PROLOG

This manual page is part of the POSIX Programmer's Manual. The Cygwin implementation of this interface may differ (consult the corresponding Cygwin manual page for details of Cygwin behavior), or the interface may not be implemented on Cygwin.

NAME

sigaction — examine and change a signal action

SYNOPSIS

#include <signal.h>

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

DESCRIPTION

The *sigaction*() function allows the calling process to examine and/or specify the action to be associated with a specific signal. The argument *sig* specifies the signal; acceptable values are defined in <*signal.h*>.

The structure **sigaction**, used to describe an action to be taken, is defined in the <*signal.h*> header to include at least the following members:

Member Type	Member Name	Description
void(*) (int)	*sa_handler*	Pointer to a signal-catching function or one of the macros SIG_IGN or SIG_DFL.
sigset_t	*sa_mask*	Additional set of signals to be blocked during execution of signal-catching function.
int	*sa_flags*	Special flags to affect behavior of signal.
void(*) (int, siginfo_t *, void *)	*sa_sigaction*	Pointer to a signal-catching function.

The storage occupied by *sa_handler* and *sa_sigaction* may overlap, and a conforming application shall not use both simultaneously.

If the argument *act* is not a null pointer, it points to a structure specifying the action to be associated with the specified signal. If the argument *oact* is not a null pointer, the action previously associated with the signal is stored in the location pointed to by the argument *oact*. If the argument *act* is a null pointer, signal handling is unchanged; thus, the call can be used to enquire about the current handling of a given signal. The SIGKILL and SIGSTOP signals shall not be added to the signal mask using this mechanism; this restriction shall be enforced by the system without causing an error to be indicated.

If the SA_SIGINFO flag (see below) is cleared in the *sa_flags* field of the **sigaction** structure, the *sa_handler* field identifies the action to be associated with the specified signal. If the SA_SIGINFO flag is set in the *sa_flags* field, the *sa_sigaction* field specifies a signal-catching function.

The *sa_flags* field can be used to modify the behavior of the specified signal.

The following flags, defined in the <*signal.h*> header, can be set in *sa_flags*:

SA_NOCLDSTOP
: Do not generate SIGCHLD when children stop or stopped children continue.

  If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is not set in *sa_flags*, and the implementation supports the SIGCHLD signal, then a SIGCHLD signal shall be generated for the calling process whenever any of its child processes stop and a SIGCHLD signal may be generated for the calling process whenever any of its stopped child processes are continued. If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is set in *sa_flags*, then the implementation shall not generate a SIGCHLD signal in this way.

SA_ONSTACK
: If set and an alternate signal stack has been declared with *sigaltstack*(), the signal shall be delivered to the calling process on that stack. Otherwise, the signal shall be delivered on the current stack.

SA_RESETHAND

If set, the disposition of the signal shall be reset to SIG_DFL and the SA_SIGINFO flag shall be cleared on entry to the signal handler.

Note: SIGILL and SIGTRAP cannot be automatically reset when delivered; the system silently enforces this restriction.

Otherwise, the disposition of the signal shall not be modified on entry to the signal handler.

In addition, if this flag is set, *sigaction*() may behave as if the SA_NODEFER flag were also set.

SA_RESTART This flag affects the behavior of interruptible functions; that is, those specified to fail with *errno* set to **[EINTR]**. If set, and a function specified as interruptible is interrupted by this signal, the function shall restart and shall not fail with **[EINTR]** unless otherwise specified. If an interruptible function which uses a timeout is restarted, the duration of the timeout following the restart is set to an unspecified value that does not exceed the original timeout value. If the flag is not set, interruptible functions interrupted by this signal shall fail with *errno* set to **[EINTR]**.

SA_SIGINFO If cleared and the signal is caught, the signal-catching function shall be entered as:

void func(int *signo***);**

where *signo* is the only argument to the signal-catching function. In this case, the application shall use the *sa_handler* member to describe the signal-catching function and the application shall not modify the *sa_sigaction* member.

If SA_SIGINFO is set and the signal is caught, the signal-catching function shall be entered as:

void func(int *signo***, siginfo_t ****info***, void ****context***);**

where two additional arguments are passed to the signal-catching function. The second argument shall point to an object of type **siginfo_t** explaining the reason why the signal was generated; the third argument can be cast to a pointer to an object of type **ucontext_t** to refer to the receiving thread's context that was interrupted when the signal was delivered. In this case, the application shall use the *sa_sigaction* member to describe the signal-catching function and the application shall not modify the *sa_handler* member.

The *si_signo* member contains the system-generated signal number.

The *si_errno* member may contain implementation-defined additional error information; if non-zero, it contains an error number identifying the condition that caused the signal to be generated.

The *si_code* member contains a code identifying the cause of the signal, as described in *Section 2.4.3*, *Signal Actions*.

SA_NOCLDWAIT

If set, and *sig* equals SIGCHLD, child processes of the calling processes shall not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited-for children that were transformed into zombie processes, it shall block until all of its children terminate, and *wait*(), *waitid*(), and *waitpid*() shall fail and set *errno* to **[ECHILD]**. Otherwise, terminating child processes shall be transformed into zombie processes, unless SIGCHLD is set to SIG_IGN.

SA_NODEFER If set and *sig* is caught, *sig* shall not be added to the thread's signal mask on entry to the signal handler unless it is included in *sa_mask*. Otherwise, *sig* shall always be added to the thread's signal mask on entry to the signal handler.

When a signal is caught by a signal-catching function installed by *sigaction*(), a new signal mask is calculated and installed for the duration of the signal-catching function (or until a call to either *sigprocmask*() or *sigsuspend*() is made). This mask is formed by taking the union of the current signal mask and the value of the *sa_mask* for the signal being delivered, and unless SA_NODEFER or SA_RESETHAND is set, then including the signal being delivered. If and when the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it shall remain installed until another action is explicitly requested (by another call to *sigaction*()), until the SA_RESETHAND flag causes resetting of the handler, or until one of the *exec* functions is called.

If the previous action for *sig* had been established by *signal*(), the values of the fields returned in the structure pointed to by *oact* are unspecified, and in particular *oact->sa_handler* is not necessarily the same value passed to *signal*(). However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to *sigaction*() via the *act* argument, handling of the signal shall be as if the original call to *signal*() were repeated.

If *sigaction*() fails, no new signal handler is installed.

It is unspecified whether an attempt to set the action for a signal that cannot be caught or ignored to SIG_DFL is ignored or causes an error to be returned with *errno* set to **[EINVAL]**.

If SA_SIGINFO is not set in *sa_flags*, then the disposition of subsequent occurrences of *sig* when it is already pending is implementation-defined; the signal-catching function shall be invoked with a single argument. If SA_SIGINFO is set in *sa_flags*, then subsequent occurrences of *sig* generated by *sigqueue*() or as a result of any signal-generating function that supports the specification of an application-defined value (when *sig* is already pending) shall be queued in FIFO order until delivered or accepted; the signal-catching function shall be invoked with three arguments. The application specified value is passed to the signal-catching function as the *si_value* member of the **siginfo_t** structure.

The result of the use of *sigaction*() and a *sigwait*() function concurrently within a process on the same signal is unspecified.

RETURN VALUE

Upon successful completion, *sigaction*() shall return 0; otherwise, −1 shall be returned, *errno* shall be set to indicate the error, and no new signal-catching function shall be installed.

ERRORS

The *sigaction*() function shall fail if:

EINVAL
The *sig* argument is not a valid signal number or an attempt is made to catch a signal that cannot be caught or ignore a signal that cannot be ignored.

ENOTSUP
The SA_SIGINFO bit flag is set in the *sa_flags* field of the **sigaction** structure.

The *sigaction*() function may fail if:

EINVAL
An attempt was made to set the action to SIG_DFL for a signal that cannot be caught or ignored (or both).

In addition, the *sigaction*() function may fail if the SA_SIGINFO flag is set in the *sa_flags* field of the **sigaction** structure for a signal not in the range SIGRTMIN to SIGRTMAX.

The following sections are informative.

EXAMPLES

Establishing a Signal Handler

The following example demonstrates the use of *sigaction*() to establish a handler for the SIGINT signal.

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

```
static void handler(int signum)
{
   /* Take appropriate actions for signal delivery */
}

int main()
{
   struct sigaction sa;

   sa.sa_handler = handler;
   sigemptyset(&sa.sa_mask);
   sa.sa_flags = SA_RESTART; /* Restart functions if
                        interrupted by handler */
   if (sigaction(SIGINT, &sa, NULL) == -1)
      /* Handle error */;

   /* Further code */
}
```

APPLICATION USAGE

The *sigaction*() function supersedes the *signal*() function, and should be used in preference. In particular, *sigaction*() and *signal*() should not be used in the same process to control the same signal. The behavior of async-signal-safe functions, as defined in their respective DESCRIPTION sections, is as specified by this volume of POSIX.1-2008, regardless of invocation from a signal-catching function. This is the only intended meaning of the statement that async-signal-safe functions may be used in signal-catching functions without restrictions. Applications must still consider all effects of such functions on such things as data structures, files, and process state. In particular, application developers need to consider the restrictions on interactions when interrupting *sleep*() and interactions among multiple handles for a file description. The fact that any specific function is listed as async-signal-safe does not necessarily mean that invocation of that function from a signal-catching function is recommended.

In order to prevent errors arising from interrupting non-async-signal-safe function calls, applications should protect calls to these functions either by blocking the appropriate signals or through the use of some programmatic semaphore (see *semget*( ), *sem_init*( ), *sem_open*( ), and so on). Note in particular that even the ''safe'' functions may modify *errno*; the signal-catching function, if not executing as an independent thread, should save and restore its value in order to avoid the possibility that delivery of a signal in between an error return from a function that sets *errno* and the subsequent examination of *errno* could result in the signal-catching function changing the value of *errno*. Naturally, the same principles apply to the async-signal-safety of application routines and asynchronous data access. Note that *longjmp*() and *siglongjmp*() are not in the list of async-signal-safe functions. This is because the code executing after *longjmp*() and *siglongjmp*() can call any unsafe functions with the same danger as calling those unsafe functions directly from the signal handler. Applications that use *longjmp*() and *siglongjmp*() from within signal handlers require rigorous protection in order to be portable. Many of the other functions that are excluded from the list are traditionally implemented using either *malloc*() or *free*() functions or the standard I/O library, both of which traditionally use data structures in a non-async-signal-safe manner. Since any combination of different functions using a common data structure can cause async-signal-safety problems, this volume of POSIX.1-2008 does not define the behavior when any unsafe function is called in a signal handler that interrupts an unsafe function.

Usually, the signal is executed on the stack that was in effect before the signal was delivered. An alternate stack may be specified to receive a subset of the signals being caught.

When the signal handler returns, the receiving thread resumes execution at the point it was interrupted unless the signal handler makes other arrangements. If *longjmp*() or *_longjmp*() is used to leave the signal handler, then the signal mask must be explicitly restored.

This volume of POSIX.1-2008 defines the third argument of a signal handling function when SA_SIGINFO is set as a **void *** instead of a **ucontext_t ***, but without requiring type checking. New applications should explicitly cast the third argument of the signal handling function to **ucontext_t ***.

The BSD optional four argument signal handling function is not supported by this volume of POSIX.1-2008. The BSD declaration would be:

```
void handler(int sig, int code, struct sigcontext *scp,
  char *addr);
```

where *sig* is the signal number, *code* is additional information on certain signals, *scp* is a pointer to the **sigcontext** structure, and *addr* is additional address information. Much the same information is available in the objects pointed to by the second argument of the signal handler specified when SA_SIGINFO is set.

Since the *sigaction*() function is allowed but not required to set SA_NODEFER when the application sets the SA_RESETHAND flag, applications which depend on the SA_RESETHAND functionality for the newly installed signal handler must always explicitly set SA_NODEFER when they set SA_RESETHAND in order to be portable.

See also the rationale for Realtime Signal Generation and Delivery in the Rationale (Informative) volume of POSIX.1-2008, *Section B.2.4.2*, *Signal Generation and Delivery*.

RATIONALE

Although this volume of POSIX.1-2008 requires that signals that cannot be ignored shall not be added to the signal mask when a signal-catching function is entered, there is no explicit requirement that subsequent calls to *sigaction*() reflect this in the information returned in the *oact* argument. In other words, if SIGKILL is included in the *sa_mask* field of *act*, it is unspecified whether or not a subsequent call to *sigaction*() returns with SIGKILL included in the *sa_mask* field of *oact*.

The SA_NOCLDSTOP flag, when supplied in the *act->sa_flags* parameter, allows overloading SIGCHLD with the System V semantics that each SIGCLD signal indicates a single terminated child. Most conforming applications that catch SIGCHLD are expected to install signal-catching functions that repeatedly call the *waitpid*() function with the WNOHANG flag set, acting on each child for which status is returned, until *waitpid*() returns zero. If stopped children are not of interest, the use of the SA_NOCLDSTOP flag can prevent the overhead from invoking the signal-catching routine when they stop.

Some historical implementations also define other mechanisms for stopping processes, such as the *ptrace*() function. These implementations usually do not generate a SIGCHLD signal when processes stop due to this mechanism; however, that is beyond the scope of this volume of POSIX.1-2008.

This volume of POSIX.1-2008 requires that calls to *sigaction*() that supply a NULL *act* argument succeed, even in the case of signals that cannot be caught or ignored (that is, SIGKILL or SIGSTOP). The System V *signal*() and BSD *sigvec*() functions return **[EINVAL]** in these cases and, in this respect, their behavior varies from *sigaction*().

This volume of POSIX.1-2008 requires that *sigaction*() properly save and restore a signal action set up by the ISO C standard *signal*() function. However, there is no guarantee that the reverse is true, nor could there be given the greater amount of information conveyed by the **sigaction** structure. Because of this, applications should avoid using both functions for the same signal in the same process. Since this cannot always be avoided in case of general-purpose library routines, they should always be implemented with *sigaction*().

It was intended that the *signal*() function should be implementable as a library routine using *sigaction*().

The POSIX Realtime Extension extends the *sigaction*() function as specified by the POSIX.1-1990 standard to allow the application to request on a per-signal basis via an additional signal action flag that the extra parameters, including the application-defined signal value, if any, be passed to the signal-catching function.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4, *Signal Concepts*, *exec*, *kill*( ), *_longjmp*( ), *longjmp*( ), *pthread_sigmask*( ), *raise*( ), *semget*( ), *sem_init*( ), *sem_open*( ), *sigaddset*( ), *sigaltstack*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ), *signal*( ), *sigsuspend*( ), *wait*( ), *waitid*( )

The Base Definitions volume of POSIX.1-2008, **<signal.h>**

COPYRIGHT