

A pointer to a character array.

nCount

The maximum number of characters to read.

Extracts bytes from the stream until the limit *nCount* is reached or until the end of file is reached. The read function is useful for binary stream input.

istream::get
istream::getline
istream::gcount
istream::ignore

istream::seekg

Syntax istream& seekg(streampos *pos*);

istream& seekg(streamoff *off*, ios::seek_dir *dir*);

Parameters

pos

The new position value; streampos is a typedef equivalent to long.

off

The new offset value; streamoff is a typedef equivalent to long.

dir

The seek direction. Must be one of the following enumerators:

Value	Meaning
ios::beg	Seek from the beginning of the stream.
ios::cur	Seek from the current position in the stream.
ios::end	Seek from the end of the stream.

Changes the get pointer for the stream. Not all derived classes of istream need support positioning; it is most often used with file-based streams.

istream::tellg
ostream::seekp
ostream::tellp

istream::sync

Syntax int sync( );

Synchronizes the stream's internal buffer with the external source of characters. This function calls the virtual streambuf::sync function so you can customize its implementation by deriving a new class from streambuf.

Return Value EOF to indicate errors.

streambuf::sync

istream::tellg

Syntax streampos tellg( );

Gets the value for the stream's get pointer.

Return Value A streampos type, corresponding to a long.

istream::seekg
ostream::tellp
ostream::seekp

istream::operator >>

Syntax istream& operator >>(char* *psz*);
istream& operator >>(unsigned char* *pusz*);
istream& operator >>(signed char* *pssz*);
istream& operator >>(char& *rch*);
istream& operator >>(unsigned char& *ruch*);
istream& operator >>(signed char& *rsch*);
istream& operator >>(short& *s*);
istream& operator >>(unsigned short& *us*);
istream& operator >>(int& *n*);
istream& operator >>(unsigned int& *un*);
istream& operator >>(long& *l*);
istream& operator >>(unsigned long& *ul*);
istream& operator >>(float& *f*);
istream& operator >>(double& *d*);
istream& operator >>(long double& *ld*); (16-bit only)
istream& operator >>(streambuf* *psb*);
istream& operator >>(istream& (**fcn*)(istream&));
istream& operator >>(ios& (**fcn*)(ios&));

These overloaded operators extract their argument from the stream. The last two variations allow the use of manipulators that are defined for both istream and ios.

istream& ws

Syntax ws

Extracts leading white space from the stream by calling the eatwhite function.

istream::eatwhite

class istream_withassign : public istream

The istream_withassign class is a variant of istream that allows object assignment. The predefined object cin is an object of this class and thus may be reassigned at run time to a different istream object.

A program that normally expects input from stdin, for example, could be temporarily directed to accept its input from a disk file.

Predefined Objects

The cin object is a predefined object of class ostream_withassign. It is connected to stdin (standard input, file descriptor 0).

The objects cin, cerr, and clog are tied to cout so that use of any of these may cause cout to be flushed.

#include <iostream.h>

ostream_withassign

istream_withassign Class Members _ All Public

	Member	Description
Construction/Destruction		
	istream_withassign	Constructs an istream_withassign object.
	~istream_withassign	Destroys an istream_withassign object.
Operators		
	operator =	Indicates an assignment operator.

istream_withassign::istream_withassign

Syntax istream_withassign(streambuf* *psb*);

istream_withassign( );

Parameters

psb

A pointer to an existing object of a streambuf-derived class.

The first constructor creates a ready-to-use object of type istream_withassign, complete with attached streambuf object.

The second constructor creates an object but does not initialize it. You must subsequently use the second variation of the istream_withassign assignment operator to attach the streambuf object, or you must use the first variation to initialize this object to match the specified istream object.

istream_withassign::operator =

istream_withassign::~istream_withassign

Syntax ~istream_withassign( );

Destructor for the istream_withassign class.

istream_withassign::operator =

Syntax istream& operator =(const istream& *ris*);

istream& operator =(streambuf* *psb*);

The first overloaded assignment operator assigns the specified istream object to this istream_withassign object.

The second operator attaches a streambuf object to an existing istream_withassign object, and it initializes the state of the istream_withassign object. This operator is often used in conjunction with the void-argument constructor.

Example 1

```
char buffer[100];
class xistream; // A special-purpose class derived from istream
extern xistream xin; // An xistream object constructed elsewhere

cin = xin; // cin is reassigned to xin
cin >> buffer; // xin used instead of cin
```

Example 2

```
char buffer[100];
extern filedesc fd; // A file descriptor for an open file
filebuf fb( fd ); // Construct a filebuf attached to fd

cin = &fb;     // fb associated with cin
cin >> buffer; // cin now gets its intput from the fb file
```

istream_withassign::istream_withassign
cin

class istrstream : public istream

The istrstream class supports input streams that have character arrays as a source. You must allocate a character array prior to the construction of an istrstream object. All the istream operators and functions (including seeking) can then be used on this character data.

You must be aware that there is a get pointer working behind the scenes in the attached strstreambuf class. This pointer advances as you extract fields from the stream's array. The only way you can make it go backwards is to use the istream::seekg function. If the get pointer reaches the end of the string (and sets the ios::eof flag), then you must call clear before seekg.

#include <strstrea.h>

strstreambuf
streambuf
strstream
ostrstream

istrstream Class Members _ All Public

	Member	Description
Construction/Destruction		
	istrstream	Constructs an istrstream object.
	~istrstream	Destroys an istrstream object.
Other Functions		
	rdbuf	Returns a pointer to the stream's associated strstreambuf object.
	str	Returns a character array pointer to the string stream's contents.

istrstream::istrstream

Syntax istrstream(char* *psz*);

istrstream(char* *pch*, int *nLength*);

Parameters

psz

A null-terminated character array (string).

pch

A character array that is not necessarily null terminated.

nLength

The size (in characters) of *pch*. If 0, then *pch* is assumed to point to a null-terminated array; if less than 0, then the array is assumed to have unlimited length.

The first constructor uses the specified *psz* buffer to make an istrstream object with length corresponding to the string length.

The second constructor makes an istrstream object out of the first *nLength* characters of the *pch* buffer.

Both constructors automatically construct a strstreambuf object that manages the specified character buffer.

istrstream::~istrstream

Syntax ~istrstream( );

Destroys an istrstream object and its associated strstreambuf object. The character buffer is not released because it was allocated by the user prior to istrstream construction.

istrstream::rdbuf

Syntax strstreambuf* rdbuf( ) const;

This function returns a pointer to the strstreambuf buffer object that is associated with this stream. Note that this is not the character buffer itself; the strstreambuf object contains a pointer to the character area.

istrstream::str

istrstream::str

Syntax char* str( );

This function returns a pointer to the string stream's character array. This pointer corresponds to the array used to construct the istrstream object.

istrstream::istrstream

class ofstream : public ostream

The ofstream class is an ostream derivative specialized for disk file output. All of its constructors automatically create and associate a filebuf buffer object.

The filebuf class documentation describes the get and put areas and their associated pointers. Only the put area and the put pointer are active for the ofstream class.

#include <fstream.h>

filebuf
streambuf
ifstream
fstream

ofstream Class Members _ All Public

	Member	Description
Construction/Destruction		
	ofstream	Constructs an ofstream object.
	~ofstream	Destroys an ofstream object.
Operations		
	open	Opens a file and attaches it to the filebuf object and thus to the stream.
	close	Flushes any waiting output and closes the stream's file.
	setbuf	Associates the specified reserve area to the stream's filebuf object.
	setmode	Sets the stream's mode to binary or text.
	attach	Attaches the stream (through the filebuf object) to an open file.
Status/Information		
	rdbuf	Gets the stream's filebuf object.
	fd	Returns the file descriptor associated with the stream.
	is_open	Tests whether the stream's file is open.

ofstream::attach

Syntax void attach(filedesc *fd*);

Parameters

fd

A file descriptor as returned by a call to the run-time function _open or _sopen; filedesc is a typedef equivalent to int.

Attaches this stream to the open file specified by *fd*. The function fails when the stream is already attached to a file. In that case, the function sets ios::failbit in the stream's error state.

filebuf::attach
ofstream::fd

ofstream::close

Syntax void close( );

Calls the close member function for the associated filebuf object. This function, in turn, flushes any waiting output, closes the file, and disconnects the file from the filebuf object. The filebuf object is not destroyed.

The stream's error state is cleared unless the call to filebuf::close fails.

filebuf::close
ofstream::open
ofstream::is_open

ofstream::fd

Syntax filedesc fd( ) const;

Returns the file descriptor associated with the stream. filedesc is a typedef equivalent to int. Its value is supplied by the underlying file system.

filebuf::fd
ofstream::attach

ofstream::is_open

Syntax int is_open( ) const;

Returns a nonzero value if this stream is attached to an open disk file identified by a file descriptor; otherwise 0.

filebuf::is_open
ofstream::open
ofstream::close

ofstream::ofstream

Syntax ofstream( );

ofstream(const char* *szName*, int *nMode* = ios::out, int *nProt* = filebuf::openprot);

ofstream(filedesc *fd*);

ofstream(filedesc *fd*, char* *pch*, int *nLength*);

Parameters

szName

The name of the file to be opened during construction.

nMode

An integer that contains mode bits defined as ios enumerators that can be combined with the bitwise-OR (|) operator:

Value	Meaning
ios::app	The function performs a seek to the end of file. When new bytes are written to the file, they are always appended to the end, even if the position is moved with the ostream::seekp function.
ios::ate	The function performs a seek to the end of file. When the first new byte is written to the file, it is appended to the end, but when subsequent bytes are written, they are written to the current position.
ios::in	If this mode is specified, then the original file (if it exists) will not be truncated.
ios::out	The file is opened for output (implied for all ofstream objects).
ios::trunc	If the file already exists, its contents are discarded. This mode is implied if ios::out is specified and ios::ate, ios::app, and ios:in are not specified.
ios::nocreate	If the file does not already exist, the function fails.
ios::noreplace	If the file already exists, the function fails.
ios::binary	Opens the file in binary mode (the default is text mode).

nProt

The file protection specification; defaults to the static integer filebuf::openprot that is equivalent to filebuf::sh_compat. The possible *nProt* values are:

Value	Meaning
filebuf::sh_compat	Compatibility share mode.
filebuf::sh_none	Exclusive mode; no sharing.
filebuf::sh_read	Read sharing allowed.
filebuf::sh_write	Write sharing allowed.

The filebuf::sh_read and filebuf::sh_write modes can be combined with the logical OR (|) operator.

fd

A file descriptor as returned by a call to the run-time function _open or _sopen. filedesc is a typedef equivalent to int.

pch

Pointer to a previously allocated reserve area of length *nLength*. A NULL value (or *nLength* = 0) indicates that the stream will be unbuffered.

nLength

The length (in bytes) of the reserve area (0 = unbuffered).

The four ofstream constructors are described as follows:

Constructor	Description
ofstream( )	Constructs an ofstream object without opening a file.
ofstream(const char*, int, int)	Contructs an ofstream object, opening the specified file.
ofstream(filedesc)	Constructs an ofstream object that is attached to an open file.
ofstream(filedesc, char*, int)	Constructs an ofstream object that is associated with a filebuf object. The filebuf object is attached to an open file and to a specified reserve area.

All ofstream constructors construct a filebuf object. The first three use an internally allocated reserve area, but the fourth uses a user-allocated area. The user-allocated area is not automatically released during destruction.

ofstream::~ofstream

Syntax ~ofstream();

Flushes the buffer, then destroys an ofstream object along with its associated filebuf object. The file is closed only if was opened by the constructor or by the open member function.

The filebuf destructor releases the reserve buffer only if it was internally allocated.

ofstream::open

Syntax void open(const char* *szName*, int *nMode* = ios::out, int *nProt* = filebuf::openprot);

Parameters

szName

The name of the file to be opened during construction.

nMode

An integer containing mode bits defined as **ios** enumerators that can be combined with the OR (|) operator. See the ofstream constructor for a list of the enumerators. The ios::out mode is implied.

nProt

The file protection specification; defaults to the static integer filebuf::openprot. See the ofstream constructor for a list of the other allowed values.

This functions opens a disk file and attaches it to the stream's filebuf object. If the filebuf object is already attached to an open file, or if a filebuf call fails, the ios::failbit is set. If the file is not found, then the ios::failbit is set only if the ios::nocreate mode was used.

filebuf::open
ofstream::ofstream
ofstream::close
ofstream::is_open

ofstream::rdbuf

Syntax filebuf* rdbuf() const;

Returns a pointer to the filebuf buffer object that is associated with this stream. (Note that this is not the character buffer; the filebuf object contains a pointer to the character area.)

```
extern ofstream ofs;
int fd = ofs.rdbuf()->fd(); // Get the file descriptor for ofs
```

ofstream::setbuf

Syntax streambuf* setbuf(char* *pch*, int *nLength*);

Parameters

pch

A pointer to a previously allocated reserve area of length *nLength*. A NULL value indicates an unbuffered stream.

nLength

The length (in bytes) of the reserve area. A length of 0 indicates an unbuffered stream.

Attaches the specified reserve area to the stream's filebuf object. If the file is open and a buffer has already been allocated, the function returns NULL; otherwise it returns a pointer to the filebuf cast as a streambuf. The reserve area will not be released by the destructor.

ofstream::setmode

Syntax int setmode(int *nMode* = filebuf::text);

Parameters

nMode

An integer that must be one of the static filebuf constants, as follows:

Value	Meaning
filebuf::text	Text mode (newline characters translated to and from carriage return\-linefeed pairs).
filebuf::binary	Binary mode (no translation).

This function sets the binary/text mode of the stream's filebuf object. It may be called only after the file is opened.

Return Value The previous mode; -1 if the parameter is invalid, the file is not open, or the mode cannnot be changed.

ios binary manipulator
ios text manipulator

class ostream : virtual public ios

The ostream class provides the basic capability for sequential and random-access output. An ostream object has a streambuf-derived object attached, and the two classes work together; the ostream class does the formatting, and the streambuf class does the low-level buffered output.

You can use ostream objects for sequential disk output if you first construct an appropriate filebuf object. (The filebuf class is derived from streambuf.) More often, you will use the predefined stream objects cout, cerr, and clog (actually objects of class ostream_withassign), or you will use objects of classes ofstream (disk file streams) and ostrstream (string streams).

All of the ostream member functions write unformatted data; formatted output is handled by the insertion operators.

Derivation

It is not always necessary to derive from ostream to add functionality to a stream; consider deriving from streambuf instead, as illustrated in Chapter 1 of the *iostream Class Library Reference*. The ofstream and ostrstream classes are examples of ostream-derived classes that construct member objects of predetermined derived streambuf classes.

You can add manipulators without deriving a new class.

If you add new insertion operators for a derived ostream class, then the rules of C++ dictate that you must reimplement all the base class insertion operators. If, however, you reimplement the operators through inline equivalence, no extra code will be generated. For example,

```
class xstream : public ostream
{
public:
    // Constructors, etc.
    // ........
    inline xstream& operator << ( char ch ) // insertion for char
    {
          return (xstream&)ostream::operator << ( ch );
    }
    // ........
    // Insertions for other types
};
```

#include <iostream.h>

streambuf
ofstream
ostrstream
cout
cerr
clog

ostream Class Members _ All Public

	Member	Description
Construction/Destruction		
	ostream	Constructs an ostream object that is attached to an existing streambuf object.
	~ostream	Destroys an ostream object.
Prefix/Suffix Functions		
	opfx	Output prefix function, called prior to insertion operations to check for error conditions, and so forth.
	osfx	Output suffix function, called after insertion operations; flushes the stream's buffer if it is unit buffered.
Unformatted Output		
	put	Inserts a single byte into the stream.
	write	Inserts a series of bytes into the stream.
Other Functions		
	flush	Flushes the buffer associated with this stream.
	seekp	Changes the stream's put pointer.
	tellp	Gets the value of the stream's put pointer.
Operators		
	operator <<	An insertion operator for various types.
Manipulators		

endl	Inserts a newline sequence and flushes the buffer.
ends	Inserts a null character to terminate a string.
flush	Flushes the stream's buffer.

ostream Class Member Categories _ All Public Members

- Construction/Destruction
- Prefix/Suffix Functions
- Unformatted Output
- Other Functions
- Operators
- Manipulators

ostream::flush

Syntax ostream& flush( );

Flushes the buffer associated with this stream. The flush function calls the sync function of the associated streambuf.

ostream flush manipulator
streambuf::sync

ostream::opfx

Syntax int opfx( );

This output prefix function is called before every insertion operation. If another ostream object is tied to this stream then the opfx function flushes that stream.

Return Value If the ostream object's error state is not 0, opfx returns 0 immediately; otherwise it returns a nonzero value.

ostream::osfx

Syntax void osfx( );

This output suffix function is called after every insertion operation. It flushes the ostream object if ios::unitbuf is set. It flushes stdout and stderr if ios::stdio is set.

ostream::ostream

Syntax Public:
ostream(streambuf* *psb*);

Protected:
ostream( );


Parameter

psb
A pointer to an existing object of a streambuf-derived class.

This function constructs an object of type ostream.

ios::init

ostream::~ostream

Syntax virtual ~ostream( );

This function destroys an ostream object. The output buffer is flushed as appropriate. The attached streambuf object is destroyed only if it was allocated internally within the ostream constructor.

ostream::put

Syntax ostream& put(char *ch*);

Parameter

ch
The character to insert.

This function inserts a single character into the output stream.

ostream::seekp

Syntax ostream& seekp(streampos *pos*);

ostream& seekp(streamoff *off*, ios::seek_dir *dir*);

Parameters

pos

The new position value; streampos is a typedef equivalent to long.

off

The new offset value; streamoff is a typedef equivalent to long.

dir

The seek direction specified by the enumerated type ios::seek_dir, as follows:

Value	Meaning
ios::beg	Seek from the beginning of the stream.
ios::cur	Seek from the current position in the stream.
ios::end	Seek from the end of the stream.

Changes the position value for the stream. Not all derived classes of ostream need support positioning. For file streams, the position is the byte offset from the beginning of the file; for string streams, it is the byte offset from the beginning of the string.

ostream::tellp
istream::seekg
istream::tellg

ostream::tellp

Syntax streampos tellp( );

This functions gets the position value for the stream. Not all derived classes of ostream need support positioning. For file streams, the position is the byte offset from the beginning of the file; for string streams, it is the byte offset from the beginning of the string. Gets the value for the stream's put pointer.

Return Value A streampos type that corresponds to a long.

ostream::seekp
istream::tellg
istream::seekg

ostream::write

Syntax ostream& write(const char* *pch*, int *nCount*);

ostream& write(const unsigned char* *puch*, int *nCount*);

ostream& write(const signed char* *psch*, int *nCount*);

Parameters

pch, *puch*, *psch*

A pointer to a character array.

nCount

The number of characters to be written.

This functions inserts a specified number of bytes from a buffer into the stream. If the underlying file was opened in text mode, additional carriage return characters may be inserted. The write function is useful for binary stream output.

ostream::operator <<

Syntax ostream& operator <<(char *ch*);
ostream& operator <<(unsigned char *uch*);
ostream& operator <<(signed char *sch*);
ostream& operator <<(const char* *psz*);
ostream& operator <<(const unsigned char **pusz*);
ostream& operator <<(const signed char **pssz*);
ostream& operator <<(short *s*);
ostream& operator <<(unsigned short *us*);
ostream& operator <<(int *n*);
ostream& operator <<(unsigned int *un*);
ostream& operator <<(long *l*);
ostream& operator <<(unsigned long *ul*);
ostream& operator <<(float *f*);
ostream& operator <<(double *d*);
ostream& operator <<(long double *ld*); (16-bit only)
ostream& operator <<(const void* *pv*);
ostream& operator <<(streambuf* *psb*);
ostream& operator <<(ostream& (**fcn*)(ostream&));
ostream& operator <<(ios& (**fcn*)(ios&));

These overloaded operators insert their argument into the stream. The last two variations allow the use of manipulators that are defined for both ostream and ios.

ostream& endl

Syntax endl

This manipulator, when inserted into an output stream, inserts a newline character and then flushes the buffer.

ostream& ends

Syntax ends

This manipulator, when inserted into an output stream, inserts a null-terminator character. It is particularly useful for ostrstream objects.

ostream& flush

Syntax flush

This manipulator, when inserted into an output stream, flushes the output buffer by calling the streambuf::sync member function.

ostream::flush
streambuf::sync

class ostream_withassign : public ostream

The ostream_withassign class is a variant of ostream that allows object assignment. The predefined objects cout, cerr, and clog are objects of this class and thus may be reassigned at run time to a different ostream object.

A program that normally sends output to stdout, for example, could be temporarily directed to send its output to a disk file.

Predefined Objects

There are three predefined objects of class ostream_withassign. They are connected as follows:

cout
Standard output (file descriptor 1).

cerr
Unit buffered standard error (file descriptor 2).

clog
Fully buffered standard error (file descriptor 2).

Unit buffering, as used by cerr, means that characters are flushed after each insertion operation. The objects cin, cerr, and clog are tied to cout so that use of any of these will cause cout to be flushed.

#include <iostream.h>

istream_withassign

ostream_withassign Class Members _ All Public

	Member	Description
Construction/Destruction		
	ostream_withassign	Constructs an ostream_withassign object.
	~ostream_withassign	Destroys an ostream_withassign object.
Operators		
	operator =	Assignment operator.

ostream_withassign::ostream_withassign

Syntax ostream_withassign(streambuf* *psb*);

ostream_withassign( );

Parameters

psb

A pointer to an existing object of a streambuf-derived class.

The first constructor makes a ready-to-use object of type ostream_withassign, complete with an attached streambuf object.

The second constructor makes an object but does not initialize it. You must subsequently use the streambuf assignment operator to attach the streambuf object, or you must use the ostream assignment operator to initialize this object to match the specified object.

ostream_withassign::operator =

ostream_withassign::~ostream_withassign

Syntax ~ostream_withassign( );

Destructor for the ostream_withassign class.

ostream_withassign::operator =

Syntax ostream& operator =(const ostream& *ros*);

ostream& operator =(streambuf* *sbp*);

The first overloaded assignment operator assigns the specified ostream object to this ostream_withassign object.

The second operator attaches a streambuf object to an existing ostream_withassign object, and it initializes the state of the ostream_withassign object. This operator is often used in conjunction with the void-argument constructor.

```
filebuf fb( "test.dat" ); // Filebuf object attached to "test.dat"
cout = &fb;          // fb associated with cout
cout << "testing"; // Message goes to "test.dat" instead of stdout
```

ostream_withassign::ostream_withassign
cout

class ostrstream : public ostream

The ostrstream class supports output streams that have character arrays as a destination. You can allocate a character array prior to construction, or the constructor can internally allocate an expandable array. All the ostream operators and functions can then be used to fill the array.

You must be aware that there is a put pointer working behind the scenes in the attached strstreambuf class. This pointer advances as you insert fields into the stream's array. The only way you can make it go backwards is to use the ostream::seekp function. If the put pointer reaches the end of user-allocated memory (and sets the ios::eof flag), then you must call clear before seekp.

#include <strstrea.h>

strstreambuf
streambuf
strstream
istrstream

ostrstream Class Members _ All Public

	Member	Description
Construction/Destruction		
	ostrstream	Constructs an ostrstream object.
	~ostrstream	Destroys an ostrstream object.
Other Functions		
	pcount	Returns the number of bytes that have been stored in the stream's buffer.
	rdbuf	Returns a pointer to the stream's associated strstreambuf object.
	str	Returns a character array pointer to the string stream's contents and freezes the array.

ostrstream::ostrstream

Syntax ostrstream( );

ostrstream(char* *pch*, int *nLength*, int *nMode* = ios::out);

Parameters

pch

A character array that is large enough to accommodate future output stream activity.

nLength

The size (in characters) of *pch*. If 0, then *pch* is assumed to point to a null-terminated array and strlen(*pch*) is used as the length; if less than 0, then the array is assumed to have infinite length.

nMode

The stream-creation mode. Must be one of the following enumerators as defined in class ios:

Value	Meaning
ios::out	Default; storing begins at *pch*
ios::ate	The *pch* parameter is assumed to be a null-terminated array; storing begins at the NULL character
ios::app	Same as ios::ate

The first constructor makes an ostrstream object that uses an internal, dynamic buffer.

The second constructor makes an ostrstream object out of the first *nLength* characters of the *pch* buffer. The stream will not accept characters once the length reaches *nLength*.

ostrstream::~ostrstream

Syntax ~ostrstream( );

This function destroys an ostrstream object and its associated strstreambuf object, thus releasing all internally allocated memory. If you used the void-argument constructor, then the internally allocated character buffer is released; otherwise, you must release it yourself.

An internally allocated character buffer will not be released if it was previously frozen by an str or strstreambuf::freeze function call.

ostrstream::str
strstreambuf::freeze

ostrstream::pcount

Syntax int pcount( ) const;

Returns the number of bytes that have been stored in the buffer. This information is especially useful when you have stored binary data in the object.

ostrstream::rdbuf

Syntax strstreambuf* rdbuf( ) const;

Returns a pointer to the strstreambuf buffer object that is associated with this stream. Note that this is not the character buffer; the strstreambuf object contains a pointer to the character area.

ostrstream::str

ostrstream::str

Syntax char* str( );

Returns a pointer to the internal character array. If the stream was built with the void-argument constructor, then str freezes the array. You must not send characters to a frozen stream, and you are responsible for deleting the array. You can, however, subsequently unfreeze the array by calling rdbuf->freeze(0).

If the stream was built with the constructor that specified the buffer, then the pointer contains the same address as the array used to construct the ostrstream object.

ostrstream::ostrstream
ostrstream::rdbuf
strstreambuf::freeze

class stdiobuf : public streambuf

The run-time library supports three conceptual sets of I/O functions: iostreams (C++ only), standard I/O (the functions declared in STDIO.H), and low-level I/O (the functions declared in IO.H). The stdiobuf class is a derived class of streambuf that is specialized for buffering to and from the standard I/O system.

Because the standard I/O system does its own internal buffering, the extra buffering level provided by stdiobuf may reduce overall input/output efficiency. The stdiobuf class is useful when you need to mix iostream I/O with standard I/O (printf and so forth).

You can avoid use of the stdiobuf class if you use the filebuf class. You must also use the stream class's ios::flags member function to set the ios::stdio format flag value.

#include <stdiostr.h>

stdiostream
filebuf
strstreambuf
ios::flags

stdiobuf Class Members _ All Public

	Member	Description
Construction/Destruction		
	stdiobuf	Constructs a stdiobuf object from a FILE pointer.
	~stdiobuf	Destroys a stdiobuf object.
Other Functions		
	stdiofile	Gets the file that is attached to the stdiofile object.

stdiobuf::stdiobuf

Syntax stdiobuf(FILE* *fp*);

Parameters

fp

A standard I/O file pointer (can be obtained through an fopen or _fsopen call).

Objects of class stdiobuf are constructed from open standard I/O files, including stdin, stdout, and stderr. The object is unbuffered by default.

stdiobuf::~stdiobuf

Syntax ~stdiobuf();

Destroys a stdiobuf object and, in the process, flushes the put area. The destructor does not close the attached file.

stdiobuf::stdiofile

Syntax FILE* stdiofile();

Returns the standard I/O file pointer associated with a stdiobuf object.

class stdiostream : public iostream

The stdiostream class makes I/O calls (through the stdiobuf class) to the standard I/O system, which does its own internal buffering. Calls to the functions declared in STDIO.H such as printf, can be mixed with stdiostream I/O calls.

This class is included for compatibility with earlier stream libraries. You can avoid use of the stdiostream class if you use the ostream or istream class with an associated filebuf class. You must also use the stream class's ios::flags member function to set the ios::stdio format flag value.

The use of the stdiobuf class may reduce efficiency because it imposes an extra level of buffering. Do not use this feature unless you need to mix iostream library calls with standard I/O calls for the same file.

#include <stdiostr.h>

stdiobuf
ios::flags

stdiostream Class Members _ All Public

	Member	Description
Construction/Destruction		
	stdiostream	Constructs a stdiostream object that is associated with a standard I/O FILE pointer.
	~stdiostream	Destroys a stdiostream object (virtual).
Other Functions		
	rdbuf	Gets the stream's stdiobuf object.

stdiostream::rdbuf

Syntax stdiobuf* rdbuf( ) const;

Returns a pointer to the stdiobuf buffer object that is associated with this stream. The rdbuf function is useful when you need to call stdiobuf member functions.

stdiostream::stdiostream

Syntax stdiostream(FILE* *fp*);

Parameters

fp

A standard I/O file pointer (can be obtained through an fopen or _fsopen call). Could be stdin, stdout, or stderr.

Objects of class stdiostream are constructed from open standard I/O files. An unbuffered stdiobuf object is automatically associated, but the standard I/O system provides its own buffering.

Example

```
stdiostream myStream( stdout );
```

stdiostream::~stdiostream

Syntax ~stdiostream( );

This destructor destroys the stdiobuf object associated with this stream; however, the attached file is not closed.

The Standard Output Stream: cout

In C++, there are facilities for performing input and output known as streams. The name **cout** represents the standard output stream. You can use **cout** to display information:

```
#include <iostream.h>
void main()
{
    cout << "Hello, world\n";
}
```

The string Hello, world\n is sent to the standard output device, which is the screen. The **<<** operator is called the insertion operator. It points from what is being sent (the string) to where it is going (the screen).

Suppose you want to print an integer instead of a string. In C, you would use **printf** with a format string that describes the parameters:

```
printf( "%d", amount );
```

In C++, you dont need the format string:

```
#include <iostream.h>

void main()
{
   int amount = 123;
   cout << amount;
}
```

The program prints 123.

The following example shows how to send a string, an integer, and a character constant to the output stream using one statement.

```
#include <iostream.h>

void main()
{
   int amount = 123;
   cout << "The value of amount is " << amount << '.';
}
```

This program sends three different data types to **cout**: a string literal, the integer amount variable, and a character constant '.' to add a period to the end of the sentence. The program prints this message:

```
The value of amount is 123.
```

Notice how multiple values are displayed using a single statement: The **<<** operator is repeated for each value.

Formatted Output

So far, the examples havent sent formatted output to **cout**. Suppose you want to display an integer using hexadecimal instead of decimal notation. The **printf** function handles this well. How does **cout** do it?

Note Whenever you wonder how to get C++ to do something that C does, remember that the entire C language is part of C++. In the absence of a better way, you can revert to C.

C++ associates a set of manipulators with the output stream. They change the default format for integer arguments. You insert the manipulators into the stream to make the change. The manipulators names are **dec**, **oct**, and **hex**.

The next example shows how you can display an integer value in its three possible configurations.

```
#include <iostream.h>

main()
{
   int amount = 123;
   cout << dec << amount << ' '
        << oct << amount << ' '
        << hex << amount;
}
```

The example inserts each of the manipulators (**dec**, **oct**, and **hex**) to convert the value in amount into different representations.

This program prints this:

```
123 173 7b
```

Each of the values shown is a different representation of the decimal value 123 from the amount variable.

cin

cerr

The Standard Input Stream: cin

At times you may want to read data from the keyboard. C++ includes its own version of standard input in the form of **cin**. The next example shows you how to use **cin** to read an integer from the keyboard.

```
#include <iostream.h>

void main()
{
   int amount;
   cout << "Enter an amount...\n";
   cin >> amount;
   cout << "The amount you entered was " << amount;
}
```

This example prompts you to enter an amount. Then **cin** sends the value that you enter to the variable amount. The next statement displays the amount using **cout** to demonstrate that the **cin** operation worked.

You can use **cin** to read other data types as well. The next example shows how to read a string from the keyboard.

```
#include <iostream.h>

void main()
{
   char name[20];
   cout << "Enter a name...\n";
   cin >> name;
   cout << "The name you entered was " << name;
}
```

The approach shown in this example has a serious flaw. The character array is only 20 characters long. If you type too many characters, the stack overflows and peculiar things happen. The **get** function solves this problem. It is explained in the *iostream Class Library Reference*. For now, the examples assume that you will not type more characters than a string can accept.

Note Recall that **printf** and **scanf** are not part of the C language proper but are functions defined in the run-time library. Similarly, the **cin** and **cout** streams are not part of the C++ language. Instead, they are defined in the stream library, which is why you must include IOSTREAM.H in order to use them. Furthermore, the meaning of the **<<** and **>>** operators depends on the context in which they are used. They can display or read data only when used with **cout** and **cin**.

cout

cerr

The Standard Error Stream: cerr

To send output to the standard error device, use **cerr** instead of **cout**. You can use this technique to send messages to the screen from programs that have their standard output redirected to another file or device.

cout

cin

Comments

C++ supports the C comment format where /* begins a comment and */ ends it. But C++ has another comment format, which is preferred by many programmers. The C++ comment token is the double-slash (*//*) sequence. Wherever this sequence appears (unless it is inside a string), everything to the end of the current line is a comment.

The following C++ example uses both C and C++ comments.

```
/* C comment */
#include <iostream.h>

void main()
{
   char name[20];                // Declare a string
   cout << "Enter a name...\n";  // Request a name
   cin >> name;                  // Read the name
   cout << "The name you entered was " << name;
}
```

class streambuf

All the iostream classes in the ios hierarchy depend on an attached streambuf class for the actual I/O processing. This class is an abstract class, but the iostream class library contains the following derived buffer classes for use with streams:

filebuf
: Buffered disk file I/O.

strstreambuf
: Stream data held entirely within an in-memory byte array.

stdiobuf
: Disk I/O with buffering done by the underlying standard I/O system.

All streambuf objects, when configured for buffered processing, maintain a fixed memory buffer, called a reserve area, that can be dynamically partitioned into a get area for input and a put area for output. These areas may or may not overlap. Protected member functions allow access and manipulation of a get pointer for character retrieval and a put pointer for character storage. The exact behavior of the buffers and pointers depends on the implementation of the derived class.

The capabilities of the iostream classes can be extended significantly through the derivation of new streambuf classes. The ios class tree supplies the programming interface and all formatting features, but the streambuf class does the real work. The ios classes call the streambuf public members, including a set of virtual functions.

The streambuf class provides a default implementation of certain virtual member functions. The "Default Implementation" section for each such function suggests function behavior for the derived class.

```
#include <iostream.h>
```

streambuf Class

	Member	Description
Public Members		
Character Input Functions		
	in_avail	Returns the number of characters in the get area.
	sgetc	Returns the character at the get pointer, but does not move the pointer.
	snextc	Advances the get pointer, then returns the next character.
	sbumpc	Returns the current character, and then advances the get pointer.
	stossc	Moves the get pointer forward one position, but does not return a character.
	sputmfcc	Attempts to move the get pointer back one position.
	sgetn	Gets a sequence of characters from the streambuf object's buffer.
Character Output Functions		
	out_waiting	Returns the number of characters in the put area.
	sputc	Stores a character in the put area and advances the put pointer.
	sputn	Stores a sequence of characters in the streambuf object's buffer and advances the put pointer.
Diagnostic Functions		
	dbp	Prints buffer statistics and pointer values.
Virtual Functions		
	sync	Empties the get area and the put area.

setbuf	Attempts to attach a reserve area to the streambuf object.
seekoff	Seeks to a specified offset.
~streambuf	Virtual destructor.
overflow	Empties the put area.
underflow	Fills the get area if necessary.
pbackfail	Augments the sputmfcc function.

Protected Members

Construction/Destruction

streambuf	Constructors for use in derived classes.

Other Protected Member Functions

base	Returns a pointer to the start of the reserve area.
ebuf	Returns a pointer to the end of the reserve area.
blen	Returns the size of the reserve area.
pbase	Returns a pointer to the start of the put area.
pptr	Returns the put pointer.
epptr	Returns a pointer to the end of the put area.
eback	Returns the lower bound of the get area.
gptr	Returns the get pointer.
egptr	Returns a pointer to the end of the get area.
setp	Sets all the put area pointers.
setg	Sets all the get area pointers.
pbump	Increments the put pointer.
gbump	Increments the get pointer.

setb	Sets up the reserve area.
unbuffered	Tests or sets the streambuf buffer state variable.
allocate	Allocates a buffer, if needed, by calling doalloc.
doallocate	Allocates a reserve area (virtual function).

streambuf Class Member Categories

Public Members

- Character Input Functions
- Character Output Functions
- Diagnostic Functions
- Virtual Functions

Private Members

- Construction/Destruction
- Other Protected Member Function

streambuf::allocate

Syntax Protected:
int allocate();

Calls the virtual function doallocate to set up a reserve area. If a reserve area already exists or if the streambuf object is unbuffered, allocate returns 0. If the space allocation fails, allocate returns EOF.

streambuf::doallocate
streambuf::unbuffered

streambuf::base

Syntax Protected:
char* base() const;

Returns a pointer to the first byte of the reserve area. The reserve area consists of space between the pointers returned by base and ebuf.

streambuf::ebuf
streambuf::setb
streambuf::blen

streambuf::blen

Syntax Protected:
int blen() const;

Returns the size, in bytes, of the reserve area.

streambuf::base
streambuf::ebuf
streambuf::setb

streambuf::dbp

Syntax void dbp();

Writes ASCII debugging information directly on stdout. Treat this function as part of the protected interface.

Some sample output follows:

```
STREAMBUF DEBUG INFO: this = 00E7:09DC
base()=00E7:0A0C, ebuf()=00E7:0C0C,  blen()=512
eback()=0000:0000, gptr()=0000:0000, egptr()=0000:0000
pbase()=00E7:0A0C, pptr()=00E7:0A22, epptr()=00E7:0C0C
```

streambuf::doallocate

Syntax Protected:
virtual int doallocate();

Called by allocate when space is needed. The doallocate function must allocate a reserve area, then call setb to attach the reserve area to the streambuf object. If the reserve area allocation fails, doallocate returns EOF.

Default Implementation Attempts to allocate a reserve area using operator new.

streambuf::allocate
streambuf::setb

streambuf::eback

Syntax Protected:
char* eback() const;

Returns the lower bound of the get area. Space between the eback and gptr pointers is available for putting a character back to the stream.

streambuf::sputmfcc
streambuf::gptr

streambuf::ebuf

Syntax Protected:
char* ebuf() const;


Returns a pointer to the byte after the last byte of the reserve area. The reserve area consists of space between the pointers returned by base and ebuf.

streambuf::base
streambuf::setb
streambuf::blen

streambuf::egptr

Syntax Protected:
char* egptr() const;


Returns a pointer to the byte after the last byte of the get area.

streambuf::setg
streambuf::eback
streambuf::gptr

streambuf::epptr

Syntax Protected:
char* epptr() const;


Returns a pointer to the byte after the last byte of the put area.

streambuf::setp
streambuf::pbase
streambuf::pptr

streambuf::gbump

Syntax Protected:
void gbump(int *nCount*);

Parameters

nCount

The number of bytes to increment the get pointer. May be positive or negative.

Increments the get pointer. No bounds checks are made on the result.

streambuf::pbump

streambuf::gptr

Syntax Protected:
char* gptr() const;

Returns a pointer to the next character to be fetched from the streambuf buffer. This pointer is known as the get pointer.

streambuf::setg
streambuf::eback
streambuf::egptr

streambuf::in_avail

Syntax int in_avail() const;

Returns the number of characters in the get area that are available for fetching. These characters are between the gptr and egptr pointers and may be fetched with a guarantee of no errors.

streambuf::out_waiting

Syntax int out_waiting() const;

Returns the number of characters in the put area that have not been sent to the final output destination. These characters are between the pbase and pptr pointers.

streambuf::overflow

Syntax virtual int overflow(int *nCh* = EOF) = 0;

Parameters

nCh

EOF or the character to output.

The virtual overflow function, together with the sync and underflow functions, defines the characteristics of the streambuf-derived class. Each derived class might implement overflow differently, but the interface with the calling stream class is the same.

The overflow function is most frequently called by public streambuf functions like sputc and sputn when they sense that the put area is full, but other classes, including the stream classes, can call overflow anytime.

The function "consumes" the characters in the put area between the pbase and pptr pointers and then reinitializes the put area. The overflow function must also consume *nCh* (if *nCh* is not EOF), or it might choose to put that character in the new put area so that it will be consumed on the next call.

The definition of "consume" varies among derived classes. The filebuf class, for example, writes its characters to a file. The strsteambuf class, on the other hand, keeps them in its buffer and (if the buffer is designated as dynamic) expands the buffer in response to a call to overflow. This expansion is achieved by freeing the old buffer and replacing it with a new, larger one. The pointers are adjusted as necessary.

Default Implementation No default implementation. Derived classes must define this function.

Return Value EOF to indicate an error.

streambuf::pbase
streambuf::pptr
streambuf::setp
streambuf::sync
streambuf::underflow

streambuf::pbackfail

Syntax virtual int pbackfail(int *nCh*);

Parameters

nCh
The character used in a previous sputmfcc call.

This function is called by sputmfcc if it fails, usually because the eback pointer equals the gptr pointer. The pbackfail function should deal with the situation, if possible, by such means as repositioning the external file pointer.

Default Implementation Returns EOF.

Return Value The *nCh* parameter if successful; otherwise EOF.

streambuf::sputmfcc

streambuf::pbase

Syntax Protected:
char* pbase() const;

Returns a pointer to the start of the put area. Characters between the pbase pointer and the pptr pointer have been stored in the buffer but not flushed to the final output destination.

streambuf::pptr
streambuf::setp
streambuf::out_waiting

streambuf::pbump

Syntax Protected:
void pbump(int *nCount*);

Parameters

nCount
The number of bytes to increment the put pointer. May be positive or negative.

This function increments the put pointer. No bounds checks are made on the result.

streambuf::gbump
streambuf::setp

streambuf::pptr

Syntax Protected:
char* pptr() const;

Returns a pointer to the first byte of the put area. This pointer is known as the put pointer and is the destination for the next character(s) sent to the streambuf object.

streambuf::epptr
streambuf::pbase
streambuf::setp

streambuf::sbumpc

Syntax int sbumpc();

Returns the current character, then advances the get pointer. Returns EOF if the get pointer is currently at the end of the sequence (equal to the egptr pointer).

streambuf::epptr
streambuf::gbump

streambuf::seekoff

Syntax virtual streampos seekoff(streamoff *off*, ios::seek_dir *dir*, int *nMode* = ios::in | ios::out);

Parameters

off

The new offset value; streamoff is a typedef equivalent to long.

dir

The seek direction specified by the enumerated type seek_dir, as follows:

Value	Meaning
ios::beg	Seek from the beginning of the stream.
ios::cur	Seek from the current position in the stream.
ios::end	Seek from the end of the stream.

nMode

An integer that contains a bitwise-OR (|) combination of the enumerators ios::in and ios::out.

This function changes the position for the streambuf object. Not all derived classes of streambuf need to support positioning; however, the filebuf, strstreambuf, and stdiobuf classes do support positioning.

Classes derived from streambuf often support independent input and output position values. The *nMode* parameter determines which value(s) is set.

Default Implementation Returns EOF.

Return Value The new position value. This is the byte offset from the start of the file (or string). If both ios::in and ios::out are specified, then the function returns the output position. If the derived class does not support positioning, then the function returns EOF.

streambuf::seekpos

streambuf::seekpos

Syntax virtual streampos seekpos(streampos *pos*, int *nMode* = ios::in | ios::out);

Parameters

pos

The new position value; streampos is a typedef equivalent to long.

nMode

An integer that contains mode bits defined as ios enumerators that can be combined with the OR (|) operator. See ofstream::ofstream for a listing of the enumerators.

This function changes the position, relative to the beginning of the stream, for the streambuf object. Not all derived classes of streambuf need to support positioning; however, the filebuf, strstreambuf, and stdiobuf classes do support positioning.

Classes derived from streambuf often support independent input and output position values. The *nMode* parameter determines which value(s) is set.

Default Implementation Calls seekoff((streamoff) *pos*, ios::beg, *nMode*). Thus, to define seeking in a derived class, it is usually necessary to redefine only seekoff.

Return Value The new position value. If both ios::in and ios::out are specified, then the function returns the output position. If the derived class does not support positioning, then the function returns EOF.

streambuf::seekoff

streambuf::setb

Syntax Protected:
void setb(char* *pb*, char* *peb*, int *nDelete* = 0);


Parameters

pb
The new value for the base pointer.

peb
The new value for the ebuf pointer.

nDelete
Flag that controls automatic deletion. If *nDelete* is not 0, then the reserve area will be deleted (1) when the base pointer is changed by another setb call or (2) when the streambuf destructor is called.

Sets the values of the reserve area pointers. If both *pb* and *peb* are NULL, then there is no reserve area. If *pb* is not NULL and *peb* is NULL, then the reserve area has a length of 0.

streambuf::base
streambuf::ebuf

streambuf::setbuf

Syntax virtual streambuf* setbuf(char* *pr*, int *nLength*);

Parameters

pr

A pointer to a previously allocated reserve area of length *nLength*. A NULL value indicates an unbuffered stream.

nLength

The length (in bytes) of the reserve area. A length of 0 indicates an unbuffered stream.

Attaches the specified reserve area to the streambuf object. Derived classes may or may not use this area.

Default Implementation Accepts the request if there is not a reserved area already.

Return Value A streambuf pointer if the buffer is accepted; otherwise NULL.

streambuf::setg

Syntax Protected:
void setg(char* *peb*, char* *pg*, char* *peg*);

Parameters

peb
The new value for the eback pointer.

pg
The new value for the gptr pointer.

peg
The new value for the egptr pointer.

This function sets the values for the get area pointers.

streambuf::eback
streambuf::gptr
streambuf::egptr

streambuf::setp

Syntax Protected:
void setp(char* *pp*, char* *pep*);

Parameters

pp
The new value for the pbase and pptr pointers.

pep
The new value for the epptr pointer.

This function sets the values for the put area pointers.

streambuf::pptr
streambuf::pbase
streambuf::epptr

streambuf::sgetc

Syntax int sgetc();

Returns the character at the get pointer. The sgetc function does not move the get pointer. Returns EOF if there is no character available.

streambuf::sbumpc
streambuf::sgetn
streambuf::snextc
streambuf::stossc

streambuf::sgetn

Syntax int sgetn(char* *pch*, int *nCount*);

Parameters

pch
A pointer to a buffer that will receive characters from the streambuf object.

nCount
The number of characters to get.

Gets the *nCount* characters that follow the get pointer and stores them in the area starting at *pch*. When fewer than *nCount* characters remain in the streambuf object, sgetn fetches whatever characters remain. The function repositions the get pointer to follow the fetched characters.

Return Value The number of characters fetched.

streambuf::sbumpc
streambuf::sgetc
streambuf::snextc
streambuf::stossc

streambuf::snextc

Syntax int snextc();

First tests the get pointer, then returns EOF if it is already at the end of the get area. Otherwise, it moves the get pointer forward one character and returns the character that follows the new position. It returns EOF if the pointer has been moved to the end of the get area.

streambuf::sbumpc
streambuf::sgetc
streambuf::sgetn
streambuf::stossc

streambuf::sputmfcc

Syntax int sputmfcc(char *ch*);

Parameters

ch

The character to be put back to the streambuf object.

Moves the get pointer back one position. The *ch* character must match the character just before the get pointer.

Return Value EOF on failure.

streambuf::sbumpc
streambuf::pbackfail

streambuf::sputc

Syntax int sputc(int *nCh*);

Parameters

nCh
The character to store in the streambuf object.

Stores a character in the put area and advances the put pointer.

This public function is available to code outside the class, including the classes derived from ios. A derived streambuf class can gain access to its buffer directly by using protected member functions.

Return Value The number of characters successfully stored; EOF on error.

streambuf::sputn

streambuf::sputn

Syntax int sputn(const char* *pch*, int *nCount*);

Parameters

pch

A pointer to a buffer that contains data to be copied to the streambuf object.

nCount

The number of characters in the buffer.

Copies *nCount* characters from *pch* to the streambuf buffer following the put pointer. The function repositions the put pointer to follow the stored characters.

Return Value The number of characters stored. This number is usually *nCount* but could be less if an error occurs.

streambuf::sputc

streambuf::stossc

Syntax void stossc();

Moves the get pointer forward one character. If the pointer is at the end of the get area already, the function has no effect.

streambuf::sbumpc
streambuf::sgetn
streambuf::snextc
streambuf::sgetc

streambuf::streambuf

Syntax Protected:
streambuf();

Protected:
streambuf(char* *pr*, int *nLength*);

Parameters

pr

A pointer to a previously allocated reserve area of length *nLength*. A NULL value indicates an unbuffered stream.

nLength

The length (in bytes) of the reserve area. A length of 0 indicates an unbuffered stream.

The first constructor makes an uninitialized streambuf object. This object is not suitable for use until a setbuf call is made. A derived class constructor usually calls setbuf or uses the second constructor.

The second constructor initializes the streambuf object with the specified reserve area or marks it as unbuffered.

streambuf::setbuf

streambuf::~streambuf

Syntax Public:
virtual ~streambuf();

The streambuf destructor flushes the buffer if the stream is being used for output.

streambuf::sync

Syntax virtual int sync();

The virtual sync function, together with the overflow and underflow functions, defines the characteristics of the streambuf-derived class. Each derived class might implement sync differently, but the interface with the calling stream class is the same.

The sync function flushes the put area. It also empties the get area and, in the process, sends any unprocessed characters back to the source, if necessary.

Default Implementation Returns 0 if the get area is empty and there are no more characters to output; otherwise, it returns EOF.

Return Value EOF if an error occurs.

streambuf::overflow

streambuf::unbuffered

Syntax Protected:
void unbuffered(int *nState*);

Protected:
int unbuffered() const;

Parameters

nState
The value of the buffering state variable; 0 = buffered, nonzero = unbuffered.

The first overloaded unbuffered function sets the value of the streambuf object's buffering state. This variable's primary purpose is to control whether the allocate function automatically allocates a reserve area.

The second function returns the current buffering state variable.

streambuf::allocate
streambuf::doallocate

streambuf::underflow

Syntax virtual int underflow() = 0;

The virtual underflow function, together with the sync and overflow functions, defines the characteristics of the streambuf-derived class. Each derived class might implement underflow differently, but the interface with the calling stream class is the same.

The underflow function is most frequently called by public streambuf functions like sgetc and sgetn when they sense that the get area is empty, but other classes, including the stream classes, can call underflow anytime.

The underflow function supplies the get area with characters from the input source. If the get area contains characters, then underflow returns the first character. If the get area is empty, then it fills the get area and returns the next character (which it leaves in the get area). If there are no more characters available, then underflow returns EOF and leaves the get area empty.

In the strstreambuf class, underflow adjusts the egptr pointer to access storage that was dynamically allocated by a call to overflow.

Default Implementation No default implementation. Derived classes must define this function.

class strstream : public iostream

The strstream class supports I/O streams that have character arrays as a source and destination. You can allocate a character array prior to construction, or the constructor can internally allocate a dynamic array. All the input and output stream operators and functions can then be used to fill the array.

You must be aware that there is a put pointer and a get pointer working independently behind the scenes in the attached strstreambuf class. The put pointer advances as you insert fields into the stream's array, and the get pointer advances as you extract fields. The ostream::seekp function moves the put pointer, and the istream::seekg function moves the get pointer. If either pointer reaches the end of the string (and sets the ios::eof flag), then you must call clear before seeking.

#include <strstrea.h>

strstreambuf
streambuf
istrstream
ostrstream

strstream Class Members _ All Public

	Member	Description
Construction/Destruction		
	strstream	Constructs a strstream object.
	~strstream	Destroys a strstream object.
Other Functions		
	pcount	Returns the number of bytes that have been stored in the stream's buffer.
	rdbuf	Returns a pointer to the stream's associated strstreambuf object.
	str	Returns a pointer to the string stream's character buffer and freezes it.

strstream::pcount

Syntax int pcount( ) const;

Returns the number of bytes that have been stored in the buffer. This information is especially useful when you have stored binary data in the object.

strstream::rdbuf

Syntax strstreambuf* rdbuf() const;

Returns a pointer to the strstreambuf buffer object that is associated with this stream. Note that this is not the character buffer; the strstreambuf object contains a pointer to the character area.

strstream::str

strstream::str

Syntax char* str();

Returns a pointer to the internal character array. If the stream was built with the void-argument constructor, then str freezes the array. You must not send characters to a frozen stream, and you are responsible for deleting the array. You can unfreeze the the stream by calling rdbuf->freeze(0).

If the stream was built with the constructor that specified the buffer, then the pointer contains the same address as the array used to construct the ostrstream object.

strstreambuf::freeze
strstream::rdbuf

strstream::strstream

Syntax strstream();

strstream(char* *pch*, int *nLength*, int *nMode*);

Parameters

pch

A character array that is large enough to accommodate future output stream activity.

nLength

The size (in characters) of *pch*. If 0, then *pch* is assumed to point to a null-terminated array; if less than 0, then the array is assumed to have infinite length.

nMode

The stream creation mode. Must be one of the following enumerators as defined in class ios:

Value	Meaning
ios::in	Retrieval begins at the beginning of the array.
ios::out	By default, storing begins at *pch*.
ios::ate	The *pch* parameter is assumed to be a null-terminated array; storing begins at the NULL character.
ios::app	Same as ios::ate.

The use of the ios::in and ios::out flags is optional for this class; both input and output are implied.

The first constructor makes an strstream object that uses an internal, dynamic buffer that is initially empty.

The second constructor makes an strstream object out of the first *nLength* characters of the *psc* buffer. The stream will not accept characters once the length reaches *nLength*.

strstream::~strstream

Syntax ~strstream();

Destroys a strstream object and its associated strstreambuf object, thus releasing all internally allocated memory. If you used the void-argument constructor, then the internally allocated character buffer is released; otherwise, you must release it yourself.

An internally allocated character buffer will not be released if it was previously frozen by calling rdbuf->freeze(0).

strstream::rdbuf

class strstreambuf : public streambuf

The strstreambuf class is a derived class of streambuf that manages an in-memory character array.

The file stream classes, ostrstream, istrstream, and strstream, use strstreambuf member functions to fetch and store characters. Some of these member functions are virtual functions defined for the streambuf class.

The reserve area, put area, and get area were introduced in the streambuf class description. For strsteambuf objects, the put area is the same as the get area, but the get pointer and the put pointer move independently.

#include <strstrea.h>

istrstream
ostrstream
filebuf
stdiobuf

strstreambuf Class Members _ All Public

	Member	Description
Construction/Destruction		
	strstreambuf	Constructs a **strstreambuf** object.
	~strstreambuf	Destroys a **strstreambuf** object.
Other Functions		
	freeze	Freezes a stream.
	str	Returns a pointer to the string.

strstreambuf::freeze

Syntax void freeze(int *n* = 1);

Parameters

n

A 0 value permits automatic deletion of the current array and also its automatic growth (if it is dynamic); a nonzero value prevents deletion.

If a strstreambuf object has a dynamic array, then memory is usually deleted on destruction and size adjustment. The freeze function provides a means of preventing that automatic deletion. Once an array is frozen, no further input or output is permitted. The results of such operations are undefined.

The freeze function can also unfreeze a frozen buffer.

strstreambuf::str

strstreambuf::str

Syntax char* str();

Returns a pointer to the object's internal character array. If the strstreambuf object was constructed with a user-supplied buffer, then that buffer address is returned. If the object has a dynamic array, then str freezes the array. You must not send characters to a frozen strstreambuf object, and you are responsible for deleting the array. If a dynamic array is empty, then str returns NULL.

You can use the freeze function with a 0 parameter to unfreeze a frozen strstreambuf object.

strstreambuf::freeze

strstreambuf::strstreambuf

Syntax strstreambuf();

strstreambuf(int *nBytes*);

strstreambuf(char* *pch*, int *n*, char* *pstart* = 0);

strstreambuf(unsigned char* *puch*, int *n*, unsigned char* *pustart* = 0);

strstreambuf(signed char* *psch*, int *n*, signed char* *psstart* = 0);

strstreambuf(void* (**falloc*)(long), void (**ffree*)(void*));

Parameters

nBytes

The initial length of a dynamic stream buffer.

pch, *puch*, *psch*

A pointer to a character buffer that will be attached to the object. The get pointer is initialized to this value.

n

An integer parameter with the following meanings:

Value	Meaning
positive	*n* bytes, starting at *pch*, is used as a fixed-length stream buffer.
0	The *pch* parameter points to the start of a null-terminated string that constitutes the stream buffer (terminator excluded).
negative	The *pch* parameter points to a stream buffer that continues indefinitely.

pstart, *pustart*, *psstart*

The initial value of the put pointer.

falloc

A memory-allocation function with the prototype void* falloc(long). The default is new.

ffree

A function that frees allocated memory with the prototype void ffree(void*). The default is delete.

The four streambuf constructors are described as follows:

Constructor	Description
strstreambuf()	Constructs an empty strstreambuf object with dynamic buffering. The buffer is allocated internally by the class and grows as needed, unless it is frozen.
strstreambuf(int)	Constructs an empty strstreambuf object with a dynamic buffer *n* bytes long to start with. The buffer is allocated internally by the class and grows

	as needed, unless it is frozen.
strstreambuf(char*, int, char*)	Constructs a strstreambuf object from already-allocated memory as specified by the arguments. There are constructor variations for both unsigned and signed character arrays.
strstreambuf(void*(*), void(*))	Constructs an empty strstreambuf object with dynamic buffering. The *falloc* function is called for allocation. The long parameter specifies the buffer length and the function returns the buffer address. If the *falloc* pointer is NULL, then operator new is used. The *ffree* function frees memory allocated by *falloc*. If the *ffree* pointer is NULL, the operator delete is used.

strstreambuf::~strstreambuf

Syntax ~strstreambuf();

Destroys a strstreambuf object and, in the process, releases all internally allocated dynamic memory unless the object is frozen. The destructor does not release user-allocated memory.