5 Signals

The UNIX operating system uses signals as a means of notifying a process that some event, often unrelated to the process's current activity, has occurred that requires the process's attention. Signals are delivered to a process asynchronously; a process cannot predict when a signal might arrive.

This chapter includes the following sections:

- POSIX Signal Functions, Section 5.1
- Signal Handling Basics, Section 5.2
- Realtime Signal Handling, Section 5.3

Signals originate from a number of sources:

- An exception, such as a divide-by-zero or segmentation violation, may be detected by hardware, causing the UNIX kernel to generate an appropriate signal (such as SIGFPE or SIGSEGV) and send it to the current process.
- A user may press certain terminal keys, such as Ctrl/C, to control the behavior of the currently running program. This causes the terminal driver program to send a signal (such as SIGINT) to the user-level process in which the program is running. (To see which signals are mapped to keys on your keyboard, issue the command `stty everything`. Signals sent from a keyboard are received by all processes in the process group currently associated with the terminal.)
- One user-level process may send a signal to another process. Traditionally, it does this using the `kill` function, although POSIX 1003.1b provides the `sigqueue` function for this purpose.
- A process may request a signal from the operating system when a timer expires, an asynchronous I/O operation completes, or a message arrives at an empty message queue.

The signal interface is also a traditional form of interprocess communication. Multitasking applications in particular take advantage of signals as a means of allowing components to coordinate activities across a number of processes. Because of the asynchronous nature of signals, a process can perform useful work while waiting for a significant event (for instance, it does not need to wait on a semaphore) and, when the event occurs, the process is notified immediately.

A process can specify what to do when it receives a signal. It can:

- Ignore the signal completely
- Handle the signal by establishing a function that is called whenever a particular signal is delivered
- Block the signal until it is able to deal with it. Typically the blocked signal has an established handler

An application can alternatively accept the default consequences of the delivery of a specific signal. These consequences vary from signal to signal, but can result in process termination, the process dumping core, the signal being ignored, or the process being restarted or continued. The default action of most signals is to terminate the process. If sudden process termination for the wide variety of conditions that cause signals is not desirable, an application should be prepared to deal with signals properly.

5.1 POSIX Signal Functions

POSIX 1003.1 standardized the reliable signal functions developed under 4.3BSD and SVR3. Table 5-1 lists the POSIX 1003.1 signal functions.

Table 5-1: POSIX 1003.1 Signal Functions

Function	Description
`kill`	Sends a signal to a process or a group of processes
`sigaction`	Specifies the action a process takes when a particular signal is delivered
`sigaddset`	Adds a signal to a signal set
`sigdelset`	Removes a signal from a signal set
`sigemptyset`	Initializes a signal set such that all signals are excluded
`sigfillset`	Initializes a signal set such that all signals are included
`sigismember`	Tests whether a signal is a member of a signal set
`sigpending`	Returns a signal set that represents those signals that are blocked from delivery to the process but are pending
`sigprocmask`	Sets the process's current blocked signal mask
`sigsuspend`	Replaces the process's current blocked signal mask, waits for a signal, and, upon its delivery, calls the handler established for the signal and returns

POSIX 1003.1b extended the POSIX 1003.1 definition to include better support for signals in realtime environments. Table 5-2 lists the POSIX 1003.1b signal functions. A realtime application uses the `sigqueue` function instead of the `kill` function. It may also use the `sigwaitinfo` or `sigtimedwait` function instead of the `sigsuspend` function.

Table 5-2: POSIX 1003.1b Signal Functions

Function	Description
`sigqueue`	Sends a signal, plus identifying information, to a process
`sigtimedwait`	Waits for a signal for the specified amount of time and, if the signal is delivered within that time, returns the signal number and any identifying information the signaling process provided
`sigwaitinfo`	Waits for a signal and, upon its delivery, returns the signal number and any identifying information the signaling process provided

To better explain the use of the POSIX 1003.1b extensions by realtime applications, this chapter first focuses on the basics of POSIX 1003.1 signal handling.

5.2 Signal Handling Basics

Example 5-1 shows the code for a process that creates a child that, in turn, creates and registers a signal handler, `catchit`.

Example 5-1: Sending a Signal to Another Process

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

#define SIG_STOP_CHILD SIGUSR1 [1]

main()
{
pid_t pid;
sigset_t newmask, oldmask;

    if ((pid = fork()) == 0) { [2]   /*Child*/
        struct sigaction action; [3]
        void catchit();

        sigemptyset(&newmask); [4]
        sigaddset(&newmask, SIG_STOP_CHILD); [5]
        sigprocmask(SIG_BLOCK, &newmask, &oldmask); [6]

        action.sa_flags = 0; [7]
        action.sa_handler = catchit;

        if (sigaction(SIG_STOP_CHILD, &action, NULL) == -1) { [8]
            perror("sigusr: sigaction");
            _exit(1);
        }
        sigsuspend(&oldmask); [9]
    }
    else {                              /* Parent */
        int stat;
        sleep(1); [10]
        kill(pid, SIG_STOP_CHILD); [11]
        pid = wait(&stat); [12]
        printf("Child exit status = %d\n", WEXITSTATUS(stat));
     _exit(0);
     }
}
void catchit(int signo) [13]
{
       printf("Signal %d received from parent\n", signo);
       _exit(0);
}
```

In this example:

1. The program defines one of the two signals POSIX 1003.1 reserves for application-specific purposes (SIGUSR1) to be a SIG_STOP_CHILD signal. [Return to example]

2. The `main` program forks, creating a child process. [Return to example]

3. The child process declares a `sigaction` structure named `action` and a signal handler named `catchit`. [Return to example]

4. The child process initializes the `newmask` `sigset_t` structure to zero. [Return to example]

5. The child process calls the `sigaddset` function to set the bit corresponding to the SIG_STOP_CHILD signal in the *newmask* `sigset_t` structure. [Return to example]

6. The child process specifies the *newmask* `sigset_t` structure to a `sigprocmask` function call, thus blocking the SIG_STOP_CHILD signal. [Return to example]

7. The child process fills in the `sigaction` structure: first by calling the `sigemptyset` function to initialize the signal set to exclude all signals, then clearing the `sa_flags` member and moving the address of the `catchit` signal handler into the `sa_handler` member. [Return to example]

8. The child process calls the `sigaction` function to set up the `catchit` signal handler so that it is called when the process receives the SIG_STOP_CHILD signal. [Return to example]

9. The child process calls the `sigsuspend` function. As a result, the SIG_STOP_CHILD signal is unblocked and the child process pauses until the SIG_STOP_CHILD signal is delivered (and causes its `catchit` signal handler to run. [Return to example]

10. The parent process sleeps for one second, allowing the child to run. [Return to example]

11. The parent process calls the `kill` function to send the SIG_STOP_CHILD signal to the child process. [Return to example]

12. It waits for the child process to terminate, printing the child's exit status when it does. Before this can occur, however, the child's `catchit` signal handler must run. [Return to example]

13. The `catchit` signal handler prints a message that acknowledges that the child received and handled the SIG_STOP_CHILD signal. [Return to example]

As in Example 5-1, under POSIX 1003.1, a process sends a signal to another process using the `kill` function. The first argument to the `kill` function is the process ID of the receiving process, or one of the following special values:

Value	Description
0	Sends the signal to all processes with the same process group ID as that of the sender
-1	Sends the signal to all processes with a process group ID equal to the effective user ID of the sender

The second argument to the `kill` function is the name or number of the signal to be sent.

The permissions checking allowed by the first argument helps ensure that signals cannot be sent that arbitrarily or accidentally terminate any process on the system. Inasmuch as a process must have the identical user ID or effective user ID as the process it is signaling, it is often the case that it has spawned these processes or explicitly called the `setuid` function to set their effective user IDs. See the `kill(2)` reference page for additional discussion of the `kill` function.

The full set of signals supported by the DIGITAL UNIX operating system is defined in `signal.h` and discussed in the `signal(4)` reference page. POSIX 1003.1 and POSIX 1003.1b require a subset of these signals; this subset is listed in Table 5-3.

Table 5-3: POSIX Signals

Signal	Description	Default Action
SIGABRT	Abort process (see abort(3))	Process termination and core dump
SIGALRM	Alarm clock expiration	Process termination
SIGFPE	Arithmetic exception (such as an integer divide-by-zero operation or a floating-point exception)	Process termination and core dump
SIGHUP	Hangup	Process termination
SIGILL	Invalid instruction	Process termination and core dump
SIGINT	Interrupt	Process termination
SIGKILL	Kill (cannot be caught, blocked, or ignored)	Process termination
SIGPIPE	Write on a pipe that has no reading process	Process termination
SIGQUIT	Quit	Process termination and core dump
SIGSEGV	Segmentation (memory access) violation	Process termination and core dump
SIGTERM	Software termination	Process termination
SIGUSR1	Application-defined	Process termination
SIGUSR2	Application-defined	Process termination
SIGCHLD	Child termination (sent to parent)	Ignored
SIGSTOP	Stop (cannot be caught, blocked, or ignored)	Process is stopped (suspended)
SIGTSTP	Interactive stop	Process is stopped (suspended)
SIGCONT	Continue if stopped (cannot be caught, blocked, or ignored)	Process is restarted (resumed)
SIGTTOU	Background write attempted to controlling terminal	Process is stopped (suspended)
SIGTTIN	Background read attempted from controlling terminal	Process is stopped (suspended)
SIGRTMIN-SIGRTMAX	Additional application-defined signals provided by POSIX 1003.1b	Process termination

---

5.2.1 Specifying a Signal Action

The `sigaction` function allows a process to specify the action to be taken for a given signal. When you set a signal-handling action with a call to the `sigaction` function, the action remains set until you explicitly reset it with another call to the `sigaction` function.

The first argument to the `sigaction` function specifies the signal for which the action is to be defined. The second and third arguments, unless specified as NULL, specify `sigaction` structures:

- The second argument is a `sigaction` structure that specifies the action to be taken when the process receives the signal specified in the first argument. If this argument is specified as NULL, signal handling is unchanged by the call to the `sigaction` function, but the call can be used to inquire about the current handling of a specified signal.

- The third argument is a `sigaction` structure that receives from the `sigaction` function the action that was previously established for the signal. An application typically specifies this argument so that it can use it in a subsequent call to the `sigaction` function that restores the previous signal state. This allows you to activate handlers only when they are needed, and deactivate them when they may interfere with other handlers set up elsewhere for the same signal.

The `sigaction` structure has two different formats, defined in `signal.h`, distinguished by whether the *sa_handler* member specifies a traditional POSIX 1003.1 signal handler or a POSIX 1003.1b realtime signal handler:

- For POSIX 1003.1 signal handling:

```
struct sigaction (
    void (*sa_handler) (int);
    sigset_t sa_mask;
    int sa_flags;
    };
```

- For POSIX 1003.1b signal handling:

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

The remainder of this section focuses on the definition of a traditional signal handler in the *sa_handler* member of the `sigaction` structure. Note that, for realtime signals (those defined as SIGRTMIN through SIGRTMAX, you define the *sa_sigaction* member, not the *sa_handler* member. Section 5.3 describes the definition of a realtime signal handler in the *sa_sigaction* member.

Use the *sa_handler* member of the `sigaction` structure to identify the action associated with a specific signal, as follows:

- To ignore the signal, specify SIG_IGN. In this case, the signal is never delivered to the process. Note that you cannot ignore the SIGKILL or SIGSTOP signals.

- To accept the default action for a signal, specify SIG_DFL.

- To handle the signal, specify a pointer to a signal handling function. When the signal handler is called, it is passed a single integer argument, the number of the signal. The handler is executed, passes control back to the process at the point where the signal was received, and execution continues. Handlers can also send error messages, save information about the status of the process when the signal was received, or transfer control to some other point in the application.

The *sa_mask* field identifies the additional set of signals to be added to the process's current signal mask before the signal handler is actually called. This signal mask, plus the current signal, is active while the process's signal handler is running (unless it modified by another call to the `sigaction` function, or a call to the `sigprocmask` or `sigsuspend` functions). If the signal handler completes successfully, the original mask is restored.

The *sa_flags* member specifies various flags that direct the operating system's dispatching of a signal. For a complete listing of these flags and a description of their meaning, see the `sigaction(2)` reference page.

5.2.2 Setting Signal Masks and Blocking Signals

A process blocks a signal to protect certain sections of code from receiving signals when the code cannot be interrupted. Unlike ignoring a signal, blocking a signal postpones the delivery of the signal until the process is ready to handle it. A blocked signal is marked as pending when it arrives and is handled as soon as the block is released. Under POSIX 1003.1, multiple occurrences of the same signal are not saved; that is, if a signal is generated again while the signal is already pending, only the one instance of the signal is delivered. The signal queuing capabilities introduced in POSIX 1003.1b allow multiple occurrences of the same signal to be preserved and distinguished (see Section 5.3).

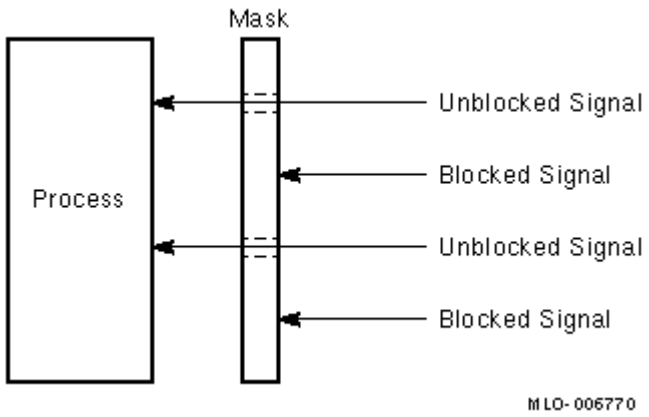
Each process has an associated signal mask that determines which signals are delivered to it and which signals are blocked from delivery. (A child process inherits its parent's signal mask when the parent forks.) Each bit represents a signal, as defined in the `signal.h` header file. For instance, if the *n*th bit in the mask is set, then signal *n* is blocked.

Note

As described in Chapter 2, the DIGITAL UNIX operating system actually schedules threads, not processes. For multithreaded applications, a signal can be delivered to a thread, using the `pthread_kill` function, and a thread signal mask can be created using the `pthread_sigmask` function. These functions are provided in the DECthreads POSIX 1003.1c library (`libpthread.so`). See the appropriate reference pages and the *Guide to DECthreads* for a discussion of using signals with multithreaded applications.

Figure 5-1 represents a mask blocking two signals. In this illustration, two signal bits are set, blocking signal delivery for the specified signals.

Figure 5-1: Signal Mask that Blocks Two Signals



The `sigprocmask` function lets you replace or alter the signal mask of the calling process; the value of the first argument to this function determines the action taken:

Value	Description
SIG_BLOCK	Adds the set of signals specified in the second argument to the process's signal mask

Value	Description
SIG_UNBLOCK	Subtracts the set of signals specified in the second argument from the process's signal mask
SIG_SETMASK	Replaces the process's signal mask with the set of signals specified in the third argument

The third argument to the `sigprocmask` function is a `sigset_t` structure that receives the process's previous signal mask.

Prior to calling the `sigprocmask` function, you use either the `sigemptyset` or `sigfillset` function to create the signal set (a `sigset_t` structure) that you provide as its second argument. The `sigemptyset` function creates a signal set with no signals in it. The `sigfillset` function creates a signal set containing all signals. You adjust the signal set you create with one of these functions by calling the `sigaddset` and `sigdelset` functions. You can determine whether a given signal is a member of a signal set by using the `sigismember` function.

The `sigprocmask` function is also useful when you want to set a mask but are uncertain as to which signals are still blocked. You can retrieve the current signal mask by calling `sigprocmask(SIG_BLOCK, NULL, &oldmask)`.

Once a signal is sent, it is delivered, unless delivery is blocked. When blocked, the signal is marked pending. Pending signals are delivered immediately once they are unblocked. To determine whether a blocked signal is pending, use the `sigpending` function.

5.2.3 Suspending a Process and Waiting for a Signal

The `sigsuspend` function replaces a process's signal mask with the mask specified as its only argument and waits for the delivery of an unblocked signal. If the signal delivery causes a signal handler to run, the `sigsuspend` function returns after the signal handler completes, having restored the process's signal mask to its previous state. If the signal delivery causes process termination, the `sigsuspend` function does not return.

Because `sigsuspend` sets the signal mask and waits for an unblocked signal in one atomic operation, the calling process does not miss delivery of a signal that may occur just before it is suspended.

A process typically uses the `sigsuspend` function to coordinate with the asynchronous completion of some work by some other process. For instance, it may block certain signals while executing a critical section and wait for a signal when it completes:

```
.
.
.
sigset_t newmask, oldmask;

sigemptyset(&newmask);
sigemptyset(&oldmask);
sigaddset(&newset, SIGUSR1);
sigaddset(&newset, SIGUSR2);
sigprocmask(SIG_BLOCK, &newmask, &oldmask);

      /* Code protected from SIGUSR1 and SIGUSR2 goes here */

      /* Release blocked signals and restore old mask */

sigsuspend(&oldmask);
```

.
.
.

5.2.4 Setting Up an Alternate Signal Stack

The XPG4-UNIX specification defines the `sigaltstack` function to allow a process to set up a discrete stack area on which signals can be processed. The alternate signal stack is used if the *sa_flags* member of the `sigaction` structure for the signal specifies the SA_ONSTACK flag.

The `stack_t` structure supplied to a call to the `sigaltstack` function determines the configuration and use of the alternate signal stack by the values of the following members:

- The *ss_sp* member contains a pointer to the location of the signal stack.
- If the *ss_flags* member is not NULL, it can specify the SS_DISABLE flag, in which case the stack is disabled upon creation.
- The *ss_size* member specifies the size of the stack.

See the `sigaltstack(2)` reference page for additional information on the `sigaltstack(2)` function.

5.3 Realtime Signal Handling

Traditional signals, as defined by POSIX 1003.1, have several limitations that make them unsuitable for realtime applications:

- There are too few user-defined signals.

  There are only two signals available for application use, SIGUSR1 and SIGUSR2. For those applications that are constructed from various general-purpose and special-purpose components, all executing concurrently, the same signal could trigger different actions, depending on the sender. To avoid the risk of calling the wrong signal handler, code must become more complex and avoid asynchronous, unpredictable signal delivery.

- There is no priority ordering to the delivery of signals.

  When multiple signals are pending to a process, the order in which they are delivered is undefined.

- Blocked signals are lost.

  A signal can be lost if it is not delivered immediately. A single bit in a signal set is set when a blocked signal arrives and is pending delivery to a process. When the signal is unblocked and delivered, this bit is cleared. While it is set, however, multiple instances of the same signal can arrive and be discarded.

- The signal delivery carries no information that distinguishes the signal from others of the same type.

  From the perspective of the receiving process, there is no information associated with signal delivery that explains where the signal came from or how it is different from other such signals it may receive.

To overcome some of these limitations, POSIX 1003.1b extends the POSIX 1003.1 signal functionality to include the following facilitators for realtime signals:

- A range of priority-ordered, application-specific signals from SIGRTMIN to SIGRTMAX
- A mechanism for queuing signals for delivery to a process
- A mechanism for providing additional information about a signal to the process to which it is delivered
- Features that allow efficient signal delivery to a process when a POSIX 1003.1b timer expires, when a message arrives on an empty message queue, or when an asynchronous I/O operation completes
- Functions that allow a process to respond more quickly to signal delivery

Example 5-2 shows some modifications to Example 5-1 that allow it to process realtime signals more efficiently.

Example 5-2: Sending a Realtime Signal to Another Process

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

#define SIG_STOP_CHILD SIGRTMIN+1 [1]

main()
{
pid_t pid;
sigset_t newmask, oldmask;

    if ((pid = fork()) == 0) { [2]   /*Child*/
        struct sigaction action;
        void catchit();

        sigemptyset(&newmask);
        sigaddset(&newmask, SIG_STOP_CHILD);
        sigprocmask(SIG_BLOCK, &newmask, &oldmask);

        action.sa_flags = SA_SIGINFO; [3]
        action.sa_sigaction = catchit;

        if (sigaction(SIG_STOP_CHILD, &action, NULL) == -1) { [4]
            perror("sigusr: sigaction");
            _exit(1);
        }
        sigsuspend(&oldmask);
    }
    else {                              /* Parent */
        union sigval sval; [5]
        sval.sigev_value.sival_int = 1;
        int stat;
        sleep(1); [6]
        sigqueue(pid, SIG_STOP_CHILD, sval); [7]
        pid = wait(&stat); [8]
        printf("Child exit status = %d\n", WEXITSTATUS(stat));
        _exit(0);
    }
}
void catchit(int signo, siginfo_t *info, void *extra) [9]
{
       void *ptr_val = info->si_value.sival_ptr;
       int int_val = info->si_value.sival_int;
       printf("Signal %d, value %d  received from parent\n", signo, int_val);
```

```
        _exit(0);
}
```

In this example:

1. The program defines one of the realtime signals defined by POSIX 1003.1b (SIGRTMIN+1) to be a SIG_STOP_CHILD signal. [Return to example]

2. The `main` program forks, creating a child process. The child process's initialization of the signal sets and creation of the process signal mask is the same as in the nonthreaded example in Example 5-1. [Return to example]

3. By specifying the SA_SIGINFO flag in the *sa_flags* member of the `sigaction` structure, the child process indicates that the associated signal will be using the realtime queued signaling behavior. [Return to example]

4. As in Example 5-1, the child process calls the `sigaction` function to set up the `catchit` signal handler so that it is called when the process receives the SIG_STOP_CHILD signal. It also calls the `sigsuspend` function to wait for a signal. [Return to example]

5. The parent process declares a `sigval` union. The member of this union can either be an integer or a pointer, depending on the value the parent sends to its child's signal handler. In this case, the value is an integer. [Return to example]

6. As in Example 5-1, the parent process sleeps for one second, allowing the child to run. [Return to example]

7. The parent process calls the `sigqueue` function to send the SIG_STOP_CHILD signal, plus a signal value, to the child process. [Return to example]

8. As in Example 5-1, the parent waits for the child process to terminate, printing the child's exit status when it does. Before this can occur, however, the child's `catchit` signal handler must run. [Return to example]

9. The `catchit` signal handler prints a message that acknowledges that the child received the SIG_STOP_CHILD signal and the signal value. [Return to example]

The following sections describe the POSIX 1003.1b extensions illustrated in this example.

5.3.1 Additional Realtime Signals

POSIX 1003.1 specified only two signals for application-specific purposes, SIGUSR1 and SIGUSR2. POSIX 1003.1b defines a range of realtime signals from SIGRTMIN to SIGRTMAX, the number of which is determined by the RTSIG_MAX constant in the `rt_limits.h` header file (which is included in the `limits.h` header file).

You specify these signals (in `sigaction` and other functions) by referring to them in terms of SIGRTMIN or SIGRTMAX: for instance, SIGRTMIN+1 or SIGRTMAX-1. Beware that SIGRTMIN and SIGRTMAX are not constants, so avoid compiler declarations that use them. You can determine the number of realtime signals on the system by calling `sysconf(_SC_RTSIG_MAX)`.

Although there is no defined delivery order for non-POSIX 1003.1b signals, the POSIX 1003.1b realtime signals are ranked from SIGRTMIN to SIGRTMAX (that is, the lowest numbered realtime signal has the highest priority). This means that, when these signals are blocked and pending, SIGRTMIN signals will be delivered first, SIGRTMIN+1 signals will be delivered next, and so on. Note that POSIX 1003.1b does not specify any

priority ordering for non-realtime signals, nor does it indicate the ordering of realtime signals relative to nonrealtime signals.

If you want a function to use only these new realtime signal numbers, you can block the old POSIX 1003.1 signal numbers in process signal masks.

5.3.2 Queuing Signals to a Process

As shown in Section 5.2.1, the `sigaction` structure a realtime process passes to the `sigaction` function has the following format:

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

A process specifies the POSIX 1003.1b realtime signaling behavior (including signal queuing and the passing of additional information about the signal to its handler) by setting the SA_SIGINFO flag in the *sa_flags* member of this structure. Setting the SA_SIGINFO bit has the following effects:

- It causes the signal, if blocked, to be queued to the process, instead of being marked as pending in the process's pending signal set.
- It causes the signal handler defined in the *sa_sigaction* member of the `sigaction` structure to be called.
- It causes the signal handler to be called with two arguments in addition to the signal number.

5.3.2.1 The siginfo_t Structure

The second argument provided to the signal handler is a `siginfo_t` structure that provides information that identifies the sender of the signal and the reason why the signal was sent. The `siginfo_t` structure is defined in the `siginfo.h` header file (included by the `signal.h` header file), as follows:

```
typedef struct siginfo {
        int si_signo;
        int si_errno;
        int si_code;
        pid_t si_pid;
        uid_t si_uid;
        int si_status;
        union sigval si_value;
        void *si_addr;
        long si_band;
        int si_fd;
} siginfo_t;
```

The following list describes the members of the `siginfo_t` structure:

- The *si_signo* member contains the signal number. It is identical to the value passed as the first argument passed to the signal handler.
- The *si_errno* member contains the `errno` value that is associated with the signal.

- The *si_code* member provides information that identifies the source of the signal. For POSIX.1b signals, it can contain one of the following values:

Value	Description
SI_ASYNCIO	The signal was sent on completion of an asynchronous I/O operation (see Section 5.3.3).
SI_MESGQ	The signal was sent on arrival of a message to an empty message queue (see Section 5.3.3).
SI_QUEUE	The signal was sent by the `sigqueue` function.
SI_TIMER	The signal was sent because of a timer expiration (see Section 5.3.3).
SI_USER	The signal was sent by `kill` function or a similar function such as `abort` or `raise`.

  For XPG4-UNIX signals, this member can contain other values, as described in the `siginfo(5)` reference page.

- The *si_pid* member contains the process identification (PID) of the sending process.

- The *si_uid* member contains the user identification (UID) of the sending process. It is valid only when the *si_code* member contains the value SI_USER.

- The *si_status* member contains the exit value returned from the child process when a SIGCHLD signal is generated.

- The *si_value* member contains an application-specific value that has been passed to the signal handler in the last argument to the `sigqueue` function that generated the signal. The *si_value* member can contain either of the following members, depending upon whether the application-specific value is an integer or a pointer:

```
typedef union sigval {
        int     sival_int;
        void    *sival_ptr;
} sigval_t;
```

- The *si_addr* member contains a pointer to the faulting instruction or memory reference. It is valid only for the SIGILL, SIGFPE, SIGSEGV, and SIGBUS signals.

- The *si_band* member contains the band event job-control character (POLL_OUT, POLL_IN, or POLL_MSG) for the SIGPOLL signal. See the `poll(2)` reference page for additional information on poll events.

- The *si_fd* member contains a pointer to the file descriptor of the poll event associated with the SIGPOLL signal.

5.3.2.2 The ucontext_t and sigcontext Structures

The third argument passed to a signal handler when the SA_SIGINFO flag is specified in the *sa_flags* member of the `sigaction` structure is defined by POSIX.1b as an "extra" argument. The DIGITAL UNIX operating system uses this field to pass a `ucontext_t` structure to a signal handler in an XPG4-UNIX environment, or a `sigcontext` structure in a BSD environment.

Both structures contain the receiving process's context at the time at which it was interrupted by the signal. The `sigcontext` structure is defined in the `signal.h` header file. The `ucontext_t` structure is defined in the `ucontext.h` header file and is fully described in the `ucontext(5)` reference page.

5.3.2.3 Sending a Realtime Signal With the sigqueue Function

Where a process uses the `kill` function to send a nonrealtime signal to another process, it uses the `sigqueue` function to send a realtime signal. The `sigqueue` function resembles the `kill` function, except that it provides an additional argument, an application-defined signal value that is passed to the signal handler in the *si_value* member of the `siginfo_t` structure if the receiving process has enabled the SA_SIGINFO flag in the *sa_flags* member of the signal's `sigaction` structure.

The `sigqueue` function queues the specified signal to the receiving process. The permissions checking for the `sigqueue` function are the same as those applied to the `kill` function (see Section 5.2). Nonprivileged callers are restricted in the number of signals they can have actively queued at any time. This per-process quota value is defined in the `rt_limits.h` header file (which is included in the `limits.h` header file) as SIGQUEUE_MAX and is configurable by the system administrator. You can retrieve its value by calling `sysconf(_SC_SIGQUEUE_MAX)`.

5.3.3 Asynchronous Delivery of Other Realtime Signals

Besides providing the `sigqueue` function to send realtime signals to processes, the POSIX 1003.1b standard defines additional features that extend realtime signal generation and delivery to functions that require asynchronous notification. Realtime functions are provided that automatically generate realtime signals for the following events:

- Asynchronous I/O completion (as initiated by the `aio_read`, `aio_write`, or `lio_listio` function)
- Timer expiration (for a timer established by the `timer_create` function)
- Arrival of a message to an empty message queue (for a message queue created by the `mq_notify` function)

When using the functions that trigger these events, you do not need to call a separate function to deliver signals. Realtime signal delivery for these events employs a `sigevent` structure, which is supplied as an argument (either directly or indirectly) to the appropriate function call. The `sigevent` structure contains information that describes the signal (or, prospectively, another mechanism of asynchronous notification to be used). It is defined in the `signal.h` header file and contains the following members:

```
int             sigev_notify;
union sigval    sigev_value;
int             sigev_signo;
```

The *sigev_notify* member specifies the notification mechanism to use when an asynchronous event occurs. There are two values defined for *sigev_notify* in POSIX 1003.1b:

Value	Description
SIGEV_SIGNAL	Indicates that a queued signal with an application-defined value is delivered when an event occurs.
SIGEV_NONE	Indicates that no asynchronous notification is delivered when an event occurs.

If the *sigev_notify* member contains SIGEV_SIGNAL, the other two members of the `sigevent` structure are meaningful.

The *sigev_value* member is an application-defined value to be passed to the signal catching function at the time of signal delivery. It can contain either of the following members, depending upon whether the application-specific value is an integer or a pointer:

```
typedef union sigval {
        int     sival_int;
        void    *sival_ptr;
} sigval_t;
```

The *sigev_signo* member specifies the signal number to be sent on completion of the asynchronous I/O operation, timer expiration, or delivery of a message to the message queue. For any of these events, you must use the `sigaction` function to set up a signal handler to execute once the signal is received. Refer to Chapter 6 and Chapter 7 for examples of using signals with these functions.

5.3.4 Responding to Realtime Signals Using the sigwaitinfo and sigtimedwait Functions

The `sigsuspend` function, described in Section 5.2.3, allows a process to block while waiting for signal delivery. When the signal arrives, the process's signal handler is called. When the handler, completes, the process is unblocked and continues execution.

The `sigwaitinfo` and `sigtimedwait` functions, defined in POSIX 1003.1b, also allow a process to block waiting for signal delivery. However, unlike `sigsuspend`, they do not call the process's signal handler when a signal arrives. Rather, they immediately unblock the process, returning the number of the received signal as a status value.

The first argument to these functions is a signal mask that specifies which signals the process is waiting for. The process must have blocked the signals specified in this mask; otherwise, they will be dispatched to any established signal handler. The second argument is an optional pointer to a location to which the function returns `siginfo_t` structure that describes the signal.

The `sigtimedwait` function further allows you to specify a timeout value, allowing you to set a limit to the time the process waits for a signal.

Example 5-3 shows a version of Example 5-2 that eliminates the signal handler that runs when the child process receives a SIG_STOP_CHILD signal from its parent. Instead, the child process blocks the signal and calls the `sigwaitinfo` function to wait for its delivery.

Example 5-3: Using the sigwaitinfo Function

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

#define SIG_STOP_CHILD SIGRTMIN+1

main()
{
pid_t pid;
sigset_t newmask;
int rcvd_sig; [1]
siginfo_t info; [2]
```

```
    if ((pid = fork()) == 0) {       /*Child*/

        sigemptyset(&newmask);
        sigaddset(&newmask, SIG_STOP_CHILD);
        sigprocmask(SIG_BLOCK, &newmask, NULL); [3]

        while (1) { [4]
            rcvd_sig = sigwaitinfo(&newmask, &info) [5]
            if (rcvd_sig == -1) {
               perror("sigusr: sigwaitinfo");
               _exit(1);
            }
            else { [6]
                 printf("Signal %d, value %d  received from parent\n",
                     rcvd_sig, info.si_value.sival_int);
               _exit(0);
            }
        }
    }
    else {                           /* Parent */
        union sigval sval;
        sval.sigev_value.sival_int = 1;
        int stat;
        sleep(1);
        sigqueue(pid, SIG_STOP_CHILD, sval);
        pid = wait(&stat);
        printf("Child exit status = %d\n", WEXITSTATUS(stat));
        _exit(0);
    }
}
```

In this example:

1. The program defines a variable to which the `sigwaitinfo` call returns the value of the delivered signal (or returns -1, indicating an error). [Return to example]

2. The program defines a variable to which the `sigwaitinfo` call returns the `siginfo_t` structure that describes the received signal. [Return to example]

3. The child process sets up a signal mask to blocks the SIG_STOP_CHILD signal. Notice that it has not defined a signal handler to run when the signal is delivered. The `sigwaitinfo` function does not call a signal handler. [Return to example]

4. The child process loops waiting for signal delivery. [Return to example]

5. The child process calls `sigwaitinfo` function, specifying the `newmask` signal mask to block the SIG_STOP_CHILD signal and wait for its delivery. [Return to example]

6. When the signal is delivered, the child process prints a message indicating that it has received the signal. It also prints the signal value that may accompany the realtime signal. [Return to example]

An additional example using the `sigwaitinfo` function is shown in Example 5-4. In this example, the child process sends to its parent the maximum number of signals that the system allows to be queued. When a SIG signal is delivered to it, the parent counts it and prints an informative message. After it has received _SC_SIGQUEUE_MAX signals, the parent prints a message that indicates the number of signals it has received.

Example 5-4: Using the sigwaitinfo Function

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

main()
{
        sigset_t        set, pend;
        int             i, sig, sigq_max, numsigs = 0;
        siginfo_t       info;
        int             SIG = SIGRTMIN;

        sigq_max = sysconf(_SC_SIGQUEUE_MAX);
        sigemptyset(&set);
        sigaddset(&set, SIG);
        sigprocmask(SIG_SETMASK, &set, NULL);
        printf("\nNow create a child to send signals...\n");
        if (fork() == 0) {      /* child */
                pid_t parent = getppid();
                printf("Child will signal parent %d\n", parent);
                for (i = 0; i < sigq_max; i++) {
                        if (sigqueue(parent, SIG, i) < 0)
                                perror("sigqueue");
                }
                exit(1);
        }
        printf("Parent sigwait for child to queue signal...\n");
        sigpending(&pend);
        printf("Is signal pending: %s\n",
               sigismember(&pend, SIG) ? "yes" : "no");
        for (i = 0; i < sigq_max; i++) {
                sig = sigwaitinfo(&set, &info);
                if (sig < 0) {
                        perror("sigwait");
                        exit(1);
                }
                printf("Main woke up after signal %d\n", sig);
                printf("signo = %d, pid = %d, uid = %d, val = %d,\n",
                       info.si_signo, info.si_pid, info.si_uid, info.si_int);
                numsigs++;
        }
        printf("Main: done after %d signals.\n", numsigs);
}
```