UNIX-Type APIs (V5R2)

Signal APIs

Table of Contents

Signal APIs

An X/Open specification defines a "signal" as a mechanism by which a process may be notified of, or affected by, an event occurring in the system. The term signal is also used to refer to the event itself.

For additional information on the Signal APIs, see:

- Signal Concepts
- OS/400 Signal Management
- Differences from Signals on UNIX Systems

The Signal APIs are:

- alarm() (Set schedule for alarm signal) generates a SIGALRM signal after the number of seconds specified by the seconds parameter have elapsed. The delivery of the SIGALRM signal is directed at the calling process.
- getitimer() (Get value of interval timer) returns the value last used to set the interval timer specified by which in the structure pointed to by value.
- kill() (Send signal to process or group of processes) sends a signal to a process or process group specified by pid.
- pause() (Suspend process until signal received) suspends processing of the calling thread.
- Qp0sDisableSignals() (Disable process for signals) prevents the process from receiving signals.
- Qp0sEnableSignals() (Enable process for signals) enables the process to receive signals.
- setitimer() (Set value of interval timer) sets the timer specified by which to the value in the structure pointed to by value and stores the previous value of the timer in the structure pointed to by ovalue.
- sigaction() (Examine and change signal action) examines, changes, or both examines and changes the action associated with a specific signal.
- sigaddset() (Add signal to signal set) is part of a family of functions that manipulate signal sets.
- sigdelset() (Delete signal from signal set) is part of a family of functions that manipulate signal sets.
- sigemptyset() (Initialize and empty signal set) is part of a family of functions that manipulate signal sets.
- sigfillset() (Initialize and fill signal set) is part of a family of functions that manipulate signal sets.
- sigismember() (Test for signal in signal set) is part of a family of functions that manipulate signal sets.
- siglongjmp() (Perform nonlocal goto with signal handling) restores the stack environment previously saved in env by sigsetjmp().
- sigpending() (Examine pending signals) returns signals that are blocked from delivery and pending for either the calling thread or the process.
- sigprocmask() (Examine and change blocked signals) examines, or changes, or both examines and changes the signal mask of the calling thread.
- sigsetjmp() (Set jump point for nonlocal goto) saves the current stack environment and, optionally, the current signal mask.
- sigsuspend() (Wait for signal) replaces the current signal mask of a thread with the signal set given

by *sigmask and then suspends processing of the calling process.

- sigtimedwait() (Synchronously accept a signal for interval of time) selects a pending signal from set, clears it from the set of pending signals for the thread or process, and returns that signal number in the si_signo member in the structure that is referenced by info.
- sigwait() (Synchronously accept a signal) selects a pending signal from set, clears it from the set of pending signals for the thread or process, and returns that signal number in the location that is referenced by sig.
- sigwaitinfo() (Synchronously accept a signal and signal data) selects a pending signal from set, clears it from the set of pending signals for the thread or process, and returns that signal number in the si_signo member in the structure that is referenced by info.
- sleep() (Suspend processing for interval of time) suspends a thread for a specified number of seconds.
- usleep() (Suspend processing for interval of time) suspends a thread for the number of microseconds specified by the of useconds parameter.

Note: These functions use header (include) files from the library QSYSINC, which is optionally installable. Make sure QSYSINC is installed on your system before using any of the functions. See Header Files for UNIX-Type Functions for the file and member name of each header file.

The term "signal" comes from X/Open CAE Specification System Interface Definitions Issue 4, Number 2, Glossary, page 27. X/Open Company Ltd., United Kingdom, 1994.

Using Signal APIs

Signal Concepts

An X/Open specification defines a "signal" as a mechanism by which a process may be notified of, or affected by, an event occurring in the system. The term signal is also used to refer to the event itself.

A signal is said to be **generated** when the event that causes the signal first occurs. Examples of such events include the following:

- System-detected errors
- Timer expiration
- Terminal (work station) activity
- Calling an API such as the X/Open **kill()** function, the American National Standard C **raise()** function, or the ILE **CEESGL** (signal a condition) function.

A **synchronous signal** is a signal that is generated by some action attributable to a program running within the thread, such as a system-detected error, **raise()**, or **CEESGL**. An **asynchronous signal** is a signal that is generated for the process by using the **kill()** function or by an asynchronous event such as terminal activity or an expired timer.

The **signal action vector** is a list of signal-handling actions for each defined signal. The signal action vector is maintained separately for each process and is inherited from the parent process. The signal action vector specifies the signal-handling actions for both synchronously and asynchronously generated signals.

A signal is said to be **delivered** to a process when the specified signal-handling action for the signal is taken. A signal is said to be **accepted** by a process when a signal is selected and returned by one of the *sigwait* functions.

Signals generated for a process are delivered to or accepted by one thread in the process.

A signal is said to be **pending** during the interval between the time the signal is generated and the time it is delivered or accepted. Ordinarily, this interval cannot be detected by an application. However, a signal can be **blocked** from being delivered to a thread. When a signal is blocked, the signal-handling action associated with the signal is not taken. If there are no threads in a call to a *sigwait* function selecting the signal and if all threads block delivery of the signal, the signal remains pending on the process. The signal remains pending until either a thread calls a *sigwait* function selecting the signal, a thread unblocks delivery of the signal, or the signal action associated with the signal is set to ignore the signal. The **signal blocking mask** defines the set of signals that are blocked from delivery to the thread. The signal blocking mask is maintained separately for each thread in the process and is inherited from the thread that created it.

OS/400 Signal Management

The set of defined signals is determined by the system. The system specifies the attributes for each defined signal. These attributes consist of a signal number, the initial signal action, and the signal default action. The system also specifies an initial signal blocking mask. The set of defined signals, the signal attributes, and signal blocking mask are referred to as **signal controls**.

A signal can be generated or delivered only to a process that has expressed an interest in signals. An error condition results under the following conditions:

- An attempt is made to generate a signal when the system signal controls have not been initialized.
- An attempt is made to generate a signal for a process that has not been enabled for signals.

A process can express an interest in signals by calling the **Qp0sEnableSignals()** API. In addition, calling particular signal APIs implicitly enables the process for signals.

If the process has not been enabled for signals, the process signal controls are set from signal controls established by the system during IPL (the system signal controls). An error condition results if an attempt is made to enable signals for the process before the system signal controls have been initialized.

Once the process signal controls have been initialized, the user is permitted to change the signal controls for the process. For example, the signal blocking mask and the signal action for a signal are commonly changed. Some signal controls, such as the number of defined signals and the signal default action for a signal, cannot be changed at the process level.

The attributes for each defined signal are stored in an object called a **signal monitor**. The system supports a maximum of 63 signal monitors for each process. The process signal action vector is a list of signal monitors, one for each defined signal. The signal monitor contains, but is not limited to, the following information:

- Signal action
- Signal default action
- Signal options

The **signal action** defines the action to be taken by the system when a process receives an unblocked signal. The user can change the signal action for a process signal monitor. The possible signal actions are:

- Handle using signal default action (SIG_DFL)

  The *handle using signal default action* signal action indicates that the system is to take the action specified by the signal default action field when the signal is eligible to be delivered.
- Ignore the signal (SIG_IGN)

  The *ignore the signal* signal action indicates that the user is not interested in handling the signal. When an ignored signal is generated for the process, the system automatically discards the signal, regardless of the blocked or unblocked state of the signal monitor.
- Handle the signal by running signal-catching function

  The *handle the signal by running signal-catching function* signal action causes the system to call the signal-catching function when a signal is received for the signal monitor. The signal-catching function is set to point to a procedure within an active activation group.

The signal default action field defines the action to be taken by the system when the signal action is set to *handle using signal default action*. The signal default action for a signal monitor is set in the system signal controls and cannot be changed for a process signal monitor. The possible signal default actions are:

- Terminate the process

  The *terminate the process* action puts the process in a phase that ends the process, allowing cancel handlers to be called. If the process is already in the end phase, the *terminate the process* action is ignored.
- End the request

  The *end the request* action results in the cancelation of all calls up to the nearest call that has a call status of request processor. If a call with a status of request processor is not present or the job is capable of having multiple threads, the *terminate the process* action is taken.

- Ignore the signal

  The *ignore the signal* action causes the system to discard the signal. A signal is discarded for a signal monitor in the blocked state when the signal action is *handle using signal default action* and the default signal action is *ignore the signal*.

- Stop the process

  The *stop the process* action causes the system to place the process in the stopped state. When a process is in the **stopped** state, it is temporarily suspended until a signal is generated for the process that has *continue the process if stopped* as its signal default action. When a process is in the stopped state, the normal process control functions remain in effect (the process can be suspended, resumed, or ended). When a signal is generated for a signal monitor that has *stop the process* as its signal default action, the system removes any pending signals for signal monitors that have *continue the process if stopped* as their default action.

- Continue the process if stopped

  The *continue the process if stopped* action causes the system to resume running the process that is in the stopped state, even if the signal monitor with the signal default action of *continue the process if stopped* is in the blocked state or has a signal action of *ignore the signal*. When a signal is generated for a signal monitor that has *continue the process if stopped* as its signal default action, the system removes any pending signals for signal monitors that have *stop the process* as their signal default action.

- Signal exception

  The *signal exception* action causes the system to send the MCH7603 escape message to the process.

The **signal options** specify an additional set of attributes for the signal monitor. The primary use of these options is to specify an additional set of actions to be taken by the system when a signal-catching function is called.

A signal is generated by sending a request to a signal monitor. Scheduling of the signal-handling action is controlled separately for each signal monitor through the **signal blocking mask**. The signal blocking mask is a bit mask that defines the set of signals to be blocked from delivery to the thread. The blocked or unblocked option specified for the *nth* bit position in the signal blocking mask is applied to the *nth* signal monitor defined for the process. When `signal is unblocked` is specified, the signal-handling action is eligible to be scheduled. When `signal is blocked` is specified, the signal-handling action is blocked from delivery.

The process to receive the signal is identified by a **process ID**. The process ID is used to indicate whether the signal should be sent to an individual process or to a group of processes (known as a process group). The process ID is a 4-byte binary number that is used to locate an entry in the system-managed process table. A process table entry contains the following information relating to the process:

- Parent process ID
- Process group ID
- Status information

The **parent process** is the logical creator of the process. A **process group** represents a collection of processes that are bound together for some common purpose. An error condition results if the process ID specified when a signal is sent does not represent a valid process or process group.

The process sending a signal must have the appropriate authority to the receiving process. The parent process is allowed to send a signal to a child process (the parent process ID of the receiving process is equal to the process ID of the process sending the signal). A child process is allowed to send a signal to its parent process (the process ID of the receiving process is equal to the parent process ID of the process sending the signal). A process can send a signal to another process if the sending process has *JOBCTL authority defined for the current process user profile or in an adopted user profile. Otherwise, the real or effective user ID of the sending process must match the real or effective user ID of the receiving process. An error condition results if the process does not have authority to send the signal to a receiving process.

Differences from Signals on UNIX Systems

The OS/400 support for signals does differ from the usual behavior of signals on UNIX systems:

- Integration of American National Standard C signal model and X/Open signal model

  On UNIX systems, the standard C signal functions (as defined by American National Standards Institute (ANSI)) and the UNIX signal functions interact. That is, the standard C **signal()** function operates on the process signal action vector. Likewise, when a signal is generated for a process using the standard C **raise()** function, the process signal blocking mask and the signal action vector are used to determine the action to be taken.

  On OS/400, the behavior of the standard C signal functions depends on a compiler option. When the compiler option **SYSIFCOPT(*ASYNCSIGNAL)** is specified, the standard C **signal()** and **raise()** functions operate like the UNIX signal functions by operating on the process signal action vector and the process signal blocking mask. However, if the **SYSIFCOPT(*ASYNCSIGNAL)** is not specified the standard C signal functions do not operate like the UNIX signal functions. Although the default C signal model does not interact with the UNIX signal functions, the UNIX signal functions **sigaction()** and **kill()** provide the same type of capability as the standard C **signal()** and **raise()** functions. For more information, see sigaction()--Examine and Change Signal Action and kill()--Send Signal to Process or Group of Processes.

- Scope of signal action vector, signal-blocking mask, and pending signals

  On most UNIX systems, a process consists of a single thread of control. When the program in control needs to perform a task that is contained in another program, the program uses the **fork()** and **exec()** functions to start a child process that runs the other program. The signal controls for the child process are inherited from the parent process. Changes to the signal controls in either the parent or the child process are isolated to the process in which the change is made.

  On OS/400, when a program needs to perform a task that is contained in another program, the program calls that program directly. The target program is run using the same process structure. As a result of this call and return mechanism, if a called program changes the process signal controls and does not restore the original signal controls when returning to its caller, the changed process signal controls remain in effect. The called program inherits the signal controls of its caller. However, there are some differences from what would be expected if **fork()** and **exec()** were used in a UNIX process:

  - The set of pending signals is not cleared.
  - Alarms are not reset.
  - Signals set to be caught are not reset to the default action.

  Programs that use signals and change the signal controls of the process should restore the old actions or signal blocking mask (or both) when they return to their callers. Programs using signals should explicitly enable the process for signals when the program begins. If the process was not enabled for signals when the program was called, the program should also disable signals when it returns to the process. For more information, see Qp0sEnableSignals()--Enable Process for Signals and Qp0sDisableSignals()--Disable Process for Signals.

- Mapping system-detected errors to signals

  On UNIX systems, system-detected errors are mapped to signal numbers. For example, a floating point error results in the SIGFPE signal being generated for the process. On OS/400, the default C signal model presents system-detected errors to the user as escape messages which can be handled with C signal handlers established with the C **signal()** function or with ILE C exception-handling functions, but not with signal handlers established with the UNIX **sigaction()** function. When the compiler option **SYSIFCOPT(*ASYNCSIGNAL)** is specified, system-detected errors are mapped to signal numbers and can be handled with signal handlers established either with the C **signal()** function or the UNIX **sigaction()** function, but not with ILE C exception-handling functions.

- Unexpected error handling in the signal-catching function

  On UNIX systems, an unhandled error condition in a signal-catching function results in ending the process. On OS/400, unhandled error conditions in the signal-catching function are implicitly handled. The signal-catching function is ended and the receiving program resumes running at the point at which it was interrupted. The error condition may be logged in the job log. Aside from the job log entry for the error, no further error notification takes place.

- Termination action

  OS/400 offers two types of termination actions. The termination action applied to most signals is to end the most recent request. This usually results in ending the current program, which is the expectation of most UNIX programmers. The second termination action is to end the process, which is more severe. The only signal with this action is SIGKILL.

- Default actions

  On OS/400, some default actions for signals are different than on typical UNIX systems. For example, the OS/400 default action for the SIGPIPE signal is to ignore the signal.

---

alarm()--Set Schedule for Alarm Signal

Syntax

```
#include <unistd.h>

unsigned int alarm( unsigned int seconds );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **alarm()** function generates a SIGALRM signal after the number of seconds specified by the *seconds* parameter have elapsed. The delivery of the SIGALRM signal is directed at the calling process.

seconds is the number of real seconds to elapse before the SIGALRM is generated. Because of processor delays, the SIGALRM may be generated slightly later than this specified time. If *seconds* is zero, any previously set alarm request is canceled.

Only one such alarm can be active at a time for the process. If a new alarm time is set, any previous alarm is canceled.

Parameters

seconds

(Input) The number of real seconds to elapse before generating the signal.

Return Value

value	**alarm()** was successful. The value returned is one of the following: ● A nonzero value that is the number of real seconds until the previous **alarm()** request would have generated a SIGALRM signal. ● A value of zero if there was no previous **alarm()** request with time remaining.
-1	**alarm()** was not successful. The *errno* variable is set to indicate the error.

Error Conditions

If **alarm()** is not successful, *errno* usually indicates the following error. Under some conditions, *errno* could indicate an error other than that listed here.

[ENOTSIGINIT]	Process not enabled for signals.

An attempt was made to call a signal function under one of the following conditions:

- The signal function is being called for a process that is not enabled for asynchronous signals.
- The signal function is being called when the system signal controls have not been initialized.

Usage Notes

The **alarm()** function enables a process for signals if the process is not already enabled for signals. For details, see Qp0sEnableSignals()--Enable Process for Signals. If the system has not been enabled for signals, **alarm()** is not successful, and an [ENOTSIGINIT] error is returned.

Related Information

- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- The <**unistd.h**> file
- pause()--Suspend Process Until Signal Received
- Qp0sDisableSignals()--Disable Process for Signals
- Qp0sEnableSignals()--Enable Process for Signals
- setitimer()--Set Value for Interval Timer
- sigaction()--Examine and Change Signal Action
- sigsuspend()--Wait for Signal
- sleep()--Suspend Processing for Interval of Time
- usleep()--Suspend Processing for Interval of Time

Example

See Code disclaimer information for information pertaining to code examples.

The following example generates a SIGALRM signal using the **alarm()** function:

```
#include <signal.h>
#include <unistd.h>
```

```
#include <stdio.h>
#include <time.h>
#include <errno.h>

#define LOOP_LIMIT  1E6

volatile int sigcount=0;

void catcher( int sig ) {
    printf( "Signal catcher called for signal %d\n", sig );
    sigcount = 1;
}

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

    struct sigaction sact;
    volatile double count;
    time_t t;

    sigemptyset( &sact.sa_mask );
    sact.sa_flags = 0;
    sact.sa_handler = catcher;
    sigaction( SIGALRM, &sact, NULL );

    alarm(5);  /* timer will pop in five seconds */

    time( &t );
    printf( "Before loop, time is %s", ctime(&t) );

    for( count=0; ((count<LOOP_LIMIT) && (sigcount==0)); count++ );

    time( &t );
    printf( "After loop, time is %s\n", ctime(&t) );

    if( sigcount == 0 )
        printf( "The signal catcher never gained control\n" );
    else
        printf( "The signal catcher gained control\n" );

    printf( "The value of count is %.0f\n", count );

    return( 0 );
}
```

Output:

```
    Before loop, time is Sun Jan 22 10:14:00 1995
    Signal catcher called for signal 14
    After loop, time is Sun Jan 22 10:14:05 1995
    The signal catcher gained control
    The value of count is 290032
```

API introduced: V3R6

Top | UNIX-Type APIs | APIs by category

getitimer()--Get Value for Interval Timer

Syntax

```
#include <sys/time.h>

int getitimer( int which, struct itimerval *value );
```

Service Program Name: QP0SSRVI

Default Public Authority: *USE

Threadsafe: Yes

The **getitimer()** function returns the value last used to set the interval timer specified by which in the structure pointed to by value.

Parameters

which

(Input) The interval timer type.

The possible values for which, which are defined in the <**sys/time.h**> header file, are as follows:

ITIMER_REAL	The interval timer value is decremented in real time. The SIGALRM signal is generated for the process when this timer expires.
ITIMER_VIRTUAL	The interval timer value is only decremented when the process is running. The SIGVTALRM signal is generated for the process when this timer expires.
ITIMER_PROF	The interval timer value is only decremented when the process is running or when the system is running on behalf of the process. The SIGPROF signal is generated for the process when this timer expires.

value

(Output) A pointer to the space where the current interval timer value is stored.

Return Value

0	**getitimer()** was successful.
-1	**getitimer()** was not successful. The errno variable is set to indicate the error.

Error Conditions

If **getitimer()** is not successful, errno usually indicates the following error. Under some conditions, errno could indicate an error other than that listed here.

[EINVAL] An invalid parameter was found.

A parameter passed to this function is not valid.

- The value of which is not equal to one of the defined values.

[ENOTSIGINIT] Process not enabled for signals.

An attempt was made to call a signal function under one of the following conditions:

- The signal function is being called for a process that is not enabled for asynchronous signals.
- The signal function is being called when the system signal controls have not been initialized.

Related Information

- The <**sys/time.h**> file (see Header Files for UNIX-Type Functions)
- alarm()--Set Schedule for Alarm Signal
- setitimer()--Set Value for Interval Timer
- sleep()--Suspend Processing for Interval of Time
- usleep()--Suspend Processing for Interval of Time

Example

See Code disclaimer information for information pertaining to code examples.

The following example returns the current interval timer value using the **getitimer()** function:

```
#include <sys/time.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include <errno.h>

#define LOOP_LIMIT  1E12

volatile int sigcount=0;
```

```
void catcher( int sig ) {

    struct itimerval value;
    int which = ITIMER_REAL;

    printf( "Signal catcher called for signal %d\n", sig );
    sigcount++;

    if( sigcount > 1 ) {

        /*
         * Disable the real time interval timer
         */

        getitimer( which, &value );

        value.it_value.tv_sec = 0;
        value.it_value.tv_usec = 0;

        setitimer( which, &value, NULL );
    }
}

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

    int result = 0;

    struct itimerval value, ovalue, pvalue;
    int which = ITIMER_REAL;

    struct sigaction sact;
    volatile double count;
    time_t t;

    sigemptyset( &sact.sa_mask );
    sact.sa_flags = 0;
    sact.sa_handler = catcher;
    sigaction( SIGALRM, &sact, NULL );

    getitimer( which, &pvalue );

    /*
     * Set a real time interval timer to repeat every 200 milliseconds
     */

    value.it_interval.tv_sec = 0;        /* Zero seconds */
    value.it_interval.tv_usec = 200000;  /* Two hundred milliseconds */
    value.it_value.tv_sec = 0;           /* Zero seconds */
    value.it_value.tv_usec = 500000;     /* Five hundred milliseconds */

    result = setitimer( which, &value, &ovalue );

    /*
     * The interval timer value returned by setitimer() should be
     * identical to the timer value returned by getitimer().
     */
    if( ovalue.it_interval.tv_sec != pvalue.it_interval.tv_sec  ||
        ovalue.it_interval.tv_usec != pvalue.it_interval.tv_usec ||
        ovalue.it_value.tv_sec != pvalue.it_value.tv_sec ||
        ovalue.it_value.tv_usec != pvalue.it_value.tv_usec ) {
        printf( "Real time interval timer mismatch\n" );
        result = -1;
    }


    time( &t );
    printf( "Before loop, time is %s", ctime(&t) );

    for( count=0; ((count<LOOP_LIMIT) && (sigcount<2)); count++ );

    time( &t );
    printf( "After loop, time is %s\n", ctime(&t) );


    if( sigcount == 0 )
        printf( "The signal catcher never gained control\n" );
    else
        printf( "The signal catcher gained control\n" );

    printf( "The value of count is %.0f\n", count );

    return( result );
}
```

Output:

```
    Before loop, time is Sun Jun 15 10:14:00 1997
    Signal catcher called for signal 14
    Signal catcher called for signal 14
    After loop, time is Sun Jun 15 10:14:01 1997
    The signal catcher gained control
    The value of count is 702943
```

API introduced: V4R2

---

kill()--Send Signal to Process or Group of Processes

Syntax

```
#include <sys/types.h>
#include <signal.h>

int kill( pid_t pid, int sig );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **kill()** function sends a signal to a process or process group specified by *pid*. The signal to be sent is specified by *sig* and is either 0 or one of the signals from the list in the <**sys/signal.h**> header file.

The process sending the signal must have appropriate authority to the receiving process or processes. The **kill()** function is successful if the process has permission to send the signal *sig* to any of the processes specified by *pid*. If **kill()** is not successful, no signal is sent.

A process can use **kill()** to send a signal to itself. If the signal is not blocked in the sending thread, and if no other thread has the *sig* unblocked or is waiting in a *sigwait* function for *sig*, either *sig* or at least one pending unblocked signal is delivered to the sender before **kill()** returns.

Parameters

pid

(Input) The process ID or process group ID to receive the signal.

sig

(Input) The signal to be sent.

pid and *sig* can be used as follows:

pid_t pid; Specifies the processes that the caller wants to send the signal to:

- If *pid* is greater than zero, **kill()** sends the signal *sig* to the process whose ID is equal to *pid*.
- If *pid* is equal to zero, **kill()** sends the signal *sig* to all processes whose process group ID is equal to that of the sender, except for those to which the sender does not have the appropriate authority to send a signal.
- If *pid* is equal to -1, **kill()** returns -1 and *errno* is set to [ESRCH].
- If *pid* is less than -1, **kill()** sends the signal *sig* to all processes whose process group ID is equal to the absolute value of *pid*, except for those to which the sender does not have appropriate authority to send a signal.

int sig; The signal that should be sent to the processes specified by *pid*. This must be zero, or one of the signals defined in the <**sys/signal.h**> header file. If *sig* is zero, **kill()** performs error checking, but does not send a signal. You can use a *sig* value of zero to check whether the *pid* argument is valid.

Authorities

The thread sending the signal must have the appropriate authority to the receiving process. A thread is allowed to send a signal to a process if at least one of the following conditions is true:

- The thread is sending a signal to its own process.
- The thread has *JOBCTL special authority defined in the currently running user profile or in a current adopted user profile.
- The thread belongs to a process that is the parent of the receiving process. (The process being signaled has a *parent process ID* equal to the *process ID* of the thread sending the signal.)
- If the receiving process is multi-threaded,
    - The real or effective user ID of the thread matches the *job user identity* of the process receiving process (the process being signaled).
- Otherwise,
    - The real or effective user ID of the thread matches the real or effective user ID of the process being signaled. If _POSIX_SAVED_IDS is defined in the <**unistd.h**> include file, the saved set user ID of the intended recipient is checked instead of its effective user ID.

The *job user identity* is the name of the user profile by which a job is known to other jobs. It is described in more detail in the Work Management book on the V5R1 Supplemental Manuals Web site.

When sending a signal affects entries for multiple processes, the signal is generated for each process to which the process sending the signal is authorized. If the process does not have permission to send the signal to any receiving process, the [EPERM] error is returned.

Regardless of user ID, a process can always send a SIGCONT signal to a process that is a member of the same process group (same *process group ID*) as the sender.

Return Value

0	**kill()** was successful. It had permission to send *sig* to one or more of the processes specified by *pid*.
-1	**kill()** was not successful. It failed to send a signal. The *errno* variable is set to indicate the error.

Error Conditions

If **kill()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EINVAL]
An invalid parameter was found.

A parameter passed to this function is not valid.

The value of *sig* is not within the range of signal numbers or is a signal that is not supported.

[ENOTSIGINIT]
Process not enabled for signals.

An attempt was made to call a signal function under one of the following conditions:

- The signal function is being called for a process that is not enabled for asynchronous signals.
- The signal function is being called when the system signal controls have not been initialized.

[ENOSYSRSC]
System resources not available to complete request.

[EPERM]
Operation not permitted.

You must have appropriate privileges or be the owner of the object or other resource to do the requested operation.

[ESRCH]
No item could be found that matches the specified value.

The process or process group specified in *pid* cannot be found.

Usage Notes

1. If the value of *pid* is 0 (so that **kill()** is used to send a signal to all processes whose process group ID is equal to that of the sender), **kill()** enables the process for signals if the process is not already enabled for signals. For details, see Qp0sEnableSignals()--Enable Process for Signals.

2. A process can use **kill()** to simulate the American National Standard C **raise()** function by using the following:

```
sigset_t  sigmask;

/*
 * Allow all signals to be delivered by unblocking all signals
*/

sigemtyset( &sigmask );
sigprocmask( SIG_SETMASK, &sigmask, NULL );

...

kill( getpid(), SIGUSR1 );
```

   The example above ensures that no signals are blocked from delivery. When the **kill()** function is called, the behavior is the same as calling the **raise()** function.

Related Information

- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- The <**sys/types.h**> file (see Header Files for UNIX-Type Functions)
- Qp0sDisableSignals()--Disable Process for Signals
- Qp0sEnableSignals()--Enable Process for Signals
- sigaction()--Examine and Change Signal Action
- sigtimedwait()--Synchronously Accept a Signal for Interval of Time
- sigwait()--Synchronously Accept a Signal
- sigwaitinfo()--Synchronously Accept a Signal and Signal Data

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses the **kill()** function:

```
#include <signal.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <time.h>

int sendsig( int );
volatile int sigcount=0;

void catcher( int sig ) {
    sigcount++;
}

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

    struct sigaction sigact;
    int result;

    /* set up a signal catching function to handle the signals */
    /* that will be sent from the sendsig() function */

    sigemptyset( &sigact.sa_mask );
    sigact.sa_flags = 0;
    sigact.sa_handler = catcher;
    sigaction( SIGUSR1, &sigact, NULL );

    /* Call the sendsig() function that will call the kill() */
    /* function for SIGUSR1 n times based on the input value */

    result = sendsig( 21 );

    printf( "Back in main\n" );
    printf( "The kill() function was called %d times\n", result );
    printf( "The signal catching function was called %d times\n", \
             sigcount );

    return( 0 );
}

int sendsig( int count ) {

    int i;
    int j=0;

    for( i=0; i < count; i++ ) {
        if( i == ((i/10)*10) ) {
            j++;
            kill( getpid(), SIGUSR1 );
        }
    }
    return( j );
}
```

Output:

```
    Back in main
    The kill() function was called 3 times
    The signal catching function was called 3 times
```

API introduced: V3R6

---

pause()--Suspend Process Until Signal Received

Syntax

```
#include <unistd.h>

int pause( void );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **pause()** function suspends processing of the calling thread. The thread does not resume until a signal is delivered whose action is to call a signal-catching function, end the request, or terminate the process. Some signals can be blocked by the thread's *signal mask*. See sigprocmask()--Examine and Change Blocked Signals for details.

If an incoming unblocked signal has an action of end the request or terminate the process, **pause()** never returns to the caller. If an incoming signal is handled by a signal-catching function, **pause()** returns after the signal-catching function returns.

Parameters

None.

Return Value

There is no return value to indicate successful completion.

Error Conditions

If **pause()** returns, *errno* indicates the following:

-1	**pause()** was not successful. The *errno* variable is set to indicate the reason.
[EINTR]	Interrupted function call.
	A signal was received and handled by a signal-catching function that returned.

[ENOTSIGINIT]	Process not enabled for signals. An attempt was made to call a signal function under one of the following conditions: ● The signal function is being called for a process that is not enabled for asynchronous signals. ● The signal function is being called when the system signal controls have not been initialized.
[EWOULDBLOCK]	Operation would have caused the process to be suspended. The current thread state would prevent the signal function from completing.

Usage Notes

The **pause()** function enables a process for signals if the process is not already enabled for signals. For details, see Qp0sEnableSignals()--Enable Process for Signals. If the system has not been enabled for signals, **pause()** is not successful, and an [ENOTSIGINIT] error is returned.

Related Information

- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- alarm()--Set Schedule for Alarm Signal
- kill()--Send Signal to Process or Group of Processes
- Qp0sDisableSignals()--Disable Process for Signals
- Qp0sEnableSignals()--Enable Process for Signals
- sigprocmask()--Examine and Change Blocked Signals
- sigsuspend()--Wait for Signal
- sigtimedwait()--Synchronously Accept a Signal for Interval of Time
- sigwait()--Synchronously Accept a Signal
- sigwaitinfo()--Synchronously Accept a Signal and Signal Data
- sleep()--Suspend Processing for Interval of Time

Example

See Code disclaimer information for information pertaining to code examples.

The following example suspends processing using the **pause()** function and determines the current time:

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <time.h>

void catcher( int sig ) {
    printf( "Signal catcher called for signal %d\n", sig );
}

void timestamp( char *str ) {
    time_t t;

    time(&t);
    printf( "The time %s is %s\n", str, ctime(&t) );
}

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

    struct sigaction sigact;

    sigemptyset( &sigact.sa_mask );
    sigact.sa_flags = 0;
    sigact.sa_handler = catcher;
    sigaction( SIGALRM, &sigact, NULL );

    alarm( 10 );

    timestamp( "before pause" );
    pause();
    timestamp( "after pause" );

    return( 0 );
}
```

Output:

```
    The time before pause is Sun Jan 22 11:09:08 1995
    Signal catcher called for signal 14
    The time after pause is Sun Jan 22 11:09:18 1995
```

API introduced: V3R6

Qp0sDisableSignals()--Disable Process for Signals

Syntax

```
#include <signal.h>

int Qp0sDisableSignals( void );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **Qp0sDisableSignals()** function prevents the process from receiving signals.

After **Qp0sDisableSignals()** is called, the process is no longer eligible to receive signals from another process or the system. Calls to functions that examine the signal action or the signal blocking mask of the thread will not return the requested information. For details on those functions, see sigaction()--Examine and Change Signal Action and sigprocmask()--Examine and Change Blocked Signals.

If the process is currently disabled for signals, a call to **Qp0sDisableSignals()** has no effect and an [ENOTSIGINIT] error is returned.

Parameters

None

Return Value

0	**Qp0sDisableSignals()** was successful.
-1	**Qp0sDisableSignals()** was not successful. The *errno* variable is set to indicate the error.

Error Conditions

If **Qp0sDisableSignals()** is not successful, *errno* usually indicates the following error. Under some conditions, *errno* could indicate an error other than that listed here.

[ENOTSIGINIT] Process not enabled for signals.

An attempt was made to call a signal function under one of the following conditions:

- The signal function is being called for a process that is not enabled for asynchronous signals.
- The signal function is being called when the system signal controls have not been initialized.

Usage Notes

1. Processes, by default, are not eligible to receive signals from other processes or the system. However, once a process has been enabled for signals, it remains eligible to receive signals until either it ends or some user action is taken to prevent the delivery of signals.

   Use of the following functions enables a process for signals:
   - alarm()
   - getpgrp()
   - getpid()
   - kill()
   - pause()
   - Qp0wGetPgrp()
   - Qp0wGetPid()
   - setitimer()
   - sigaction()
   - sigprocmask()
   - sigsuspend()
   - sigtimedwait()
   - sigwait()
   - sigwaitinfo()
   - sleep()

   Any of the Pthread APIs. See Pthread APIs for more information.
2. The user of signals can prevent the signals from being delivered to the process by calling the **sigprocmask()** function. The user can also ignore the signal by calling the **sigaction()** function. However, not all signals can be blocked or ignored. For details, see sigaction()--Examine and Change Signal Action and sigprocmask()--Examine and Change Blocked Signals. The **Qp0sDisableSignals()** function provides a means of preventing the calling process from receiving any signal from other processes or the system.
3. If a process has not been enabled for signals, the signal blocking mask for any thread created in the process will be set to the empty set.
4. If a process with multiple threads is disabled for signals by calling **Qp0sDisableSignals()** and then later re-enabled for signals, only the thread that causes signals to be enabled will have its signal blocking mask changed. The signal blocking mask for all other threads will be the value last used to

set the signal blocking mask for those threads.

Related Information

- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- alarm()--Set Schedule for Alarm Signal
- kill()--Send Signal to Process or Group of Processes
- pause()--Suspend Process Until Signal Received
- Qp0sEnableSignals()--Enable Process for Signals
- setitimer()--Set Value for Interval Timer
- sigaction()--Examine and Change Signal Action
- sigprocmask()--Examine and Change Blocked Signals
- sigsuspend()--Wait for Signal
- sigtimedwait()--Synchronously Accept a Signal for Interval of Time
- sigwait()--Synchronously Accept a Signal
- sigwaitinfo()--Synchronously Accept a Signal and Signal Data
- sleep()--Suspend Processing for Interval of Time

Example

See Code disclaimer information for information pertaining to code examples.

The following example shows how a process can reset its signal vector and signal blocking mask.

```
#include <signal.h>
#include <time.h>
#include <unistd.h>
#include <stdio.h>

void timestamp( char *str ) {

    time_t t;
    time( &t );
    printf( "%s the time is %s\n", str, ctime(&t) );
}

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

    unsigned int ret;

    timestamp( "before sleep()" );

    /*
     *  The sleep() function implicitly enables the process to
     *  receive signals.
     */

    ret = sleep( 10 );

    timestamp( "after sleep()" );

    printf( "sleep() returned %d\n", ret );

    /*
     *  Qp0sDisableSignals() prevents the process from receiving
     *  signals.  If the call to the Qp0sDisableSignals() function
     *  is not done, the process would remain eligible to receive
     *  signals after the return from main().
     */

    Qp0sDisableSignals();

    return( 0 );

}
```

Output:

```
    before sleep() the time is Sun Jan 22 17:25:17 1995
    after sleep() the time is Sun Jan 22 17:25:28 1995
    sleep() returned 0
```

API introduced: V3R6

Qp0sEnableSignals()--Enable Process for Signals

Syntax

```
#include <signal.h>

int Qp0sEnableSignals( void );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **Qp0sEnableSignals()** function enables the process to receive signals.

The **Qp0sEnableSignals()** function causes the process signal vector to be initialized for the set of supported signals. The signal handling action for each supported signal is set to the default action, as defined by **sigaction()** (see sigaction()--Examine and Change Signal Action). The signal blocking mask of the calling thread is set to the empty signal set (see sigemptyset()--Initialize and Empty Signal Set).

If the process is currently enabled for signals, a call to the **Qp0sEnableSignals()** has no effect. That is, the process signal vector and the signal blocking mask of the calling thread are unchanged and an [EALREADY] error is returned.

Parameters

None

Return Value

0	**Qp0sEnableSignals()** was successful.
-1	**Qp0sEnableSignals()** was not successful. The *errno* variable is set to indicate the error.

Error Conditions

If **Qp0sEnableSignals()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EALREADY]	Operation already in progress.
	The calling process is currently enabled for signals.
[ENOTSIGINIT]	Process not enabled for signals.
	An attempt was made to call a signal function under one of the following conditions: ● The signal function is being called for a process that is not enabled for asynchronous signals. ● The signal function is being called when the system signal controls have not been initialized.

Usage Notes

1. Processes, by default, are not eligible to receive signals from other processes or the system. The **Qp0sEnableSignals**() function allows the calling process to receive signals from other processes or the system without having to call other signal functions that enable the process for signals.

   Use of the following functions enable a process for signals:

   - alarm()
   - getpgrp()
   - getpid()
   - kill()
   - pause()
   - Qp0wGetPgrp()
   - Qp0wGetPid()
   - setitimer()
   - sigaction()
   - sigprocmask()
   - sigsuspend()
   - sigtimedwait()
   - sigwait()
   - sigwaitinfo()
   - sleep()

   Any of the Pthread APIs. See Pthread APIs for more information.
2. Once a process has been enabled for signals, it remains eligible to receive signals until either it ends or some user action is taken to prevent the delivery of signals. The user of signals can prevent the signals from being delivered by calling the **sigprocmask()** function. The user can also ignore the signal by calling the **sigaction()** function. However, not all signals can be blocked or ignored. For details, see sigaction()--Examine and Change Signal Action and sigprocmask()--Examine and Change Blocked Signals.
3. If a process has not been enabled for signals, the signal blocking mask for any thread created in the process will be set to the empty set.
4. If a process with multiple threads is disabled for signals by calling **Qp0sDisableSignals()** and then

later re-enabled for signals, only the thread that causes signals to be enabled will have its signal blocking mask changed. The signal blocking mask for all other threads will be the value last used to set the signal blocking mask for those threads.

Related Information

- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- alarm()--Set Schedule for Alarm Signal
- kill()--Send Signal to Process or Group of Processes
- pause()--Suspend Process Until Signal Received
- Qp0sDisableSignals()--Disable Process for Signals
- setitimer()--Set Value for Interval Timer
- sigaction()--Examine and Change Signal Action
- sigprocmask()--Examine and Change Blocked Signals
- sigsuspend()--Wait for Signal
- sigtimedwait()--Synchronously Accept a Signal for Interval of Time
- sigwait()--Synchronously Accept a Signal
- sigwaitinfo()--Synchronously Accept a Signal and Signal Data
- sleep()--Suspend Processing for Interval of Time

Example

See Code disclaimer information for information pertaining to code examples.

The following example shows how a process can reset its signal vector and signal blocking mask:

```
#include <signal.h>
#include <errno.h>

int resetSignals( void ) {

    int return_value;
    return_value = Qp0sEnableSignals();
    if( return_value == -1 ) {
        Qp0sDisableSignals();
        return_value = Qp0sEnableSignals();
    }
    return( return_value );
}
```

API introduced: V3R6

---

setitimer()--Set Value for Interval Timer

Syntax

```
 #include <sys/time.h>

 int setitimer( int which,
                const struct itimerval *value,
                struct itimerval *ovalue );
```

Service Program Name: QP0SSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **setitimer()** function sets the timer specified by which to the value in the structure pointed to by value and stores the previous value of the timer in the structure pointed to by ovalue.

Parameters

which

(Input) The interval timer type.

The possible values for which, which are defined in the <**sys/time.h**> header file, are as follows:

ITIMER_REAL	The interval timer value is decremented in real time. The SIGALRM signal is generated for the process when this timer expires.
ITIMER_VIRTUAL	The interval timer value is only decremented when the process is running. The SIGVTALRM signal is generated for the process when this timer expires.
ITIMER_PROF	The interval timer value is only decremented when the process is running or when the system is running on behalf of the process. The SIGPROF signal is generated for the process when this timer expires.

value

(Input) A pointer to the interval timer structure to be used to change the interval timer value.

The timer value is defined by the itimerval structure. If it_value is non-zero, it indicates the time to the next timer expiration. If it_interval is non-zero, it indicates the time to be used to reset the timer when the it_value time elapses. If it_value is zero, the timer is disabled and the value of it_interval is ignored. If it_interval is zero, the timer is disabled after the next timer expiration.

ovalue

(Output) A pointer to the space where the previous interval timer value is stored. This value may be

NULL.

Return Value

0	**setitimer()** was successful.
-1	**setitimer()** was not successful. The errno variable is set to indicate the error.

Error Conditions

If **setitimer()** is not successful, errno usually indicates the following error. Under some conditions, errno could indicate an error other than that listed here.

[EINVAL] An invalid parameter was found.

A parameter passed to this function is not valid.

- The value of which is not equal to one of the defined values.
- The tv_usec member of the it_value structure has a value greater than or equal to 1,000,000.
- The tv_usec member of the it_interval structure has a value greater than or equal to 1,000,000.

[ENOSYSRSC] System resources not available to complete request.

- The ITIMER_VIRTUAL value for which is not supported on this implementation.
- The ITIMER_PROF value for which is not supported on this implementation.

[ENOTSIGINIT] Process not enabled for signals.

An attempt was made to call a signal function under one of the following conditions:

- The signal function is being called for a process that is not enabled for asynchronous signals.
- The signal function is being called when the system signal controls have not been initialized.

Usage Notes

The **setitimer()** function enables a process for signals if the process is not already enabled for signals. For details, see Qp0sEnableSignals()--Enable Process for Signals. If the system has not been enabled for signals, **setitimer()** is not successful, and an [ENOTSIGINIT] error is returned.

Related Information

- The <**sys/time.h**> file (see Header Files for UNIX-Type Functions)
- alarm()--Set Schedule for Alarm Signal
- setitimer()--Set Value for Interval Timer
- sleep()--Suspend Processing for Interval of Time
- usleep()--Suspend Processing for Interval of Time

Example

See Code disclaimer information for information pertaining to code examples.

The following example returns the current interval timer value using the **setitimer()** function:

```
#include <sys/time.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include <errno.h>

#define LOOP_LIMIT  1E12

volatile int sigcount=0;

void catcher( int sig ) {

    struct itimerval value;
    int which = ITIMER_REAL;

    printf( "Signal catcher called for signal %d\n", sig );
    sigcount++;

    if( sigcount > 1 ) {

        /*
         * Disable the real time interval timer
         */
        getitimer( which, &value );

        value.it_value.tv_sec = 0;
        value.it_value.tv_usec = 0;

        setitimer( which, &value, NULL );
    }
}

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

    int result = 0;

    struct itimerval value, ovalue, pvalue;
    int which = ITIMER_REAL;

    struct sigaction sact;
    volatile double count;
    time_t t;

    sigemptyset( &sact.sa_mask );
    sact.sa_flags = 0;
    sact.sa_handler = catcher;
    sigaction( SIGALRM, &sact, NULL );

    getitimer( which, &pvalue );

    /*
     * Set a real time interval timer to repeat every 200 milliseconds
     */

    value.it_interval.tv_sec = 0;        /* Zero seconds */
    value.it_interval.tv_usec = 200000;  /* Two hundred milliseconds */
    value.it_value.tv_sec = 0;           /* Zero seconds */
    value.it_value.tv_usec = 500000;     /* Five hundred milliseconds */

    result = setitimer( which, &value, &ovalue );

    /*
     * The interval timer value returned by setitimer() should be
     * identical to the timer value returned by getitimer().
     */

    if( ovalue.it_interval.tv_sec != pvalue.it_interval.tv_sec  ||
        ovalue.it_interval.tv_usec != pvalue.it_interval.tv_usec ||
        ovalue.it_value.tv_sec != pvalue.it_value.tv_sec ||
        ovalue.it_value.tv_usec != pvalue.it_value.tv_usec ) {
        printf( "Real time interval timer mismatch\n" );
        result = -1;
    }


    time( &t );
    printf( "Before loop, time is %s", ctime(&t) );

    for( count=0; ((count<LOOP_LIMIT) && (sigcount<2)); count++ );
    time( &t );
    printf( "After loop, time is %s\n", ctime(&t) );


    if( sigcount == 0 )
        printf( "The signal catcher never gained control\n" );
    else
        printf( "The signal catcher gained control\n" );

    printf( "The value of count is %.0f\n", count );

    return( result );
}
```

Output:

```
    Before loop, time is Sun Jun 15 10:14:00 1997
    Signal catcher called for signal 14
    Signal catcher called for signal 14
    After loop, time is Sun Jun 15 10:14:01 1997
    The signal catcher gained control
    The value of count is 702943
```

API introduced: V4R2

---

sigaction()--Examine and Change Signal Action

Syntax

```
#include <signal.h>
#include <sys/signal.h>

int sigaction( int sig, const struct sigaction *act,
               struct sigaction *oact );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **sigaction()** function examines, changes, or both examines and changes the action associated with a specific signal.

The *sig* argument must be one of the macros defined in the <**sys/signal.h**> header file.

If **sigaction()** fails, the action for the signal *sig* is not changed.

Parameters

sig

(Input) A signal from the list defined in Control Signals Table .

****act***

(Input) A pointer to the sigaction structure that describes the action to be taken for the signal. Can be NULL.

If *act* is a NULL pointer, signal handling is unchanged. **sigaction()** can be used to inquire about the current handling of signal *sig*.

If *act* is not NULL, the action specified in the sigaction structure becomes the new action associated with *sig*.

****oact***

(Output) A pointer to a storage location where **sigaction()** can store a sigaction structure. This structure contains the action currently associated with *sig*. Can be NULL.

If *oact* is a NULL pointer, **sigaction()** does not store this information.

The **sigaction()** function uses structures of the sigaction type. The following is an example of a sigaction() structure:

```
struct sigaction {
    void      (*sa_handler)(int);
    sigset_t  sa_mask;
    int       sa_flags;
    void      (*sa_sigaction)(int, siginfo_t *,void *);
};
```

The members of the sigaction structure are as follows:

Member name	Description
void (*) (int) sa_handler	A pointer to the function assigned to handle the signal. The value of this member also can be SIG_DFL (indicating the default action) or SIG_IGN (indicating that the signal should be ignored).
sigset_t sa_mask	A signal set (set of signals) to be added to the signal mask of the calling process before the signal-catching function sa_handler is called. For more on signal sets, see sigprocmask()--Examine and Change Blocked Signals. You cannot use this mechanism to block the SIGKILL or SIGSTOP signals. If sa_mask includes these signals, they are ignored and **sigaction()** does not return an error. sa_mask must be set by using one or more of the signal set manipulation functions: **sigemptyset()**, **sigfillset()**, **sigaddset()**, or **sigdelset()**
int sa_flags	A collection of flag bits that affect the behavior of signals. The following flag bits can be set in sa_flags: *SA_NOCLDSTOP* If this flag is set, the system does not generate a SIGCHLD signal when child processes stop. This is relevant only when the *sig* argument of **sigaction()** is SIGCHLD. *SA_NODEFER* If this flag is set and *sig*is caught, *sig* is not added to the signal mask of the process on entry to the signal catcher unless it is included in **sa_mask**. If this flag is not set, *sig* is always added to the signal mask of the process on entry to the signal catcher. This flag is supported for compatibility with applications that use **signal()** to set the signal action. *SA_RESETHAND* If this flag is set, the signal-handling action for the signal is reset to SIG_DFL and the SA_SIGINFO flag is cleared on entry to the signal-catching function. Otherwise, the signal-handling action is not changed on entry to the signal-catching function. This flag is supported for compatibility with applications that use **signal()** to set the signal action. *SA_SIGINFO* If this flag is not set and the signal is caught, the signal-catching function identified by sa_handler is entered. If this flag is set and the signal is caught, the signal-catching function identified by sa_sigaction is entered.
*void (*) (int, siginfo_t *, void *) sa_sigaction*	A pointer to the function assigned to handle the signal. If SA_SIGINFO is set, the signal-catching function identified by sa_sigaction is entered with additional arguments and sa_handler is ignored. If SA_SIGINFO is not set, sa_sigaction is ignored. If **sig_action()** is called from a program using data model LLP64, the parameters to sa_sigaction must be declared as siginfo_t *__ptr128 and void *__ptr128.

When a signal catcher installed by **sigaction()**, with the SA_RESETHAND flag set off, catches a signal, the system calculates a new signal mask by taking the union of the following:

- The current signal mask
- The signals specified by sa_mask
- The signal that was just caught if the SA_NODEFER flag is set off

This new mask stays in effect until the signal handler returns, or until **sigprocmask()**, **sigsuspend()**, or **siglongjmp()** is called. When the signal handler ends, the original signal mask is restored.

After an action has been specified for a particular signal, it remains installed until it is explicitly changed with another call to **sigaction()**.

There are three types of actions that can be associated with a signal: SIG_DFL, SIG_IGN, or a pointer to a function. Initially, all signals are set to SIG_DFL or SIG_IGN. The actions prescribed by these values are as follows:

Action	Description
SIG_DFL (signal-specific default action)	● The default actions for the supported signals are specified in Control Signals Table ● If the default action is to stop the process, that process is temporarily suspended. When a process stops, a SIGCHLD signal is generated for its parent process, unless the parent process has set the SA_NOCLDSTOP flag. While a process is stopped, any additional signals sent to the process are not delivered. The one exception is SIGKILL, which always ends the receiving process. When the process resumes, any unblocked signals that were not delivered are then delivered to it. ● If the default action is to ignore the signal, setting a signal action to SIG_DFL causes any pending signals for that signal to be discarded, whether or not the signal is blocked.
SIG_IGN (ignore signal)	● Delivery of the signal has no effect on the process. The behavior of a process is undefined if it ignores a SIGFPE, SIGILL, or SIGSEGV signal that was not generated by **kill()** or **raise()**. ● If the default action is to ignore the signal, setting a signal action to SIG_DFL causes any pending signals for that signal to be discarded, whether or not the signal is blocked. ● The signal action for the signals SIGKILL and SIGTOP cannot be set to SIG_IGN.

Pointer to a function (catch signal)	● On delivery of the signal, the receiving process runs the signal-catching function. When the signal-catching function returns, the receiving process resumes processing at the point at which it was interrupted.
	
	● If SA_SIGINFO is not set, the signal-catching function identified by sa_handler is entered as follows:
	
	`void func( int signo );`
	where the following is true:
	
	○ *func* is the specified signal-catching function.
	○ *signo* is the signal number of the signal being delivered.
	
	● If SA_SIGINFO is set, the signal-catching function identified by sa_sigaction is entered as follows:
	
	`void func( int signo, siginfo_t *info, void *context );`
	where the following is true:
	
	○ *func* is the specified signal-catching function.
	○ *signo* is the signal number of the signal being delivered.
	○ **info* points to an object of type siginfo_t associated with the signal being delivered.
	○ *context* is set to the NULL pointer.
	
	● The behavior of a process is undefined if it returns normally from a signal-catching function for a SIGFPE, SIGILL, or SIGSEGV signal that was not generated by **kill()** or **raise()**.
	
	● The signals SIGKILL and SIGSTOP cannot be caught.

The following is an example of the siginfo_t structure:

```
typedef struct siginfo_t {
   int      si_signo;               /* Signal number                 */
   int      si_source   :   1;      /* Signal source                 */
   int      reserved1   :  15;      /* Reserved (binary 0)           */
   short    si_data_size;           /* Size of additional signal
                                       related data (if available)   */
   _MI_Time si_time;                /* Time of signal                */
   struct {
       char reserved2[2]  /* Pad (reserved)              */
       char si_job[10];   /* Simple job name             */
       char si_user[10];  /* User name                   */
       char si_jobno[6];  /* Job number                  */
       char reserved3[4]; /* Pad (reserved)              */
   } si_QJN;                        /* Qualified job name            */
   int      si_code;                /* Cause of signal               */
   int      si_errno;               /* Error number                  */
   pid_t    si_pid;                 /* Process ID of sender          */
   uid_t    si_uid;                 /* Real user ID of sender        */
   char     si_data[1];   /* Additional signal related
                                       data (if available)           */
} siginfo_t;
```

The members of the siginfo_t structure are as follows:

int si_signo	The system-generated signal number.
int si_source	Indicates whether the source of the signal is being generated by the system or another process on the system. When the signal source is another process, the members si_QJN, si_pid, and si_uid contain valid data. When the signal source is the system, those members are set to binary 0.
short si_data_size	The length of si_errno, si_code, si_pid, si_uid, and any additional signal-related data. If this member is set to 0, this signal-related information is not available.
struct si_QJN	The fully qualified OS/400 job name of the process sending the signal.
int si_errno	If not zero, this member contains an *errno* value associated with the signal, as defined in **<errno.h>**.
int si_code	If not zero, this member contains a code identifying the cause of the signal. Possible code values are defined in the **<sys/signal.h>** header file.
pidt_t si_pid	The process ID of the process sending the signal.
uid_t si_uid	The real user ID of the process sending the signal.
char si_data[1]	If present, the member contains any additional signal-related data.

Control Signals Table

See Default actions for a description of the value given.

Value	**Default Action**	**Meaning**
SIGABRT	2	Abnormal termination
SIGFPE	2	Arithmetic exceptions that are not masked (for example, overflow, division by zero, and incorrect operation)
SIGILL	2	Detection of an incorrect function image
SIGINT	2	Interactive attention
SIGSEGV	2	Incorrect access to storage

SIGTERM	2	Termination request sent to the program
SIGUSR1	2	Intended for use by user applications
SIGUSR2	2	Intended for use by user applications
SIGALRM	2	A timeout signal (sent by **alarm()**)
SIGHUP	2	A controlling terminal is hung up, or the controlling process ended
SIGKILL	1	A termination signal that cannot be caught or ignored
SIGPIPE	3	A write to a pipe that is not being read
SIGQUIT	2	A quit signal for a terminal
SIGCHLD	3	An ended or stopped child process (SIGCLD is an alias name for this signal)
SIGCONT	5	If stopped, continue
SIGSTOP	4	A stop signal that cannot be caught or ignored
SIGTSTP	4	A stop signal for a terminal
SIGTTIN	4	A background process attempted to read from a controlling terminal
SIGTTOU	4	A background process attempted to write to a controlling terminal
SIGIO	3	Completion of input or output
SIGURG	3	High bandwidth data is available at a socket
SIGPOLL	2	Pollable event
SIGBUS	2	Specification exception
SIGPRE	2	Programming exception
SIGSYS	2	Bad system call
SIGTRAP	2	Trace or breakpoint trap
SIGPROF	2	Profiling timer expired
SIGVTALRM	2	Virtual timer expired
SIGXCPU	2	Processor time limit exceeded
SIGXFSZ	2	File size limit exceeded
SIGDANGER	2	System crash imminent
SIGPCANCEL	2	Thread termination signal that cannot be caught or ignored

Default Actions:

1	End the process immediately.
2	End the request.
3	Ignore the signal.
4	Stop the process.
5	Continue the process if it is currently stopped. Otherwise, ignore the signal.

Return Value

0	**sigaction()** was successful.
-1	**sigaction()** was not successful. The *errno* variable is set to indicate the error.

Error Conditions

If **sigaction()** is not successful, *errno* usually indicates one of the following errors. Under some conditions, *errno* could indicate an error other than those listed here.

[EINVAL] An invalid parameter was found.

A parameter passed to this function is not valid.

[ENOTSIGINIT] Process not enabled for signals.

An attempt was made to call a signal function under one of the following conditions:

- The signal function is being called for a process that is not enabled for asynchronous signals.
- The signal function is being called when the system signal controls have not been initialized.

Usage Notes

1. When the **sigaction** function is used to change the action associated with a specific signal, it enables a process for signals if the process is not already enabled for signals. For details, see Qp0sEnableSignals()--Enable Process for Signals. If the system has not been enabled for signals, **sigaction()** is not successful, and an [ENOTSIGINIT] error is returned.

2. The **sigaction()** function can be used to set the action for a particular signal with the same semantics as a call to **signal()**. The sigaction structure indicated by the parameter **act* should contain the following:

   - A sa_handler equal to the *func* specified on **signal()**.
   - A sa_mask containing the signal mask set by **sigemptyset()**.

    - A sa_flag with the SA_RESETHAND flag set on.
    - A sa_flag with the SA_NODEFER flag set on.

3. Some of the functions have been restricted to be serially reusable with respect to asynchronous signals. That is, the library does not allow an asynchronous signal to interrupt the processing of one of these functions until it has completed.

   This restriction needs to be taken into consideration when a signal-catching function is called asynchronously, because it causes the behavior of some of the library functions to become unpredictable.

   Because of this, when producing a strictly compliant POSIX application, only the following functions should be assumed to be reentrant with respect to asynchronous signals. Your signal-catching functions should be restricted to using only these functions:

accept()	**access()**	**alarm()**	**chdir()**
chmod()	**chown()**	**close()**	**connect()**
creat()	**dup()**	**dup2()**	**fcntl()**
fstat()	**getegid()**	**geteuid()**	**getgid()**
getgroups()	**getpgrp()**	**getpid()**	**getppid()**
getuid()	**kill()**	**link()**	**lseek()**
mkdir()	**open()**	**pathconf()**	**pause()**
read()	**readv()**	**recv()**	**recvfrom()**
recvmsg()	**rename()**	**rmdir()**	**select()**
send()	**sendmsg()**	**sendto()**	**sigaction()**
sigaddset()	**sigdelset()**	**sigemptyset()**	**sigfillset()**
sigismember()	**sigpending()**	**sigprocmask()**	**sigsuspend()**
sigtimedwait()	**sigwait()**	**sigwaitinfo()**	**setitimer()**
sleep()	**stat()**	**sysconf()**	**time()**
times()	**umask()**	**uname()**	**unlink()**
utime()	**write()**	**writev()**	

   In addition to the above functions, the macro versions of **getc()** and **putc()** are not reentrant. However, the library versions of these functions are reentrant.

Related Information

- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- kill()--Send Signal to Process or Group of Processes

- Qp0sDisableSignals()--Disable Process for Signals
- Qp0sEnableSignals()--Enable Process for Signals
- sigprocmask()--Examine and Change Blocked Signals
- sigsuspend()--Wait for Signal

Example

See Code disclaimer information for information pertaining to code examples.

The following example shows how signal catching functions can be established using the **sigaction()** function:

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

void check_mask( int sig, char *signame ) {

    sigset_t sigset;

    sigprocmask( SIG_SETMASK, NULL, &sigset );
    if( sigismember( &sigset, sig ) )
        printf( "the %s signal is blocked\n", signame );
    else
        printf( "the %s signal is unblocked\n", signame );
}

void catcher( int sig ) {

    printf( "inside catcher() function\n" );
    check_mask( SIGUSR1, "SIGUSR1" );
    check_mask( SIGUSR2, "SIGUSR2" );
}

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

    struct sigaction sigact, old_sigact;
    sigset_t sigset;

   /*
    * Set up an American National Standard C style signal handler
    * by setting the signal mask to the empty signal set and
    * using the do-not-defer signal, and reset the signal handler
    * to the SIG_DFL signal flag options.
    */

    sigemptyset( &sigact.sa_mask );
    sigact.sa_flags = 0;
    sigact.sa_flags = sigact.sa_flags | SA_NODEFER | SA_RESETHAND;
    sigact.sa_handler = catcher;
    sigaction( SIGUSR1, &sigact, NULL );
   /*
    * Send a signal to this program by using
    *    kill(getpid(), SIGUSR1)
    * which is the equivalent of the American
    * National Standard C raise(SIGUSR1)
    * function call.
    */

    printf( "raise SIGUSR1 signal\n" );
    kill( getpid(), SIGUSR1 );

   /*
    * Get the current value of the signal handling action for
    * SIGUSR1.  The signal-catching function should have been
    * reset to SIG_DFL
    */

    sigaction( SIGUSR1, NULL, &old_sigact );
    if ( old_sigact.sa_handler != SIG_DFL )
        printf( "signal handler was not reset\n" );

   /*
    * Reset the signal-handling action for SIGUSR1
    */

    sigemptyset( &sigact.sa_mask );
    sigaddset( &sigact.sa_mask, SIGUSR2 );
    sigact.sa_flags = 0;
    sigact.sa_handler = catcher;
    sigaction( SIGUSR1, &sigact, NULL );

    printf( "raise SIGUSR1 signal\n" );
    kill( getpid(), SIGUSR1 );

   /*
    * Get the current value of the signal-handling action for
    * SIGUSR1.  catcher() should still be the signal catching
    * function.
    */

    sigaction( SIGUSR1, NULL, &old_sigact );
    if( old_sigact.sa_handler != catcher )
        printf( "signal handler was reset\n" );

    return( 0 );

}
```

Output:

```
    raise SIGUSR1 signal
    inside catcher() function
    the SIGUSR1 signal is unblocked
    the SIGUSR2 signal is unblocked
    raise SIGUSR1 signal
```

```
inside catcher() function
the SIGUSR1 signal is blocked
the SIGUSR2 signal is blocked
```

API introduced: V3R6

---

sigaddset()--Add Signal to Signal Set

Syntax

```
#include <signal.h>

int sigaddset( sigset_t *set, int signo );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **sigaddset()** function is part of a family of functions that manipulate signal sets. **Signal sets** are data objects that let a thread keep track of groups of signals. For example, a thread might create a signal set to record which signals it is blocking, and another signal set to record which signals are pending. Signal sets are used to manipulate groups of signals used by other functions (such as **sigprocmask()**) or to examine signal sets returned by other functions (such as **sigpending()**).

sigaddset() adds a signal to the set of signals already recorded in *set*.

Parameters

set

(Input) A pointer to a signal set.

signo

(Input) A signal from the list defined in Control Signals Table.

Return Value

0	**sigaddset()** successfully added to the signal set.
-1	**sigaddset()** was not successful. The *errno* variable is set to indicate the error.

Error Conditions

If **sigaddset()** is not successful, *errno* usually indicates the following error. Under some conditions, *errno* could indicate an error other than that listed here.

[EINVAL]	An invalid parameter was found. A parameter passed to this function is not valid. The value of *signo* is not within the range of valid signals or specifies a signal that is not supported.

Related Information

- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- sigaction()--Examine and Change Signal Action
- sigdelset()--Delete Signal from Signal Set
- sigemptyset()--Initialize and Empty Signal Set
- sigfillset()--Initialize and Fill Signal Set
- sigismember()--Test for Signal in Signal Set
- sigprocmask()--Examine and Change Blocked Signals
- sigpending()--Examine Pending Signals
- sigsuspend()--Wait for Signal
- sigtimedwait()--Synchronously Accept a Signal for Interval of Time
- sigwait()--Synchronously Accept a Signal
- sigwaitinfo()--Synchronously Accept a Signal and Signal Data

Example

See Code disclaimer information for information pertaining to code examples.

The following example adds a signal to a set of signals:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
```

```
void catcher( int sig ) {

    printf( "catcher() has gained control\n" );
}

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

    struct sigaction sigact;
    sigset_t sigset;

    sigemptyset( &sigact.sa_mask );
    sigact.sa_flags = 0;
    sigact.sa_handler = catcher;
    sigaction( SIGUSR1, &sigact, NULL );

    printf( "before first kill()\n" );
    kill( getpid(), SIGUSR1 );

    /*
     * Blocking SIGUSR1 signals prevents the signals
     * from being delivered until they are unblocked,
     * so the catcher will not gain control.
     */

    sigemptyset( &sigset );
    sigaddset( &sigset, SIGUSR1 );
    sigprocmask( SIG_SETMASK, &sigset, NULL );

    printf( "before second kill()\n" );
    kill( getpid(), SIGUSR1 );
    printf( "after second kill()\n" );

    return( 0 );
}
```

Output:

```
    before first kill()
    catcher() has gained control
    before second kill()
    after second kill()
```

API introduced: V3R6

---

sigdelset()--Delete Signal from Signal Set

Syntax

```
#include <signal.h>

int sigdelset( sigset_t *set, int signo );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **sigdelset()** function is part of a family of functions that manipulate signal sets. **Signal sets** are data objects that let a thread keep track of groups of signals. For example, a thread might create a signal set to record which signals it is blocking, and another signal set to record which signals are pending. Signal sets are used to manipulate groups of signals used by other functions (such as **sigprocmask()**) or to examine signal sets returned by other functions (such as **sigpending()**).

sigdelset() removes the specified *signo* from the list of signals recorded in *set*.

Parameters

set

(Input) A pointer to a signal set.

signo

(Input) A signal from the list defined in Control Signals Table.

Return Value

0	**sigdelset()** successfully deleted from the signal set.
-1	**sigdelset()** was not successful. The *errno* variable is set to indicate the error.

Error Conditions

If **sigdelset()** is not successful, *errno* usually indicates the following error. Under some conditions, *errno* could indicate an error other than that listed here.

[EINVAL] An invalid parameter was found.

A parameter passed to this function is not valid.

The value of *signo* is not within the range of valid signals or specifies a signal that is not supported.

Related Information

- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- sigaction()--Examine and Change Signal Action
- sigaddset()--Add Signal to Signal Set
- sigemptyset()--Initialize and Empty Signal Set
- sigfillset()--Initialize and Fill Signal Set
- sigismember()--Test for Signal in Signal Set
- sigprocmask()--Examine and Change Blocked Signals
- sigpending()--Examine Pending Signals
- sigsuspend()--Wait for Signal
- sigtimedwait()--Synchronously Accept a Signal for Interval of Time
- sigwait()--Synchronously Accept a Signal
- sigwaitinfo()--Synchronously Accept a Signal and Signal Data

Example

See Code disclaimer information for information pertaining to code examples.

The following example deletes a signal from a set of signals:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
```

```
void catcher( int sig ) {

    printf( "catcher() has gained control\n" );
}

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

    struct sigaction sigact;
    sigset_t sigset;

    sigemptyset( &sigact.sa_mask );
    sigact.sa_flags = 0;
    sigact.sa_handler = catcher;
    sigaction( SIGUSR1, &sigact, NULL );

    /*
     * Blocking all signals prevents the blockable
     * signals from being delivered until they are
     * unblocked, so the catcher will not gain
     * control.
     */

    sigfillset( &sigset );
    sigaddset( &sigset, SIGUSR1 );
    sigprocmask( SIG_SETMASK, &,sigset, NULL );

    printf( "before kill()\n" );
    kill( getpid(), SIGUSR1 );

    printf( "before unblocking SIGUSR1\n" );
    sigdelset( &sigset, SIGUSR1 );
    sigprocmask( SIG_SETMASK, &sigset, NULL );

    printf( "after unblocking SIGUSR1\n" );

    return( 0 );
}
```

Output:

```
    before kill()
    before unblocking SIGUSR1
    catcher() has gained control
    after unblocking SIGUSR1
```

API introduced: V3R6

---

sigemptyset()--Initialize and Empty Signal Set

Syntax

```
#include <signal.h>

int sigemptyset( sigset_t *set );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **sigemptyset()** function is part of a family of functions that manipulate signal sets. **Signal sets** are data objects that let a thread keep track of groups of signals. For example, a thread might create a signal set to record which signals it is blocking, and another signal set to record which signals are pending. Signal sets are used to manipulate groups of signals used by other functions (such as **sigprocmask()**) or to examine signal sets returned by other functions (such as **sigpending()**).

sigemptyset() initializes the signal set specified by *set* to an empty set. That is, all supported signals are excluded (see Control Signals Table).

Parameters

set

(Input) A pointer to a signal set.

Return Value

0	**sigemptyset()** was successful.

Error Conditions

The **sigemptyset()** function does not return an error.

Related Information

- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- sigaction()--Examine and Change Signal Action

- sigaddset()--Add Signal to Signal Set
- sigdelset()--Delete Signal from Signal Set
- sigfillset()--Initialize and Fill Signal Set
- sigismember()--Test for Signal in Signal Set
- sigprocmask()--Examine and Change Blocked Signals
- sigpending()--Examine Pending Signals
- sigsuspend()--Wait for Signal
- sigtimedwait()--Synchronously Accept a Signal for Interval of Time
- sigwait()--Synchronously Accept a Signal
- sigwaitinfo()--Synchronously Accept a Signal and Signal Data

Example

See Code disclaimer information for information pertaining to code examples.

The following example initializes a set of signals to the empty set:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

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

    struct sigaction sigact;
    sigset_t sigset;

    sigemptyset( &sigact.sa_mask );
    sigact.sa_flags = 0;
    sigact.sa_handler = SIG_IGN;
    sigaction( SIGUSR2, &sigact, NULL );

    /*
     * Unblocking all signals ensures that the signal
     * handling action will be taken when the signal
     * is generated.
     */

    sigemptyset( &sigset );
    sigprocmask( SIG_SETMASK, &sigset, NULL );

    printf( "before kill()\n" );
    kill( getpid(), SIGUSR2 );
    printf( "after kill()\n" );

    return( 0 );
}
```

Output:

```
    before kill()
    after kill()
```

API introduced: V3R6

---

sigfillset()--Initialize and Fill Signal Set

Syntax

```
#include <signal.h>

int sigfillset( sigset_t *set );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **sigfillset()** function is part of a family of functions that manipulate signal sets. **Signal sets** are data objects that let a thread keep track of groups of signals. For example, a thread might create a signal set to record which signals it is blocking, and another signal set to record which signals are pending. Signal sets are used to manipulate groups of signals used by other functions (such as **sigprocmask()**) or to examine signal sets returned by other functions (such as **sigpending()**).

sigfillset() initializes the signal set specified by set to a complete set. That is, the set includes all supported signals (see Control Signals Table).

Parameters

set

(Input) A pointer to a signal set.

Return Value

0	**sigfillset()** was successful.

Error Conditions

The **sigfillset()** function does not return an error.

Related Information

- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- sigaction()--Examine and Change Signal Action
- sigaddset()--Add Signal to Signal Set
- sigdelset()--Delete Signal from Signal Set
- sigemptyset()--Initialize and Empty Signal Set
- sigismember()--Test for Signal in Signal Set
- sigprocmask()--Examine and Change Blocked Signals
- sigpending()--Examine Pending Signals
- sigsuspend()--Wait for Signal
- sigtimedwait()--Synchronously Accept a Signal for Interval of Time
- sigwait()--Synchronously Accept a Signal
- sigwaitinfo()--Synchronously Accept a Signal and Signal Data

Example

See Code disclaimer information for information pertaining to code examples.

The following example initializes a set of signals to the complete set:

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

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

    sigset_t sigset;

    /*
     * Blocking all signals ensures that the signal
     * handling action for the signals in the set is
     * not taken until the signals are unblocked.
     */

    sigfillset( &sigset );
    sigprocmask( SIG_SETMASK, &sigset, NULL );

    printf( "before kill()\n" );
    kill( getpid(), SIGUSR2 );
    printf( "after kill()\n" );

    return( 0 );
}
```

Output:

```
    before kill()
    after kill()
```

API introduced: V3R6

---

sigismember()--Test for Signal in Signal Set

Syntax

```
#include <signal.h>

int sigismember( const sigset_t *set, int signo );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **sigismember()** function is part of a family of functions that manipulate signal sets. **Signal sets** are data objects that let a thread keep track of groups of signals. For example, a thread might create a signal set to record which signals it is blocking, and another signal set to record which signals are pending. Signal sets are used to manipulate groups of signals used by other functions (such as **sigprocmask()**) or to examine signal sets returned by other functions (such as **sigpending()**).

sigismember() tests whether a signal number specified by *signo* is a member of a signal set specified by *set*.

Parameters

set

(Input) A pointer to a signal set.

signo

(Input) A signal from the list defined in Control Signals Table.

Return Value

1	The specified signal is in the specified signal set.
0	The specified signal is not in the specified signal set.
-1	An error occurred. The *errno* variable is set to indicate the error.

Error Conditions

If **sigismember()** is not successful, *errno* usually indicates the following error. Under some conditions, *errno* could indicate an error other than that listed here.

[EINVAL]	An invalid parameter was found.
	A parameter passed to this function is not valid.
	The value of *signo* is not within the range of valid signals or specifies a signal that is not supported.

Related Information

- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- sigaction()--Examine and Change Signal Action
- sigaddset()--Add Signal to Signal Set
- sigdelset()--Delete Signal from Signal Set
- sigemptyset()--Initialize and Empty Signal Set
- sigfillset()--Initialize and Fill Signal Set
- sigprocmask()--Examine and Change Blocked Signals
- sigpending()--Examine Pending Signals
- sigsuspend()--Wait for Signal
- sigtimedwait()--Synchronously Accept a Signal for Interval of Time
- sigwait()--Synchronously Accept a Signal
- sigwaitinfo()--Synchronously Accept a Signal and Signal Data

Example

The following example uses the **sigismember()** function to test for the presence of signals in a signal set:

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

```
void check( sigset_t set, int signo, char *signame ) {

    printf( "%s is ", signame );
    if( !sigismember( &set, signo ) )
      printf( "not ");
    printf( "in the set" );
}

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

    sigset_t sigset;

    sigemptyset( &sigset );

    sigaddset( &sigset, SIGUSR1 );
    sigaddset( &sigset, SIGKILL );
    sigaddset( &sigset, SIGCHLD );

    check( sigset, SIGUSR1, "SIGUSR1" );
    check( sigset, SIGUSR2, "SIGUSR2" );
    check( sigset, SIGCHLD, "SIGCHLD" );
    check( sigset, SIGFPE,  "SIGFPE" );
    check( sigset, SIGKILL, "SIGKILL" );

    return( 0 );
}
```

Output:

```
    SIGUSR1 is in the set
    SIGUSR2 is not in the set
    SIGCHLD is in the set
    SIGFPE is not in the set
    SIGKILL is in the set
```

API introduced: V3R6

---

siglongjmp()--Perform Nonlocal Goto with Signal Handling

Syntax

```
#include <setjmp.h>

void siglongjmp( sigjmp_buf env, int val );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **siglongjmp()** function restores the stack environment previously saved in *env* by **sigsetjmp()**. **siglongjmp()** also provides the option to restore the signal mask, depending on whether the signal mask was saved by **sigsetjmp()**.

siglongjmp() is similar to **longjmp()**, except for the optional capability of restoring the signal mask.

The **sigsetjmp()** and **siglongjmp()** functions provide a way to perform a nonlocal "goto."

A call to **sigsetjmp()** causes the current stack environment (including, optionally, the signal mask) to be saved in *env*. A subsequent call to **siglongjmp()** does the following:

- Restores the saved environment and signal mask (if saved by **sigsetjmp()**).
- Returns control to a point in the program corresponding to the **sigsetjmp()** call.

Processing resumes as if the **sigsetjmp()** call had just returned the given *val*. All variables, (except register variables) that are accessible to the function that receives control contain the values they had when **siglongjmp()** was called. The values of register variables are unpredictable. Nonvolatile auto variables that are changed between calls to **sigsetjmp()** and **siglongjmp()** are also unpredictable.

Note: When using **siglongjmp()**, the function in which the corresponding call to **sigsetjmp()** was made must not have returned first. Unpredictable program behavior occurs if **siglongjmp()** is called after the function calling **sigsetjmp()** has returned.

The *val* argument passed to **siglongjmp()** must be nonzero. If the *val* argument is equal to zero, **siglongjmp()** substitutes a 1 in its place.

siglongjmp() does not use the normal function call and return mechanisms. **siglongjmp()** restores the saved signal mask only if the *env* parameter was initialized by a call to **sigsetjmp()** with a nonzero *savemask* argument.

Parameters

env

(Input) An array type that holds the information needed to restore a calling environment.

val

(Input) The return value.

Return Value

None.

Error Conditions

The **siglongjmp()** function does not return an error.

Usage Notes

The **sigsetjmp()-siglongjmp()** pair and the **setjmp()-longjmp()** pair cannot be intermixed. A stack environment and signal mask saved by **sigsetjmp()** can be restored only by **siglongjmp()**.

Related Information

- The <**setjmp.h**> file (see Header Files for UNIX-Type Functions)
- sigaction()--Examine and Change Signal Action
- sigprocmask()--Examine and Change Blocked Signals
- sigsetjmp()--Set Jump Point for Nonlocal Goto
- sigsuspend()--Wait for Signal
- sleep()--Suspend Processing for Interval of Time

Example

See Code disclaimer information for information pertaining to code examples.

This example saves the stack environment and signal mask at the following statement:

```
    if( sigsetjmp(mark,1) != 0 ) { ...
```

When the system first performs the if statement, it saves the environment and signal mask in *mark* and sets the condition to false because **sigsetjmp()** returns a 0 when it saves the environment. The program prints the following message:

```
    sigsetjmp() has been called
```

The subsequent call to function **p()** tests for a local error condition, which can cause it to perform **siglongjmp()** (in this example as a result of calling a signal catching function). Control is returned to the original **sigsetjmp()** function using the environment saved in *mark* and the restored signal mask. This time, the condition is true because -1 is the return value from **siglongjmp()**. The program then performs the statements in the block and prints the following:

```
    siglongjmp() function was called
```

Then the program performs the **recover()** function and exits.

Here is the program:

```
#include <signal.h>
#include <setjmp.h>
#include <unistd.h>
#include <stdio.h>

sigset_t sigset;
sigjmp_buf mark;

void catcher( int );
void p( void );
void recover( void );

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

    int result;

    /*
     * Block the SIGUSR1 and SIGUSR2 signals.  This set of
     * signals will be saved as part of the environment
     * by the sigsetjmp() function.
     */

     sigemptyset( &sigset );
     sigaddset( &sigset, SIGUSR1 );
     sigaddset( &sigset, SIGUSR2 );
     sigprocmask( SIG_SETMASK, &sigset, NULL );

     if( sigsetjmp( mark, 1 ) != 0 ) {
         printf( "siglongjmp() function was called\n" );
         recover();
         result=0;
     }
     else {
         printf( "sigsetjmp() has been called\n" );
         p();
```

```
            sigprocmask( SIG_SETMASK, NULL, &sigset );
            if( sigismember( &sigset, SIGUSR2 ) )
                printf( "siglongjmp() was not called\n" );
            result=-1;
        }

        printf( "return to main with result %d\n", result );

        return( result );
    }

    void p( void ) {

        struct sigaction sigact;
        int error=0;

        printf( "performing function p()\n" );

        /* Send signal handler in case error condition is detected */

        sigemptyset( &sigact.sa_mask );
        sigact.sa_flags = 0;
        sigact.sa_handler = catcher;
        sigaction( SIGUSR2, &sigact, NULL );

        sigdelset( &sigset, SIGUSR2 );
        sigprocmask( SIG_SETMASK, &sigset, NULL );

        /* After some processing an error condition is detected */

        error=-1;

        /* Call catcher() function if error is detected */

        if( error != 0 ) {
            printf( "error condition detected, send SIGUSR2 signal\n" );
            kill( getpid(), SIGUSR2 );
        }
        printf( "return from catcher() function is an error\n" );
    }

    void recover( void ) {

         printf( "taking recovery action\n" );

         sigprocmask( SIG_SETMASK, NULL, &sigset );
         if( sigismember( &sigset, SIGUSR2 ) )
             printf( "signal mask was restored after siglongjmp()\n" );
    }

    void catcher( int signo ) {

        printf( "in catcher() before siglongjmp()\n" );

        siglongjmp( mark, -1 );

        printf( "in catcher() after siglongjmp() is an error\n" );
```

```
}
```

Output

```
    sigsetjmp() has been called
    performing function p()
    error condition detected, send SIGUSR2 signal
    in catcher() before siglongjmp()
    siglongjmp() function was called
    taking recovery action
    signal mask was restored after siglongjmp()
    return to main with result 0
```

API introduced: V3R6

---

sigpending()--Examine Pending Signals

Syntax

```
#include <signal.h>

int sigpending( sigset_t *set );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **sigpending()** function returns signals that are blocked from delivery and pending for either the calling thread or the process. This information is represented as a signal set stored in *set*. For more information on examining the signal set pointed to by *set*, see sigismember()--Test for Signal in Signal Set.

Parameters

***set**

(Output) A pointer to the space where the signal set information is stored.

Return Value

0	**sigpending()** was successful.
-1	**sigpending()** was not successful. The *errno* variable is set to indicate the error.

Error Conditions

If **sigpending()** is not successful, *errno* usually indicates the following error. Under some conditions, *errno* could indicate an error other than that listed here.

[ENOTSIGINIT] Process not enabled for signals.

An attempt was made to call a signal function under one of the following conditions:

- The signal function is being called for a process that is not enabled for asynchronous signals.
- The signal function is being called when the system signal controls have not been initialized.

Related Information

- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- sigaddset()--Add Signal to Signal Set
- sigdelset()--Delete Signal from Signal Set
- sigemptyset()--Initialize and Empty Signal Set
- sigfillset()--Initialize and Fill Signal Set
- sigismember()--Test for Signal in Signal Set
- sigprocmask()--Examine and Change Blocked Signals

Example

See Code disclaimer information for information pertaining to code examples.

The following example returns blocked and pending signals:

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

void catcher( int sig ) {
    puts( "inside catcher() function\n" );
}

void check_pending( int sig, char *signame ) {

    sigset_t sigset;

    if( sigpending( &sigset ) != 0 )
        perror( "sigpending() error\n" );

    else if( sigismember( &sigset, sig ) )
             printf( "a %s signal is pending\n", signame );
         else
             printf( "no %s signals are pending\n", signame );
}

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

    struct sigaction sigact;
    sigset_t sigset;
    sigemptyset( &sigact.sa_mask );
    sigact.sa_flags = 0;
    sigact.sa_handler = catcher;

    if( sigaction( SIGUSR1, &sigact, NULL ) != 0 )
        perror( "sigaction() error\n" );

    else {
        sigemptyset( &sigset );
        sigaddset( &sigset, SIGUSR1 );
        if ( sigprocmask( SIG_SETMASK, &sigset, NULL ) != 0)
           perror( "sigprocmask() error\n" );

       else {
            printf( "SIGUSR1 signals are now blocked\n" );

            kill( getpid(), SIGUSR1 );
            printf( "after kill()\n" );

            check_pending( SIGUSR1, "SIGUSR1" );

            sigemptyset( &sigset );
            sigprocmask( SIG_SETMASK, &sigset, NULL );
            printf( "SIGUSR1 signals are no longer blocked\n" );

            check_pending( SIGUSR1, "SIGUSR1" );
        }
    }
    return( 0 );
}
```

Output:

```
    SIGUSR1 signals are now blocked
    after kill()
    a SIGUSR1 signal is pending
    inside catcher() function
    SIGUSR1 signals are no longer blocked
    no SIGUSR1 signals are pending
```

API introduced: V3R6

---

sigprocmask()--Examine and Change Blocked Signals

Syntax

```
 #include <signal.h>

 int sigprocmask( int how, const sigset_t *set,
                  sigset_t *oset );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **sigprocmask()** function examines, or changes, or both examines and changes the signal mask of the calling thread.

The signals SIGKILL or SIGSTOP cannot be blocked. Any attempt to use **sigprocmask()** to block these signals is simply ignored, and no error is returned.

SIGFPE, SIGILL, and SIGSEGV signals that are not artificially generated by **kill()** or **raise()** (that is, were generated by the system as a result of a hardware or software exception) are not blocked.

If there are any pending unblocked signals after **sigprocmask()** has changed the signal mask, at least one of those signals is delivered to the thread before **sigprocmask()** returns.

If **sigprocmask()** fails, the process's signal mask is not changed.

Parameters

how

(Input) The way in which the signal set is changed.

set

(Input) A pointer to a set of signals to be used to change the currently blocked set. May be NULL.

oset

(Output) A pointer to the space where the previous signal mask is stored. May be NULL.

The possible values for *how*, which are defined in the <**sys/signal.h**> header file, are as follows:

SIG_BLOCK	Indicates that the set of signals given by *set* should be blocked, in addition to the set currently being blocked.

SIG_UNBLOCK	Indicates that the set of signals given by *set* should not be blocked. These signals are removed from the current set of signals being blocked.
SIG_SETMASK	Indicates that the set of signals given by *set* should replace the old set of signals being blocked.

The *set* parameter points to a signal set giving the new signals that should be blocked or unblocked (depending on the value of *how*), or it points to the new signal mask if the value of *how* was SIG_SETMASK. Signal sets are described in sigemptyset()--Initialize and Empty Signal Set. If *set* is a NULL pointer, the set of blocked signals is not changed. If *set* is NULL, the value of *how*is ignored.

The signal set manipulation functions (**sigemptyset()**, **sigfillset()**, **sigaddset()**, and **sigdelset()**) must be used to establish the new signal set pointed to by *set*.

sigprocmask() determines the current signal set and returns this information in **oset*. If *set* is NULL, *oset* returns the current set of signals being blocked. When *set* is not NULL, the set of signals pointed to by *oset* is the previous set.

Return Value

0	**sigprocmask()** was successful.
-1	**sigprocmask()** was not successful. The *errno* variable is set to indicate the error.

Error Conditions

If **sigprocmask()** is not successful, *errno* usually indicates the following error. Under some conditions, *errno* could indicate an error other than that listed here.

[EINVAL]

An invalid parameter was found.

A parameter passed to this function is not valid.

One of the following has occurred:

- The value of *how* is not equal to one of the defined values.
- The signal set pointed to by *set* contains a signal that is not within the valid range or a signal that is not supported.

[ENOTSIGINIT]

Process not enabled for signals.

An attempt was made to call a signal function under one of the following conditions:

- The signal function is being called for a process that is not enabled for asynchronous signals.
- The signal function is being called when the system signal controls have not been initialized.

Usage Notes

1. When the **sigprocmask** function is used to change the signal mask of the calling process, it enables the process for signals if the process is not already enabled for signals. For details, see Qp0sEnableSignals()--Enable Process for Signals. If the system has not been enabled for signals, **sigprocmask()** is not successful, and an [ENOTSIGINIT] error is returned.

2. Typically, sigprocmask(SIG_BLOCK, ...) is used to block signals during a critical section of code. At the end of the critical section of code, sigprocmask(SIG_SETMASK, ...) is used to restore the mask to the previous value returned by sigprocmask(SIG_BLOCK, ...).

Related Information

- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- sigaction()--Examine and Change Signal Action
- Qp0sDisableSignals()--Disable Process for Signals
- Qp0sEnableSignals()--Enable Process for Signals
- sigaddset()--Add Signal to Signal Set
- sigdelset()--Delete Signal from Signal Set
- sigemptyset()--Initialize and Empty Signal Set
- sigfillset()--Initialize and Fill Signal Set
- sigismember()--Test for Signal in Signal Set
- sigpending()--Examine Pending Signals
- sigsuspend()--Wait for Signal
- sigtimedwait()--Synchronously Accept a Signal for Interval of Time
- sigwait()--Synchronously Accept a Signal
- sigwaitinfo()--Synchronously Accept a Signal and Signal Data

Example

See Code disclaimer information for information pertaining to code examples.

The following example changes the signal mask:

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

void catcher( int sig ) {
    printf( "inside catcher() function\n" );
}

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

    time_t start, finish;
    struct sigaction sact;
    sigset_t new_set, old_set;
    double diff;

    sigemptyset( &sact.sa_mask );
    sact.sa_flags = 0;
    sact.sa_handler = catcher;
    sigaction( SIGALRM, &sact, NULL );

    sigemptyset( &new_set );
    sigaddset( &new_set, SIGALRM );
    sigprocmask( SIG_BLOCK, &new_set, &old_set);

    time( &start );
    printf( "SIGALRM signals blocked at %s\n", ctime(&start) );

    alarm( 1 );     /* SIGALRM will be sent in 1 second */

    do {
         time( &finish );
         diff = difftime( finish, start );
    } while (diff < 10);

    sigprocmask( SIG_SETMASK, &old_set, NULL );
    printf( "SIGALRM signals unblocked at %s\n", ctime(&finish) );

    return( 0 );
}
```

Output:

```
    SIGALRM signals blocked at Sun Jan 22 16:53:40 1995
    inside catcher() function
    SIGALRM signals unblocked at Sun Jan 22 16:53:50 1995
```

API introduced: V3R6

---

sigsetjmp()--Set Jump Point for Nonlocal Goto

Syntax

```
#include <setjmp.h>

int sigsetjmp( sigjmp_buf env, int savemask );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **sigsetjmp()** function saves the current stack environment and, optionally, the current signal mask. The stack environment and signal mask saved by **sigsetjmp()** can subsequently be restored by **siglongjmp()**.

sigsetjmp() is similar to **setjmp()**, except for the optional capability of saving the signal mask. Like **setjmp()** and **longjmp()**, the **sigsetjmp()** and **siglongjmp()** functions provide a way to perform a nonlocal "goto."

A call to **sigsetjmp()** causes it to save the current stack environment in *env*. If the value of the *savemask* parameter is nonzero, **sigsetjmp()** also saves the current signal mask in *env*. A subsequent call to **siglongjmp()** does the following:

- Restores the saved environment and signal mask (if saved by **sigsetjmp()**).
- Returns control to a point corresponding to the **sigsetjmp()** call.

The values of all variables (except register variables) accessible to the function receiving control contain the values they had when **siglongjmp()** was called. The values of register variables are unpredictable. Nonvolatile automatic storage variables that are changed between calls to **sigsetjmp()** and **siglongjmp()** are also unpredictable.

Parameters

env

(Input) An array type for holding the information needed to restore a calling environment.

savemask

(Input) An indicator used to determine if the current signal mask of the thread is to be saved. This value may be zero.

Return Value

0	**sigsetjmp()** was called to save the stack environment and, optionally, the signal mask. It may have been either successful or not successful.
val	**siglongjmp()** caused control to be transferred to the place in the user's program where **sigsetjmp()** was issued. The value returned is the value specified on **siglongjmp()** for the *val* parameter (or 1 if the value of *val* is zero).

Error Conditions

The **sigsetjmp()** function does not return an error.

Usage Notes

The **sigsetjmp()-siglongjmp()** pair and the **setjmp()-longjmp()** pair cannot be intermixed. A stack environment and signal mask saved by **sigsetjmp()** can be restored only by **siglongjmp()**.

Related Information

- The <**setjmp.h**> file (see Header Files for UNIX-Type Functions)
- sigaction()--Examine and Change Signal Action
- siglongjmp()--Perform Nonlocal Goto with Signal Handling
- sigprocmask()--Examine and Change Blocked Signals
- sigsuspend()--Wait for Signal

Example

See Code disclaimer information for information pertaining to code examples.

This example saves the stack environment and signal mask at the following statement:

```
if( sigsetjmp(mark,1) != 0 ) { ...
```

When the system first performs the if statement, it saves the environment and signal mask in *mark* and sets the condition to false because **sigsetjmp()** returns a 0 when it saves the environment. The program prints the following message:

```
sigsetjmp() has been called
```

The subsequent call to function **p()** tests for a local error condition, which can cause it to perform **siglongjmp()** (in this example as a result of calling a signal catching function). Control is returned to the original **sigsetjmp()** function using the environment saved in *mark* and the restored signal mask. This time, the condition is true because -1 is the return value from **siglongjmp()**. The program then performs the statements in the block and prints the following:

```
    siglongjmp() function was called
```

Then the program performs the **recover()** function and exits.

Here is the program:

```
#include <signal.h>
#include <setjmp.h>
#include <unistd.h>
#include <stdio.h>

sigset_t sigset;
sigjmp_buf mark;

void catcher( int );
void p( void );
void recover( void );

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

    int result;

    /*
     * Block the SIGUSR1 and SIGUSR2 signals.  This set of
     * signals will be saved as part of the environment
     * by the sigsetjmp() function.
     */

     sigemptyset( &sigset );
     sigaddset( &sigset, SIGUSR1 );
     sigaddset( &sigset, SIGUSR2 );
     sigprocmask( SIG_SETMASK, &sigset, NULL );

     if( sigsetjmp( mark, 1 ) != 0 ) {
         printf( "siglongjmp() function was called\n" );
         recover();
         result=0;
     }
     else {
         printf( "sigsetjmp() has been called\n" );
         p();
         sigprocmask( SIG_SETMASK, NULL, &sigset );
         if( sigismember( &sigset, SIGUSR2 ) )
             printf( "siglongjmp() was not called\n" );
         result=-1;
    }
    printf( "return to main with result %d\n", result);

    return( result );
}
```

```
void p( void ) {

    struct sigaction sigact;
    int error=0;

    printf( "performing function p()\n" );

    /* Send signal handler in case error condition is detected */

    sigemptyset( &sigact.sa_mask );
    sigact.sa_flags = 0;
    sigact.sa_handler = catcher;
    sigaction( SIGUSR2, &sigact, NULL );

    sigdelset( &sigset, SIGUSR2 );
    sigprocmask( SIG_SETMASK, &sigset, NULL );

    /* After some processing an error condition is detected */

    error=-1;

    /* Call catcher() function if error is detected */

    if( error != 0 ) {
        printf( "error condition detected, send SIGUSR2 signal\n" );
        kill( getpid(), SIGUSR2 );
    }
    printf( "return from catcher() function is an error\n" );
}

void recover( void ) {

     printf( "taking recovery action\n" );

     sigprocmask( SIG_SETMASK, NULL, &sigset );
     if( sigismember( &sigset, SIGUSR2 ) )
         printf( "signal mask was restored after siglongjmp()\n" );
}

void catcher( int signo ) {

    printf( "in catcher() before siglongjmp()\n" );

    siglongjmp( mark, -1 );

    printf( "in catcher() after siglongjmp() is an error\n" );
}
```

Output:

```
    sigsetjmp() has been called
    performing function p()
    error condition detected, send SIGUSR2 signal
    in catcher() before siglongjmp()
    siglongjmp() function was called
    taking recovery action
    signal mask was restored after siglongjmp()
    return to main with result 0
```

API introduced: V3R6

Top | UNIX-Type APIs | APIs by category

sigsuspend()--Wait for Signal

Syntax

```
#include <signal.h>

int sigsuspend( const sigset_t *sigmask );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **sigsuspend()** function replaces the current signal mask of a thread with the signal set given by **sigmask* and then suspends processing of the calling process. The thread does not resume running until a signal is delivered whose action is to call a signal-catching function, to end the request, or to terminate the process. (Signal sets are described in more detail in sigemptyset()--Initialize and Empty Signal Set.)

The signal mask indicates a set of signals that should be blocked. Such signals do not "wake up" the suspended function. The signals SIGSTOP and SIGKILL cannot be blocked or ignored; they are delivered to the thread regardless of what the *sigmask* argument specifies.

If an incoming unblocked signal has an action of end the request of terminate the process, **sigsuspend()** never returns to the caller. If an incoming signal is handled by a signal-catching function, **sigsuspend()** returns after the signal-catching function returns. In this case, the signal mask of the thread is restored to whatever it was before **sigsuspend()** was called.

Parameters

****sigmask***

(Input) A pointer to a set of signals to be used to replace the current signal mask of the process.

Return Value

-1	**sigsuspend()** was not successful. The *errno* variable is set to indicate the reason.

There is no return value to indicate successful completion.

Error Conditions

If **sigsuspend()** returns, *errno* indicates the following:

[EINTR]	Interrupted function call. A signal was received and handled by a signal-catching function that returned.
[EINVAL]	An invalid parameter was found. A parameter passed to this function is not valid. The signal set pointed to by *sigmask* contains a signal that is not within the valid range or a signal that is not supported.
[ENOTSIGINIT]	Process not enabled for signals. An attempt was made to call a signal function under one of the following conditions: ● The signal function is being called for a process that is not enabled for asynchronous signals. ● The signal function is being called when the system signal controls have not been initialized.
[EWOULDBLOCK]	Operation would have caused the process to be suspended. The current thread state would prevent the signal function from completing.

Usage Notes

The **sigsuspend** function enables a process for signals if the process is not already enabled for signals. For details, see Qp0sEnableSignals()--Enable Process for Signals. If the system has not been enabled for signals, **sigsuspend()** is not successful, and an [ENOTSIGINIT] error is returned.

Related Information

- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- alarm()--Set Schedule for Alarm Signal
- pause()--Suspend Process Until Signal Received
- Qp0sDisableSignals()--Disable Process for Signals
- Qp0sEnableSignals()--Enable Process for Signals
- sigaction()--Examine and Change Signal Action

- sigaddset()--Add Signal to Signal Set
- sigdelset()--Delete Signal from Signal Set
- sigemptyset()--Initialize and Empty Signal Set
- sigfillset()--Initialize and Fill Signal Set
- sigismember()--Test for Signal in Signal Set
- sigpending()--Examine Pending Signals
- sigprocmask()--Examine and Change Blocked Signals
- sigtimedwait()--Synchronously Accept a Signal for Interval of Time
- sigwait()--Synchronously Accept a Signal
- sigwaitinfo()--Synchronously Accept a Signal and Signal Data
- sleep()--Suspend Processing for Interval of Time

Example

See Code disclaimer information for information pertaining to code examples.

The following example replaces the signal mask and then suspends processing:

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

void catcher( int sig ) {
    printf( "inside catcher() function\n" );
}

void timestamp( char *str ) {

    time_t t;

    time( &t );
    printf( "%s the time is %s\n", str, ctime(&t) );
}

int main( int argc, char *argv[] ) {
    struct sigaction sigact;
    sigset_t block_set;

    sigfillset( &block_set );
    sigdelset( &block_set, SIGALRM );

    sigemptyset( &sigact.sa_mask );
    sigact.sa_flags = 0;
    sigact.sa_handler = catcher;
    sigaction( SIGALRM, &sigact, NULL );

    timestamp( "before sigsuspend()" );
    alarm( 10 );
    sigsuspend( &block_set );
    timestamp( "after sigsuspend()" );

    return( 0 );
}
```

Output:

```
    before sigsuspend() the time is Sun Jan 22 17:11:41 1995
    inside catcher() function
    after sigsuspend() the time is Sun Jan 22 17:11:51 1995
```

API introduced: V3R6

---

sigtimedwait()--Synchronously Accept a Signal for Interval of Time

Syntax

```
 #include <signal.h>

 int sigtimedwait( const sigset_t *set,
                   siginfo_t *info,
                   const struct timespec *timeout );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **sigtimedwait()** function selects a pending signal from *set*, clears it from the set of pending signals for the thread or process, and returns that signal number in the *si_signo* member in the structure that is referenced by *info*. If prior to the call to **sigtimedwait()** there are multiple pending instances of a single signal number, upon successful return the number of remaining signals for that signal number is decremented by one.

If no signal in *set* is pending at the time of the call, the thread shall be suspended for the time interval in the *timespec* structure referenced by *timeout*. The thread does not resume until either one or more signals in *set* become pending or the time interval has elapsed. If the *timespec* structure referenced by *timeout* has a value of zero and none of the signals specified by *set* are pending, then **sigtimedwait()** is not successful and an [EAGAIN] error is returned.

The signals defined by *set* are required to be blocked at the time of the call to **sigtimedwait()**; otherwise, **sigtimedwait()** is not successful, and an [EINVAL] error is returned. The signal SIGKILL or SIGSTOP cannot be selected. Any attempt to use **sigprocmask()** to select these signals is simply ignored, and no error is returned.

The signal action for the signal in *set* that is returned in the member *si_signo* in the structure referenced by *info* is not taken.

If more than one thread is using a *sigwait* function to wait for the same signal, only one of these threads will return from the *sigwait* function with the signal number. If more than one thread is waiting for the same signal, the first thread to wait on the signal will return from the *sigwait* function.

Parameters

set

(Input) A pointer to a signal set to be waited upon.

info

(Output) A pointer to the storage location where **sigtimedwait()** can store the signal related information for the signal number that completed the wait. This value may be NULL. The *siginfo_t* structure is described in sigaction()--Examine and Change Signal Action.

***timeout**

(Input) A pointer to the storage location specifying the time interval **sigtimedwait()** should wait. This value may be NULL. If *timeout* is NULL, the thread will be suspended until one or more signals in *set* become pending.

Return Value

0	**sigtimedwait()** was successful.
-1	**sigtimedwait()** was not successful. The *errno* variable is set to indicate the reason.

Error Conditions

If **sigtimedwait()** is not successful, *errno* usually indicates the following error. Under some conditions, *errno* could indicate an error other than that listed here.

[EINVAL] An invalid parameter was found.

A parameter passed to this function is not valid.

One of the following has occurred:

- The signal set pointed to by *set* contains a signal that is not within the valid range or a signal that is not supported.
- A signal in the signal set pointed to by *set* contains a signal that is not blocked.
- The *tv_nsec* member in the *timespec* structure pointed to by *timeout* is greater than or equal to 1,000,000,000.

[EAGAIN] Operation would have caused the process to be suspended.

[ENOTSIGINIT] Process not enabled for signals.

An attempt was made to call a signal function under one of the following conditions:

- The signal function is being called for a process that is not enabled for asynchronous signals.
- The signal function is being called when the system signal controls have not been initialized.

Usage Notes

The **sigtimedwait()** function enables a process for signals if the process is not already enabled for signals. For details, see Qp0sEnableSignals()--Enable Process for Signals. If the system has not been enabled for signals, **sigtimedwait()** is not successful, and an [ENOTSIGINIT] error is returned.

Related Information

- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- Qp0sDisableSignals()--Disable Process for Signals
- Qp0sEnableSignals()--Enable Process for Signals
- sigaction()--Examine and Change Signal Action
- sigpending()--Examine Pending Signals
- sigprocmask()--Examine and Change Blocked Signals
- sigsuspend()--Wait for Signal
- sigwait()--Synchronously Accept a Signal
- sigwaitinfo()--Synchronously Accept a Signal and Signal Data

Example

See Code disclaimer information for information pertaining to code examples.

The following example suspends processing by using the **sigtimedwait()** function and determines the current time:

Note: The signal catching function is not called.

```
#include <signal.h>
#include <stdio.h>
#include <time.h>

void catcher( int sig ) {
    printf( "Signal catcher called for signal %d\n", sig );
}

void timestamp( char *str ) {
    time_t t;
    time( T );
    printf( "The time %s is %s\n", str, ctime(T) );
}

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

    int result = 0;

    struct sigaction sigact;
    struct sigset_t waitset;
    siginfo_t info;
    struct timespec timeout;

    sigemptyset( &sigact.sa_mask );
    sigact.sa_flags = 0;
    sigact.sa_handler = catcher;
    sigaction( SIGALRM, &sigact, NULL );

    sigemptyset( &waitset );
    sigaddset( &waitset, SIGALRM );

    sigprocmask( SIG_BLOCK, &waitset, NULL );

    timeout.tv_sec = 10;     /* Number of seconds to wait */
    timeout.tv_nsec = 1000;  /* Number of nanoseconds to wait */

    alarm( 10 );

    timestamp( "before sigtimedwait()" );

    result = sigtimedwait( &waitset, &info, &timeout );
    printf("sigtimedwait() returned for signal %d\n",
        info.si_signo );

    timestamp( "after sigtimedwait()" );

    return( result );
}
```

Output:

```
    The time before sigtimedwait() is Mon Feb 17 11:09:08 1997
    sigtimedwait() returned for signal 14
    The time after sigtimedwait() is Mon Feb 17 11:09:18 1997
```

API introduced: V4R2

---

sigwait()--Synchronously Accept a Signal

Syntax

```
#include <signal.h>

int sigwait( const sigset_t *set, int *sig );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **sigwait()** function selects a pending signal from *set*, clears it from the set of pending signals for the thread or process, and returns that signal number in the location that is referenced by *sig*. If prior to the call to **sigwait()** there are multiple pending instances of a single signal number, upon successful return the number of remaining signals for that signal number is decremented by one.

If no signal in *set* is pending at the time of the call, the thread shall be suspended. The thread does not resume until one or more signals in *set* become pending.

The signals defined by *set* are required to be blocked at the time of the call to **sigwait()**; otherwise, **sigwait()** is not successful, and an [EINVAL] error is returned. The signals SIGKILL or SIGSTOP cannot be selected. Any attempt to use **sigwait()** to select these signals is simply ignored, and no error is returned.

The signal action for the signal in *set* that is returned in the location referenced by *sig* is not taken.

If more than one thread is using a *sigwait* function to wait for the same signal, only one of these threads will return from the *sigwait* function with the signal number. If more than one thread is waiting for the same signal, the first thread to wait on the signal will return from the *sigwait* function.

Parameters

set

(Input) A pointer to a signal set to be waited upon.

sig

(Output) A pointer to the storage location where **sigwait()** can store the signal number that completed the wait.

Return Value

0	**sigwait()** was successful.
-1	**sigwait()** was not successful. The *errno* variable is set to indicate the reason.

Error Conditions

If **sigwait()** is not successful, *errno* usually indicates the following error. Under some conditions, *errno* could indicate an error other than that listed here.

[EINVAL]	An invalid parameter was found. A parameter passed to this function is not valid. One of the following has occurred: ● The signal set pointed to by *set* contains a signal that is not within the valid range or a signal that is not supported. ● A signal in the signal set pointed to by *set* contains a signal that is not blocked.
[ENOTSIGINIT]	Process not enabled for signals. An attempt was made to call a signal function under one of the following conditions: ● The signal function is being called for a process that is not enabled for asynchronous signals. ● The signal function is being called when the system signal controls have not been initialized.

Usage Notes

The **sigwait()** function enables a process for signals if the process is not already enabled for signals. For details, see Qp0sEnableSignals()--Enable Process for Signals. If the system has not been enabled for signals, **sigwait()** is not successful, and an [ENOTSIGINIT] error is returned.

Related Information

- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- Qp0sDisableSignals()--Disable Process for Signals
- Qp0sEnableSignals()--Enable Process for Signals
- sigaction()--Examine and Change Signal Action
- sigpending()--Examine Pending Signals
- sigprocmask()--Examine and Change Blocked Signals

- sigsuspend()--Wait for Signal
- sigtimedwait()--Synchronously Accept a Signal for Interval of Time
- sigwaitinfo()--Synchronously Accept a Signal and Signal Data

Example

See Code disclaimer information for information pertaining to code examples.

The following example suspends processing by using the **sigwait()** function and determines the current time:

Note: The signal catching function is not called.

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

extern int errno;

void catcher( int sig ) {
    printf( "Signal catcher called for signal %d\n", sig );
}

void timestamp( char *str ) {
    time_t t;

    time( T );
    printf( "The time %s is %s\n", str, ctime(T) );
}

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

    struct sigaction sigact;
    sigset_t waitset;
    int sig;
    int result = 0;

    sigemptyset( &sigact.sa_mask );
    sigact.sa_flags = 0;
    sigact.sa_handler = catcher;
    sigaction( SIGALRM, &sigact, NULL );

    sigemptyset( &waitset );
    sigaddset( &waitset, SIGALRM );

    sigprocmask( SIG_BLOCK, &waitset, NULL );

    alarm( 10 );

    timestamp( "before sigwait()" );
    result = sigwait( &waitset, &sig );
    if( result == 0 )
        printf( "sigwait() returned for signal %d\n", sig );
    else {
        printf( "sigwait() returned error number %d\n", errno );
        perror( "sigwait() function failed\n" );
    }

    timestamp( "after sigwait()" );

    return( result );
}
```

Output:

```
    The time before sigwait() is Tue Jul 15 11:15:43 1997
    sigwait() returned for signal 14
    The time after sigwait() is Tue Jul 15 11:15:54 1997
```

API introduced: V4R2

---

sigwaitinfo()--Synchronously Accept a Signal and Signal Data

Syntax

```
#include <signal.h>

int sigwaitinfo( const sigset_t *set,
                 siginfo_t *info);
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **sigwaitinfo()** function selects a pending signal from *set*, clears it from the set of pending signals for the thread or process, and returns that signal number in the *si_signo* member in the structure that is referenced by *info*. If prior to the call to **sigwaitinfo()** there are multiple pending instances of a single signal number, upon successful return the number of remaining signals for that signal number is decremented by one.

If no signal in *set* is pending at the time of the call, the thread shall be suspended. The thread does not resume until one or more signals in *set* become pending.

The signals defined by *set* are required to be blocked at the time of the call to **sigwaitinfo()**; otherwise, **sigwaitinfo()** is not successful, and an [EINVAL] error is returned. The signals SIGKILL or SIGSTOP cannot be selected. Any attempt to use **sigwaitinfo()** to select these signals is simply ignored, and no error is returned.

The signal action for the signal in *set* that is returned in the member *si_signo* in the structure referenced by *info* is not taken.

If more than one thread is using a *sigwait* function to wait for the same signal, only one of these threads will return from the *sigwait* function with the signal number. If more than one thread is waiting for the same signal, the first thread to wait on the signal will return from the *sigwait* function.

Parameters

set

(Input) A pointer to a signal set to be waited upon.

info

(Output) A pointer to the storage location where **sigwaitinfo()** can store the signal related information for the signal number that completed the wait. This value may be NULL. The *siginfo_t* structure is described in sigaction()--Examine and Change Signal Action.

Return Value

0	**sigwaitinfo()** was successful.
-1	**sigwaitinfo()** was not successful. The *errno* variable is set to indicate the reason.

Error Conditions

If **sigwaitinfo()** is not successful, *errno* usually indicates the following error. Under some conditions, *errno* could indicate an error other than that listed here.

[EINVAL] An invalid parameter was found.

A parameter passed to this function is not valid.

One of the following has occurred:

- The signal set pointed to by *set* contains a signal that is not within the valid range or a signal that is not supported.
- A signal in the signal set pointed to by *set* contains a signal that is not blocked.

[ENOTSIGINIT] Process not enabled for signals.

An attempt was made to call a signal function under one of the following conditions:

- The signal function is being called for a process that is not enabled for asynchronous signals
- The signal function is being called when the system signal controls have not been initialized.

Usage Notes

The **sigwaitinfo()** function enables a process for signals if the process is not already enabled for signals. For details, see Qp0sEnableSignals()--Enable Process for Signals. If the system has not been enabled for signals, **sigwaitinfo()** is not successful, and an [ENOTSIGINIT] error is returned.

Related Information

- The <**signal.h**> file (see Header Files for UNIX-Type Functions)
- Qp0sDisableSignals()--Disable Process for Signals
- Qp0sEnableSignals()--Enable Process for Signals
- sigaction()--Examine and Change Signal Action
- sigpending()--Examine Pending Signals

- sigprocmask()--Examine and Change Blocked Signals
- sigsuspend()--Wait for Signal
- sigtimedwait()--Synchronously Accept a Signal for Interval of Time
- sigwait()--Synchronously Accept a Signal

Example

See Code disclaimer information for information pertaining to code examples.

The following example suspends processing by using the **sigwaitinfo()** function and determines the current time:

Note: The signal catching function is not called.

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

extern int errno;

void catcher( int sig ) {
    printf( "Signal catcher called for signal %d\n", sig );
}

void timestamp( char *str ) {
    time_t t;

    time( T );
    printf( "The time %s is %s\n", str, ctime(T) );
}

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

    int result = 0;

    struct sigaction sigact;
    sigset_t waitset;
    siginfo_t info;

    sigemptyset( &sigact.sa_mask );
    sigact.sa_flags = 0;
    sigact.sa_handler = catcher;
    sigaction( SIGALRM, &sigact, NULL );

    sigemptyset( &waitset );
    sigaddset( &waitset, SIGALRM );

    sigprocmask( SIG_BLOCK, &waitset, NULL );
    alarm( 10 );

    timestamp( "before sigwaitinfo(" );

    result = sigwaitinfo( &waitset, &info );

    if( result == 0 )
        printf( "sigwaitinfo() returned for signal %d\n",
                 info.si_signo );
    else {
        printf( "sigwait() returned error number %d\n", errno );
        perror( "sigwait() function failed\n" );
    }

    timestamp( "after sigwaitinfo()" );

    return( result );
}
```

Output:

```
    The time before sigwaitinfo() is Tue Jul 15 11:22:56 1997
    sigwaitinfo() returned for signal 14
    The time after sigwaitinfo() is Tue Jul 15 11:23:07 1997
```

API introduced: V4R2

---

sleep()--Suspend Processing for Interval of Time

Syntax

```
#include <unistd.h>

unsigned int sleep( unsigned int seconds );
```

Service Program Name: QPOSSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **sleep()** function suspends a thread for a specified number of *seconds*. (Because of processor delays, the thread can sleep slightly longer than this specified time.) If an unblocked signal is received during this time and its action is to call a signal-catching function, to end the request, or to end the process, **sleep()** returns immediately with the amount of sleep time remaining.

If a SIGALRM signal is generated for the calling process while **sleep()** is running and if the SIGALRM signal is being ignored or blocked from delivery, **sleep()** does not return when the SIGALRM signal is scheduled. If the SIGALRM signal is blocked from delivery, the SIGALRM remains pending after **sleep()** returns.

If a SIGALRM signal is generated for the calling process while **sleep()** is running (except as a result of a previous call to **alarm()**) and if the SIGALRM is not being ignored or blocked from delivery, the SIGALRM signal has no effect on **sleep()** other than causing it to return.

A signal-catching function that interrupts **sleep()** can examine and change the time a SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, and whether SIGALRM is blocked from delivery.

If a signal-catching function interrupts **sleep()** and calls **siglongjmp()** or **longjmp()** to restore an environment saved prior to **sleep()**, the **sleep()** function is canceled. The action associated with the SIGALRM signal and the time at which a SIGALRM signal is scheduled to be generated are unchanged. The SIGALRM blocking action remains unchanged, unless the thread's signal mask is restored as part of the environment.

Parameters

seconds

(Input) The number of real seconds for which the process is to be suspended.

Return Value

0	The thread slept for the full time specified.
value	The thread did not sleep the full time because of a signal whose action is to run a signal-catching function, to end the request, or to terminate the process. The value returned is the number of seconds remaining in the specified sleep time; that is, the value of *seconds* minus the actual number of seconds that the thread was suspended.
-1	**sleep()** was not successful. The *errno* variable is set to indicate the error.

Error Conditions

If **sleep()** is not successful, *errno* usually indicates the following error. Under some conditions, *errno* could indicate an error other than that listed here.

[ENOTSIGINIT]

Process not enabled for signals.

An attempt was made to call a signal function under one of the following conditions:

- The signal function is being called for a process that is not enabled for asynchronous signals.
- The signal function is being called when the system signal controls have not been initialized.

[ETIMEDOUT]

A remote host did not respond within the timeout period.

[EWOULDBLOCK]

Operation would have caused the process to be suspended.

The current thread state would prevent the signal function from completing.

Usage Notes

The **sleep()** function enables a process for signals if the process is not already enabled for signals. For details, see Qp0sEnableSignals()--Enable Process for Signals. If the system has not been enabled for signals, **sleep()** is not successful, and an [ENOTSIGINIT] error is returned.

Related Information

- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- alarm()--Set Schedule for Alarm Signal

- pause()--Suspend Process Until Signal Received
- Qp0sDisableSignals()--Disable Process for Signals
- Qp0sEnableSignals()--Enable Process for Signals
- sigaction()--Examine and Change Signal Action
- siglongjmp()--Perform Nonlocal Goto with Signal Handling
- sigsetjmp()--Set Jump Point for Nonlocal Goto
- sigsuspend()--Wait for Signal
- sigtimedwait()--Synchronously Accept a Signal for Interval of Time
- sigwait()--Synchronously Accept a Signal
- sigwaitinfo()--Synchronously Accept a Signal and Signal Data
- usleep()--Suspend Processing for Interval of Time

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses the **sleep()** function to suspend processing for a specified time:

```
#include <unistd.h>
#include <stdio.h>
#include <time.h>

void timestamp( char *str ) {

    time_t t;

    time( &t );
    printf( "%s the time is %s\n", str, ctime(&t) );
}

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

    unsigned int ret;

    timestamp( "before sleep()" );
    ret = sleep( 10 );
    timestamp( "after sleep()" );
    printf( "sleep() returned %d\n", ret );

    return( 0 );
}
```

Output:

```
    before sleep() the time is Sun Jan 22 17:25:17 1995
    after sleep() the time is Sun Jan 22 17:25:28 1995
    sleep() returned 0
```

API introduced: V3R6

Top | UNIX-Type APIs | APIs by category

usleep()--Suspend Processing for Interval of Time

Syntax

```
#include <unistd.h>

unsigned int usleep( useconds_t useconds );
```

Service Program Name: QP0SSRV1

Default Public Authority: *USE

Threadsafe: Yes

The **usleep()** function suspends a thread for the number of microseconds specified by the of useconds parameter. (Because of processor delays, the thread can be suspended slightly longer than this specified time.)

The **usleep()** function uses the process's real-time interval timer to indicate when the thread should be resumed.

There is one real-time interval timer for each process. The **usleep()** function will not interfere with a previous setting of this timer.

Parameters

useconds

(Input) The number of microseconds for which the thread is to be suspended.

Return Value

0	The thread slept for the full time specified.
-*1*	**sleep()** was not successful. The errno variable is set to indicate the error.

Error Conditions

If **usleep()** is not successful, errno usually indicates the following error. Under some conditions, errno could indicate an error other than that listed here.

[EINVAL] An invalid parameter was found.

A parameter passed to this function is not valid.

- The time interval specified 1,000,000 or more microseconds.

Usage Notes

The **usleep()** function is included for its historical usage. The **setitimer()** function is preferred over this function.

Related Information

- The <**unistd.h**> file (see Header Files for UNIX-Type Functions)
- alarm()--Set Schedule for Alarm Signal
- getitimer()--Get Value for Interval Timer
- setitimer()--Set Value for Interval Timer
- sleep()--Suspend Processing for Interval of Time

Example

See Code disclaimer information for information pertaining to code examples.

The following example uses the **usleep()** function to suspend processing for a specified time:

```
#include <unistd.h>
#include <stdio.h>
#include <time.h>

void timestamp( char *str ) {

    time_t t;

    time( &t );
    printf( "%s the time is %s\nquot;, str, ctime(&t) );
}

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

    int result = 0;

    timestamp( quot;before usleep()quot; );
    result = usleep( 999999 );
    timestamp( quot;after usleep()quot; );

    printf( quot;usleep() returned %d\nquot;, result );

    return( result );
}
```

Output:

```
    before usleep() the time is Sun Jun 15 17:25:17 1995
    after usleep() the time is Sun Jun 15 17:25:18 1995
    usleep() returned 0
```

API introduced: V4R2

Header Files for UNIX-Type Functions

Programs using the UNIX-type functions must include one or more header files that contain information needed by the functions, such as:

- Macro definitions
- Data type definitions
- Structure definitions
- Function prototypes

The header files are provided in the QSYSINC library, which is optionally installable. Make sure QSYSINC is on your system before compiling programs that use these header files. For information on installing the QSYSINC library, see Data structures and the QSYSINC Library.

The table below shows the file and member name in the QSYSINC library for each header file used by the UNIX-type APIs in this publication.

Name of Header File	Name of File in QSYSINC	Name of Member
arpa/inet.h	ARPA	INET
arpa/nameser.h	ARPA	NAMESER
bse.h	H	BSE
bsedos.h	H	BSEDOS
bseerr.h	H	BSEERR
dirent.h	H	DIRENT
errno.h	H	ERRNO
fcntl.h	H	FCNTL
grp.h	H	GRP
»inttypes.h	H	INTTYPES«
limits.h	H	LIMITS
»mman.h	H	MMAN«
netdbh.h	H	NETDB
»netinet/icmp6.h	NETINET	ICMP6«
net/if.h	NET	IF
netinet/in.h	NETINET	IN
netinet/ip_icmp.h	NETINET	IP_ICMP
netinet/ip.h	NETINET	IP
»netinet/ip6.h	NETINET	IP6«
netinet/tcp.h	NETINET	TCP
netinet/udp.h	NETINET	UDP
netns/idp.h	NETNS	IDP
netns/ipx.h	NETNS	IPX
netns/ns.h	NETNS	NS
netns/sp.h	NETNS	SP
net/route.h	NET	ROUTE
nettel/tel.h	NETTEL	TEL

os2.h	H	OS2
os2def.h	H	OS2DEF
pwd.h	H	PWD
Qlg.h	H	QLG
qp0lflop.h	H	QP0LFLOP
»qp0ljrnl.h	H	QP0LJRNL«
»qp0lror.h	H	QP0LROR«
Qp0lstdi.h	H	QP0LSTDI
qp0wpid.h	H	QP0WPID
qp0zdipc.h	H	QP0ZDIPC
qp0zipc.h	H	QP0ZIPC
qp0zolip.h	H	QP0ZOLIP
qp0zolsm.h	H	QP0ZOLSM
qp0zripc.h	H	QP0ZRIPC
qp0ztrc.h	H	QP0ZTRC
qp0ztrml.h	H	QP0ZTRML
qp0z1170.h	H	QP0Z1170
»qsoasync.h	H	QSOASYNC«
qtnxaapi.h	H	QTNXAAPI
qtnxadtp.h	H	QTNXADTP
qtomeapi.h	H	QTOMEAPI
qtossapi.h	H	QTOSSAPI
resolv.h	H	RESOLVE
semaphore.h	H	SEMAPHORE
signal.h	H	SIGNAL
spawn.h	H	SPAWN
ssl.h	H	SSL
sys/errno.h	H	ERRNO
sys/ioctl.h	SYS	IOCTL
sys/ipc.h	SYS	IPC
sys/layout.h	H	LAYOUT
sys/limits.h	H	LIMITS
sys/msg.h	SYS	MSG
sys/param.h	SYS	PARAM
»sys/resource.h	SYS	RESOURCE«
sys/sem.h	SYS	SEM
sys/setjmp.h	SYS	SETJMP
sys/shm.h	SYS	SHM
sys/signal.h	SYS	SIGNAL
sys/socket.h	SYS	SOCKET
sys/stat.h	SYS	STAT
sys/statvfs.h	SYS	STATVFS

sys/time.h	SYS	TIME
sys/types.h	SYS	TYPES
sys/uio.h	SYS	UIO
sys/un.h	SYS	UN
sys/wait.h	SYS	WAIT
»ulimit.h	H	ULIMIT«
unistd.h	H	UNISTD
utime.h	H	UTIME

You can display a header file in QSYSINC by using one of the following methods:

- Using your editor. For example, to display the **unistd.h** header file using the Source Entry Utility editor, enter the following command:

  ```
  STRSEU SRCFILE(QSYSINC/H) SRCMBR(UNISTD) OPTION(5)
  ```

- Using the Display Physical File Member command. For example, to display the **sys/stat.h** header file, enter the following command:

  ```
  DSPPFM FILE(QSYSINC/SYS) MBR(STAT)
  ```

You can print a header file in QSYSINC by using one of the following methods:

- Using your editor. For example, to print the **unistd.h** header file using the Source Entry Utility editor, enter the following command:

  ```
  STRSEU SRCFILE(QSYSINC/H) SRCMBR(UNISTD) OPTION(6)
  ```

- Using the Copy File command. For example, to print the **sys/stat.h** header file, enter the following command:

  ```
  CPYF FROMFILE(QSYSINC/SYS) TOFILE(*PRINT) FROMMBR(STAT)
  ```

Symbolic links to these header files are also provided in directory /QIBM/include.

Errno Values for UNIX-Type Functions

Programs using the UNIX-type functions may receive error information as *errno* values. The possible values returned are listed here in ascending *errno* value sequence.

Name	Value	Text
EDOM	3001	A domain error occurred in a math function.
ERANGE	3002	A range error occurred.
ETRUNC	3003	Data was truncated on an input, output, or update operation.
ENOTOPEN	3004	File is not open.
ENOTREAD	3005	File is not opened for read operations.
EIO	3006	Input/output error.
ENODEV	3007	No such device.
ERECIO	3008	Cannot get single character for files opened for record I/O.
ENOTWRITE	3009	File is not opened for write operations.
ESTDIN	3010	The stdin stream cannot be opened.
ESTDOUT	3011	The stdout stream cannot be opened.
ESTDERR	3012	The stderr stream cannot be opened.
EBADSEEK	3013	The positioning parameter in fseek is not correct.
EBADNAME	3014	The object name specified is not correct.
EBADMODE	3015	The type variable specified on the open function is not correct.
EBADPOS	3017	The position specifier is not correct.
ENOPOS	3018	There is no record at the specified position.
ENUMMBRS	3019	Attempted to use ftell on multiple members.
ENUMRECS	3020	The current record position is too long for ftell.
EINVAL	3021	The value specified for the argument is not correct.
EBADFUNC	3022	Function parameter in the signal function is not set.
ENOENT	3025	No such path or directory.
ENOREC	3026	Record is not found.
EPERM	3027	The operation is not permitted.
EBADDATA	3028	Message data is not valid.
EBUSY	3029	Resource busy.
EBADOPT	3040	Option specified is not valid.
ENOTUPD	3041	File is not opened for update operations.
ENOTDLT	3042	File is not opened for delete operations.

EPAD	3043	The number of characters written is shorter than the expected record length.
EBADKEYLN	3044	A length that was not valid was specified for the key.
EPUTANDGET	3080	A read operation should not immediately follow a write operation.
EGETANDPUT	3081	A write operation should not immediately follow a read operation.
EIOERROR	3101	A nonrecoverable I/O error occurred.
EIORECERR	3102	A recoverable I/O error occurred.
EACCES	3401	Permission denied.
ENOTDIR	3403	Not a directory.
ENOSPC	3404	No space is available.
EXDEV	3405	Improper link.
EAGAIN	3406	Operation would have caused the process to be suspended.
EWOULDBLOCK	3406	Operation would have caused the process to be suspended.
EINTR	3407	Interrupted function call.
EFAULT	3408	The address used for an argument was not correct.
ETIME	3409	Operation timed out.
ENXIO	3415	No such device or address.
EAPAR	3418	Possible APAR condition or hardware failure.
ERECURSE	3419	Recursive attempt rejected.
EADDRINUSE	3420	Address already in use.
EADDRNOTAVAIL	3421	Address is not available.
EAFNOSUPPORT	3422	The type of socket is not supported in this protocol family.
EALREADY	3423	Operation is already in progress.
ECONNABORTED	3424	Connection ended abnormally.
ECONNREFUSED	3425	A remote host refused an attempted connect operation.
ECONNRESET	3426	A connection with a remote socket was reset by that socket.
EDESTADDRREQ	3427	Operation requires destination address.
EHOSTDOWN	3428	A remote host is not available.
EHOSTUNREACH	3429	A route to the remote host is not available.
EINPROGRESS	3430	Operation in progress.
EISCONN	3431	A connection has already been established.
EMSGSIZE	3432	Message size is out of range.
ENETDOWN	3433	The network currently is not available.
ENETRESET	3434	A socket is connected to a host that is no longer available.

ENETUNREACH	3435	Cannot reach the destination network.
ENOBUFS	3436	There is not enough buffer space for the requested operation.
ENOPROTOOPT	3437	The protocol does not support the specified option.
ENOTCONN	3438	Requested operation requires a connection.
ENOTSOCK	3439	The specified descriptor does not reference a socket.
ENOTSUP	3440	Operation is not supported.
EOPNOTSUPP	3440	Operation is not supported.
EPFNOSUPPORT	3441	The socket protocol family is not supported.
EPROTONOSUPPORT	3442	No protocol of the specified type and domain exists.
EPROTOTYPE	3443	The socket type or protocols are not compatible.
ERCVDERR	3444	An error indication was sent by the peer program.
ESHUTDOWN	3445	Cannot send data after a shutdown.
ESOCKTNOSUPPORT	3446	The specified socket type is not supported.
ETIMEDOUT	3447	A remote host did not respond within the timeout period.
EUNATCH	3448	The protocol required to support the specified address family is not available at this time.
EBADF	3450	Descriptor is not valid.
EMFILE	3452	Too many open files for this process.
ENFILE	3453	Too many open files in the system.
EPIPE	3455	Broken pipe.
ECANCEL	3456	Operation cancelled.
EEXIST	3457	File exists.
EDEADLK	3459	Resource deadlock avoided.
ENOMEM	3460	Storage allocation request failed.
EOWNERTERM	3462	The synchronization object no longer exists because the owner is no longer running.
EDESTROYED	3463	The synchronization object was destroyed, or the object no longer exists.
ETERM	3464	Operation was terminated.
ENOENT1	3465	No such file or directory.
ENOEQFLOG	3466	Object is already linked to a dead directory.
EEMPTYDIR	3467	Directory is empty.
EMLINK	3468	Maximum link count for a file was exceeded.

ESPIPE	3469	Seek request is not supported for object.
ENOSYS	3470	Function not implemented.
EISDIR	3471	Specified target is a directory.
EROFS	3472	Read-only file system.
EUNKNOWN	3474	Unknown system state.
EITERBAD	3475	Iterator is not valid.
EITERSTE	3476	Iterator is in wrong state for operation.
EHRICLSBAD	3477	HRI class is not valid.
EHRICLBAD	3478	HRI subclass is not valid.
EHRITYPBAD	3479	HRI type is not valid.
ENOTAPPL	3480	Data requested is not applicable.
EHRIREQTYP	3481	HRI request type is not valid.
EHRINAMEBAD	3482	HRI resource name is not valid.
EDAMAGE	3484	A damaged object was encountered.
ELOOP	3485	A loop exists in the symbolic links.
ENAMETOOLONG	3486	A path name is too long.
ENOLCK	3487	No locks are available.
ENOTEMPTY	3488	Directory is not empty.
ENOSYSRSC	3489	System resources are not available.
ECONVERT	3490	Conversion error.
E2BIG	3491	Argument list is too long.
EILSEQ	3492	Conversion stopped due to input character that does not belong to the input codeset.
ETYPE	3493	Object type mismatch.
EBADDIR	3494	Attempted to reference a directory that was not found or was destroyed.
EBADOBJ	3495	Attempted to reference an object that was not found, was destroyed, or was damaged.
EIDXINVAL	3496	Data space index used as a directory is not valid.
ESOFTDAMAGE	3497	Object has soft damage.
ENOTENROLL	3498	User is not enrolled in system distribution directory.
EOFFLINE	3499	Object is suspended.
EROOBJ	3500	Object is a read-only object.
EEAHDDSI	3501	Hard damage on extended attribute data space index.
EEASDDSI	3502	Soft damage on extended attribute data space index.
EEAHDDS	3503	Hard damage on extended attribute data space.
EEASDDS	3504	Soft damage on extended attribute data space.
EEADUPRC	3505	Duplicate extended attribute record.

ELOCKED	3506	Area being read from or written to is locked.
EFBIG	3507	Object too large.
EIDRM	3509	The semaphore, shared memory, or message queue identifier is removed from the system.
ENOMSG	3510	The queue does not contain a message of the desired type and (msgflg logically ANDed with IPC_NOWAIT).
EFILECVT	3511	File ID conversion of a directory failed.
EBADFID	3512	A file ID could not be assigned when linking an object to a directory.
ESTALE	3513	File handle was rejected by server.
ESRCH	3515	No such process.
ENOTSIGINIT	3516	Process is not enabled for signals.
ECHILD	3517	No child process.
EBADH	3520	Handle is not valid.
ETOOMANYREFS	3523	The operation would have exceeded the maximum number of references allowed for a descriptor.
ENOTSAFE	3524	Function is not allowed.
EOVERFLOW	3525	Object is too large to process.
EJRNDAMAGE	3526	Journal is damaged.
EJRNINACTIVE	3527	Journal is inactive.
EJRNRCVSPC	3528	Journal space or system storage error.
EJRNRMT	3529	Journal is remote.
ENEWJRNRCV	3530	New journal receiver is needed.
ENEWJRN	3531	New journal is needed.
EJOURNALED	3532	Object already journaled.
EJRNENTTOOLONG	3533	Entry is too large to send.
EDATALINK	3534	Object is a datalink object.
ENOTAVAIL	3535	IASP is not available.
ENOTTY	3536	I/O control operation is not appropriate.
EFBIG2	3540	Attempt to write or truncate file past its sort file size limit.
ETXTBSY	3543	Text file busy.
EASPGRPNOTSET	3544	ASP group not set for thread.
ERESTART	3545	A system call was interrupted and may be restarted.
