Microsoft® C Optimizing Compiler

for the MS® OS/2 Operating System

Run-Time Library Reference Update

Microsoft Corporation

Document No. 510830024-100-O00-0587
Part No. 048-014-087

Introduction

Important Note

READ THIS FIRST BEFORE INSTALLING YOUR PRODUCT

This update notice provides information about additions and changes to the C run-time library that have been made to support a protected-mode, multitasking environment. These changes supplement the draft-quality manuals that document the DOS 3.*x* version of the product.

There are several changes to the run-time library for the Microsoft® C Optimizing Compiler that are not documented in the Software Developer's Kit manual. These changes are covered in the following paragraphs.

Chapter 2—Using Library Functions

In Section 2.10, note that the **dosexterr** function is not supported in the MS® OS/2 protected mode. When using the **exec** and **spawn** families, the variable *arg0* will contain the command name.

Chapter 3—Global Variables and Standard Data Types

In Section 3.6, note the addition of another system variable, **_osmode**, which stores a 0 if the program is running in real mode or a 1 if running in protected mode. Also, in Section 3.8, there is no **_psp** variable in the protected mode of MS OS/2.

Functions Not Supported

Although the following functions are described in the reference section of the manual, they are *not* supported in this MS OS/2 SDK release:

atexit	**fgetpos**	**_nheapchk**
clock	**fsetpos**	**_nheapset**
div	**ldiv**	**_nheapwalk**
_fheapchk	**_memmax**	**raise**
_fheapset	**memmove**	**strtoul**
_fheapwalk	**mktime**	

Function Changes

■ Real Mode Only

The following run-time functions are supported only in the MS OS/2 real mode:

bdos	**int86**	**intdosx**
dosexterr	**int86x**	**outp**
inp	**intdos**	

■ Non-Redirectable in Real Mode

The following functions are not redirectable in the MS OS/2 real mode:

getch **getche** **putch**

■ Protected Mode Thread Termination

The **exit** and **_exit** functions terminate all threads of the calling program.

■ Re-entrant Functions

The existing C run-time library is designed for single-thread execution only. Most of the functions are not re-entrant, and will not work properly in a multi-thread process. The functions below are re-entrant and therefore may be used in multi-thread programs:

abs[1]	**inp**[1]	**memicmp**	**strcmpi**	**strrev**
atol	**itoa**	**memset**	**strcpy**	**strset**
atoi	**kbhit**[1]	**movedata**[1]	**stricmp**	**strstr**
bdos[1]	**labs**[1]	**outp**[1]	**strlen**	**strupr**
bsearch	**lfind**	**putch**[1]	**strlwr**	**swab**
chdir	**lsearch**	**qsort**	**strncat**	**tolower**[1]
getch[1]	**longjmp**[1]	**segread**	**strncmp**	**toupper**[1]
getche[1]	**memccpy**	**setjmp**[1]	**strnicmp**	**utime**
getpid[1]	**memchr**	**strcat**	**strncpy**	
halloc[1]	**memcmp**	**strchr**	**strnset**	
hfree[1]	**memcpy**	**strcmp**	**strrchr**	

[1] Works properly in both far- and near- data-model program threads.

All of the routines listed above will work properly in execution threads of programs written in a far-data model (compact- and large-model). However, in near-data-model memory programs (small- and medium-model), only the footnoted routines are guaranteed to work properly. The other routines have model-dependent pointers in their interface and have the potential of allocating stack space outside the default segment (**SS!=DS**). As a result, these routines are not guaranteed to work properly in multi-thread near-data-model programs.

Note

The multi-thread restriction applies *only* to multi-thread programs. Single-thread programs can use any routine in the run-time library.

■ New Reference Pages

New reference pages are included for the following functions because they behave differently under MS OS/2 or are completely new functions for MS OS/2:

cwait **spawn** **system**
signal **stat** **wait**

■ Summary

int cwait(*stat-loc, process-id, action-code***);**

int ***stat-loc;**
int *process-id;*
int *action-code;*

■ Description

The **cwait** function suspends the calling process until the specified child process terminates.

Important

The **cwait** function is supported only in the 286DOS protected mode.

The *process-id* argument specifies which child process termination to wait for. This value is returned by the call to **spawn** that started the child process. If the specified child process terminates before **cwait** is called, **cwait** returns immediately.

If non-**NULL**, *stat-loc* points to a buffer containing the termination status word and return code of the terminated child process.

cwait (protected mode only)

The termination status word indicates whether or not the child process terminated normally (i.e., by calling the 286 **DOSEXIT** function). If terminated normally, the low-order and high-order bytes of the termination status word are as follows:

Byte	Contents
Low order	0
High order	Low-order byte of the result code that the child code passed to **DOSEXIT**. **DOSEXIT** is called if the child process called **exit** or **_exit**, returned from **main**, or reached the end of **main**. The low-order byte of the result code is either low-order byte of the argument to **_exit** or **exit**, the low-order byte of the return value from **main**, or, if the child process reached the end of **main**, a random value.

If the child process terminated for any other reason, the low-order and high-order bytes of the termination status word are as follows:

Byte	Contents
Low order	Termination code from **DOSCWAIT**:
High order	0

Code	Meaning
1	Hard-error abort
2	Trap operation
3	**SIGTERM** signal not intercepted

The *action-code* argument specifies when the parent process resumes execution, as shown in the following list:

Action Code	Meaning
0	The parent process waits until the specified child process has ended.
1	The parent process waits until the specified child process and all of the child processes of that child process have ended.

WAIT-CHILD and **WAIT-GRANDCHILD** are defined as 0 and 1, respectively, in **process.h**.

■ Return Value

If **cwait** returns after abnormal termination of the child process, it returns –1 and sets **errno** to **EINTR**.

If **cwait** returns after normal termination of the child process, it returns the child's process ID.

Otherwise, **cwait** returns –1 immediately and sets **errno** to one of the following error codes:

Value	Meaning
EINVAL	Invalid action code
ECHILD	No child process exists, or invalid process ID

■ See Also

exit, **_exit**, **spawn**, **wait**

signal

■ Summary

 # include <signal.h>

int (*signal(*sig, func***))( );**
int *sig*; Signal value
int (* *func* **)( );** Function to be executed

■ Description

The **signal** function allows a process to choose one of several ways to handle an interrupt signal from the operating system.

On MS-DOS® versions 3.*x* and earlier, the *sig* argument must be one of the manifest constants **SIGINT** or **SIGFPE**, defined in **signal.h**. On a 286DOS system, the *sig* argument must be one of the manifest constants **SIGINT**, **SIGFPE**, **SIGTERM**, **SIGUSR1**, **SIGUSR2**, **SIGUSR3**, or **SIGBREAK**, defined in **signal.h**.

On MS-DOS versions 3.*x* and earlier, the *func* argument must be one of the manifest constants **SIG_DFL** or **SIG_IGN** (also defined in **signal.h**), or a function address. On 286DOS, **SIG_ERR** and **SIG_ACK** are allowed in addition to the *func* arguments allowed on MS-DOS Version 3.*x*.

The meaning of the *sig* values are as follows:

Value	Meaning
SIGINT	CTRL-C signal. On MS-DOS version 3.*x* or earlier, this specifies the INT 23H interrupt signal. On 286DOS, this specifies the **SIGINTR** signal.
SIGFPE	This specifies floating-point exceptions that are not masked, such as overflow, division by zero, and invalid operation.
SIGTERM	286DOS only. This specifies the 286DOS **SIGTERM** signal.
SIGUSR1	286DOS only. This specifies the 286DOS Process flag A signal.
SIGUSR2	286DOS only. This specifies the 286DOS Process flag B signal.

SIGUSR3 286DOS only. This specifies the 286DOS Process flag C signal.

Note

SIGUSR1, **SIGUSR2**, and **SIGUSR3** are user-defined signals that can be sent by means of **DOSFLAGPROCESS**. See the **DOSCALLS** documentation for details.

SIGBREAK CTRL-BREAK signal (286DOS protected mode only).

The action taken when the interrupt signal is received depends on the value of *func*, as follows:

Value	Meaning
SIG_IGN	Ignores interrupt signal. The value should never be given for **SIGFPE**, since the floating-point state of the process is left undefined.
SIG_DFL	System default response. On MS-DOS Version 3.*x* or earlier, the calling process is terminated and control returns to the MS-DOS command level. If the calling program uses stream **I/O**, buffers created by the run-time library are not flushed (MS-DOS buffers are flushed). On 286DOS, the system-default response is to ignore all signals, with the exception of **SIGTERM** and **SIGFPE**. The system-default response for **SIGTERM** and **SIGFPE** is the same as the termination described above for all signals on MS-DOS Version 3.*x* or earlier.

SIG_ERR	286DOS only. Equivalent to **SIG_IGN** except that an error also occurs when a process sends this signal to this process by means of **DOSFLAGPROCESS** (if the signal is **SIGUSR1**, **SIGUSR2**, or **SIGUSR3**) or by means of **DOSKILLPROCESS** (if the signal is **SIGTERM**). Has exactly the same effect as **SIG_IGN** for **SIGINT** and **SIGBREAK** signals, since these can be sent only by the operating system. Not supported for **SIGFPE** signals.
SIG_ACK	Acknowledges receipt of a signal. Once a process receives a given signal, the operating system will not send any more signals of this type until it receives a **SIG_ACK** acknowledgement back from the process. The operating system does not queue up signals of a given type, so if more than one signal of a given type accumulates before the process sends back a **SIG_ACK**, only the most recent signal will be sent to the process after the **SIG_ACK** is received by the operating system. This option has no effect on which handler is installed for a given signal. Not supported for **SIGFPE** signals.
Function address	Installs specified function as the handler for the given signal. For **SIGFPE** signals, the function is passed two arguments: **SIGFPE** and the floating-point error code identifying the type of exception that occurred. On MS-DOS Version 3.*x* or earlier, for **SIGINT** signals, the function is passed the single argument **SIGINT** and executed. On 286DOS, for all signals other than **SIGFPE**, the function is passed two arguments: the signal number and the argument furnished by **DOSFLAGPROCESS**, if appropriate. The second argument is only meaningful if the signal is **SIGUSR1**, **SIGUSR2**, or **SIGUSR3**.

For all types of signals and on all versions of DOS, if the function returns, the calling process resumes execution immediately following the point at which it received the interrupt signal.

On MS-DOS Version 3.*x* or earlier, before the specified function is executed, the value of *func* is set to **SIG_DFL**; the next interrupt signal is treated as described above for **SIG_DFL**, unless an intervening call to **signal** specifies otherwise. This allows the user to reset signals in the called function if desired.

On 286DOS, the signal handler is not reset to the system-default response. Instead, no signals of a given type will be received by a process until the process sends a **SIG_ACK** to the operating system. The user can restore the default system response from the handler by first sending **SIG_DFL** and then sending **SIG_ACK** to the operating system.

For **SIGFPE**, the function pointed to by *func* is passed two arguments, **SIGFPE** and an integer error subcode, **FPE_** *xxx*, then executed. (See the include file **float.h** for definitions of the **FPE_** *xxx* subcodes.) The value of *func* is not reset upon receiving the signal; to recover from floating-point exceptions, use **setjmp** in conjunction with **longjmp**. (See the example under **_fpreset** in this reference.) If the function returns, the calling process resumes execution with the floating-point state of the process left in an undefined state.

■ Return Value

On MS-DOS Version 3.*x* or earlier, the **signal** function returns the previous value of *func* associated with the given signal number.

On 286DOS, for **SIGFPE** signals, the **signal** function returns the previous value of *func* for **SIGFPE**. For all other signals, the return value is as follows:

Previous *func*	**Return Value**
SIG_IGN	**SIG_IGN**
SIG_DFL	**SIG_DFL**
SIG_ERR	**SIG_ERR**
SIG_ACK	Address of currently installed handler
Function address	Function address

A return value of –1 indicates an error, and **errno** is set to **EINVAL**. The possible error causes are an invalid *sig* value; an invalid *func* value (i.e., a value that is less than **SIG_ACK** but not defined); or a *func* value of **SIG_ACK** used when there is no currently installed handler.

■ See Also

abort, **exit**, **_exit**, **_fpreset**, **spawnl**, **spawnle**, **spawnlp**, **spawnv**, **spawnve**, **spawnvp**

Note

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.

■ Example

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <process.h>

int handler();

main()
 {

  /* Set so interrupt calls "handler": */

  if (signal(SIGINT,handler) == (int(*)())-1)
   {
    fprintf(stderr,"Couldn't set SIGINT");
    abort();
   }
  for(;;)printf("Hit control C:\n");
 }
```

```
int handler()      /* Function called at system interrupt */
 {
  char ch;

  signal(SIGINT, SIG_IGN);    /* Disallow ctrl-c
                                 during handler    */

  printf("Terminate processing? ");
  ch = getch();
  if (( ch == 'y' ) || ( ch == 'Y')) exit(0);

  signal(SIGINT,handler);        /* By calling "signal" here,
                                 ** the next operating system
                                 ** interrupt signal sends
                                 ** control to handler(), not
                                 ** to the operating system
                                 */
 }
```

This program uses **signal** to set up the `handler` function as the routine that is called to execute an operating-system interrupt. When the user presses CTRL-C, `handler` is called to handle the interrupt.

spawnl – spawnvpe

■ Summary

 # include <stdio.h>
 # include <process.h>

int spawnl(*modeflag, pathname, arg0, arg1..., argn,***NULL);**

int spawnle(*modeflag, pathname, arg0, arg1..., argn,***NULL, *envp*);**

int spawnlp(*modeflag, pathname, arg0, arg1..., argn,***NULL);**

int spawnlpe(*modeflag, pathname, arg0, arg1..., argn,***NULL, *envp*);**

int spawnv(*modeflag, pathname, argv*);**

int spawnve(*modeflag, pathname, argv, envp*);**

int spawnvp(*modeflag, pathname, argv*);**

int spawnvpe(*modeflag, pathname, argv, envp*);**

int *modeflag*;	Execution mode for parent process
char **pathname*;	Path name of file to be executed
char **arg0*,**arg1*,...,**argn* ;	List of pointers to arguments
char **argv*[ ];	Array of pointers to arguments
char **envp*[ ];	Array of pointers to environment settings

■ Description

The **spawn** functions create and execute a new child process. Enough memory must be available for loading and executing the child process. The *modeflag* argument determines the action taken by the parent process before and during the **spawn**. The following values for *modeflag* are defined in **process.h**:

Value	Meaning
P_WAIT	Suspend parent process until execution of child process is complete (synchronous **spawn**)
P_NOWAIT	Continue to execute parent process concurrently with child process (asynchronous **spawn**—valid only in protected mode)

P_NOWAITO	Continue to execute parent process and ignore **wait** and **cwait** calls against child process (asynchronous **spawn**—valid only in protected mode)
P_OVERLAY	Overlay parent process with child, destroying the parent (same effect as **exec** calls)

The *pathname* argument specifies the file to be executed as the child process. The *pathname* can specify a full path (from the root), a partial path (from the current working directory), or just a file name. If *pathname* does not have a file-name extension or end with a period (.), search for the file; if unsuccessful, the extension **.EXE** is attempted. If *pathname* has an extension, only that extension is used. If *pathname* ends with a period, the **spawn** calls search for *pathname* with no extension. The **spawnlp**, **spawnlpe**, **spawnvp**, and **spawnvpe** routines search for *pathname* (using the same procedures) in the directories specified by the **PATH** environment variable.

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. 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 a copy of the *pathname* argument. (A different value will not produce an error.) Under versions of MS-DOS earlier than 3.0, the passed value of *arg0* or *arg*[**0**] is not available for use in the child process. However, under MS-DOS versions 3.0 and later, the *pathname* is available as *arg0* or *arg*[**0**].

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 *pathname*. 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 *pathname*, 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.

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 allow 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 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. The C start-up code normally processes this entry and then deletes it from the environment. However, if a **spawn** function spawns a non-C process (such as **COMMAND.COM**), this entry will remain in the environment. In this case, since the environment information is passed in binary form, printing the environment will show graphics characters in the definition string for this entry. It should not have any other effect on normal operations.

■ Return Value

The return value from a synchronous **spawn** (**P_WAIT** specified for *modeflag*) is the exit status of the child process.

The return value from an asynchronous **spawn** (**P_NOWAIT** or **P_NOWAITO** specified for *modeflag*) is the process ID. To obtain the exit code for the spawned process, you must call the **wait** or **cwait** function and specify the process ID.

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 set a positive exit status, the 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	The argument list exceeds 128 bytes, or the space required for the environment information exceeds 32K.
EINVAL	The *modeflag* argument is invalid.
ENOENT	The file or path name 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

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.

■ See Also

abort, **atexit**, **execl**, **execle**, **execlp**, **execlpe**, **execv**, **execve**, **execvp**, **execvpe**, **exit**, **_exit**, **onexit**, **system**

■ Example

```
#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
                  };

main(argc, argv)
int argc;
char *argv[];
 {
  char *args[4];
  int result;

  /* 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, "child.exe","child ","spawnl","two",
               NULL);
break;
    case '2':
      spawnle (P_WAIT, "child.exe","child","spawnle","two",
               NULL,my_env);
break;
    case '3':
      spawnlp (P_WAIT, "child.exe","child","spawnlp","two",
               NULL);
break;
    case '4':
      spawnlpe (P_WAIT, "child.exe","child","spawnlpe","two",
                NULL,my_env);
break;
    case '5':
      spawnv  (P_OVERLAY, "child.exe",args);
```

```
break;
    case '6':
      spawnve (P_OVERLAY, "child.exe",args,my_env);
break;
    case '7':
      spawnvp (P_OVERLAY, "child.exe",args);
break;
    case '8':
      spawnvpe(P_OVERLAY, "child.exe",args,my_env);
break;
    default:
      printf("Enter a number from 1 to 8 as a command line
      parameter."); exit();
   }
 printf("\n\nReturned from SPAWN!\n");
 }
```

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.

stat

■ Summary

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

int stat(*pathname***, *buffer*);**
char *pathname; Path name of existing file
struct stat *buffer; Pointer to structure to receive results

■ Description

The **stat** function obtains information about the file or directory specified by *pathname* and stores it in the structure pointed to by *buffer*. The **stat** structure, defined in **sys\stat.h**, contains the following fields:

Field	Value
st_mode	Bit mask for file mode information. **S_IFDIR** bit set if *pathname* specifies a directory; **S_IFREG** bit set if *pathname* specifies an ordinary file. User read/write bits set according to the file's permission mode; user execute bits set using the file-name extension.
st_dev	Drive number of the disk containing the file (same as **st_rdev**).
st_rdev	Drive number of the disk containing the file (same as **st_dev**).
st_nlink	Always 1.
st_size	Size of the file in bytes.
st_atime	Time of last modification of file (same as **st_mtime** and **st_ctime**).
st_mtime	Time of last modification of file (same as **st_atime** and **st_ctime**).
st_ctime	Time of last modification of file (same as **st_atime** and **st_mtime**).

There are three additional fields in the **stat** structure type that do not contain meaningful values under MS-DOS.

■ Return Value

The **stat** function returns the value 0 if the file-status information is obtained. A return value of –1 indicates an error, and **errno** is set to **ENOENT**, indicating that the file name or path name could not be found.

■ See Also

access, **fstat**

Note

If the given *pathname* refers to a device, the size and time fields in the **stat** structure are not meaningful.

■ Example

```
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

struct stat buf;
int fh, result;
char *buffer = "A line to output";

main()
 {

  /* Get data associated with "data": */

  result = stat("data",&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("Time modified : %s",ctime(&buf.st_atime));
    }
}
```

This program uses **stat** to report the size and last modification time for the file named `data`.

■ Summary

# include <process.h>	Required only for function declarations
# include <stdlib.h>	Use either **process.h** or **stdlib.h**
int system(*string*);	
char **string*;	Command to be executed

■ Description

The **system** function passes *string* to the command interpreter and executes the string as an MS-DOS command. The **system** function refers to the **COMSPEC** and **PATH** environment variables which locate the command interpreter file—**COMMAND.COM** in MS-DOS or **CMD.EXE** in 286DOS.

■ Return Value

The **system** function returns the value 0 if *string* is successfully executed. A return value of –1 indicates an error, and **errno** is set to one of the following values:

Value	Meaning
E2BIG	The argument list for the command 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; or the available memory has been corrupted; or an invalid block exists, indicating that the process making the call was not allocated properly.

■ See Also

execl, **execle**, **execlp**, **execv**, **execve**, **execvp**, **exit**, **_exit**, **spawnl**, **spawnle**, **spawnlp**, **spawnv**, **spawnve**, **spawnvp**

■ Example

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

int result;

main()

{
 /* Place version number in "result.log"*/
 result = system("ver >>result.log");

 /* Type "result.log" to the screen     */
 result = system("type result.log");
}
```

This program uses **system** to place the DOS version number in a file named `result.log` and then displays `result.log` on the screen.

■ Summary

 # include <process.h>

int wait(*stat_loc***);**
**int *****stat_loc*;

■ Description

The **wait** function suspends the calling process until any of its immediate child processes terminates. If all of the caller's children have terminated before it calls **wait**, the function returns immediately.

Important

The **wait** function is supported only in the 286DOS protected mode.

If non-**NULL**, *stat-loc* points to a buffer containing a termination status word and return code for the terminated child process. The termination status word indicates whether or not the child terminated normally (i.e., by calling the 286DOS **DOSEXIT** function). If it did terminate normally, the low-order and high-order bytes of the termination status word are as follows:

Byte	Contents
Low order	0
High order	Low-order byte of the result code that the child process passed to **DOSEXIT**. **DOSEXIT** is called if the child process called **exit** or **_exit**, returned from **main**, or reached the end of **main**. The low-order byte of the result code is either the low-order byte of the argument to **_exit** or **exit**, the low-order byte of the return value from **main**, or, if the child process reached the end of **main**, a random value.

If the child process terminated for any other reason, the low-order and high-order bytes of the termination status word are as follows:

Byte	Contents
Low order	Termination code from **DOSWAIT**:
High order	0

Code	Meaning
1	Hard-error abort
2	Trap operation
3	**SIGTERM** signal not intercepted

■ Return Value

If **wait** returns after abnormal termination of a child process, it returns –1 and sets **errno** to **EINTR**.

If **wait** returns after normal termination of a child process, it returns the child's process ID.

Otherwise, **wait** returns –1 immediately and sets **errno** to **ECHILD**, indicating that no child processes exist for the calling process.

■ See Also

cwait, **exit**, **_exit**