# Signal Handling

A *signal* is a software interrupt delivered to a process. The operating system uses signals to report exceptional situations to an executing program. Some signals report errors such as references to invalid memory addresses; others report asynchronous events, such as disconnection of a phone line.

The GNU C library defines a variety of signal types, each for a particular kind of event. Some kinds of events make it inadvisable or impossible for the program to proceed as usual, and the corresponding signals normally abort the program. Other kinds of signals that report harmless events are ignored by default.

If you anticipate an event that causes signals, you can define a handler function and tell the operating system to run it when that particular type of signal arrives.

Finally, one process can send a signal to another process; this allows a parent process to abort a child, or two related processes to communicate and synchronize.

## Basic Concepts of Signals

This section explains basic concepts of how signals are generated, what happens after a signal is delivered, and how programs can handle signals.

### Some Kinds of Signals

A signal reports the occurrence of an exceptional event. These are some of the events that can cause (or *generate*, or *raise*) a signal:

- A program error such as dividing by zero or issuing an address outside the valid range.
- A user request to interrupt or terminate the program. Most environments are set up to let a user suspend the program by typing `C-z`, or terminate it with `C-c`. Whatever key sequence is used, the operating system sends the proper signal to interrupt the process.
- The termination of a child process.
- Expiration of a timer or alarm.
- A call to `kill` or `raise` by the same process.
- A call to `kill` from another process. Signals are a limited but useful form of interprocess communication.

Each of these kinds of events (excepting explicit calls to `kill` and `raise`) generates its own particular kind of signal. The various kinds of signals are listed and described in detail in section Standard Signals.

### Concepts of Signal Generation

In general, the events that generate signals fall into three major categories: errors, external events, and explicit requests.

An error means that a program has done something invalid and cannot continue execution. But not all kinds of errors generate signals--in fact, most do not. For example, opening a nonexistant file is an error, but it does not raise a signal; instead, `open` returns `-1`. In general, errors that are necessarily associated with certain library functions are reported by returning a value that indicates an error. The errors which raise signals are those which can happen anywhere in the program, not just in library calls. These include division by zero and invalid memory addresses.

An external event generally has to do with I/O or other processes. These include the arrival of input, the expiration of a timer, and the termination of a child process.

An explicit request means the use of a library function such as `kill` whose purpose is specifically to generate a signal.

Signals may be generated *synchronously* or *asynchronously*. A synchronous signal pertains to a specific action in the program, and is delivered (unless blocked) during that action. Errors generate signals synchronously, and so do explicit requests by a process to generate a signal for that same process.

Asynchronous signals are generated by events outside the control of the process that receives them. These signals arrive at unpredictable times during execution. External events generate signals asynchronously, and so do explicit requests that apply to some other process.

A given type of signal is either typically synchrous or typically asynchronous. For example, signals for errors are typically synchronous because errors generate signals synchronously. But any type of signal can be generated synchronously or asynchronously with an explicit request.

## How Signals Are Delivered

When a signal is generated, it becomes *pending*. Normally it remains pending for just a short period of time and then is *delivered* to the process that was signaled. However, if that kind of signal is currently *blocked*, it may remain pending indefinitely--until signals of that kind are *unblocked*. Once unblocked, it will be delivered immediately. See section Blocking Signals.

When the signal is delivered, whether right away or after a long delay, the *specified action* for that signal is taken. For certain signals, such as `SIGKILL` and `SIGSTOP`, the action is fixed, but for most signals, the program has a choice: ignore the signal, specify a *handler function*, or accept the *default action* for that kind of signal. The program specifies its choice using functions such as `signal` or `sigaction` (see section Specifying Signal Actions). We sometimes say that a handler *catches* the signal. While the handler is running, that particular signal is normally blocked.

If the specified action for a kind of signal is to ignore it, then any such signal which is generated is discarded immediately. This happens even if the signal is also blocked at the time. A signal discarded in this way will never be delivered, not even if the program subsequently specifies a different action for that kind of signal and then unblocks it.

If a signal arrives which the program has neither handled nor ignored, its *default action* takes place. Each kind of signal has its own default action, documented below (see section Standard Signals). For most kinds of signals, the default action is to terminate the process. For certain kinds of signals that represent "harmless" events, the default action is to do nothing.

When a signal terminates a process, its parent process can determine the cause of termination by examining the termination status code reported by the `wait` or `waitpid` functions. (This is discussed in more detail in section Process Completion.) The information it can get includes the fact that termination was due to a signal, and the kind of signal involved. If a program you run from a shell is terminated by a signal, the shell typically prints some kind of error message.

The signals that normally represent program errors have a special property: when one of these signals terminates the process, it also writes a *core dump file* which records the state of the process at the time of termination. You can examine the core dump with a debugger to investigate what caused the error.

If you raise a "program error" signal by explicit request, and this terminates the process, it makes a core dump file just as if the signal had been due directly to an error.

## Standard Signals

This section lists the names for various standard kinds of signals and describes what kind of event they mean. Each signal name is a macro which stands for a positive integer--the *signal number* for that kind of signal. Your programs should never make assumptions about the numeric code for a particular kind of signal, but rather refer to them always by the names defined here. This is because the number for a given kind of signal can vary from system to system, but the meanings of the names are standardized and fairly uniform.

The signal names are defined in the header file `` `signal.h' ``.

Macro: int **NSIG**

The value of this symbolic constant is the total number of signals defined. Since the signal numbers are allocated consecutively, `NSIG` is also one greater than the largest defined signal number.

### Program Error Signals

The following signals are generated when a serious program error is detected by the operating system or the computer itself. In general, all of these signals are indications that your program is seriously broken in some way, and there's usually no way to continue the computation which encountered the error.

Some programs handle program error signals in order to tidy up before terminating; for example, programs that turn off echoing of terminal input should handle program error signals in order to turn echoing back on. The handler should end by specifying the default action for the signal that happened and then reraising it; this will cause the program to terminate with that signal, as if it had not had a handler. (See section Handlers That Terminate the Process.)

Termination is the sensible ultimate outcome from a program error in most programs. However, programming systems such as Lisp that can load compiled user programs might need to keep executing even if a user program incurs an error. These programs have handlers which use `longjmp` to return control to the command level.

The default action for all of these signals is to cause the process to terminate. If you block or ignore these signals or establish handlers for them that return normally, your program will probably break horribly when such signals happen, unless they are generated by `raise` or `kill` instead of a real error.

When one of these program error signals terminates a process, it also writes a *core dump file* which records the state of the process at the time of termination. The core dump file is named `` `core' `` and is written in whichever directory is current in the process at the time. (On the GNU system, you can specify the file name for core dumps with the environment variable `COREFILE`.) The purpose of core dump files is so that you can examine them with a debugger to investigate what caused the error.

<u>Macro:</u> int **SIGFPE**

The `SIGFPE` signal reports a fatal arithmetic error. Although the name is derived from "floating-point exception", this signal actually covers all arithmetic errors, including division by zero and overflow. If a program stores integer data in a location which is then used in a floating-point operation, this often causes an "invalid operation" exception, because the processor cannot recognize the data as a floating-point number.

Actual floating-point exceptions are a complicated subject because there are many types of exceptions with subtly different meanings, and the `SIGFPE` signal doesn't distinguish between them. The *IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)* defines various floating-point exceptions and requires conforming computer systems to report their occurrences. However, this standard does not specify how the exceptions are reported, or what kinds of handling and control the operating system can offer to the programmer.

BSD systems provide the `SIGFPE` handler with an extra argument that distinguishes various causes of the exception. In order to access this argument, you must define the handler to accept two arguments, which means you must cast it to a one-argument function type in order to establish the handler. The GNU library does provide this extra argument, but the value is meaningful only on operating systems that provide the information (BSD systems and GNU systems).

`FPE_INTOVF_TRAP`
: Integer overflow (impossible in a C program unless you enable overflow trapping in a hardware-specific fashion).

`FPE_INTDIV_TRAP`
: Integer division by zero.

`FPE_SUBRNG_TRAP`
: Subscript-range (something that C programs never check for).

`FPE_FLTOVF_TRAP`
: Floating overflow trap.

`FPE_FLTDIV_TRAP`
: Floating/decimal division by zero.

`FPE_FLTUND_TRAP`
: Floating underflow trap. (Trapping on floating underflow is not normally enabled.)

`FPE_DECOVF_TRAP`
: Decimal overflow trap. (Only a few machines have decimal arithmetic and C never uses it.)

<u>Macro:</u> int **SIGILL**

The name of this signal is derived from "illegal instruction"; it means your program is trying to execute garbage or a privileged instruction. Since the C compiler generates only valid instructions, `SIGILL` typically indicates that the executable file is corrupted, or that you are trying to execute data. Some common ways of getting into the latter situation are by passing an invalid object where a pointer to a function was expected, or by writing past the end of an automatic array (or similar problems with pointers to automatic variables) and corrupting other data on the stack such as the return address of a stack frame.

<u>Macro:</u> int **SIGSEGV**

This signal is generated when a program tries to read or write outside the memory that is allocated for it. (Actually, the signals only occur when the program goes far enough outside to be detected by the system's memory protection mechanism.) The name is an abbreviation for "segmentation violation".

The most common way of getting a `SIGSEGV` condition is by dereferencing a null or uninitialized pointer. A null pointer refers to the address 0, and most operating systems make sure this address is always invalid precisely so that dereferencing a null pointer will cause `SIGSEGV`. (Some operating systems place valid memory at address 0, and dereferencing a null pointer does not cause a signal on these systems.) As for uninitialized pointer variables, they contain random addresses which may or may not be valid.

Another common way of getting into a `SIGSEGV` situation is when you use a pointer to step through an array, but fail to check for the end of the array.

Macro: int **SIGBUS**

This signal is generated when an invalid pointer is dereferenced. Like `SIGSEGV`, this signal is typically the result of dereferencing an uninitialized pointer. The difference between the two is that `SIGSEGV` indicates an invalid access to valid memory, while `SIGBUS` indicates an access to an invalid address. In particular, `SIGBUS` signals often result from dereferencing a misaligned pointer, such as referring to a four-word integer at an address not divisible by four. (Each kind of computer has its own requirements for address alignment.)

The name of this signal is an abbreviation for "bus error".

Macro: int **SIGABRT**

This signal indicates an error detected by the program itself and reported by calling `abort`. See section Aborting a Program.

## Termination Signals

These signals are all used to tell a process to terminate, in one way or another. They have different names because they're used for slightly different purposes, and programs might want to handle them differently.

The reason for handling these signals is usually so your program can tidy up as appropriate before actually terminating. For example, you might want to save state information, delete temporary files, or restore the previous terminal modes. Such a handler should end by specifying the default action for the signal that happened and then reraising it; this will cause the program to terminate with that signal, as if it had not had a handler. (See section Handlers That Terminate the Process.)

The (obvious) default action for all of these signals is to cause the process to terminate.

Macro: int **SIGHUP**

The `SIGHUP` ("hang-up") signal is used to report that the user's terminal is disconnected, perhaps because a network or telephone connection was broken. For more information about this, see section Control Modes.

This signal is also used to report the termination of the controlling process on a terminal to jobs associated with that session; this termination effectively disconnects all processes in the session from the controlling terminal. For more information, see section Termination Internals.

Macro: int **SIGINT**

The `SIGINT` ("program interrupt") signal is sent when the user types the INTR character (normally `C-c`). See section Special Characters, for information about terminal driver support for `C-c`.

Macro: int **SIGQUIT**

The `SIGQUIT` signal is similar to `SIGINT`, except that it's controlled by a different key--the QUIT character, usually `C-\`---and produces a core dump when it terminates the process, just like a program error signal. You can think of this as a program error condition "detected" by the user.

See section Program Error Signals, for information about core dumps. See section Special Characters, for information about terminal driver support.

Certain kinds of cleanups are best omitted in handling `SIGQUIT`. For example, if the program creates temporary files, it should handle the other termination requests by deleting the temporary files. But it is better for `SIGQUIT` not to delete them, so that the user can examine them in conjunction with the core dump.

Macro: int **SIGTERM**

The `SIGTERM` signal is a generic signal used to cause program termination. Unlike `SIGKILL`, this signal can be blocked, handled, and ignored.

The shell command `kill` generates `SIGTERM` by default.

Macro: int **SIGKILL**

The `SIGKILL` signal is used to cause immediate program termination. It cannot be handled or ignored, and is therefore always fatal. It is also not possible to block this signal.

This signal is generated only by explicit request. Since it cannot be handled, you should generate it only as a last resort, after first trying a less drastic method such as `C-c` or `SIGTERM`. If a process does not respond to any other termination signals, sending it a `SIGKILL` signal will almost always cause it to go away.

In fact, if `SIGKILL` fails to terminate a process, that by itself constitutes an operating system bug which you should report.

## Alarm Signals

These signals are used to indicate the expiration of timers. See section Setting an Alarm, for information about functions that cause these signals to be sent.

The default behavior for these signals is to cause program termination. This default is rarely useful, but no other default would be useful; most of the ways of using these signals would require handler functions in any case.

Macro: int **SIGALRM**

This signal typically indicates expiration of a timer that measures real or clock time. It is used by the `alarm` function, for example.

Macro: int **SIGVTALRM**

This signal typically indicates expiration of a timer that measures CPU time used by the current process. The name is an abbreviation for "virtual time alarm".

Macro: int **SIGPROF**

This signal is typically indicates expiration of a timer that measures both CPU time used by the current process, and CPU time expended on behalf of the process by the system. Such a timer is used to implement code profiling facilities, hence the name of this signal.

## Asynchronous I/O Signals

The signals listed in this section are used in conjunction with asynchronous I/O facilities. You have to take explicit action by calling `fcntl` to enable a particular file descriptor to generate these signals (see section Interrupt-Driven Input). The default action for these signals is to ignore them.

Macro: int **SIGIO**

This signal is sent when a file descriptor is ready to perform input or output.

On most operating systems, terminals and sockets are the only kinds of files that can generate `SIGIO`; other kinds, including ordinary files, never generate `SIGIO` even if you ask them to.

Macro: int **SIGURG**

This signal is sent when "urgent" or out-of-band data arrives on a socket. See section Out-of-Band Data.

## Job Control Signals

These signals are used to support job control. If your system doesn't support job control, then these macros are defined but the signals themselves can't be raised or handled.

You should generally leave these signals alone unless you really understand how job control works. See section Job Control.

Macro: int **SIGCHLD**

This signal is sent to a parent process whenever one of its child processes terminates or stops.

The default action for this signal is to ignore it. If you establish a handler for this signal while there are child processes that have terminated but not reported their status via `wait` or `waitpid` (see section Process Completion), whether your new handler applies to those processes or not depends on the particular operating system.

Macro: int **SIGCONT**

You can send a `SIGCONT` signal to a process to make it continue. The default behavior for this signal is to make the process continue if it is stopped, and to ignore it otherwise.

Most programs have no reason to handle `SIGCONT`; they simply resume execution without realizing they were ever stopped. You can use a handler for `SIGCONT` to make a program do something special when it is stopped and continued--for example, to reprint a prompt when it is suspended while waiting for input.

Macro: int **SIGSTOP**

The `SIGSTOP` signal stops the process. It cannot be handled, ignored, or blocked.

Macro: int **SIGTSTP**

The `SIGTSTP` signal is an interactive stop signal. Unlike `SIGSTOP`, this signal can be handled and ignored.

Your program should handle this signal if you have a special need to leave files or system tables in a secure state when a process is stopped. For example, programs that turn off echoing should handle `SIGTSTP` so they can turn echoing back on before stopping.

This signal is generated when the user types the SUSP character (normally `C-z`). For more information about terminal driver support, see section Special Characters.

Macro: int **SIGTTIN**

A process cannot read from the the user's terminal while it is running as a background job. When any process in a background job tries to read from the terminal, all of the processes in the job are sent a `SIGTTIN` signal. The default action for this signal is to stop the process. For more information about how this interacts with the terminal driver, see section Access to the Controlling Terminal.

Macro: int **SIGTTOU**

This is similar to `SIGTTIN`, but is generated when a process in a background job attempts to write to the terminal or set its modes. Again, the default action is to stop the process.

While a process is stopped, no more signals can be delivered to it until it is continued, except `SIGKILL` signals and (obviously) `SIGCONT` signals. The `SIGKILL` signal always causes termination of the process and can't be blocked or ignored. You can block or ignore `SIGCONT`, but it always causes the process to be continued anyway if it is stopped. Sending a `SIGCONT` signal to a process causes any pending stop signals for that process to be discarded. Likewise, any pending `SIGCONT` signals for a process are discarded when it receives a stop signal.

When a process in an orphaned process group (see section Orphaned Process Groups) receives a `SIGTSTP`, `SIGTTIN`, or `SIGTTOU` signal and does not handle it, the process does not stop. Stopping the process would be unreasonable since there would be no way to continue it. What happens instead depends on the operating system you are using. Some systems may do nothing; others may deliver another signal instead, such as `SIGKILL` or `SIGHUP`.

## Miscellaneous Signals

These signals are used to report various other conditions. The default action for all of them is to cause the process to terminate.

Macro: int **SIGPIPE**

If you use pipes or FIFOs, you have to design your application so that one process opens the pipe for reading before another starts writing. If the reading process never starts, or terminates unexpectedly, writing to the pipe or FIFO raises a `SIGPIPE` signal. If `SIGPIPE` is blocked, handled or ignored, the offending call fails with `EPIPE` instead.

Pipes and FIFO special files are discussed in more detail in section Pipes and FIFOs.

Another cause of `SIGPIPE` is when you try to output to a socket that isn't connected. See section Sending Data.

Macro: int **SIGUSR1**

Macro: int **SIGUSR2**

The `SIGUSR1` and `SIGUSR2` signals are set aside for you to use any way you want. They're useful for interprocess communication. Since these signals are normally fatal, you should write a signal handler for them in the program that receives the signal.

There is an example showing the use of `SIGUSR1` and `SIGUSR2` in section Signaling Another Process.

## Nonstandard Signals

Particular operating systems support additional signals not listed above. The ANSI C standard reserves all identifiers beginning with `` `SIG' `` followed by an uppercase letter for the names of signals. You should consult the documentation or header files for your particular operating system and processor type to find out about the specific signals it supports.

For example, some systems support extra signals which correspond to hardware traps. Some other kinds of signals commonly supported are used to implement limits on CPU time or file system usage, asynchronous changes to terminal configuration, and the like. Systems may also define signal names that are aliases for standard signal names.

You can generally assume that the default action (or the action set up by the shell) for implementation-defined signals is reasonable, and not worry about them yourself. In fact, it's usually a bad idea to ignore or block signals you don't know anything about, or try to establish a handler for signals whose meanings you don't know.

Here are some of the other signals found on commonly used operating systems:

`SIGCLD`
: Obsolete name for `SIGCHLD`.

`SIGTRAP`
: Generated by the machine's breakpoint instruction. Used by debuggers. Default action is to dump core.

`SIGIOT`
: Generated by the PDP-11 "iot" instruction; equivalent to `SIGABRT`. Default action is to dump core.

`SIGEMT`
: Emulator trap; this results from certain unimplemented instructions. It is a program error signal.

`SIGSYS`
: Bad system call; that is to say, the instruction to trap to the operating system was executed, but the code number for the system call to perform was invalid. This is a program error signal.

`SIGPOLL`
: This is a System V signal name, more or less similar to `SIGIO`.

`SIGXCPU`
: CPU time limit exceeded. This is used for batch processing. Default action is program termination.

`SIGXFSZ`
: File size limit exceeded. This is used for batch processing. Default action is program termination.

`SIGWINCH`
: Window size change. This is generated on certain systems when the size of the current window on the screen is changed. Default action is to ignore it.

## Signal Messages

We mentioned above that the shell prints a message describing the signal that terminated a child process. The clean way to print a message describing a signal is to use the functions `strsignal` and `psignal`. These functions use a signal number to specify which kind of signal to describe. The signal number may come from the termination status of a child process (see section Process Completion) or it may come from a signal handler in the same process.

Function: char * **strsignal** *(int signum)*

This function returns a pointer to a statically-allocated string containing a message describing the signal *signum*. You should not modify the contents of this string; and, since it can be rewritten on subsequent calls, you should save a copy of it if you need to reference it later.

This function is a GNU extension, declared in the header file `` `string.h' ``.

Function: void **psignal** *(int signum, const char *message)*

This function prints a message describing the signal *signum* to the standard error output stream `stderr`; see section Standard Streams.

If you call `psignal` with a *message* that is either a null pointer or an empty string, `psignal` just prints the message corresponding to *signum*, adding a trailing newline.

If you supply a non-null *message* argument, then `psignal` prefixes its output with this string. It adds a colon and a space character to separate the *message* from the string corresponding to *signum*.

This function is a BSD feature, declared in the header file `` `stdio.h' ``.

There is also an array `sys_siglist` which contains the messages for the various signal codes. This array exists on BSD systems, unlike `strsignal`.

# Specifying Signal Actions

The simplest way to change the action for a signal is to use the `signal` function. You can specify a built-in action (such as to ignore the signal), or you can *establish a handler*.

The GNU library also implements the more versatile `sigaction` facility. This section describes both facilities and gives suggestions on which to use when.

## Basic Signal Handling

The `signal` function provides a simple interface for establishing an action for a particular signal. The function and associated macros are declared in the header file `` `signal.h' ``.

Data Type: **sighandler_t**

This is the type of signal handler functions. Signal handlers take one integer argument specifying the signal number, and have return type `void`. So, you should define handler functions like this:

```
void handler (int signum) { ... }
```

The name `sighandler_t` for this data type is a GNU extension.

Function: sighandler_t **signal** *(int signum, sighandler_t action)*

The `signal` function establishes *action* as the action for the signal *signum*.

The first argument, *signum*, identifies the signal whose behavior you want to control, and should be a signal number. The proper way to specify a signal number is with one of the symbolic signal names described in section Standard Signals---don't use an explicit number, because the numerical code for a given kind of signal may vary from operating system to operating system.

The second argument, *action*, specifies the action to use for the signal *signum*. This can be one of the following:

`SIG_DFL`

`SIG_DFL` specifies the default action for the particular signal. The default actions for various kinds of signals are stated in section Standard Signals.

`SIG_IGN`

`SIG_IGN` specifies that the signal should be ignored.

Your program generally should not ignore signals that represent serious events or that are normally used to request termination. You cannot ignore the `SIGKILL` or `SIGSTOP` signals at all. You can ignore program error signals like `SIGSEGV`, but ignoring the error won't enable the program to continue executing meaningfully. Ignoring user requests such as `SIGINT`, `SIGQUIT`, and `SIGTSTP` is unfriendly.

When you do not wish signals to be delivered during a certain part of the program, the thing to do is to block them, not ignore them. See section Blocking Signals.

*handler*

Supply the address of a handler function in your program, to specify running this handler as the way to deliver the signal.

For more information about defining signal handler functions, see section Defining Signal Handlers.

If you set the action for a signal to `SIG_IGN`, or if you set it to `SIG_DFL` and the default action is to ignore that signal, then any pending signals of that type are discarded (even if they are blocked). Discarding the pending signals means that they will never be delivered, not even if you subsequently specify another action and unblock this kind of signal.

The `signal` function returns the action that was previously in effect for the specified *signum*. You can save this value and restore it later by calling `signal` again.

If `signal` can't honor the request, it returns `SIG_ERR` instead. The following `errno` error conditions are defined for this function:

`EINVAL`

You specified an invalid *signum*; or you tried to ignore or provide a handler for `SIGKILL` or `SIGSTOP`.

Here is a simple example of setting up a handler to delete temporary files when certain fatal signals happen:

```
#include <signal.h>

Void termination_handler (int signum)
{
  struct temp_file *p;

  for (p = temp_file_list; p; p = p->next)
    unlink (p->name);
}

int
main (void)
{
  ...
  if (signal (SIGINT, termination_handler) == SIG_IGN)
    signal (SIGINT, SIG_IGN);
  if (signal (SIGHUP, termination_handler) == SIG_IGN)
    signal (SIGHUP, SIG_IGN);
  if (signal (SIGTERM, termination_handler) == SIG_IGN)
    signal (SIGTERM, SIG_IGN);
  ...
}
```

Note how if a given signal was previously set to be ignored, this code avoids altering that setting. This is because non-job-control shells often ignore certain signals when starting children, and it is important for the children to respect this.

We do not handle `SIGQUIT` or the program error signals in this example because these are designed to provide information for debugging (a core dump), and the temporary files may give useful information.

Function: sighandler_t **ssignal** *(int signum, sighandler_t action)*

The `ssignal` function does the same thing as `signal`; it is provided only for compatibility with SVID.

Macro: sighandler_t **SIG_ERR**

The value of this macro is used as the return value from `signal` to indicate an error.

## Advanced Signal Handling

The `sigaction` function has the same basic effect as `signal`: to specify how a signal should be handled by the process. However, `sigaction` offers more control, at the expense of more complexity. In particular, `sigaction` allows you to specify additional flags to control when the signal is generated and how the handler is invoked.

The `sigaction` function is declared in `` `signal.h' ``.

Data Type: **struct sigaction**

Structures of type `struct sigaction` are used in the `sigaction` function to specify all the information about how to handle a particular signal. This structure contains at least the following members:

`sighandler_t sa_handler`
: This is used in the same way as the *action* argument to the `signal` function. The value can be `SIG_DFL`, `SIG_IGN`, or a function pointer. See section Basic Signal Handling.

`sigset_t sa_mask`
: This specifies a set of signals to be blocked while the handler runs. Blocking is explained in section Blocking Signals for a Handler. Note that the signal that was delivered is automatically blocked by default before its handler is started; this is true regardless of the value in `sa_mask`. If you want that signal not to be blocked within its handler, you must write code in the handler to unblock it.

`int sa_flags`
: This specifies various flags which can affect the behavior of the signal. These are described in more detail in section Flags for `sigaction`.

Function: int **sigaction** *(int signum, const struct sigaction *action, struct sigaction *old_action)*

The *action* argument is used to set up a new action for the signal *signum*, while the *old_action* argument is used to return information about the action previously associated with this symbol. (In other words, *old_action* has the same purpose as the `signal` function's return value--you can check to see what the old action in effect for the signal was, and restore it later if you want.)

Either *action* or *old_action* can be a null pointer. If *old_action* is a null pointer, this simply suppresses the return of information about the old action. If *action* is a null pointer, the action associated with the signal *signum* is unchanged; this allows you to inquire about how a signal is being handled without changing that handling.

The return value from `sigaction` is zero if it succeeds, and `-1` on failure. The following `errno` error conditions are defined for this function:

`EINVAL`
: The *signum* argument is not valid, or you are trying to trap or ignore `SIGKILL` or `SIGSTOP`.

## Interaction of `signal` and `sigaction`

It's possible to use both the `signal` and `sigaction` functions within a single program, but you have to be careful because they can interact in slightly strange ways.

The `sigaction` function specifies more information than the `signal` function, so the return value from `signal` cannot express the full range of `sigaction` possibilities. Therefore, if you use `signal` to save and later reestablish an action, it may not be able to reestablish properly a handler that was established with `sigaction`.

To avoid having problems as a result, always use `sigaction` to save and restore a handler if your program uses `sigaction` at all. Since `sigaction` is more general, it can properly save and reestablish any action, regardless of whether it was established originally with `signal` or `sigaction`.

If you establish an action with `signal` and then examine it with `sigaction`, the handler address that you get may not be the same as what you specified with `signal`. It may not even be suitable for use as an action argument with `signal`. But you can rely on using it as an argument to `sigaction`.

So, you're better off using one or the other of the mechanisms consistently within a single program.

**Portability Note:** The basic `signal` function is a feature of ANSI C, while `sigaction` is part of the POSIX.1 standard. If you are concerned about portability to non-POSIX systems, then you should use the `signal` function instead.

## `sigaction` Function Example

In section Basic Signal Handling, we gave an example of establishing a simple handler for termination signals using `signal`. Here is an equivalent example using `sigaction`:

```
#include <signal.h>

Void termination_handler (int signum)
{
  struct temp_file *p;

  for (p = temp_file_list; p; p = p->next)
    unlink (p->name);
}

Int main (void)
{
  ...
  struct sigaction new_action, old_action;

  /* Set up the structure to specify the new action. */
  new_action.sa_handler = termination_handler;
  sigemptyset (&new_action.sa_mask);
  new_action.sa_flags = 0;

  sigaction (SIGINT, NULL, &old_action);
  if (old_action.sa_handler != SIG_IGN)
    sigaction (SIGINT, &new_action, NULL);
  sigaction (SIGHUP, NULL, &old_action);
  if (old_action.sa_handler != SIG_IGN)
    sigaction (SIGHUP, &new_action, NULL);
  sigaction (SIGTERM, NULL, &old_action);
  if (old_action.sa_handler != SIG_IGN)
    sigaction (SIGTERM, &new_action, NULL);
  ...
}
```

The program just loads the `new_action` structure with the desired parameters and passes it in the `sigaction` call. The usage of `sigemptyset` is described later; see section Blocking Signals.

As in the example using `signal`, we avoid handling signals previously set to be ignored. Here we can avoid altering the signal handler even momentarily, by using the feature of `sigaction` that lets us examine the current action without specifying a new one.

Here is another example. It retrieves information about the current action for `SIGINT` without changing that action.

```
struct sigaction query_action;

if (sigaction (SIGINT, NULL, &query_action) < 0)
  /* sigaction returns -1 in case of error. */
else if (query_action.sa_handler == SIG_DFL)
  /* SIGINT is handled in the default, fatal manner. */
else if (query_action.sa_handler == SIG_IGN)
  /* SIGINT is ignored. */
else
  /* A programmer-defined signal handler is in effect. */
```

## Flags for `sigaction`

The `sa_flags` member of the `sigaction` structure is a catch-all for special features. Most of the time, `SA_RESTART` is a good value to use for this field.

The value of `sa_flags` is interpreted as a bit mask. Thus, you should choose the flags you want to set, OR those flags together, and store the result in the `sa_flags` member of your `sigaction` structure.

Each signal number has its own set of flags. Each call to `sigaction` affects one particular signal number, and the flags that you specify apply only to that particular signal.

In the GNU C library, establishing a handler with `signal` sets all the flags to zero except for `SA_RESTART`, whose value depends on the settings you have made with `siginterrupt`. See section Primitives Interrupted by Signals, to see what this is about.

These macros are defined in the header file `` `signal.h' ``.

Macro: int **SA_NOCLDSTOP**

This flag is meaningful only for the `SIGCHLD` signal. When the flag is set, the system delivers the signal for a terminated child process but not for one that is stopped. By default, `SIGCHLD` is delivered for both terminated children and stopped children.

Setting this flag for a signal other than `SIGCHLD` has no effect.

Macro: int **SA_ONSTACK**

If this flag is set for a particular signal number, the system uses the signal stack when delivering that kind of signal. See section BSD Signal Handling.

Macro: int **SA_RESTART**

This flag controls what happens when a signal is delivered during certain primitives (such as `open`, `read` or `write`), and the signal handler returns normally. There are two alternatives: the library function can resume, or it can return failure with error code `EINTR`.

The choice is controlled by the `SA_RESTART` flag for the particular kind of signal that was delivered. If the flag is set, returning from a handler resumes the library function. If the flag is clear, returning from a handler makes the function fail. See section Primitives Interrupted by Signals.

### Initial Signal Actions

When a new process is created (see section Creating a Process), it inherits handling of signals from its parent process. However, when you load a new process image using the `exec` function (see section Executing a File), any signals that you've defined your own handlers for revert to their `SIG_DFL` handling. (If you think about it a little, this makes sense; the handler functions from the old program are specific to that program, and aren't even present in the address space of the new program image.) Of course, the new program can establish its own handlers.

When a program is run by a shell, the shell normally sets the initial actions for the child process to `SIG_DFL` or `SIG_IGN`, as appropriate. It's a good idea to check to make sure that the shell has not set up an initial action of `SIG_IGN` before you establish your own signal handlers.

Here is an example of how to establish a handler for `SIGHUP`, but not if `SIGHUP` is currently ignored:

```
...
struct sigaction temp;

sigaction (SIGHUP, NULL, &temp);

if (temp.sa_handler != SIG_IGN)
  {
    temp.sa_handler = handle_sighup;
    sigemptyset (&temp.sa_mask);
    sigaction (SIGHUP, &temp, NULL);
  }
```

## Defining Signal Handlers

This section describes how to write a signal handler function that can be established with the `signal` or `sigaction` functions.

A signal handler is just a function that you compile together with the rest of the program. Instead of directly invoking the function, you use `signal` or `sigaction` to tell the operating system to call it when a signal arrives. This is known as *establishing* the handler. See section Specifying Signal Actions.

There are two basic strategies you can use in signal handler functions:

- You can have the handler function note that the signal arrived by tweaking some global data structures, and then return normally.

- You can have the handler function terminate the program or transfer control to a point where it can recover from the situation that caused the signal.

You need to take special care in writing handler functions because they can be called asynchronously. That is, a handler might be called at any point in the program, unpredictably. If two signals arrive during a very short interval, one handler can run within another. This section describes what your handler should do, and what you should avoid.

## Signal Handlers That Return

Handlers which return normally are usually used for signals such as `SIGALRM` and the I/O and interprocess communication signals. But a handler for `SIGINT` might also return normally after setting a flag that tells the program to exit at a convenient time.

It is not safe to return normally from the handler for a program error signal, because the behavior of the program when the handler function returns is not defined after a program error. See section Program Error Signals.

Handlers that return normally must modify some global variable in order to have any effect. Typically, the variable is one that is examined periodically by the program during normal operation. Its data type should be `sig_atomic_t` for reasons described in section Atomic Data Access and Signal Handling.

Here is a simple example of such a program. It executes the body of the loop until it has noticed that a `SIGALRM` signal has arrived. This technique is useful because it allows the iteration in progress when the signal arrives to complete before the loop exits.

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

/* This flag controls termination of the main loop.  */
volatile sig_atomic_t keep_going = 1;

/* The signal handler just clears the flag and re-enables itself.  */
Void catch_alarm (int sig)
{
  keep_going = 0;
  signal (sig, catch_alarm);
}

void
do_stuff (void)
{
  puts ("Doing stuff while waiting for alarm....");
}

int
main (void)
{
  /* Establish a handler for SIGALRM signals.  */
  signal (SIGALRM, catch_alarm);

  /* Set an alarm to go off in a little while.  */
  alarm (2);
  /* Check the flag once in a while to see when to quit.  */
  while (keep_going)
    do_stuff ();

  return EXIT_SUCCESS;
}
```

## Handlers That Terminate the Process

Handler functions that terminate the program are typically used to cause orderly cleanup or recovery from program error signals and interactive interrupts.

The cleanest way for a handler to terminate the process is to raise the same signal that ran the handler in the first place. Here is how to do this:

```
volatile sig_atomic_t fatal_error_in_progress = 0;

void
fatal_error_signal (int sig)
{
  /* Since this handler is established for more than one kind of signal,
     it might still get invoked recursively by delivery of some other kind
     of signal.  Use a static variable to keep track of that. */
  if (fatal_error_in_progress)
    raise (sig);
  fatal_error_in_progress = 1;

  /* Now do the clean up actions:
     - reset terminal modes
     - kill child processes
     - remove lock files */
  ...

  /* Now reraise the signal.  Since the signal is blocked,
     it will receive its default handling, which is
     to terminate the process.  We could just call
     exit or abort, but reraising the signal
     sets the return status from the process correctly. */
  raise (sig);
}
```

## Nonlocal Control Transfer in Handlers

You can do a nonlocal transfer of control out of a signal handler using the `setjmp` and `longjmp` facilities (see section Non-Local Exits).

When the handler does a nonlocal control transfer, the part of the program that was running will not continue. If this part of the program was in the middle of updating an important data structure, the data structure will remain inconsistent. Since the program does not terminate, the inconsistency is likely to be noticed later on.

There are two ways to avoid this problem. One is to block the signal for the parts of the program that update important data structures. Blocking the signal delays its delivery until it is unblocked, once the critical updating is finished. See section Blocking Signals.

The other way to re-initialize the crucial data structures in the signal handler, or make their values consistent.

Here is a rather schematic example showing the reinitialization of one global variable.

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

jmp_buf return_to_top_level;

volatile sig_atomic_t waiting_for_input;

void
handle_sigint (int signum)
{
  /* We may have been waiting for input when the signal arrived,
     but we are no longer waiting once we transfer control. */
  waiting_for_input = 0;
  longjmp (return_to_top_level, 1);
}

int
main (void)
{
  ...
  signal (SIGINT, sigint_handler);
  ...
  while (1) {
    prepare_for_command ();
    if (setjmp (return_to_top_level) == 0)
      read_and_execute_command ();
  }
}

/* Imagine this is a subroutine used by various commands. */
char *
read_data ()
{
  if (input_from_terminal) {
    waiting_for_input = 1;
    ...
    waiting_for_input = 0;
  } else {
    ...
  }
}
```

## Signals Arriving While a Handler Runs

What happens if another signal arrives when your signal handler function is running?

When the handler for a particular signal is invoked, that signal is normally blocked until the handler returns. That means that if two signals of the same kind arrive close together, the second one will be held until the first has been handled. (The handler can explicitly unblock the signal using `sigprocmask`, if you want to allow more signals of this type to arrive; see section Process Signal Mask.)

However, your handler can still be interrupted by delivery of another kind of signal. To avoid this, you can use the `sa_mask` member of the action structure passed to `sigaction` to explicitly specify which signals should be blocked while the signal handler runs. These signals are in addition to the signal for which the handler was invoked, and any other signals that are normally blocked by the process. See section Blocking Signals for a Handler.

**Portability Note:** Always use `sigaction` to establish a handler for a signal that you expect to receive asynchronously, if you want your program to work properly on System V Unix. On this system, the handling of a signal whose handler was established with `signal` automatically sets the signal's action back to `SIG_DFL`, and the handler must re-establish itself each time it runs. This practice, while inconvenient, does work when signals cannot arrive in succession. However, if another signal can arrive right away, it may arrive before the handler can re-establish itself. Then the second signal would receive the default handling, which could terminate the process.

## Signals Close Together Merge into One

If multiple signals of the same type are delivered to your process before your signal handler has a chance to be invoked at all, the handler may only be invoked once, as if only a single signal had arrived. In effect, the signals merge into one. This situation can arise when the signal is blocked, or in a multiprocessing environment where the system is busy running some other processes while the signals are delivered. This means, for example, that you cannot reliably use a signal handler to count signals. The only distinction you can reliably make is whether at least one signal has arrived since a given time in the past.

Here is an example of a handler for `SIGCHLD` that compensates for the fact that the number of signals recieved may not equal the number of child processes generate them. It assumes that the program keeps track of all the child processes with a chain of structures as follows:

```
struct process
{
  struct process *next;
  /* The process ID of this child.  */
  int pid;
  /* The descriptor of the pipe or pseudo terminal
     on which output comes from this child.  */
  int input_descriptor;
  /* Nonzero if this process has stopped or terminated.  */
  sig_atomic_t have_status;
  /* The status of this child; 0 if running,
     otherwise a status value from waitpid.  */
  int status;
};

struct process *process_list;
```

This example also uses a flag to indicate whether signals have arrived since some time in the past--whenever the program last cleared it to zero.

```
/* Nonzero means some child's status has changed
   so look at process_list for the details.  */
int process_status_change;
```

Here is the handler itself:

```
Void sigchld_handler (int signo)
{
  int old_errno = errno;

  while (1) {
    register int pid;
    int w;
    struct process *p;

    /* Keep asking for a status until we get a definitive result.  */
    do
      {
        errno = 0;
        pid = waitpid (WAIT_ANY, &w, WNOHANG | WUNTRACED);
      }
    while (pid <= 0 && errno == EINTR);

    if (pid <= 0) {
      /* A real failure means there are no more
         stopped or terminated child processes, so return.  */
      errno = old_errno;
      return;
    }

    /* Find the process that signaled us, and record its status.  */

    for (p = process_list; p; p = p->next)
      if (p->pid == pid) {
        p->status = w;
        /* Indicate that the status field
           has data to look at.  We do this only after storing it.  */
        p->have_status = 1;

        /* If process has terminated, stop waiting for its output.  */
        if (WIFSIGNALED (w) || WIFEXITED (w))
          if (p->input_descriptor)
            FD_CLR (p->input_descriptor, &input_wait_mask);

        /* The program should check this flag from time to time
           to see if there is any news in process_list.  */
        ++process_status_change;
      }

    /* Loop around to handle all the processes
       that have something to tell us.  */
  }
}
```

Here is the proper way to check the flag `process_status_change`:

```
if (process_status_change) {
  struct process *p;
  process_status_change = 0;
  for (p = process_list; p; p = p->next)
    if (p->have_status) {
      ... Examine p->status ...
    }
}
```

It is vital to clear the flag before examining the list; otherwise, if a signal were delivered just before the clearing of the flag, and after the appropriate element of the process list had been checked, the status change would go unnoticed until the next signal arrived to set the flag again. You could, of course, avoid this problem by blocking the signal while scanning the list, but it is much more elegant to guarantee correctness by doing things in the right order.

The loop which checks process status avoids examining `p->status` until it sees that status has been validly stored. This is to make sure that the status cannot change in the middle of accessing it. Once `p->have_status` is set, it means that the child process is stopped or terminated, and in either case, it cannot stop or terminate again until the program has taken notice. See section Atomic Usage Patterns, for more information about coping with interruptions during accessings of a variable.

Here is another way you can test whether the handler has run since the last time you checked. This technique uses a counter which is never changed outside the handler. Instead of clearing the count, the program remembers the previous value and sees whether it has changed since the previous check. The advantage of this method is that different parts of the program can check independently, each part checking whether there has been a signal since that part last checked.

```
sig_atomic_t process_status_change;

sig_atomic_t last_process_status_change;

...
{
  sig_atomic_t prev = last_process_status_change;
  last_process_status_change = process_status_change;
  if (last_process_status_change != prev) {
    struct process *p;
    for (p = process_list; p; p = p->next)
      if (p->have_status) {
        ... Examine p->status ...
      }
  }
}
```

## Signal Handling and Nonreentrant Functions

Handler functions usually don't do very much. The best practice is to write a handler that does nothing but set an external variable that the program checks regularly, and leave all serious work to the program. This is best because the handler can be called at asynchronously, at unpredictable times--perhaps in the middle of a system call, or even between the beginning and the end of a C operator that requires multiple instructions. The data structures being manipulated might therefore be in an inconsistent state when the handler function is invoked. Even copying one `int` variable into another can take two instructions on most machines.

This means you have to be very careful about what you do in a signal handler.

- If your handler needs to access any global variables from your program, declare those variables `volatile`. This tells the compiler that the value of the variable might change asynchronously, and inhibits certain optimizations that would be invalidated by such modifications.
- If you call a function in the handler, make sure it is *reentrant* with respect to signals, or else make sure that the signal cannot interrupt a call to a related function.

A function can be non-reentrant if it uses memory that is not on the stack.

- If a function uses a static variable or a global variable, or a dynamically-allocated object that it finds for itself, then it is non-reentrant and any two calls to the function can interfere.

  For example, suppose that the signal handler uses `gethostbyname`. This function returns its value in a static object, reusing the same object each time. If the signal happens to arrive during a call to `gethostbyname`, or even after one (while the program is still using the value), it will clobber the value that the program asked for.

  However, if the program does not use `gethostbyname` or any other function that returns information in the same object, or if it always blocks signals around each use, then you are safe.

  There are a large number of library functions that return values in a fixed object, always reusing the same object in this fashion, and all of them cause the same problem. The description of a function in this manual always mentions this behavior.

- If a function uses and modifies an object that you supply, then it is potentially non-reentrant; two calls can interfere if they use the same object.

  This case arises when you do I/O using streams. Suppose that the signal handler prints a message with `fprintf`. Suppose that the program was in the middle of an `fprintf` call using the same stream when the signal was delivered. Both the signal handler's message and the program's data could be corrupted, because both calls operate on the same data structure--the stream itself.

  However, if you know that the stream that the handler uses cannot possibly be used by the program at a time when signals can arrive, then you are safe. It is no problem if the program uses some other stream.

- On most systems, `malloc` and `free` are not reentrant, because they use a static data structure which records what memory blocks are free. As a result, no library functions that allocate or free memory are reentrant. This includes functions that allocate space to store a result.

  The best way to avoid the need to allocate memory in a handler is to allocate in advance space for signal handlers to use.

  The best way to avoid freeing memory in a handler is to flag or record the objects to be freed, and have the program check from time to time whether anything is waiting to be freed. But this must be done with care, because placing an object on a chain is not atomic, and if it is interrupted by another signal handler that does the same thing, you could "lose" one of the objects.

  On the GNU system, `malloc` and `free` are safe to use in signal handlers because it blocks signals. As a result, the library functions that allocate space for a result are also safe in signal handlers. The obstack allocation functions are safe as long as you don't use the same obstack both inside and outside of a signal handler.

  The relocating allocation functions (see section Relocating Allocator) are certainly not safe to use in a signal handler.

- Any function that modifies `errno` is non-reentrant, but you can correct for this: in the handler, save the original value of `errno` and restore it before returning normally. This prevents errors that occur within the signal handler from being confused with errors from system calls at the point the program is interrupted to run the handler.

  This technique is generally applicable; if you want to call in a handler a function that modifies a particular object in memory, you can make this safe by saving and restoring that object.

- Merely reading from a memory object is safe provided that you can deal with any of the values that might appear in the object at a time when the signal can be delivered. Keep in mind that assignment to some data types requires more than one instruction, which means that the handler could run "in the middle of" an assignment to the variable if its type is not atomic. See section Atomic Data Access and Signal Handling.
- Merely writing into a memory object is safe as long as a sudden change in the value, at any time when the handler might run, will not disturb anything.

## Atomic Data Access and Signal Handling

Whether the data in your application concerns atoms, or mere text, you have to be careful about the fact that access to a single datum is not necessarily *atomic*. This means that it can take more than one instruction to read or write a single object. In such cases, a signal handler can run in the middle of reading or writing the object.

There are three ways you can cope with this problem. You can use data types that are always accessed atomically; you can carefully arrange that nothing untoward happens if an access is interrupted, or you can block all signals around any access that had better not be interrupted (see section Blocking Signals).

### Example of Problems with Non-Atomic Access

Here is an example which shows what can happen if a signal handler runs in the middle of modifying a variable. (Interrupting the reading of a variable can also lead to paradoxical results, but here we only show writing.)

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

struct two_words { int a, b; } memory;

void handler(int signum)
{
   printf ("%d,%d\n", memory.a, memory.b);
   alarm (1);
}

Int main (void)
{
   static struct two_words zeros = { 0, 0 }, ones = { 1, 1 };
   signal (SIGALRM, handler);
   memory = zeros;
   alarm (1);
   while (1)
     {
       memory = zeros;
       memory = ones;
     }
}
```

This program fills `memory` with zeros, ones, zeros, ones, alternating forever; meanwhile, once per second, the alarm signal handler prints the current contents. (Calling `printf` in the handler is safe in this program because it is certainly not being called outside the handler when the signal happens.)

Clearly, this program can print a pair of zeros or a pair of ones. But that's not all it can do! On most machines, it takes several instructions to store a new value in `memory`, and the value is stored one word at a time. If the signal is delivered in between these instructions, the handler might find that `memory.a` is zero and `memory.b` is one (or vice versa).

On some machines it may be possible to store a new value in `memory` with just one instruction that cannot be interrupted. On these machines, the handler will always print two zeros or two ones.

## Atomic Types

To avoid uncertainty about interrupting access to a variable, you can use a particular data type for which access is always atomic: `sig_atomic_t`. Reading and writing this data type is guaranteed to happen in a single instruction, so there's no way for a handler to run "in the middle" of an access.

The type `sig_atomic_t` is always an integer data type, but which one it is, and how many bits it contains, may vary from machine to machine.

Data Type: **sig_atomic_t**

This is an integer data type. Objects of this type are always accessed atomically.

In practice, you can assume that `int` and other integer types no longer than `int` are atomic. You can also assume that pointer types are atomic; that is very convenient. Both of these are true on all of the machines that the GNU C library supports, and on all POSIX systems we know of.

## Atomic Usage Patterns

Certain patterns of access avoid any problem even if an access is interrupted. For example, a flag which is set by the handler, and tested and cleared by the main program from time to time, is always safe even if access actually requires two instructions. To show that this is so, we must consider each access that could be interrupted, and show that there is no problem if it is interrupted.

An interrupt in the middle of testing the flag is safe because either it's recognized to be nonzero, in which case the precise value doesn't matter, or it will be seen to be nonzero the next time it's tested.

An interrupt in the middle of clearing the flag is no problem because either the value ends up zero, which is what happens if a signal comes in just before the flag is cleared, or the value ends up nonzero, and subsequent events occur as if the signal had come in just after the flag was cleared. As long as the code handles both of these cases properly, it can also handle a signal in the middle of clearing the flag. (This is an example of the sort of reasoning you need to do to figure out whether non-atomic usage is safe.)

Sometimes you can insure uninterrupted access to one object by protecting its use with another object, perhaps one whose type guarantees atomicity. See section Signals Close Together Merge into One, for an example.

## Primitives Interrupted by Signals

A signal can arrive and be handled while an I/O primitive such as `open` or `read` is waiting for an I/O device. If the signal handler returns, the system faces the question: what should happen next?

POSIX specifies one approach: make the primitive fail right away. The error code for this kind of failure is `EINTR`. This is flexible, but usually inconvenient. Typically, POSIX applications that use signal handlers must check for `EINTR` after each library function that can return it, in order to try the call again. Often programmers forget to check, which a common source of error is.

The GNU library provides a convenient way to retry a call after a temporary failure, with the macro `TEMP_FAILURE_RETRY`:

Macro: **TEMP_FAILURE_RETRY** *(expression)*

This macro evaluates *expression* once. If it fails and reports error code `EINTR`, `TEMP_FAILURE_RETRY` evaluates it again, and over and over until the result is not a temporary failure.

The value returned by `TEMP_FAILURE_RETRY` is whatever value *expression* produced.

BSD avoids `EINTR` entirely and provides a more convenient approach: to restart the interrupted primitive, instead of making it fail. If you choose this approach, you need not be concerned with `EINTR`.

You can choose either approach with the GNU library. If you use `sigaction` to establish a signal handler, you can specify how that handler should behave. If you specify the `SA_RESTART` flag, return from that handler will resume a primitive; otherwise, return from that handler will cause `EINTR`. See section Flags for `sigaction`.

Another way to specify the choice is with the `siginterrupt` function. See section POSIX and BSD Signal Facilities.

When you don't specify with `sigaction` or `siginterrupt` what a particular handler should do, it uses a default choice. The default choice in the GNU library depends on the feature test macros you have defined. If you define `_BSD_SOURCE` or `_GNU_SOURCE` before calling `signal`, the default is to resume primitives; otherwise, the default is to make them fail with `EINTR`. (The library contains alternate versions of the `signal` function, and the feature test macros determine which one you really call.) See section Feature Test Macros.

The primitives affected by this issue are `close`, `fcntl` (operation `F_SETLK`), `open`, `read`, `recv`, `recvfrom`, `select`, `send`, `sendto`, `tcdrain`, `waitpid`, `wait`, and `write`.

There is one situation where resumption never happens no matter which choice you make: when a data-transfer function such as `read` or `write` is interrupted by a signal after transferring part of the data. In this case, the function returns the number of bytes already transferred, indicating partial success.

This might at first appear to cause unreliable behavior on record-oriented devices (including datagram sockets; see section Datagram Socket Operations), where splitting one `read` or `write` into two would read or write two records. Actually, there is no problem, because interruption after a partial transfer cannot happen on such devices; they always transfer an entire record in one burst, with no waiting once data transfer has started.

## Generating Signals

Besides signals that are generated as a result of a hardware trap or interrupt, your program can explicitly send signals to itself or to another process.

### Signaling Yourself

A process can send itself a signal with the `raise` function. This function is declared in `` `signal.h' ``.

Function: int **raise** *(int signum)*

The `raise` function sends the signal *signum* to the calling process. It returns zero if successful and a nonzero value if it fails. About the only reason for failure would be if the value of *signum* is invalid.

Function: int **gsignal** *(int signum)*

The `gsignal` function does the same thing as `raise`; it is provided only for compatibility with SVID.

One convenient use for `raise` is to reproduce the default behavior of a signal that you have trapped. For instance, suppose a user of your program types the SUSP character (usually `C-z`; see section Special Characters) to send it an interactive stop stop signal (`SIGTSTP`), and you want to clean up some internal data buffers before stopping. You might set this up like this:

```
#include <signal.h>

/* When a stop signal arrives, set the action back to the default
   and then resend the signal after doing cleanup actions. */

void tstp_handler (int sig)
{
  signal (SIGTSTP, SIG_DFL);
  /* Do cleanup actions here. */
  ...
  raise (SIGTSTP);
}

/* When the process is continued again, restore the signal handler. */

Void cont_handler (int sig)
{
  signal (SIGCONT, cont_handler);
  signal (SIGTSTP, tstp_handler);
}

/* Enable both handlers during program initialization. */

Int main (void)
{
  signal (SIGCONT, cont_handler);
  signal (SIGTSTP, tstp_handler);
  ...
}
```

**Portability note:** `raise` was invented by the ANSI C committee. Older systems may not support it, so using `kill` may be more portable. See section Signaling Another Process.

## Signaling Another Process

The `kill` function can be used to send a signal to another process. In spite of its name, it can be used for a lot of things other than causing a process to terminate. Some examples of situations where you might want to send signals between processes are:

- A parent process starts a child to perform a task--perhaps having the child running an infinite loop--and then terminates the child when the task is no longer needed.
- A process executes as part of a group, and needs to terminate or notify the other processes in the group when an error or other event occurs.
- Two processes need to synchronize while working together.

This section assumes that you know a little bit about how processes work. For more information on this subject, see section Child Processes.

The `kill` function is declared in `` `signal.h' ``.

Function: int **kill** *(pid_t pid, int signum)*

The `kill` function sends the signal *signum* to the process or process group specified by *pid*. Besides the signals listed in section Standard Signals, *signum* can also have a value of zero to check the validity of the *pid*.

The *pid* specifies the process or process group to receive the signal:

*pid* `> 0`
 The process whose identifier is *pid*.

*pid* `== 0`
 All processes in the same process group as the sender. The sender itself does not receive the signal.

*pid* `< -1`
 The process group whose identifier is -*pid*.

*pid* `== -1`
 If the process is privileged, send the signal to all processes except for some special system processes. Otherwise, send the signal to all processes with the same effective user ID.

A process can send a signal to itself with `kill (getpid(),` *signum*`);`. If `kill` is used by a process to send a signal to itself, and the signal is not blocked, then `kill` delivers at least one signal (which might be some other pending unblocked signal instead of the signal *signum*) to that process before it returns.

The return value from `kill` is zero if the signal can be sent successfully. Otherwise, no signal is sent, and a value of `-1` is returned. If *pid* specifies sending a signal to several processes, `kill` succeeds if it can send the signal to at least one of them. There's no way you can tell which of the processes got the signal or whether all of them did.

The following `errno` error conditions are defined for this function:

`EINVAL`

The *signum* argument is an invalid or unsupported number.

`EPERM`

You do not have the privilege to send a signal to the process or any of the processes in the process group named by *pid*.

`ESCRH`

The *pid* argument does not refer to an existing process or group.

Function: int **killpg** *(int pgid, int signum)*

This is similar to `kill`, but sends signal *signum* to the process group *pgid*. This function is provided for compatibility with BSD; using `kill` to do this is more portable.

As a simple example of `kill`, the call `kill (getpid (), sig)` has the same effect as `raise (sig)`.

## Permission for using `kill`

There are restrictions that prevent you from using `kill` to send signals to any random process. These are intended to prevent antisocial behavior such as arbitrarily killing off processes belonging to another user. In typical use, `kill` is used to pass signals between parent, child, and sibling processes, and in these situations you normally do have permission to send signals. The only common execption is when you run a setuid program in a child process; if the program changes its real UID as well as its effective UID, you may not have permission to send a signal. The `su` program does this.

Whether a process has permission to send a signal to another process is determined by the user IDs of the two processes. This concept is discussed in detail in section The Persona of a Process.

Generally, for a process to be able to send a signal to another process, either the sending process must belong to a privileged user (like `` `root' ``), or the real or effective user ID of the sending process must match the real or effective user ID of the receiving process. If the receiving process has changed its effective user ID from the set-user-ID mode bit on its process image file, then the owner of the process image file is used in place of its current effective user ID. In some implementations, a parent process might be able to send signals to a child process even if the user ID's don't match, and other implementations might enforce other restrictions.

The `SIGCONT` signal is a special case. It can be sent if the sender is part of the same session as the receiver, regardless of user IDs.

## Using `kill` for Communication

Here is a longer example showing how signals can be used for interprocess communication. This is what the `SIGUSR1` and `SIGUSR2` signals are provided for. Since these signals are fatal by default, the process that is supposed to receive them must trap them through `signal` or `sigaction`.

In this example, a parent process forks a child process and then waits for the child to complete its initialization. The child process tells the parent when it is ready by sending it a `SIGUSR1` signal, using the `kill` function.

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

```
/* When a SIGUSR1 signal arrives, set this variable.   */
volatile sig_atomic_t usr_interrupt = 0;

void synch_signal (int sig)
{
  usr_interrupt = 1;
}

/* The child process executes this function.  */
void child_function (void)
{
  /* Perform initialization.  */
  printf ("I'm here!!!  My pid is %d.\n", (int) getpid ());

  /* Let parent know you're done.  */
  kill (getppid (), SIGUSR1);

  /* Continue with execution.  */
  puts ("Bye, now....");
  exit (0);
}

int main (void)
{
  struct sigaction usr_action;
  sigset_t block_mask;
  pid_t child_id;

  /* Establish the signal handler.  */
  sigfillset (&block_mask);
  usr_action.sa_handler = synch_signal;
  usr_action.sa_mask = block_mask;
  usr_action.sa_flags = 0;
  sigaction (SIGUSR1, &usr_action, NULL);

  /* Create the child process.  */
  child_id = fork ();
  if (child_id == 0)
    child_function ();         /* Does not return.   */

  /* Busy wait for the child to send a signal.  */
  while (!usr_interrupt) ;

  /* Now continue execution.  */
  puts ("That's all, folks!");

  return 0;
}
```

This example uses a busy wait, which is bad, because it wastes CPU cycles that other programs could otherwise use. It is better to ask the system to wait until the signal arrives. See the example in section Waiting for a Signal.

## Blocking Signals

Blocking a signal means telling the operating system to hold it and deliver it later. Generally, a program does not block signals indefinitely--it might as well ignore them by setting their actions to `SIG_IGN`. But it is useful to block signals briefly, to prevent them from interrupting sensitive operations. For instance:

- You can use the `sigprocmask` function to block signals while you modify global variables that are also modified by the handlers for these signals.
- You can set `sa_mask` in your `sigaction` call to block certain signals while a particular signal handler runs. This way, the signal handler can run without being interrupted itself by signals.

## Why Blocking Signals is Useful

Temporary blocking of signals with `sigprocmask` gives you a way to prevent interrupts during critical parts of your code. If signals arrive in that part of the program, they are delivered later, after you unblock them.

One example where this is useful is for sharing data between a signal handler and the rest of the program. If the type of the data is not `sig_atomic_t` (see section Atomic Data Access and Signal Handling), then the signal handler could run when the rest of the program has only half finished reading or writing the data. This would lead to confusing consequences.

To make the program reliable, you can prevent the signal handler from running while the rest of the program is examining or modifying that data--by blocking the appropriate signal around the parts of the program that touch the data.

Blocking signals is also necessary when you want to perform a certain action only if a signal has not arrived. Suppose that the handler for the signal sets a flag of type `sig_atomic_t`; you would like to test the flag and perform the action if the flag is not set. This is unreliable. Suppose the signal is delivered immediately after you test the flag, but before the consequent action: then the program will perform the action even though the signal has arrived.

The only way to test reliably for whether a signal has yet arrived is to test while the signal is blocked.

## Signal Sets

All of the signal blocking functions use a data structure called a *signal set* to specify what signals are affected. Thus, every activity involves two stages: creating the signal set, and then passing it as an argument to a library function.

These facilities are declared in the header file `` `signal.h' ``.

Data Type: **sigset_t**

The `sigset_t` data type is used to represent a signal set. Internally, it may be implemented as either an integer or structure type.

For portability, use only the functions described in this section to initialize, change, and retrieve information from `sigset_t` objects--don't try to manipulate them directly.

There are two ways to initialize a signal set. You can initially specify it to be empty with `sigemptyset` and then add specified signals individually. Or you can specify it to be full with `sigfillset` and then delete specified signals individually.

You must always initialize the signal set with one of these two functions before using it in any other way. Don't try to set all the signals explicitly because the `sigset_t` object might include some other information (like a version field) that needs to be initialized as well. (In addition, it's not wise to put into your program an assumption that the system has no signals aside from the ones you know about.)

Function: int **sigemptyset** *(sigset_t *set)*

This function initializes the signal set *set* to exclude all of the defined signals. It always returns `0`.

Function: int **sigfillset** *(sigset_t *set)*

This function initializes the signal set *set* to include all of the defined signals. Again, the return value is `0`.

Function: int **sigaddset** *(sigset_t *set, int signum)*

This function adds the signal *signum* to the signal set *set*. All `sigaddset` does is modify *set*; it does not block or unblock any signals.

The return value is `0` on success and `-1` on failure. The following `errno` error condition is defined for this function:

`EINVAL`
: The *signum* argument doesn't specify a valid signal.

Function: int **sigdelset** *(sigset_t *set, int signum)*

This function removes the signal *signum* from the signal set *set*. All `sigdelset` does is modify *set*; it does not block or unblock any signals. The return value and error conditions are the same as for `sigaddset`.

Finally, there is a function to test what signals are in a signal set:

Function: int **sigismember** *(const sigset_t *set, int signum)*

The `sigismember` function tests whether the signal *signum* is a member of the signal set *set*. It returns `1` if the signal is in the set, `0` if not, and `-1` if there is an error.

The following `errno` error condition is defined for this function:

`EINVAL`
: The *signum* argument doesn't specify a valid signal.

## Process Signal Mask

The collection of signals that are currently blocked is called the *signal mask*. Each process has its own signal mask. When you create a new process (see section Creating a Process), it inherits its parent's mask. You can block or unblock signals with total flexibility by modifying the signal mask.

The prototype for the `sigprocmask` function is in `` `signal.h' ``.

Function: int **sigprocmask** *(int how, const sigset_t *set, sigset_t *oldset)*

The `sigprocmask` function is used to examine or change the calling process's signal mask. The *how* argument determines how the signal mask is changed, and must be one of the following values:

`SIG_BLOCK`
Block the signals in `set`---add them to the existing mask. In other words, the new mask is the union of the existing mask and *set*.

`SIG_UNBLOCK`
Unblock the signals in *set*---remove them from the existing mask.

`SIG_SETMASK`
Use *set* for the mask; ignore the previous value of the mask.

The last argument, *oldset*, is used to return information about the old process signal mask. If you just want to change the mask without looking at it, pass a null pointer as the *oldset* argument. Similarly, if you want to know what's in the mask without changing it, pass a null pointer for *set* (in this case the *how* argument is not significant). The *oldset* argument is often used to remember the previous signal mask in order to restore it later. (Since the signal mask is inherited over `fork` and `exec` calls, you can't predict what its contents are when your program starts running.)

If invoking `sigprocmask` causes any pending signals to be unblocked, at least one of those signals is delivered to the process before `sigprocmask` returns. The order in which pending signals are delivered is not specified, but you can control the order explicitly by making multiple `sigprockmask` calls to unblock various signals one at a time.

The `sigprocmask` function returns `0` if successful, and `-1` to indicate an error. The following `errno` error conditions are defined for this function:

`EINVAL`
The *how* argument is invalid.

You can't block the `SIGKILL` and `SIGSTOP` signals, but if the signal set includes these, `sigprocmask` just ignores them instead of returning an error status.

Remember, too, that blocking program error signals such as `SIGFPE` leads to undesirable results for signals generated by an actual program error (as opposed to signals sent with `raise` or `kill`). This is because your program may be too broken to be able to continue executing to a point where the signal is unblocked again. See section Program Error Signals.

## Blocking to Test for Delivery of a Signal

Now for a simple example. Suppose you establish a handler for `SIGALRM` signals that sets a flag whenever a signal arrives, and your main program checks this flag from time to time and then resets it. You can prevent additional `SIGALRM` signals from arriving in the meantime by wrapping the critical part of the code with calls to `sigprocmask`, like this:

```
/* This variable is set by the SIGALRM signal handler. */
volatile sig_atomic_t flag = 0;

int main (void)
{
  sigset_t block_alarm;
  ...

  /* Initialize the signal mask. */
  sigemptyset (&block_alarm);
  sigaddset (&block_alarm, SIGALRM);

  while (1)
    {
      /* Check if a signal has arrived; if so, reset the flag. */
      sigprocmask (SIG_BLOCK, &block_alarm, NULL);
      if (flag)
        {
          actions-if-not-arrived
          flag = 0;
        }
      sigprocmask (SIG_UNBLOCK, &block_alarm, NULL);

      ...
    }
}
```

## Blocking Signals for a Handler

When a signal handler is invoked, you usually want it to be able to finish without being interrupted by another signal. From the moment the handler starts until the moment it finishes, you must block signals that might confuse it or corrupt its data.

When a handler function is invoked on a signal, that signal is automatically blocked (in addition to any other signals that are already in the process's signal mask) during the time the handler is running. If you set up a handler for `SIGTSTP`, for instance, then the arrival of that signal forces further `SIGTSTP` signals to wait during the execution of the handler.

However, by default, other kinds of signals are not blocked; they can arrive during handler execution.

The reliable way to block other kinds of signals during the execution of the handler is to use the `sa_mask` member of the `sigaction` structure.

Here is an example:

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

void catch_stop ();

void
install_handler (void)
{
  struct sigaction setup_action;
  sigset_t block_mask;

  sigemptyset (&block_mask);
  /* Block other terminal-generated signals while handler runs. */
  sigaddset (&block_mask, SIGINT);
  sigaddset (&block_mask, SIGQUIT);
  setup_action.sa_handler = catch_stop;
  setup_action.sa_mask = block_mask;
  setup_action.sa_flags = 0;
  sigaction (SIGTSTP, &setup_action, NULL);
}
```

This is more reliable than blocking the other signals explicitly in the code for the handler. If you block signals explicity in the handler, you can't avoid at least a short interval at the beginning of the handler where they are not yet blocked.

You cannot remove signals from the process's current mask using this mechanism. However, you can make calls to `sigprocmask` within your handler to block or unblock signals as you wish.

In any case, when the handler returns, the system restores the mask that was in place before the handler was entered.

## Checking for Pending Signals

You can find out which signals are pending at any time by calling `sigpending`. This function is declared in `` `signal.h' ``.

Function: int **sigpending** *(sigset_t *set)*

The `sigpending` function stores information about pending signals in *set*. If there is a pending signal that is blocked from delivery, then that signal is a member of the returned set. (You can test whether a particular signal is a member of this set using `sigismember`; see section Signal Sets.)

The return value is `0` if successful, and `-1` on failure.

Testing whether a signal is pending is not often useful. Testing when that signal is not blocked is almost certainly bad design.

Here is an example.

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

sigset_t base_mask, waiting_mask;

sigemptyset (&base_mask);
sigaddset (&base_mask, SIGINT);
sigaddset (&base_mask, SIGTSTP);

/* Block user interrupts while doing other processing. */
sigprocmask (SIG_SETMASK, &base_mask, NULL);
...

/* After a while, check to see whether any signals are pending. */
sigpending (&waiting_mask);
if (sigismember (&waiting_mask, SIGINT)) {
  /* User has tried to kill the process. */
}
else if (sigismember (&waiting_mask, SIGTSTP)) {
  /* User has tried to stop the process. */
}
```

Remember that if there is a particular signal pending for your process, additional signals of that same type that arrive in the meantime might be discarded. For example, if a `SIGINT` signal is pending when another `SIGINT` signal arrives, your program will probably only see one of them when you unblock this signal.

**Portability Note:** The `sigpending` function is new in POSIX.1. Older systems have no equivalent facility.

## Remembering a Signal to Act On Later

Instead of blocking a signal using the library facilities, you can get almost the same results by making the handler set a flag to be tested later, when you "unblock". Here is an example:

```
/* If this flag is nonzero, don't handle the signal right away. */
volatile sig_atomic_t signal_pending;

/* This is nonzero if a signal arrived and was not handled. */
volatile sig_atomic_t defer_signal;

void handler (int signum)
{
  if (defer_signal)
    signal_pending = signum;
  else
    ... /* "Really" handle the signal. */
}

...

Void update_mumble (int frob)
{
  /* Prevent signals from having immediate effect. */
  defer_signal++;
  /* Now update mumble, without worrying about interruption. */
  mumble.a = 1;
  mumble.b = hack ();
  mumble.c = frob;
  /* We have updated mumble.  Handle any signal that came in. */
  defer_signal--;
  if (defer_signal == 0 && signal_pending != 0)
    raise (signal_pending);
}
```

Note how the particular signal that arrives is stored in `signal_pending`. That way, we can handle several types of inconvenient signals with the same mechanism.

We increment and decrement `defer_signal` so that nested critical sections will work properly; thus, if `update_mumble` were called with `signal_pending` already nonzero, signals would be deferred not only within `update_mumble`, but also within the caller. This is also why we do not check `signal_pending` if `defer_signal` is still nonzero.

The incrementing and decrementing of `defer_signal` require more than one instruction; it is possible for a signal to happen in the middle. But that does not cause any problem. If the signal happens early enough to see the value from before the increment or decrement, that is equivalent to a signal which came before the beginning of the increment or decrement, which is a case that works properly.

It is absolutely vital to decrement `defer_signal` before testing `signal_pending`, because this avoids a subtle bug. If we did these things in the other order, like this,

```
  if (defer_signal == 1 && signal_pending != 0)
    raise (signal_pending);
  defer_signal--;
```

then a signal arriving in between the `if` statement and the decrement would be effetively "lost" for an indefinite amount of time. The handler would merely set `defer_signal`, but the program having already tested this variable, it would not test the variable again.

Bugs like these are called *timing errors*. They are especially bad because they happen only rarely and are nearly impossible to reproduce. You can't expect to find them with a debugger as you would find a reproducible bug. So it is worth being especially careful to avoid them.

(You would not be tempted to write the code in this order, given the use of `defer_signal` as a counter which must be tested along with `signal_pending`. After all, testing for zero is cleaner than testing for one. But if you did not use `defer_signal` as a counter, and gave it values of zero and one only, then either order might seem equally simple. This is a further advantage of using a counter for `defer_signal`: it will reduce the chance you will write the code in the wrong order and create a subtle bug.)

## Waiting for a Signal

If your program is driven by external events, or uses signals for synchronization, then when it has nothing to do it should probably wait until a signal arrives.

### Using `pause`

The simple way to wait until a signal arrives is to call `pause`. Please read about its disadvantages, in the following section, before you use it.

Function: int **pause** *()*

The `pause` function suspends program execution until a signal arrives whose action is either to execute a handler function, or to terminate the process.

If the signal causes a handler function to be executed, then `pause` returns. This is considered an unsuccessful return (since "successful" behavior would be to suspend the program forever), so the return value is `-1`. Even if you specify that other primitives should resume when a system handler returns (see section Primitives Interrupted by Signals), this has no effect on `pause`; it always fails when a signal is handled.

The following `errno` error conditions are defined for this function:

`EINTR`
The function was interrupted by delivery of a signal.

If the signal causes program termination, `pause` doesn't return (obviously).

The `pause` function is declared in `` `unistd.h' ``.

## Problems with `pause`

The simplicity of `pause` can conceal serious timing errors that can make a program hang mysteriously.

It is safe to use `pause` if the real work of your program is done by the signal handlers themselves, and the "main program" does nothing but call `pause`. Each time a signal is delivered, the handler will do the next batch of work that is to be done, and then return, so that the main loop of the program can call `pause` again.

You can't safely use `pause` to wait until one more signal arrives, and then resume real work. Even if you arrange for the signal handler to cooperate by setting a flag, you still can't use `pause` reliably. Here is an example of this problem:

```
/* usr_interrupt is set by the signal handler.  */
if (!usr_interrupt)
  pause ();

/* Do work once the signal arrives.  */
...
```

This has a bug: the signal could arrive after the variable `usr_interrupt` is checked, but before the call to `pause`. If no further signals arrive, the process would never wake up again.

You can put an upper limit on the excess waiting by using `sleep` in a loop, instead of using `pause`. (See section Sleeping, for more about `sleep`.) Here is what this looks like:

```
/* usr_interrupt is set by the signal handler.
while (!usr_interrupt)
  sleep (1);

/* Do work once the signal arrives.  */
...
```

For some purposes, that is good enough. But with a little more complexity, you can wait reliably until a particular signal handler is run, using `sigsuspend`.

## Using `sigsuspend`

The clean and reliable way to wait for a signal to arrive is to block it and then use `sigsuspend`. By using `sigsuspend` in a loop, you can wait for certain kinds of signals, while letting other kinds of signals be handled by their handlers.

Function: int **sigsuspend** *(const sigset_t *set)*

This function replaces the process's signal mask with *set* and then suspends the process until a signal is delivered whose action is either to terminate the process or invoke a signal handling function. In other words, the program is effectively suspended until one of the signals that is not a member of *set* arrives.

If the process is woken up by deliver of a signal that invokes a handler function, and the handler function returns, then `sigsuspend` also returns.

The mask remains *set* only as long as sigsuspend is waiting. The function sigsuspend always restores the previous signal mask when it returns.

The return value and error conditions are the same as for pause.

With sigsuspend, you can replace the pause or sleep loop in the previous section with something completely reliable:

```
sigset_t mask, oldmask;

...

/* Set up the mask of signals to temporarily block. */
sigemptyset (&mask);
sigaddset (&mask, SIGUSR1);

...

/* Wait for a signal to arrive. */
sigprocmask (SIG_BLOCK, &mask, &oldmask);
while (!usr_interrupt)
  sigsuspend (&oldmask);
sigprocmask (SIG_UNBLOCK, &mask, NULL);
```

This last piece of code is a little tricky. The key point to remember here is that when sigsuspend returns, it resets the process's signal mask to the original value, the value from before the call to sigsuspend---in this case, the SIGUSR1 signal is once again blocked. The second call to sigprocmask is necessary to explicitly unblock this signal.

One other point: you may be wondering why the while loop is necessary at all, since the program is apparently only waiting for one SIGUSR1 signal. The answer is that the mask passed to sigsuspend permits the process to be woken up by the delivery of other kinds of signals, as well--for example, job control signals. If the process is woken up by a signal that doesn't set usr_interrupt, it just suspends itself again until the "right" kind of signal eventually arrives.

This technique takes a few more lines of preparation, but that is needed just once for each kind of wait criterion you want to use. The code that actually waits is just four lines.

## BSD Signal Handling

This section describes alternative signal handling functions derived from BSD Unix. These facilities were an advance, in their time; today, they are mostly obsolete, and supported mainly for compatibility with BSD Unix.

They do provide one feature that is not available through the POSIX functions: You can specify a separate stack for use in certain signal handlers. Using a signal stack is the only way you can handle a signal caused by stack overflow.

### POSIX and BSD Signal Facilities

There are many similarities between the BSD and POSIX signal handling facilities, because the POSIX facilities were inspired by the BSD facilities. Besides having different names for all the functions to avoid conflicts, the main differences between the two are:

- BSD Unix represents signal masks as an `int` bit mask, rather than as a `sigset_t` object.
- The BSD facilities use a different default for whether an interrupted primitive should fail or resume. The POSIX facilities make system calls fail unless you specify that they should resume. With the BSD facility, the default is to make system calls resume unless you say they should fail. See section Primitives Interrupted by Signals.
- BSD Unix has a concept of a *signal stack*. This is an alternate stack that is used during the execution of signal handler functions, instead of its normal execution stack.

The BSD facilities are declared in `` `signal.h' ``.

## BSD Function to Establish a Handler

Data Type: **struct sigvec**

This data type is the BSD equivalent of `struct sigaction` (see section Advanced Signal Handling); it is used to specify signal actions to the `sigvec` function. It contains the following members:

`sighandler_t sv_handler`

This is the handler function.

`int sv_mask`

This is the mask of additional signals to be blocked while the handler function is being called.

`int sv_flags`

This is a bit mask used to specify various flags which affect the behavior of the signal. You can also refer to this field as `sv_onstack`.

These symbolic constants can be used to provide values for the `sv_flags` field of a `sigvec` structure. This field is a bit mask value, so you bitwise-OR the flags of interest to you together.

Macro: int **SV_ONSTACK**

If this bit is set in the `sv_flags` field of a `sigvec` structure, it means to use the signal stack when delivering the signal.

Macro: int **SV_INTERRUPT**

If this bit is set in the `sv_flags` field of a `sigvec` structure, it means that system calls interrupted by this kind of signal should not be restarted if the handler returns; instead, the system calls should return with a `EINTR` error status. See section Primitives Interrupted by Signals.

Macro: int **SV_RESETHAND**

If this bit is set in the `sv_flags` field of a `sigvec` structure, it means to reset the action for the signal back to `SIG_DFL` when the signal is received.

Function: int **sigvec** *(int signum, const struct sigvec *action,struct sigvec *old_action)*

This function is the equivalent of `sigaction` (see section Advanced Signal Handling); it installs the action *action* for the signal *signum*, returning information about the previous action in effect for that signal in *old_action*.

Function: int **siginterrupt** *(int signum, int failflag)*

This function specifies which approach to use when certain primitives are interrupted by handling signal *signum*. If *failflag* is false, signal *signum* restarts primitives. If *failflag* is true, handling *signum* causes these primitives to fail with error code `EINTR`. See section Primitives Interrupted by Signals.

## BSD Functions for Blocking Signals

Macro: int **sigmask** *(int signum)*

This macro returns a signal mask that has the bit for signal *signum* set. You can bitwise-OR the results of several calls to `sigmask` together to specify more than one signal. For example,

```
(sigmask (SIGTSTP) | sigmask (SIGSTOP) | sigmask (SIGTTIN) | sigmask (SIGTTOU))
```

specifies a mask that includes all the job-control stop signals.

Function: int **sigblock** *(int mask)*

This function is the equivalent of `sigprocmask` (see section Process Signal Mask) with a *how* argument of `SIG_BLOCK`: it adds the signals specified by *mask* to the calling process's signal mask. The return value is the previous set of blocked signals.

Function: int **sigsetmask** *(int mask)*

This function is the equivalent of `sigprocmask` (see section Process Signal Mask) with a *how* argument of `SIG_SETMASK`: it sets the calling process's signal mask to *mask*. The return value is the previous set of blocked signals.

Function: int **sigpause** *(int mask)*

This function is the equivalent of `sigsuspend` (see section Waiting for a Signal): it sets the calling process's signal mask to *mask*, and waits for a signal to arrive. On return the previous set of blocked signals is restored.

## Using a Separate Signal Stack

A signal stack is a special area of memory to be used as the execution stack during signal handlers. It should be fairly large, to avoid any danger that it will overflow in turn--we recommend at least 16,000 bytes. You can use `malloc` to allocate the space for the stack. Then call `sigstack` to tell the system to use that space for the signal stack.

You don't need to write signal handlers differently in order to use a signal stack. Switching from one stack to the other happens automatically. However, some debuggers on some machines may get confused if you examine a stack trace while a handler that uses the signal stack is running.

Data Type: **struct sigstack**

This structure describes a signal stack. It contains the following members:

`void *ss_sp`
This is the stack pointer.

`int ss_onstack`
This field is true if the process is currently using this stack.

Function: int **sigstack** *(const struct sigstack *stack, struct sigstack *oldstack)*

The `sigstack` function specifies an alternate stack for use during signal handling. When a signal is received by the process and its action indicates that the signal stack is used, the system arranges a switch to the currently installed signal stack while the handler for that signal is executed.

If *oldstack* is not a null pointer, information about the currently installed signal stack is returned in the location it points to. If *stack* is not a null pointer, then this is installed as the new stack for use by signal handlers.

The return value is `0` on success and `1` on failure.

Go to the previous, next section.