



Full credit is given to the above companies including the Operating System (OS) that this PDF file was generated!

Rocky Enterprise Linux 9.2 Manual Pages on command 'rt_sigaction.2'

\$ man rt_sigaction.2

SIGACTION(2)

Linux Programmer's Manual

SIGACTION(2)

NAME

sigaction, rt_sigaction - examine and change a signal action

SYNOPSIS

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
             struct sigaction *oldact);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

```
sigaction(): _POSIX_C_SOURCE
siginfo_t: _POSIX_C_SOURCE >= 199309L
```

DESCRIPTION

The `sigaction()` system call is used to change the action taken by a process on receipt of a specific signal. (See `signal(7)` for an overview of signals.)

`signum` specifies the signal and can be any valid signal except `SIGKILL` and `SIGSTOP`.

If `act` is non-NULL, the new action for signal `signum` is installed from `act`. If `oldact` is non-NULL, the previous action is saved in `oldact`.

The `sigaction` structure is defined as something like:

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

};

On some architectures a union is involved: do not assign to both `sa_handler` and `sa_sigac?`
tion.

The `sa_restorer` field is not intended for application use. (POSIX does not specify a `sa_restorer` field.) Some further details of the purpose of this field can be found in `si?`
`greturn(2)`.

`sa_handler` specifies the action to be associated with `signum` and is be one of the follow?
ing:

- * `SIG_DFL` for the default action.
- * `SIG_IGN` to ignore this signal.
- * A pointer to a signal handling function. This function receives the signal number as its only argument.

If `SA_SIGINFO` is specified in `sa_flags`, then `sa_sigaction` (instead of `sa_handler`) speci?
fies the signal-handling function for `signum`. This function receives three arguments, as
described below.

`sa_mask` specifies a mask of signals which should be blocked (i.e., added to the signal
mask of the thread in which the signal handler is invoked) during execution of the signal
handler. In addition, the signal which triggered the handler will be blocked, unless the
`SA_NODEFER` flag is used.

`sa_flags` specifies a set of flags which modify the behavior of the signal. It is formed
by the bitwise OR of zero or more of the following:

`SA_NOCLDSTOP`

If `signum` is `SIGCHLD`, do not receive notification when child processes stop (i.e.,
when they receive one of `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, or `SIGTTOU`) or resume (i.e.,
they receive `SIGCONT`) (see `wait(2)`). This flag is meaningful only when establish?
ing a handler for `SIGCHLD`.

`SA_NOCLDWAIT` (since Linux 2.6)

If `signum` is `SIGCHLD`, do not transform children into zombies when they terminate.

See also `waitpid(2)`. This flag is meaningful only when establishing a handler for
`SIGCHLD`, or when setting that signal's disposition to `SIG_DFL`.

If the `SA_NOCLDWAIT` flag is set when establishing a handler for `SIGCHLD`, POSIX.1
leaves it unspecified whether a `SIGCHLD` signal is generated when a child process
terminates. On Linux, a `SIGCHLD` signal is generated in this case; on some other

implementations, it is not.

SA_NODEFER

Do not add the signal to the thread's signal mask while the handler is executing, unless the signal is specified in `act.sa_mask`. Consequently, a further instance of the signal may be delivered to the thread while it is executing the handler. This flag is meaningful only when establishing a signal handler.

`SA_NOMASK` is an obsolete, nonstandard synonym for this flag.

SA_ONSTACK

Call the signal handler on an alternate signal stack provided by `sigaltstack(2)`.

If an alternate stack is not available, the default stack will be used. This flag is meaningful only when establishing a signal handler.

SA_RESETHAND

Restore the signal action to the default upon entry to the signal handler. This flag is meaningful only when establishing a signal handler.

`SA_ONESHOT` is an obsolete, nonstandard synonym for this flag.

SA_RESTART

Provide behavior compatible with BSD signal semantics by making certain system calls restartable across signals. This flag is meaningful only when establishing a signal handler. See `signal(7)` for a discussion of system call restarting.

SA_RESTORER

Not intended for application use. This flag is used by C libraries to indicate that the `sa_restorer` field contains the address of a "signal trampoline". See `si_greturn(2)` for more details.

SA_SIGINFO (since Linux 2.2)

The signal handler takes three arguments, not one. In this case, `sa_sigaction` should be set instead of `sa_handler`. This flag is meaningful only when establishing a signal handler.

The `siginfo_t` argument to a SA_SIGINFO handler

When the `SA_SIGINFO` flag is specified in `act.sa_flags`, the signal handler address is passed via the `act.sa_sigaction` field. This handler takes three arguments, as follows:

void

`handler(int sig, siginfo_t *info, void *ucontext)`

{

...

}

These three arguments are as follows

sig The number of the signal that caused invocation of the handler.

info A pointer to a **siginfo_t**, which is a structure containing further information about the signal, as described below.

ucontext

This is a pointer to a **ucontext_t** structure, cast to **void ***. The structure pointed to by this field contains signal context information that was saved on the user-space stack by the kernel; for details, see **sigreturn(2)**. Further information about the **ucontext_t** structure can be found in **getcontext(3)** and **signal(7)**. Commonly, the handler function doesn't make any use of the third argument.

The **siginfo_t** data type is a structure with the following fields:

```
siginfo_t {  
    int     si_signo;    /* Signal number */  
    int     si_errno;    /* An errno value */  
    int     si_code;    /* Signal code */  
    int     si_trapno;  /* Trap number that caused  
                        hardware-generated signal  
                        (unused on most architectures) */  
    pid_t   si_pid;    /* Sending process ID */  
    uid_t   si_uid;    /* Real user ID of sending process */  
    int     si_status;  /* Exit value or signal */  
    clock_t si_utime;  /* User time consumed */  
    clock_t si_stime;  /* System time consumed */  
    union sigval si_value; /* Signal value */  
    int     si_int;    /* POSIX.1b signal */  
    void   *si_ptr;    /* POSIX.1b signal */  
    int     si_overrun; /* Timer overrun count;  
                        POSIX.1b timers */  
    int     si_timerid; /* Timer ID; POSIX.1b timers */  
    void   *si_addr;    /* Memory location which caused fault */  
    long    si_band;    /* Band event (was int in
```

```

glibc 2.3.2 and earlier) */

int    si_fd;    /* File descriptor */

short  si_addr_lsb; /* Least significant bit of address

                     (since Linux 2.6.32) */

void   *si_lower;  /* Lower bound when address violation

                     occurred (since Linux 3.19) */

void   *si_upper;  /* Upper bound when address violation

                     occurred (since Linux 3.19) */

int    si_pkey;   /* Protection key on PTE that caused

                     fault (since Linux 4.6) */

void   *si_call_addr; /* Address of system call instruction

                     (since Linux 3.5) */

int    si_syscall; /* Number of attempted system call

                     (since Linux 3.5) */

unsigned int si_arch; /* Architecture of attempted system call

                     (since Linux 3.5) */

}

}

```

si_signo, si_errno and si_code are defined for all signals. (si_errno is generally unused on Linux.) The rest of the struct may be a union, so that one should read only the fields that are meaningful for the given signal:

* Signals sent with kill(2) and sigqueue(3) fill in si_pid and si_uid. In addition, sig? nals sent with sigqueue(3) fill in si_int and si_ptr with the values specified by the sender of the signal; see sigqueue(3) for more details.

* Signals sent by POSIX.1b timers (since Linux 2.6) fill in si_overrun and si_timerid. The si_timerid field is an internal ID used by the kernel to identify the timer; it is not the same as the timer ID returned by timer_create(2). The si_overrun field is the timer overrun count; this is the same information as is obtained by a call to timer_getoverrun(2). These fields are nonstandard Linux extensions.

* Signals sent for message queue notification (see the description of SIGEV_SIGNAL in mq_notify(3)) fill in si_int/si_ptr, with the sigev_value supplied to mq_notify(3); si_pid, with the process ID of the message sender; and si_uid, with the real user ID of the message sender.

* SIGCHLD fills in si_pid, si_uid, si_status, si_utime, and si_stime, providing informa?

tion about the child. The `si_pid` field is the process ID of the child; `si_uid` is the child's real user ID. The `si_status` field contains the exit status of the child (if `si_code` is `CLD_EXITED`), or the signal number that caused the process to change state. The `si_utime` and `si_stime` contain the user and system CPU time used by the child process; these fields do not include the times used by waited-for children (unlike `getrusage(2)` and `times(2)`). In kernels up to 2.6, and since 2.6.27, these fields report CPU time in units of `sysconf(_SC_CLK_TCK)`. In 2.6 kernels before 2.6.27, a bug meant that these fields reported time in units of the (configurable) system jiffy (see `time(7)`).

* `SIGILL`, `SIGFPE`, `SIGSEGV`, `SIGBUS`, and `SIGTRAP` fill in `si_addr` with the address of the fault. On some architectures, these signals also fill in the `si_trapno` field.

Some suberrors of `SIGBUS`, in particular `BUS_MCEERR_AO` and `BUS_MCEERR_AR`, also fill in `si_addr_lsb`. This field indicates the least significant bit of the reported address and therefore the extent of the corruption. For example, if a full page was corrupted, `si_addr_lsb` contains $\log_2(\text{sysconf}(\text{SC_PAGESIZE}))$. When `SIGTRAP` is delivered in response to a `ptrace(2)` event (`PTRACE_EVENT_foo`), `si_addr` is not populated, but `si_pid` and `si_uid` are populated with the respective process ID and user ID responsible for delivering the trap. In the case of `seccomp(2)`, the tracee will be shown as delivering the event.

`BUS_MCEERR_*` and `si_addr_lsb` are Linux-specific extensions.

The `SEGV_BNDERR` suberror of `SIGSEGV` populates `si_lower` and `si_upper`.

The `SEGV_PKUERR` suberror of `SIGSEGV` populates `si_pkey`.

* `SIGIO/SIGPOLL` (the two names are synonyms on Linux) fills in `si_band` and `si_fd`. The `si_band` event is a bit mask containing the same values as are filled in the `revents` field by `poll(2)`. The `si_fd` field indicates the file descriptor for which the I/O event occurred; for further details, see the description of `F_SETSIG` in `fcntl(2)`.

* `SIGSYS`, generated (since Linux 3.5) when a seccomp filter returns `SECCOMP_RET_TRAP`, fills in `si_call_addr`, `si_syscall`, `si_arch`, `si_errno`, and other fields as described in `seccomp(2)`.

The `si_code` field

The `si_code` field inside the `siginfo_t` argument that is passed to a `SA_SIGINFO` signal handler is a value (not a bit mask) indicating why this signal was sent. For a `ptrace(2)` event, `si_code` will contain `SIGTRAP` and have the `ptrace` event in the high byte: `(SIGTRAP | PTRACE_EVENT_foo << 8)`.

For a non-ptrace(2) event, the values that can appear in `si_code` are described in the remainder of this section. Since glibc 2.20, the definitions of most of these symbols are obtained from `<signal.h>` by defining feature test macros (before including any header file) as follows:

- * `_XOPEN_SOURCE` with the value 500 or greater;
- * `_XOPEN_SOURCE` and `_XOPEN_SOURCE_EXTENDED`; or
- * `_POSIX_C_SOURCE` with the value 200809L or greater.

For the `TRAP_*` constants, the symbol definitions are provided only in the first two cases.

Before glibc 2.20, no feature test macros were required to obtain these symbols.

For a regular signal, the following list shows the values which can be placed in `si_code` for any signal, along with the reason that the signal was generated.

`SI_USER`

`kill(2)`.

`SI_KERNEL`

Sent by the kernel.

`SI_QUEUE`

`sigqueue(3)`.

`SI_TIMER`

POSIX timer expired.

`SI_MESGQ` (since Linux 2.6.6)

POSIX message queue state changed; see `mq_notify(3)`.

`SI_ASYNCIO`

AIO completed.

`SI_SIGIO`

Queued SIGIO (only in kernels up to Linux 2.2; from Linux 2.4 onward SIGIO/SIG?

POLL fills in `si_code` as described below).

`SI_TKILL` (since Linux 2.4.19)

`tkill(2)` or `tgkill(2)`.

The following values can be placed in `si_code` for a SIGILL signal:

`ILL_ILLOPC`

Illegal opcode.

`ILL_ILLOPN`

Illegal operand.

ILL_ILLADR

Illegal addressing mode.

ILL_ILLTRP

Illegal trap.

ILL_PRVOPC

Privileged opcode.

ILL_PRVREG

Privileged register.

ILL_COPROC

Coprocessor error.

ILL_BADSTK

Internal stack error.

The following values can be placed in `si_code` for a SIGFPE signal:

FPE_INTDIV

Integer divide by zero.

FPE_INTOVF

Integer overflow.

FPE_FLTDIV

Floating-point divide by zero.

FPE_FLTOVF

Floating-point overflow.

FPE_FLTUND

Floating-point underflow.

FPE_FLTRES

Floating-point inexact result.

FPE_FLTINV

Floating-point invalid operation.

FPE_FLTSUB

Subscript out of range.

The following values can be placed in `si_code` for a SIGSEGV signal:

SEGV_MAPERR

Address not mapped to object.

SEGV_ACCERR

Invalid permissions for mapped object.

SEGV_BNDERR (since Linux 3.19)

Failed address bound checks.

SEGV_PKUERR (since Linux 4.6)

Access was denied by memory protection keys. See `pkeys(7)`. The protection key which applied to this access is available via `si_pkey`.

The following values can be placed in `si_code` for a SIGBUS signal:

BUS_ADRALN

Invalid address alignment.

BUS_ADRERR

Nonexistent physical address.

BUS_OBJERR

Object-specific hardware error.

BUS_MCEERR_AR (since Linux 2.6.32)

Hardware memory error consumed on a machine check; action required.

BUS_MCEERR_AO (since Linux 2.6.32)

Hardware memory error detected in process but not consumed; action optional.

The following values can be placed in `si_code` for a SIGTRAP signal:

TRAP_BRKPT

Process breakpoint.

TRAP_TRACE

Process trace trap.

TRAP_BRANCH (since Linux 2.4, IA64 only)

Process taken branch trap.

TRAP_HWBKPT (since Linux 2.4, IA64 only)

Hardware breakpoint/watchpoint.

The following values can be placed in `si_code` for a SIGCHLD signal:

CLD_EXITED

Child has exited.

CLD_KILLED

Child was killed.

CLD_DUMPED

Child terminated abnormally.

CLD_TRAPPED

Traced child has trapped.

CLD_STOPPED

Child has stopped.

CLD_CONTINUED (since Linux 2.6.9)

Stopped child has continued.

The following values can be placed in `si_code` for a SIGIO/SIGPOLL signal:

POLL_IN

Data input available.

POLL_OUT

Output buffers available.

POLL_MSG

Input message available.

POLL_ERR

I/O error.

POLL_PRI

High priority input available.

POLL_HUP

Device disconnected.

The following value can be placed in `si_code` for a SIGSYS signal:

SYS_SECCOMP (since Linux 3.5)

Triggered by a seccomp(2) filter rule.

RETURN VALUE

`sigaction()` returns 0 on success; on error, -1 is returned, and `errno` is set to indicate the error.

ERRORS

EFAULT `act` or `oldact` points to memory which is not a valid part of the process address space.

EINVAL An invalid signal was specified. This will also be generated if an attempt is made to change the action for SIGKILL or SIGSTOP, which cannot be caught or ignored.

CONFORMING TO

POSIX.1-2001, POSIX.1-2008, SVr4.

NOTES

A child created via `fork(2)` inherits a copy of its parent's signal dispositions. During an `execve(2)`, the dispositions of handled signals are reset to the default; the dispositions of ignored signals are left unchanged.

According to POSIX, the behavior of a process is undefined after it ignores a `SIGFPE`, `SIGILL`, or `SIGSEGV` signal that was not generated by `kill(2)` or `raise(3)`. Integer division by zero has undefined result. On some architectures it will generate a `SIGFPE` signal. (Also dividing the most negative integer by `-1` may generate `SIGFPE`.) Ignoring this signal might lead to an endless loop.

POSIX.1-1990 disallowed setting the action for `SIGCHLD` to `SIG_IGN`. POSIX.1-2001 and later allow this possibility, so that ignoring `SIGCHLD` can be used to prevent the creation of zombies (see `wait(2)`). Nevertheless, the historical BSD and System V behaviors for ignoring `SIGCHLD` differ, so that the only completely portable method of ensuring that terminated children do not become zombies is to catch the `SIGCHLD` signal and perform a `wait(2)` or similar.

POSIX.1-1990 specified only `SA_NOCLDSTOP`. POSIX.1-2001 added `SA_NOCLDSTOP`, `SA_NOCLDWAIT`, `SA_NODEFER`, `SA_ONSTACK`, `SA_RESETHAND`, `SA_RESTART`, and `SA_SIGINFO`. Use of these latter values in `sa_flags` may be less portable in applications intended for older UNIX implementations.

The `SA_RESETHAND` flag is compatible with the SVr4 flag of the same name.

The `SA_NODEFER` flag is compatible with the SVr4 flag of the same name under kernels 1.3.9 and later. On older kernels the Linux implementation allowed the receipt of any signal, not just the one we are installing (effectively overriding any `sa_mask` settings). `sigaction()` can be called with a `NULL` second argument to query the current signal handler.

It can also be used to check whether a given signal is valid for the current machine by calling it with `NULL` second and third arguments.

It is not possible to block `SIGKILL` or `SIGSTOP` (by specifying them in `sa_mask`). Attempts to do so are silently ignored.

See `sigsetops(3)` for details on manipulating signal sets.

See `signal-safety(7)` for a list of the async-signal-safe functions that can be safely called inside from inside a signal handler.

C library/kernel differences

The glibc wrapper function for `sigaction()` gives an error (`EINVAL`) on attempts to change the disposition of the two real-time signals used internally by the NPTL threading imple?

mentation. See `nptl(7)` for details.

On architectures where the signal trampoline resides in the C library, the glibc wrapper function for `sigaction()` places the address of the trampoline code in the `act.sa_restorer` field and sets the `SA_RESTORER` flag in the `act.sa_flags` field. See `sigreturn(2)`.

The original Linux system call was named `sigaction()`. However, with the addition of real-time signals in Linux 2.2, the fixed-size, 32-bit `sigset_t` type supported by that system call was no longer fit for purpose. Consequently, a new system call, `rt_sigaction()`, was added to support an enlarged `sigset_t` type. The new system call takes a fourth argument, `size_t sigsetsize`, which specifies the size in bytes of the signal sets in `act.sa_mask` and `oldact.sa_mask`. This argument is currently required to have the value `sizeof(sigset_t)` (or the error `EINVAL` results). The glibc `sigaction()` wrapper function hides these details from us, transparently calling `rt_sigaction()` when the kernel provides it.

Undocumented

Before the introduction of `SA_SIGINFO`, it was also possible to get some additional information about the signal. This was done by providing an `sa_handler` signal handler with a second argument of type `struct sigcontext`, which is the same structure as the one that is passed in the `uc_mcontext` field of the `ucontext` structure that is passed (via a pointer) in the third argument of the `sa_sigaction` handler. See the relevant Linux kernel sources for details. This use is obsolete now.

BUGS

When delivering a signal with a `SA_SIGINFO` handler, the kernel does not always provide meaningful values for all of the fields of the `siginfo_t` that are relevant for that signal.

In kernels up to and including 2.6.13, specifying `SA_NODEFER` in `sa_flags` prevents not only the delivered signal from being masked during execution of the handler, but also the signals specified in `sa_mask`. This bug was fixed in kernel 2.6.14.

EXAMPLES

See `mprotect(2)`.

SEE ALSO

`kill(1)`, `kill(2)`, `pause(2)`, `pidfd_send_signal(2)`, `restart_syscall(2)`, `seccomp(2)`, `sigaltstack(2)`, `signal(2)`, `signalfd(2)`, `sigpending(2)`, `sigprocmask(2)`, `sigreturn(2)`, `sigsuspend(2)`, `wait(2)`, `killpg(3)`, `raise(3)`, `siginterrupt(3)`, `sigqueue(3)`, `sigsetops(3)`, `sigvec(3)`, `core(5)`, `signal(7)`

COLOPHON

This page is part of release 5.10 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at <https://www.kernel.org/doc/man-pages/>.

Linux

2020-12-21

SIGACTION(2)