**sigaction() – Examine or change a signal action**

**Format**

#define \_POSIX\_SOURCE

#include <signal.h>

int sigaction(int *sig*,

const struct sigaction \*\_\_restrict\_\_ *new*,

struct sigaction \*\_\_restrict\_\_ *old*);

**General description**

Examines and changes the action associated with a specific signal.

int *sig* is the number of a recognized signal. sigaction() examines and sets the action to be associated with this signal. See [Table 1](https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.bpxbd00/rtsigac.htm?view=kc#rtsigac__trs1) for the values of *sig*, as well as the signals supported by z/OS® UNIX services. The *sig* argument must be one of the macros defined in the signal.h header file.

const struct sigaction *\*new* may be a NULL pointer. If so, sigaction() merely determines the action currently defined to handle *sig*. It does not change this action. If *new* is not NULL, it should point to a sigaction structure. The action specified in this structure becomes the new action associated with *sig*.

struct sigaction *\*old* points to a memory location where sigaction() can store a sigaction structure. sigaction() uses this memory location to store a sigaction structure describing the action currently associated with *sig*. *old* can also be a NULL pointer, in which case sigaction() does not store this information.

This function is supported only in a POSIX program.

**Special behavior for C++:**

* The behavior when mixing signal-handling with C++ exception handling is undefined. Also, the use of signal-handling with constructors and destructors is undefined.
* C++ and C language linkage conventions are incompatible, and therefore sigaction() cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to sigaction(), the compiler will flag it as an error. Therefore, to use the sigaction() function in the C++ language, you must ensure that signal handler routines established have C linkage, by declaring them as extern "C".

**Signals:** [Table 1](https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.bpxbd00/rtsigac.htm?view=kc#rtsigac__trs1) lists signal values and their default action and meaning.

| *Table 1. Signal values and signals supported by z/OS UNIX services* | | |
| --- | --- | --- |
| **Value** | **Default Action** | **Meaning** |
| **SIGABND** | 1 | Abend. |
| **SIGABRT** | 1 | Abnormal termination (sent by abort()). |
| **SIGALRM** | 1 | A timeout signal (sent by alarm()). |
| **SIGBUS** | 1 | Bus error (available only when running on MVS™ 5.2 or higher). |
| **SIGFPE** | 1 | Arithmetic exceptions that are not masked, for example, overflow, division by zero, and incorrect operation. |
| **SIGHUP** | 1 | A controlling terminal is suspended, or the controlling process ended. |
| **SIGILL** | 1 | Detection of an incorrect function image. |
| **SIGINT** | 1 | Interactive attention. |
| **SIGKILL** | 1 | A termination signal that cannot be caught or ignored. |
| **SIGPIPE** | 1 | A write to a pipe that is not being read. |
| **SIGPOLL** | 1 | Pollable event occurred (available only when running on MVS 5.2 or higher). |
| **SIGPROF** | 1 | Profiling timer expired (available only when running on MVS 5.2 or higher). |
| **SIGQUIT** | 1 | A quit signal for a terminal. |
| **SIGSEGV** | 1 | Incorrect access to memory. |
| **SIGSYS** | 1 | Bad system call issued (available only when running on MVS 5.2 or higher). |
| **SIGTERM** | 1 | Termination request sent to the program. |
| **SIGTRAP** | 1 | Internal for use by dbx or ptrace. |
| **SIGURG** | 2 | High bandwidth data is available at a socket (available only when running on MVS 5.2 or higher). |
| **SIGUSR1** | 1 | Intended for use by user applications. |
| **SIGUSR2** | 1 | Intended for use by user applications. |
| **SIGVTALRM** | 1 | Virtual timer has expired (available only when running on MVS 5.2 or higher). |
| **SIGXCPU** | 1 | CPU time limit exceeded (available only when running on MVS 5.2 or higher). If a process runs out of CPU time and SIGXCPU is caught or ignored, a SIGKILL is generated. |
| **SIGXFSZ** | 1 | File size limit exceeded. |
| **SIGCHLD** | 2 | An ended or stopped child process (SIGCLD is an alias name for this signal). |
| **SIGIO** | 2 | Completion of input or output. |
| **SIGIOERR** | 2 | A serious I/O error was detected. |
| **SIGWINCH** | 2 | Window size has changed (available only when running on MVS 5.2 or higher). |
| **SIGSTOP** | 3 | A stop signal that cannot be caught or ignored. |
| **SIGTSTP** | 3 | A stop signal for a terminal. |
| **SIGTTIN** | 3 | A background process attempted to read from a controlling terminal. |
| **SIGTTOU** | 3 | A background process attempted to write to a controlling terminal. |
| **SIGCONT** | 4 | If stopped, continue. |

The **Default Actions** in [Table 1](https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.bpxbd00/rtsigac.htm?view=kc#rtsigac__trs1) are:

**1:** Normal termination of the process.

**2:** Ignore the signal.

**3:** Stop the process.

**4:** Continue the process if it is currently stopped. Otherwise, ignore the signal.

If the main program abends in a way that is not caught or handled by the operating system or application, z/OS UNIX terminates the running application with a KILL -9. If z/OS UNIX gets control in EOT or EOM and the terminating status has not been set, z/OS UNIX sets it to appear as if a KILL -9 occurred.

If a signal catcher for a SIGABND, SIGFPE, SIGILL or SIGSEGV signal runs as a result of a program check or an ABEND, and the signal catcher executes a RETURN statement, the process will be terminated.

**sigaction structure:** The sigaction structure is defined as follows:

struct sigaction {

void (\*sa\_handler)(int);

sigset\_t sa\_mask;

int sa\_flags;

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);

};

The following are members of the structure:

**void (\*)(int) sa\_handler**

A pointer to the function assigned to handle the signal. The value of this member can also be SIG\_DFL (indicating the default action) or SIG\_IGN (indicating that the signal is to be ignored).

**Special behavior for XPG4.2:** This member and sa\_sigaction are mutually exclusive of each other. When the SA\_SIGINFO flag is set in sa\_flags then sa\_sigaction is used. Otherwise, sa\_handler is used.

**sigset\_t sa\_mask**

A signal set identifies a set of signals that are to be added to the signal mask of the calling process before the signal-handling function sa\_handler or **sa\_sigaction** (in XPG4.2) is invoked. For more on signal sets, see [sigemptyset() — Initialize a signal mask to exclude all signals](https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.bpxbd00/rtsige.htm?view=kc" \l "rtsige). You cannot use this mechanism to block SIGKILL, SIGSTOP, or SIGTRACE. If sa\_mask includes these signals, they will simply be ignored; sigaction() will not return an error.

sa\_mask must be set by using one or more of the signal set manipulation functions: sigemptyset(), sigfillset(), sigaddset(), or sigdelset()

**int sa\_flags**

A collection of flag bits that affect the behavior of signals. The following flag bits can be set in sa\_flags:

**\_SA\_IGNORE**

This bit is output only and cannot be specified by the application. The handler value will be saved and returned on subsequent calls, but the signal is ignored.

**SA\_NOCLDSTOP**

Tells the system not to issue a SIGCHLD signal when child processes stop. This is relevant only when the *sig* argument of sigaction() is SIGCHLD.

**SA\_NOCLDWAIT**

Tells the system not to create 'zombie' processes when a child process dies. This is relevant only when the *sig* argument of sigaction() is SIGCHLD. If the calling process subsequently waits for its children, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of its children terminate. The wait(), waitid(), or waitpid() will fail and set errno to **ECHILD**.

**SA\_NODEFER**

Tells the system to bypass automatically blocking this signal when invoking a signal handler function.

**\_SA\_OLD\_STYLE**

Tells the C runtime library to use ANSI signal delivery rules, instead of POSIX rules. It is supported for compatibility with applications that use signal() to set signal action. (See [signal() — Handle interrupts](https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.bpxbd00/signal.htm?view=kc#signal).) For a description of ANSI and POSIX.1 signal delivery rules, find “Handling Error Conditions and Signals” in [z/OS XL C/C++ Programming Guide](https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.cbcpx01/toc.htm?view=kc).

**SA\_ONSTACK**

Tells the system to use the alternate signal stack (see [sigaltstack() — Set or get signal alternate stack context](https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.bpxbd00/rsigal.htm?view=kc" \l "rsigal) or [sigstack() — Set or get signal stack context](https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.bpxbd00/rsigst.htm?view=kc" \l "rsigst)) when invoking a signal handler function. If an alternate signal stack has not been declared, the signal handler function will be invoked with the current stack.

**SA\_RESETHAND**

Tells the system to reset the signal's action to SIG\_DFL and clear the SA\_SIGINFO flag before invoking a signal handler function (Note: SIGILL and SIGTRAP cannot be automatically reset when delivered. However, no error will be generated should this situation exist). Otherwise, the disposition of the signal will not be modified on entry to the signal handler.

In addition, if this flag is set, sigaction() behaves as if the SA\_NODEFER flag were also set.

**SA\_RESTART**

Tells the system to restart certain library functions if they should be interrupted by a signal. The functions that this restartability applies to are all of those that are defined as interruptible by signals and set errno to **EINTR** (except pause(), sigpause(), and sigsuspend()).

[Table 2](https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.bpxbd00/rtsigac.htm?view=kc#rtsigac__sigat1) lists functions that are restartable if interrupted by a signal.

|  |  |  |
| --- | --- | --- |
| *Table 2. Functions that are restartable if interrupted by a signal* | | |
| **accept()** | fstatvfs() | recvmsg() |
| **catclose()** | fsync() | select() |
| **catgets()** | ftruncate() | semop() |
| **chmod()** | getgrgid() | send() |
| **chown()** | getgrnam() | sendmsg() |
| **close()** | getmsg() | sendto() |
| **closedir()** | getpass() | statvfs() |
| **connect()** | getpwnam() | tcdrain() |
| **creat()** | getpwuid() | tcflow() |
| **dup2()** | ioctl() | tcflush() |
| **endgrent()** | lchown() | tcgetattr() |
| **fchmod()** | lockf() | tcgetpgrp() |
| **fchown()** | mkfifo() | tcsendbreak() |
| **fclose()** | msgrcv() | tcsetattr() |
| **fcntl()** | msgxrcv() | tcsetpgrp() |
| **fflush()** | msgsnd() | tmpfile() |
| **fgetc()** | open() | umount() |
| **fgetwc()** | poll() | wait() |
| **fopen()** | putmsg() | waitid() |
| **fputc()** | read() | waitpid() |
| **fputwc()** | readv() | write() |
| **freopen()** | recv() |  |
| **fseek()** | recvfrom() |  |

**SA\_SIGINFO**

Tells the system to use the signal action specified by sa\_sigaction instead of sa\_handler.

When this flag is off and the action is to catch the signal, the signal handler function specified by sa\_handler is invoked as:

   void  function(int *signo*);

Where *signo* is the only argument to the signal handler and it specifies the type of signal that has caused the signal handler function to be invoked.

When this flag is on and the action is to catch the signal, the signal handler function specified by sa\_sigaction is invoked as:

   void  function(int *signo*, siginfo\_t \**info*, void \**context*);

Where two additional arguments are passed to the signal handler function. If the second argument is not a NULL pointer, it will point to an object of type siginfo\_t which provides additional information about the source of the signal. A siginfo\_t object is a structure contains the following members:

**si\_signo**

Contains the system-generated signal number

**si\_errno**

Contains the implementation-specific error information (it is not used on this implementation)

**si\_code**

Contains a code identifying the cause of the signal (refer to the <signal.h> include file for a list of these codes and for their meanings, see [Table 1](https://www.ibm.com/support/knowledgecenter/SSLTBW_2.2.0/com.ibm.zos.v2r2.bpxbd00/signal.htm?view=kc#signal__trs2)).

If si\_signo contains SIGPOLL then si\_code can be set to SI\_ASYNCIO. Otherwise, if the value of si\_code is less than or equal to zero then the signal was generated by another process and the si\_pid and si\_uid members respectively indicate the process ID and the real user ID of the sender of this signal.

If the value of si\_code is less than or equal to zero, then the signal was generated by another process and the si\_pid and si\_uid members respectively indicate the process ID and the real user ID of the sender of this signal.

**si\_pid**

If the value of si\_code is less than or equal to zero, then this member will indicate the process ID of the sender of this signal. Otherwise, this member is meaningless.

**si\_uid**

If the value of si\_code is less than or equal to zero, then this member will indicate the real user ID of the sender of this signal. Otherwise, this member is meaningless.

**si\_value**

If si\_code is SI\_ASYNCIO the si\_value contains the application specified value. Otherwise, the contents of si\_value are undefined

The third argument will point to an object of type ucontext\_t (refer to the <ucontext.h> include file for a description of the contents of this object).

**Note:** The remaining flag bits are reserved for system use. There is no guarantee that the integer value of "int sa\_flags" will be the same upon return from sigaction(). However, all flag bits defined above will remain unchanged.

**void (\*)(int, siginfo\_t \*, void \*) sa\_sigaction**

A pointer to the function assigned to handle the signal, or SIG\_DFL, or SIG\_IGN. This function will be invoked passing three parameters. The first is of type 'int' that contains the signal type for which this function is being invoked. The second is of type 'pointer to siginfo\_t' where the siginfo\_t contain additional information about the source of the signal. The third is of type 'pointer to void' but will actually point to a ucontext\_t containing the context information at the time of the signal interrupt.

**Notes:**

1. The user must cast SIG\_IGN or SIG\_DFL to match the sa\_sigaction definition. (indicating that the signal is to be ignored).
2. **Special behavior for XPG4.2:** This member and sa\_handler are mutually exclusive of each other. When the SA\_SIGINFO flag is set in sa\_flags then sa\_sigaction is used. Otherwise, sa\_handler is used.

When a signal handler installed by sigaction(), with the \_SA\_OLD\_STYLE flag set off, catches a signal, the system calculates a new signal mask by taking the union of the current signal mask, the signals specified by sa\_mask, and the signal that was just caught (if the SA\_NODEFER flag is not set). This new mask stays in effect until the signal handler returns, or sigprocmask(), sigsuspend(), siglongjmp(), sighold(), sigpause(), or sigrelse() is called. When the signal handler ends, the original signal mask is restored.

After an action has been specified for a particular signal, using sigaction() or signal(), it remains installed until it is explicitly changed with another call to sigaction(), signal(), one of the exec functions, bsd\_signal(), sigignore(), sigset(), or until the SA\_RESETHAND flag causes it to be reset to SIG\_DFL.

After an action has been specified for a particular signal, using sigaction() with the \_SA\_OLD\_STYLE flag not set, it remains installed until it is explicitly changed with another call to sigaction(), signal(), or one of the exec functions.

After an action has been specified for a particular signal, using sigaction() with the \_SA\_OLD\_STYLE flag set or using signal(), it remains installed until it is explicitly changed with another call to sigaction(), signal(), or one of the exec functions, or a signal catcher is driven, where it will be reset to SIG\_DFL.

Successful setting of signal action to SIG\_IGN for a signal that is pending causes the pending signal to be discarded, whether or not it is blocked. This provides the ability to discard signals that are found to be blocked and pending by sigpending().

**Special behavior for XPG4.2:**

* If a process sets the action of the SIGCHLD signal to SIG\_IGN, child processes of the calling process will 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 will block until all of its children terminate. The wait(), waitid(), or waitpid() function will fail and set errno to **ECHILD**.
* If the **SA\_SIGINFO** flag is set, the signal-catching function specified by sa\_sigaction is invoked as:

       void  *function*(int *signo*, siginfo\_t \**info*, void \**context*);

Where *function* is the specified signal-catching function, *signo* is the signal number of the signal being delivered, *info* points to an object of type siginfo\_t associated with the signal being delivered, and *context* points to an object of type ucontext\_t.

**Considerations for asynchronous signal-catching functions:** Some of the functions have been restricted to be serially reusable with respect to asynchronous signals. That is, the library will not allow an asynchronous signal to interrupt the execution of one of these functions until it has completed.

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

Thus, when you are producing a strictly compliant POSIX C or X/Open application, only the following functions should be assumed to be reentrant with respect to asynchronous signals. Use only these functions in your signal-catching functions:

|  |  |  |
| --- | --- | --- |
| **access()** | alarm() | cfgetispeed() |
| **cfgetospeed()** | cfsetispeed() | cfsetospeed() |
| **chdir()** | chmod() | chown() |
| **close()** | creat() | dup() |
| **dup2()** | execle() | execve() |
| **\_exit()** | fcntl() | fork() |
| **fstat()** | getegid() | geteuid() |
| **getgid()** | getgroups() | getpgrp() |
| **getpid()** | getppid() | getuid() |
| **kill()** | link() | lseek() |
| **mkdir()** | mkfifo() | open() |
| **pathconf()** | pause() | pipe() |
| **pthread\_cond\_broadcast()** | pthread\_cond\_signal() | pthread\_mutex\_trylock() |
| **read()** | rename() | rmdir() |
| **setgid()** | setpgid() | setsid() |
| **setuid()** | sigaction() | sigaddset() |
| **sigdelset()** | sigemptyset() | sigfillset() |
| **sigismember()** | sigpending() | sigprocmask() |
| **sigsuspend()** | sleep() | stat() |
| **sysconf()** | tcdrain() | tcflow() |
| **tcflush()** | tcgetattr() | tcgetpgrp() |
| **tcsendbreak()** | tcsetattr() | tcsetpgrp() |
| **time()** | times() | umask() |
| **uname()** | unlink() | utime() |
| **wait()** | waitpid() | write() |

**Special behavior for XPG4.2:** Adds the following functions to the list of functions above that may be used in signal-catching functions in strictly compliant X/Open applications:

* fpathconf()
* raise()
* signal()

The macro versions of getc() and putc() are not reentrant, even though the library versions of these functions are.

For nonportable POSIX applications, most of the library functions can be used in a signal-catching function. However, do not use the following functions:

* getenv()
* getgrent()
* getgrgid()
* getgrnam()
* getpwent()
* getpwnam()
* getpwuid()
* ttyname()

**Special behavior for XPLINK-compiled C++:** Restrictions concerning setjmp.h and ucontext.h:

1. All XPLINK programs compiled with the V2R10 or later C compilers that are to run with Language Environment V2R10 or later libraries and use the **jmp\_buf**, **sigjmp\_buf** or **ucontext\_t** types must not be compiled with C headers from Language Environment V2R9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp\_buf**, **sigjmp\_buf** or **ucontext\_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), \_longjmp(), setjmp(), \_setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When \_\_XPLINK\_\_ is defined, the Language Environment V2R10 and later headers define a larger **jmp\_buf**, **sigjmp\_buf** or **ucontext\_t** area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If \_\_XPLINK\_\_ is not defined, the Language Environment V2R10 and later headers define a shorter **jmp\_buf**, **sigjmp\_buf** or **ucontext\_t** area. The Language Environment headers before V2R10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp\_buf**, **sigjmp\_buf** or **ucontext\_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.
4. The sigaction() function supersedes the signal() interface, and should be the preferred usage. In particular, sigaction() and signal() must not be used in the same process to control the same signal.

**Usage notes**

The use of the SIGTHSTOP and SIGTHCONT signal is not supported with this function.

**Returned value**

If successful, sigaction() returns 0.

If unsuccessful, no new signal handler is installed, sigaction() returns -1, and sets errno to one of the following values:

**Error Code**

**Description**

**EINVAL**

The value of *sig* is not a valid signal for one of the following reasons:

* The *sig* is not recognized.
* The process tried to ignore a signal that cannot be ignored.
* The process tried to catch a signal that cannot be caught.

The default action for SIGCHILD and SIGIO is for the signal to be ignored. A sigaction() to set the action to SIG\_IGN for SIGIO will result in an error, with errno equal to EINVAL.

**Example**

**CELEBS13**

/\* CELEBS13

The first part of this example determines whether the SIGCHLD

signal is currently being ignored.

With a NULL pointer for the new argument, the current signal

handler action is not changed.

\*/

#define \_POSIX\_SOURCE

#define \_XOPEN\_SOURCE\_EXTENDED 1

#include <stdio.h>

#include <signal.h>

void main(void) {

struct sigaction info;

if (sigaction(SIGCHLD,NULL,&info) != -1)

if (info.sa\_handler == SIG\_IGN)

printf("SIGCHLD being ignored.\n");

else if (info.sa\_handler == SIG\_DFL)

printf("SIGCHLD being defaulted.\n");

}

**CELEBS14**

/\* CELEBS14

This fragment initializes a sigaction structure to specify

mysig as a signal handler and then sets the signal handler

for SIGCHLD.

Information on the previous signal handler for SIGCHLD is

stored in info.

\*/

#define \_XOPEN\_SOURCE\_EXTENDED 1

#include <signal.h>

#include <stdio.h>

void mysig(int a) { printf("In mysig\n"); }

void main(void) {

struct sigaction info, newhandler;

if (sigaction(SIGCHLD,NULL,&info) != -1)

if (info.sa\_handler == SIG\_IGN)

printf("SIGCHLD being ignored.\n");

else if(info.sa\_handler == SIG\_DFL)

printf("SIGCHLD being defaulted.\n");

newhandler.sa\_handler = &mysig;

sigemptyset(&(newhandler.sa\_mask));

newhandler.sa\_flags = 0;

if (sigaction(SIGCHLD,&newhandler,&info) != -1)

printf("New handler set.\n"); }

  Syntax

#include <signal.h>

int sigaction( int sig, const struct sigaction \*act,

struct sigaction \*oact );

  Service Program Name: QP0SSRV1  
  
  Default Public Authority: \*USE  
  
  Threadsafe: Yes

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

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

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

## Authorities and Locks

None.

## Parameters

**sig**

(Input) A signal from the list defined in [Control Signals Table](https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_74/apis/sigactn.htm?view=kc#TBLSIGTBL1).

**\*act**

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

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

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

**\*oact**

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

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

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

struct sigaction {

void (\*sa\_handler)(int);

sigset\_t sa\_mask;

int sa\_flags;

void (\*sa\_sigaction)(int, siginfo\_t \*,void \*);

};

The members of the sigaction structure are as follows:

|  |  |
| --- | --- |
| **Member name** | **Description** |
| void (\*) (int) sa\_handler | A pointer to the function assigned to handle the signal. The value of this member also can be SIG\_DFL (indicating the default action) or SIG\_IGN (indicating that the signal should be ignored). |
| sigset\_t sa\_mask | A signal set (set of signals) to be added to the signal mask of the calling process before the signal-catching function sa\_handler is called. For more information about signal sets, see [sigprocmask()--Examine and Change Blocked Signals](https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_74/apis/sigpmsk.htm?view=kc). You cannot use this mechanism to block the SIGKILL or SIGSTOP signals. If sa\_mask includes these signals, they are ignored and **sigaction()** does not return an error.  sa\_mask must be set by using one or more of the signal set manipulation functions: **sigemptyset()**, **sigfillset()**, **sigaddset()**, or **sigdelset()** |
| int sa\_flags | A collection of flag bits that affect the behavior of signals. The following flag bits can be set in sa\_flags:   |  |  | | --- | --- | | SA\_NOCLDSTOP | If this flag is set, the system does not generate a SIGCHLD signal when child processes stop. This is relevant only when the sig argument of **sigaction()** is SIGCHLD. | | SA\_NODEFER | If this flag is set and sigis caught, sig is not added to the signal mask of the process on entry to the signal catcher unless it is included in **sa\_mask**. If this flag is not set, sig is always added to the signal mask of the process on entry to the signal catcher. This flag is supported for compatibility with applications that use **signal()** to set the signal action. | | SA\_RESETHAND | If this flag is set, the signal-handling action for the signal is reset to SIG\_DFL and the SA\_SIGINFO flag is cleared on entry to the signal-catching function. Otherwise, the signal-handling action is not changed on entry to the signal-catching function. This flag is supported for compatibility with applications that use **signal()** to set the signal action. | | SA\_SIGINFO | If this flag is not set and the signal is caught, the signal-catching function identified by sa\_handler is entered. If this flag is set and the signal is caught, the signal-catching function identified by sa\_sigaction is entered. | |
| void (\*) (int, siginfo\_t \*, void \*) sa\_sigaction | A pointer to the function assigned to handle the signal. If SA\_SIGINFO is set, the signal-catching function identified by sa\_sigaction is entered with additional arguments and sa\_handler is ignored. If SA\_SIGINFO is not set, sa\_sigaction is ignored. If **sig\_action()** is called from a program using data model LLP64, the parameters to sa\_sigaction must be declared as siginfo\_t \*\_\_ptr128 and void \*\_\_ptr128. |

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

* The current signal mask
* The signals specified by sa\_mask
* The signal that was just caught if the SA\_NODEFER flag is set off

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

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

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

|  |  |
| --- | --- |
| **Action** | **Description** |
| SIG\_DFL (signal-specific default action) | * The default actions for the supported signals are specified in [Control Signals Table](https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_74/apis/sigactn.htm?view=kc#TBLSIGTBL1) * If the default action is to stop the process, that process is temporarily suspended. When a process stops, a SIGCHLD signal is generated for its parent process, unless the parent process has set the SA\_NOCLDSTOP flag. While a process is stopped, any additional signals sent to the process are not delivered. The one exception is SIGKILL, which always ends the receiving process. When the process resumes, any unblocked signals that were not delivered are then delivered to it. * If the default action is to ignore the signal, setting a signal action to SIG\_DFL causes any pending signals for that signal to be discarded, whether or not the signal is blocked. |
| SIG\_IGN (ignore signal) | * Delivery of the signal has no effect on the process. The behavior of a process is undefined if it ignores a SIGFPE, SIGILL, or SIGSEGV signal that was not generated by **kill()** or **raise()**. * If the default action is to ignore the signal, setting a signal action to SIG\_DFL causes any pending signals for that signal to be discarded, whether or not the signal is blocked. * The signal action for the signals SIGKILL and SIGSTOP cannot be set to SIG\_IGN. |
| Pointer to a function (catch signal) | * On delivery of the signal, the receiving process runs the signal-catching function. When the signal-catching function returns, the receiving process resumes processing at the point at which it was interrupted. * If SA\_SIGINFO is not set, the signal-catching function identified by sa\_handler is entered as follows: * void func( int signo );   where the following is true:   * + func is the specified signal-catching function.   + signo is the signal number of the signal being delivered. * If SA\_SIGINFO is set, the signal-catching function identified by sa\_sigaction is entered as follows: * void func( int signo, siginfo\_t \*info, void \*context );   where the following is true:   * + func is the specified signal-catching function.   + signo is the signal number of the signal being delivered.   + \*info points to an object of type siginfo\_t associated with the signal being delivered.   + context is set to the NULL pointer. * The behavior of a process is undefined if it returns normally from a signal-catching function for a SIGFPE, SIGILL, or SIGSEGV signal that was not generated by **kill()** or **raise()**. * The signals SIGKILL and SIGSTOP cannot be caught. |

The following is an example of the siginfo\_t structure:

typedef struct siginfo\_t {

int si\_signo; /\* Signal number \*/

int si\_source : 1; /\* Signal source \*/

int reserved1 : 15; /\* Reserved (binary 0) \*/

short si\_data\_size; /\* Size of additional signal

related data (if available) \*/

\_MI\_Time si\_time; /\* Time of signal \*/

struct {

char reserved2[2] /\* Pad (reserved) \*/

char si\_job[10]; /\* Simple job name \*/

char si\_user[10]; /\* User name \*/

char si\_jobno[6]; /\* Job number \*/

char reserved3[4]; /\* Pad (reserved) \*/

} si\_QJN; /\* Qualified job name \*/

int si\_code; /\* Cause of signal \*/

int si\_errno; /\* Error number \*/

pid\_t si\_pid; /\* Process ID of sender \*/

uid\_t si\_uid; /\* Real user ID of sender \*/

char si\_data[1]; /\* Additional signal related

data (if available) \*/

} siginfo\_t;

The members of the siginfo\_t structure are as follows:

|  |  |
| --- | --- |
| int si\_signo | The system-generated signal number. |
| int si\_source | Indicates whether the source of the signal is being generated by the system or another process on the system. When the signal source is another process, the members si\_QJN, si\_pid, and si\_uid contain valid data. When the signal source is the system, those members are set to binary 0. |
| short si\_data\_size | The length of si\_errno, si\_code, si\_pid, si\_uid, and any additional signal-related data. If this member is set to 0, this signal-related information is not available. |
| struct si\_QJN | The fully qualified IBM® i job name of the process sending the signal. |
| int si\_errno | If not zero, this member contains an errno value associated with the signal, as defined in **<errno.h>**. |
| int si\_code | If not zero, this member contains a code identifying the cause of the signal. Possible code values are defined in the **<signal.h>** header file. |
| pidt\_t si\_pid | The process ID of the process sending the signal. |
| uid\_t si\_uid | The real user ID of the process sending the signal. |
| char si\_data[1] | If present, the member contains any additional signal-related data. |

## Control Signals Table

See [Default Actions](https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_74/apis/sigactn.htm?view=kc#DEFAULT) for a description of the value given.

|  |  |  |
| --- | --- | --- |
| **Value** | **Default Action** | **Meaning** |
| SIGABRT | 2 | Abnormal termination |
| SIGFPE | 2 | Arithmetic exceptions that are not masked (for example, overflow, division by zero, and incorrect operation) |
| SIGILL | 2 | Detection of an incorrect function image |
| SIGINT | 2 | Interactive attention |
| SIGSEGV | 2 | Incorrect access to storage |
| SIGTERM | 2 | Termination request sent to the program |
| SIGUSR1 | 2 | Intended for use by user applications |
| SIGUSR2 | 2 | Intended for use by user applications |
| SIGALRM | 2 | A timeout signal (sent by **alarm()**) |
| SIGHUP | 2 | A controlling terminal is hung up, or the controlling process ended |
| SIGKILL | 1 | A termination signal that cannot be caught or ignored |
| SIGPIPE | 3 | A write to a pipe that is not being read |
| SIGQUIT | 2 | A quit signal for a terminal |
| SIGCHLD | 3 | An ended or stopped child process (SIGCLD is an alias name for this signal) |
| SIGCONT | 5 | If stopped, continue |
| SIGSTOP | 4 | A stop signal that cannot be caught or ignored |
| SIGTSTP | 4 | A stop signal for a terminal |
| SIGTTIN | 4 | A background process attempted to read from a controlling terminal |
| SIGTTOU | 4 | A background process attempted to write to a controlling terminal |
| SIGIO | 3 | Completion of input or output |
| SIGURG | 3 | High bandwidth data is available at a socket |
| SIGPOLL | 2 | Pollable event |
| SIGBUS | 2 | Specification exception |
| SIGPRE | 2 | Programming exception |
| SIGSYS | 2 | Bad system call |
| SIGTRAP | 2 | Trace or breakpoint trap |
| SIGPROF | 2 | Profiling timer expired |
| SIGVTALRM | 2 | Virtual timer expired |
| SIGXCPU | 2 | Processor time limit exceeded |
| SIGXFSZ | 2 | File size limit exceeded |
| SIGDANGER | 2 | System crash imminent |
| SIGPCANCEL | 2 | Thread termination signal that cannot be caught or ignored |

### **Default Actions:**

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

## Return Value

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

## Error Conditions

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

[EINVAL]

The value specified for the argument is not correct.

A function was passed incorrect argument values, or an operation was attempted on an object and the operation specified is not supported for that type of object.

An argument value is not valid, out of range, or NULL.

[ENOTSIGINIT]

Process not enabled for signals.

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

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

[ENOTSUP]

Operation not supported.

The operation cannot be performed while running in a system job. An attempt was made to change a signal action while running in a system job.

## Usage Notes

1. When the **sigaction** function is used to change the action associated with a specific signal, it enables a process for signals if the process is not already enabled for signals. For details, see [Qp0sEnableSignals()--Enable Process for Signals](https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_74/apis/sigesig.htm?view=kc). If the system has not been enabled for signals, **sigaction()** is not successful, and an [ENOTSIGINIT] error is returned.
2. The **sigaction()** function can be used to set the action for a particular signal with the same semantics as a call to **signal()**. The sigaction structure indicated by the parameter \*act should contain the following:  
     
   * A sa\_handler equal to the func specified on **signal()**.
   * A sa\_mask containing the signal mask set by **sigemptyset()**.
   * A sa\_flag with the SA\_RESETHAND flag set on.
   * A sa\_flag with the SA\_NODEFER flag set on.
3. Some of the functions have been restricted to be serially reusable with respect to asynchronous signals. That is, the library does not allow an asynchronous signal to interrupt the processing of one of these functions until it has completed.

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

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

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

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