

N3984: Thread-safe signals handling rev 5

Document #: N3984
Date: 2026-10-11
Project: Programming Language C
Reply-to: Niall Douglas
<s_sourceforge@nedprod.com>

This paper revises [N3924] *Thread-safe signals handling* to incorporate committee feedback. This paper proposes composable signal handling which is thread-safe and optionally thread-local for the standard C library. Its reference implementation library can be found at https://github.com/ned14/wg14_signals, and it has been shipping in production open source libraries since 2025.

The reference implementation library is designed to be easily mergeable into any standard C library as well as used standalone. It requires a C11 compiler to compile, but only a C89 compiler to use. It works on POSIX and Microsoft Windows on at least the x64 and AArch64 architectures (and likely many more). It works well in the Fil-C guaranteed memory safe C compiler.

Performance is reasonable: `sigguarded()` costs 16 to 20 nanoseconds; global signal deciders cost 8 to 20 nanoseconds depending on platform and architecture.

Since the August meeting the reference implementation has been merged as a test into two production C standard library implementations: the GNU C Library (<https://github.com/ned14/glibc>, branch `replace_signals_handling`) and LLVM libc (<https://github.com/ned14/llvm-project>, branch `replace_signals_handling`). No problems were found: the reference library seamlessly extends both standard C library implementations with the proposed APIs with no ABI churn and all existing code using those standard C libraries being unaffected, as all existing signal handling functions work exactly as before.

I then went ahead and ported LLVM over to exclusively use the proposed APIs. This was greatly eased by LLVM already internally implementing a thread-safe, thread-local, composable signals infrastructure which is semantically identical to this proposed one (this isn't surprising: any mature production codebase usually reinvents this proposed API). It was therefore a one-for-one swap. LLVM's test suite passed without issue, and I hacked clang to segfault as a quick smoke test, and it all 'just worked'.

This revision completes merging WG14 committee member feedback from last year as well as Joseph Myers' feedback on N3924. I *think* that the proposed normative wording now correctly refers to the C abstract machine, though I'm sure Joseph will find many places where I have done poorly. I once again thank Joseph for his diligent and detailed work on this proposal, without which we wouldn't have made so many improvements as quickly as we have.

Changes since N3924:

- Rebased onto Working Draft N3886.

- Removed the `_t` suffix from `sig_decision` enum.
- Applied the remainder of the WG14 member feedback outstanding, or at least the low hanging fruit.
- `sigfence()` can now take arguments.
- Rewrote the *async-signal-safe* definition to remove the circularity.
- Added the four kinds of signal handling state (the `signal` handler, thread-local deciders, global deciders, and the activation count per signal number) and specified that activation and deactivation create or remove none of the other kinds of state.
- The signal receipt sequences in the wording are now keyed on whether the signal number of the signal received is activated.
- `raise` and `stdc_raise` are now equivalent wherever either is permitted: the `raise` function is *async-signal-safe*.
- Split `stdc_raise` into `sighandle` (7.14.2.9), which invokes the signal deciders directly, and `stdc_raise` (7.14.2.10), which behaves as if it called `raise`.
- `stdc_raise` and `sighandle` now construct an object containing an implementation-defined default signal information for the signal number when the `raw_info` argument is a null pointer, and an object representing the calling stack frame when the `raw_context` argument is a null pointer, rather than setting the corresponding `struct stdc_siginfo` member to a null pointer (7.14.2.9, 7.14.2.10).
- Signal deciders may call the `raise`, `stdc_raise`, and `abort` functions.
- Formal definition of *signal decider* added to 7.14.1, together with the subterms *thread-local signal decider* and *global signal decider*.
- Formal definition of *valid signal number* added to 7.14.1: a positive integer that is the signal number of one of the signals defined by the implementation.
- Formal definition of *reentrant* added to clause 3 (numbered 3.23), applying to all uses of "reentrant" and "reentrancy" throughout the document, including the stream lock sentence of 7.24.3 and the requirement on the `tss_async_signal_safe_attr` function pointers in 7.30.1.
- Decider ordering clarified: thread-local deciders are called most recently installed first, and global deciders with `callfirst == true` most recently installed first, followed by those with `callfirst == false` most recently installed last.
- `sig_decision_call_recovery` from a thread-local decider now specifies a non-local jump (5.2.2.4) to the saved calling environment instead of an as-if `setjmp/longjmp` description.
- Added `sigdecider_abandon()` and `sigdecider_abandon_resume()` (7.14.3.2 and 7.14.3.3) and specified the abandonment of a `sigguarded()` call.

- The `signal` function paragraph 5 restrictions were replaced by "The signal handler shall meet the requirements of 7.14.4", and it is now undefined behavior to call `signal` for a signal number that is activated.
- `siginstall` with a null pointer argument now means all signals.
- A signal number that `siginstall` cannot install for reasons specific to the implementation is silently not installed and the call reports success; it is implementation-defined which signal numbers are not installed, and an implementation should document the set of signal numbers for which it silently does not install (7.14.2.5).
- The opaque handles returned by `siginstall` and `signal_decider_create` now have the implementation-defined pointer types `sig_handlers_t` and `signal_decider_t`, respectively, rather than `void *` (7.14.1).
- `signal_decider_create` never invokes a decider for a signal number that is not activated: a decider installed for a signal number that is not activated at the time of the call is not invoked until the signal number becomes activated (7.14.1, 7.14.2.7). It shall not be called from within a decider; `signal_decider_destroy` may be called from within a decider, and the storage for a currently executing decider entry is released after that decider returns.
- `signal_decider_destroy` is now described as async-signal-safe only when called by a decider upon itself, so a signal decider may call it to destroy its own storage; for any other call it is not async-signal-safe (7.14.2.8, 7.14.4).
- `sigguarded` now specifies undefined behavior when called during the handling of a signal whose signal number is not activated.
- Added `SIGGUARDED_FAILURE_VALUE`, a `constexpr` variable of type `union stdc_siginfo_value` with an implementation-defined value, to 7.14.1.
- `sigset_t`: copying the representation of a `sigset_t` object, by assignment or otherwise, yields an object representing the same set of signals, independent of the address or continued existence of the source object.
- `tss_async_signal_safe` renamed `tss_async_signal_safe_t`.
- A global decider returning `sig_decision_call_recovery` now causes the default action for that signal number to be performed.
- If every signal decider in the ordered sequence returns `sig_decision_next_decider`, or if the ordered sequence is empty, the signal is passed to the signal handler installed before the signal number was activated; if that handler was the default handling, the default handling for that signal number on that implementation is performed (7.14.1).
- Defined the interruption point of 5.2.2.4 as the moment the handling of the signal begins, and spelled out in 7.14.4 how a signal decider can use `sigfence` to keep the representation of the objects it modifies from becoming indeterminate.

- It is now unspecified (was implementation-defined) whether the `sighandle` function ever returns.
- The `sigemptyset` and `sigfillset` functions now specify, like the `sigfillset_*` functions, that it is permitted for the memory pointed to by `setp` to be uninitialized.

Contents

1	Overview	5
2	Proposed wording	6
3	References	32

1 Overview

This paper takes a conservative approach to modernising signal handling, and fixes these specific problems only:

1. `signal()` (and indeed POSIX `sigaction()`) are not suited for programs containing multiple threads, as installing and removing signal handlers is not threadsafe.
2. `signal()` (and indeed POSIX `sigaction()`) are not suited for programs containing shared libraries, as dynamically loading and unloading a shared library cannot install and remove signal handlers easily i.e. they don't compose well in modern software architecture.
3. There is no ability to install thread local signal handlers so a signal caused by one thread can be handled locally, and it not affect other threads of execution.
4. Finally, `_Thread_local` may, or may not, be async signal handler safe depending on platform. As the previous functionality needs to store thread-local state which is safe to use from within a signal handler, we might as well expose that as a public library API for other code to use.

There is a quarter century of existing practice being standardised here:

1. Microsoft Windows Structured Exception Handling implements exactly what this pure library extension to the C standard library would now implement portably. On Microsoft Windows, the reference library is a thin wrap of Microsoft Windows facilities and does little other work.
2. IBM z/OS appears to also implement exactly what this pure library extension to the C standard library would now implement portably. On z/OS, the reference library should be a thin wrap of z/OS APIs and would do little other work¹.

On POSIX, it exclusively uses `sigaction()` and does not require any new functionality in the C standard library. The reference implementation internally uses `setjmp()` and `longjmp()` on POSIX to implement signal recovery, but this paper does not mandate that and an implementation can use any mechanism it likes internally.

It is expected that after this paper is merged, I will propose a second paper *Even more thread-safe signal handling* to WG14 which takes a less conservative approach than this paper does. It will enable some signal handlers to execute arbitrary code not unencumbered by async signal handling

¹My thanks to Rajan for illuminating the z/OS documentation to me. I unfortunately have no access to such a system to port this library to z/OS, however from reading its documentation, such a port should be trivial. I asked a LLM to plan how to implement the reference library for z/OS based on its reading of the z/OS documentation, and it came up with https://github.com/ned14/wg14_signals/blob/main/plans/ibm_zos_backend.md which would seem to confirm that such a port would indeed be trivial.

restrictions. This is possible because every major platform provides proprietary APIs to provide signal handling by a background kernel thread, so this is another area ripe for standardisation seeing as this facility is ubiquitous across the major platforms. That proposal will overlay this proposal, and can be considered entirely an extension to this proposal which may be separately accepted or rejected by WG14.

Finally, there is a very great deal of existing code out there using existing signals, and the API has been complicated by needing to be as good a citizen as possible to existing code. We allow users to partially opt-in, partially opt-out, and entirely opt-out. We also allow existing signal handling code to call into this modern signals handling, so people can still have their code be the first layer of signal handling, and then vector into this code after. As a quick summary of the relevant APIs:

- `sighandle()` initiates signal handling by this framework – existing code can install its own global signal handling, do its own processing, then delegate to `sighandle()`.
- `stdc_raise()` reinitiates signal handling preserving the original `siginfo` and context – this lets handling try to process a signal up to a point where it cannot continue, and then ‘re-throw’ the signal from the beginning. You would almost never do this in code fully ported over to the new API, but for mixed implementation signals handling this facility can be a lifesaver.

It is proposed that for one major C standard that these facilities shall be opt-in with `signal()` marked deprecated. The following C standard these facilities would become explicit opt-out.

2 Proposed wording

The following changes are made against [N3886], *Working Draft*, 2026-05-24.

Legend

Deletions in the shown standard text are struck through in orange; additions are underlined in teal green.

Add a definition to clause 3 Terms, definitions, and symbols

Insert the following new entry between 3.22 (recommended practice) and 3.23 (runtime-constraint); the new entry is numbered 3.23, and the entries 3.23 through 3.33 are renumbered 3.24 through 3.34:

3.23 reentrant

reentrant able to be entered or invoked again before a previous entry or invocation has completed, without interference between the entries or invocations.

Note 1 to entry: Reentrancy does not by itself imply that calls can be executed concurrently by multiple threads; that property is addressed by the data race rules (5.2.2.5).

Modifications in clause 5.2.2.4 paragraph 6

When the processing of the abstract machine is interrupted by receipt of a signal, the values of objects that are ~~neither lock-free atomic objects nor of type `volatile sig_atomic_t`~~

not in any of the following categories:

1. A lock-free atomic object.
2. An object of type `volatile sig_atomic_t`.
3. An object not modified after a call to the `sigfence` macro (7.14.1) that describes the memory of the object and that is sequenced before the interruption.
4. An object pointed to by the `void *` value returned by `tss_async_signal_safe_get` (7.30.6.8).

are unspecified, as is the state of the dynamic floating-point environment. The representation of any object modified by the handler that is ~~neither a lock-free atomic object nor of type `volatile sig_atomic_t`~~

not in any of the following categories:

1. A lock-free atomic object.
2. An object of type `volatile sig_atomic_t`.
3. An object not modified after a call to the `sigfence` macro (7.14.1) that describes the memory of the object and that is sequenced before the handler exits.
4. An object pointed to by the `void *` value returned by `tss_async_signal_safe_get` (7.30.6.8).

becomes indeterminate when the handler exits, as does the state of the dynamic floating-point environment if it is modified by the handler and not restored to its original state. For the purposes of this paragraph, the processing of the abstract machine is interrupted when the handling of the signal (7.14.1) begins.

Insert the following new paragraphs after paragraph 6; the existing paragraphs 7 through 17 are renumbered 11 through 21:

7 At a call to the `sigfence` macro (7.14.1), the implementation shall not deviate from the abstract machine for the memory that the call describes: the value of the memory at the point of the call is the value most recently stored to the memory before the call; an access to the memory that is sequenced after the call reads from the memory; and no access to the memory is performed on the other side of the call from where it is sequenced.

8 NOTE The two category 3 restrictions differ: the first requires the call of the `sigfence` macro to be sequenced before the interruption, because only then is the value of the memory at the interruption guaranteed; the second has no such requirement: a signal handler or signal decider can instead preserve the representation of an object it modifies by calling the `sigfence` macro during the handling of the signal (7.14.1), after the modification.

9 A *non-local jump* is a transfer of control that restores a calling environment saved by an earlier operation and resumes thread execution at the point of that operation. A non-local jump is performed by the `longjmp` function (7.13.3.1), and by the signal recovery mechanism specified in 7.14.3.1; the restrictions of 7.13.3.1 do not apply to a non-local jump performed by signal recovery. The environment saved by such an operation consists of information sufficient to return execution to the correct block and invocation of that block, were it called recursively; it does not include the

state of the floating-point environment, of open files, or of any other component of the abstract machine.

10 When a non-local jump is performed, all accessible objects have values, and all other components of the abstract machine have state, as of the time the non-local jump was performed, except that the representation of objects of automatic storage duration that are local to the function containing the operation that saved the environment and that do not have volatile-qualified type and have been changed between the saving of the environment and the non-local jump is indeterminate. The representation of an object that is not modified after a call to the `sigfence` macro (7.14.1) that describes the memory of the object and that is sequenced before the non-local jump is not indeterminate.

In the forward references of 5.2.2.4, replace "the signal function (7.14), files (7.24.3)" with "non-local jumps (7.13), the `sigfence` macro (7.14.1), the signal function (7.14.2.3), the signal recovery mechanism (7.14.3.1), files (7.24.3), the `tss_async_signal_safe_get` function (7.30.6.8)":

Modifications in clause 7.13 Non-local jumps <setjmp.h>

In 7.13.1 General

3 The type declared is

```
1 jmp_buf
```

which is an array type suitable for holding the information needed to restore a calling environment. ~~The environment of an invocation of the `setjmp` macro consists of information sufficient for a call to the `longjmp` function to return execution to the correct block and invocation of that block, were it called recursively. It does not include the state of the floating-point environment, of open files, or of any other component of the abstract machine.~~

In 7.13.2.1: The `setjmp` macro

Description

The `setjmp` macro saves its calling environment in its `jmp_buf` argument ~~for later use by the `longjmp` function~~for later restoration by a non-local jump (5.2.2.4).

Returns

If the return is from a direct invocation, the `setjmp` macro returns the value zero. If the return is from ~~a call to the `longjmp` function~~a non-local jump (5.2.2.4), the `setjmp` macro returns a nonzero value.

In 7.13.3.1: The `longjmp` function

Description

The `longjmp` function ~~restores the environment~~performs a non-local jump (5.2.2.4) to the environment saved by the most recent invocation of the `setjmp` macro in the same invocation of

the program with the corresponding `jmp_buf` argument. If there has been no such invocation, or if the invocation was from another thread of execution, or if the function containing the invocation of the `setjmp` macro has terminated execution in the interim, or if the invocation of the `setjmp` macro was within the scope of an identifier with variably modified type and execution has left that scope in the interim, the behavior is undefined.

~~All accessible objects have values, and all other components of the abstract machine have state, as of the time the `longjmp` function was called, except that the representation of objects of automatic storage duration that are local to the function containing the invocation of the corresponding `setjmp` macro that do not have volatile-qualified type and have been changed between the `setjmp` invocation and `longjmp` call is indeterminate.~~

Returns

After `longjmp` is completed, thread execution continues as if the corresponding invocation of the `setjmp` macro had just returned the value specified by `val` (5.2.2.4). The `longjmp` function cannot cause the `setjmp` macro to return the value 0; if `val` is 0, the `setjmp` macro returns the value 1.

Modifications in clause 7.14.1

1 The header `<signal.h>` declares ~~a type and two functions and defines several macros~~ types, enumeration constants, and functions, and defines macros, for handling various *signals* (conditions that may be reported during program execution).

Insert the following new paragraphs after paragraph 1:

2 A signal can be received by a thread within the program. When a signal is received, the processing of the abstract machine is interrupted (5.2.2.4) and the signal is handled as described below.

3 There are the following categories of signal:

1. *Synchronous*: these are caused by the execution of a thread. Standard signals are:

- abnormal termination,
- erroneous arithmetic operation,
- detection of an invalid function image, and
- invalid access to storage.

The implementation may define additional signals in the synchronous category.

2. *Asynchronous non-debug*: these are generated by the environment and are not usually generated by testing. Standard signals are:

- receipt of an interactive attention signal, and
- termination request sent to the program.

The implementation may define additional signals in the asynchronous non-debug category.

3. *Asynchronous debug*: these are generated by the environment and are usually generated by testing.

The implementation may define additional signals in the asynchronous debug category.

4 *Async-signal-safe* functions and macros are those safe to call during the handling of a signal. Only the functions and macros in the standard library listed below are required to be async-signal-safe; all other functions and macros in the standard library need not be async-signal-safe. This document does not specify whether functions of the program that are not in the standard library are async-signal-safe. The following functions and macros are required to be async-signal-safe:

- the functions in `<stdatomic.h>` (except where explicitly stated otherwise) when the atomic arguments are lock-free,
- the `atomic_is_lock_free` function with any atomic argument,
- the `signal` function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the signal handler, or
- any function or macro in this document explicitly described as async-signal-safe.

If such a call to the `signal` function results in a `SIG_ERR` return, the object designated by `errno` has an indeterminate representation.

5 There are two ways to specify the handling of a signal:

1. The `signal` function installs a single signal handler for the whole program, replacing any handler previously installed by the `signal` function.
2. The `siginstall` function enables an alternative, thread-safe, composable signal handling mechanism.

6 A *signal decider* is a function of type `sig_decide_t` that is installed by a call of the `sigguarded` function (7.14.3.1) or of the `signal_decider_create` function (7.14.2.7) and that is invoked during the handling of a signal to decide how that signal is to be handled. Signal deciders installed by the `sigguarded` function are *thread-local signal deciders*; signal deciders installed by the `signal_decider_create` function are *global signal deciders*.

7 Signal handling state persists for the entire execution of the program and consists of the following kinds, all of which coexist:

- the signal handler installed by the `signal` function (7.14.2.3): a single handler per signal number, replaced by each call of the `signal` function, which persists whether or not the signal number is activated;
- the thread-local signal deciders installed by the `sigguarded` function (7.14.3.1), one ordered sequence per thread, which persist whether or not the signal numbers in their signal sets are activated;
- the global signal deciders installed by the `signal_decider_create` function (7.14.2.7), which persist whether or not the signal numbers in their signal sets are activated; and
- the activation count for each signal number: the number of calls of `siginstall` that install that signal number, minus the number of calls of `siguninstall` that uninstall it; a signal number is *activated* while its activation count is nonzero, and is *deactivated* while its activation count is zero.

8 The kinds of state coexist whether or not a signal number is activated: activation and deactivation create or remove none of the other kinds of state. In particular, a call of the `signal` function replaces only the signal handler installed by a previous call of the `signal` function; it does not remove thread-local or global signal deciders, nor does it change the activation count for any signal number. For the purposes of this subclause, the equivalent of `signal(sig, SIG_DFL)` that an implementation may perform prior to calling a signal handler (7.14.2.3), and the equivalents of `signal(sig, SIG_IGN)` and `signal(sig, SIG_DFL)` that an implementation may perform at program startup (7.14.2.3), are considered calls of the `signal` function.

9 Concurrent calls of the `siginstall` (7.14.2.5), `siguninstall` (7.14.2.6), `signal_decider_create` (7.14.2.7), and `signal_decider_destroy` (7.14.2.8) functions from multiple threads shall not introduce data races.

10 If the signal number of the signal received is not activated, then the following sequence occurs on receipt of the signal:

1. The handler installed by the most recent call of the `signal` function for that signal number before the receipt of the signal is called, unless that handler is `SIG_IGN`, in which case the signal is ignored, or is `SIG_DFL`, in which case the default handling for that signal number on that implementation is performed. The thread on which the handler is called is unspecified.

11 If the signal number of the signal received is activated, then the following sequence occurs on receipt of the signal:

1. For synchronous category signals, the signal deciders are invoked on the thread that caused the signal; for asynchronous category signals, the thread on which the signal deciders are invoked is unspecified.
2. An ordered sequence of signal deciders is invoked on that thread to decide how to handle the signal. The ordered sequence begins with the thread-local signal deciders whose signal sets include the signal number, most recently installed first, followed by the global signal deciders whose signal sets include the signal number. The ordered sequence is determined when the signal is received: a thread-local signal decider installed while an earlier occurrence of a signal is being processed is not invoked for that occurrence, but is invoked for occurrences received after its installation. Each decider is called with a pointer to a `struct stdc_siginfo` object whose `signo` member is set to the number of the signal received and whose `value` member is set to the `value` argument of the call of `sigguarded` (7.14.3.1) or `signal_decider_create` (7.14.2.7) that installed the decider. For a signal other than one raised by the `stdc_raise` function (7.14.2.10) and other than one for which deciders are invoked by a call of the `sighandle` function (7.14.2.9), the values of the remaining members of the `struct stdc_siginfo` object are implementation-defined, except that the `raw_info` and `raw_context` members may be null pointers. The `struct stdc_siginfo` object passed to a signal decider is not subject to the rules of 5.2.2.4 for objects modified by a signal handler or by a signal decider, and its lifetime extends until the handling of the signal completes; if the recovery function of the `sigguarded` function call (7.14.3.1) that installed the decider is called, its lifetime extends instead until the recovery function returns.

- For thread-local signal deciders, if a decider function returns:

- sig_decision_resume_execution: execution of the interrupted thread is resumed.
- sig_decision_call_recovery: a non-local jump (5.2.2.4) is performed to the calling environment saved when that thread-local decider was installed, and the recovery function specified at that time shall be called, with a pointer to the struct stdc_sinfo object that was passed to the decider, to implement recovery from the signal raise for that thread.
- sig_decision_next_decider: the next decider in the sequence is called.
- For global signal deciders, the deciders are called in the following order: first, those installed with the callfirst argument true, most recently installed first; then, those installed with the callfirst argument false, in the order in which they were installed.

If a decider function returns:

- sig_decision_resume_execution: execution of the interrupted thread is resumed.
- sig_decision_call_recovery: the default handling for that signal number on that implementation is performed.
- sig_decision_next_decider: the next decider in the sequence is called.

12 It is implementation-defined whether, when a signal occurs while an earlier occurrence of the same signal has not yet been completely processed, the later occurrence is blocked until the processing of the earlier occurrence has completed, is discarded, or is processed immediately, in which case its processing may be nested within that of the earlier occurrence. In particular, an implementation may block the signal during the execution of a signal handler or of signal deciders for that signal, and may discard repeated occurrences before the processing of the signal completes, so that only one occurrence is processed.

13 A signal decider is not required to return. Signal deciders shall meet the requirements of 7.14.4. A signal decider that determines that it will never return should call the sigdecider_abandon function (7.14.3.2) before it does so, and may call the sigdecider_abandon_resume function (7.14.3.3) to retract that declaration if it later determines that it will return. 14 If every signal decider in the ordered sequence returns sig_decision_next_decider, or if the ordered sequence is empty, the signal is passed to the signal handler that was installed for that signal number before it was activated; if that signal handler was the default handling, the default handling for that signal number on that implementation is performed. While a signal number is activated, the signal handler installed by the signal function for that signal number is invoked only as that continuation; it is not invoked directly by the implementation's signal delivery mechanism. Conversely, while a signal number is not activated, signal deciders are not used to handle signals received with that number, except by a call of the sighandle function (7.14.2.9).

15 The activation count for a signal number is incremented by each call of siginstall that installs that signal number, and is decremented by each call of siguninstall that uninstalls it; a signal number shall not be deactivated until the last uninstallation for that signal number is performed, that is, until its activation count reaches zero. While a signal number is not activated, a signal with that number is handled as specified for a signal whose signal number is not activated, and no signal deciders are invoked for it, except by a call of the sighandle function (7.14.2.9); when a signal

number becomes activated again, the signal deciders whose signal sets include that signal number are invoked again. Activation and deactivation of signal numbers do not install or remove signal deciders or signal handlers.

16 Recommended practice It is recommended practice that pre-existing programs be upgraded to use `siginstall`, and that newly written code prefer `siginstall` over `signal`.

17 EXAMPLE 1 Use `sigguarded` to recover from a `SIGFPE`

```
1  #include <signal.h>
2  #include <assert.h>
3  #include <stdlib.h>
4
5  /* Marker used to verify the value returned by recovery */
6  static int sigfpe_marker;
7
8  /* Recovery function for SIGFPE */
9  static union stdc_siginfo_value
10 sigfpe_recovery_func(const struct stdc_siginfo *rsi)
11 {
12     /* Recover from the signal raise */
13     return rsi->value;
14 }
15
16 /* Decider function for SIGFPE */
17 static enum sig_decision
18 sigfpe_decider_func(struct stdc_siginfo *rsi)
19 {
20     /* Verify we got SIGFPE */
21     if (rsi->signo != SIGFPE)
22     {
23         abort();
24     }
25     rsi->value.ptr_value = &sigfpe_marker;
26     /* Please recover */
27     return sig_decision_call_recovery;
28 }
29
30 /* Guarded function that triggers SIGFPE via division by zero */
31 static union stdc_siginfo_value
32 sigfpe_func(union stdc_siginfo_value value)
33 {
34     /* volatile is needed to prevent elision under optimization */
35     volatile int divisor = 0;
36     /* This should trigger SIGFPE */
37     volatile int result = 42 / divisor;
38     /* If we get here, this architecture doesn't trap integer divide by zero */
39     sighandle(SIGFPE, nullptr, nullptr);
40     return value;
41 }
42 ...
43 int main() {
44     union stdc_siginfo_value value = { .ptr_value = nullptr };
45     sigset_t guarded;
46     sigemptyset(&guarded);
47     sigaddset(&guarded, SIGFPE);
```

```

48  /* Activate SIGFPE */
49  siginstall(&guarded);
50  value = sigguarded(&guarded,
51                    sigfpe_func,
52                    sigfpe_recovery_func,
53                    sigfpe_decider_func,
54                    value);
55  assert(value.ptr_value == &sigfpe_marker);
56  }

```

18 NOTE The integer division by zero in this example is undefined behavior (6.5.5); the example relies on the implementation trapping the operation and raising SIGFPE, and falls back to sighandle when it does not.

19 (base paragraph 2) The ~~type-defined~~ types defined are

stdc_siginfo_error_code_t which is an implementation-defined complete object type that represents a system-specific error code.

The stdc_siginfo_value union shall contain at least a member `void *ptr_value`, and, if `intptr_t` is defined (7.23.2.5), a member `intptr_t int_value`, in any order.

stdc_siginfo_siginfo_t which is an implementation-defined possibly incomplete object type that represents the system-specific signal information.

stdc_siginfo_context_t which is an implementation-defined possibly incomplete object type that represents the system-specific context.

The `stdc_siginfo` structure shall contain at least the following members, in any order. The semantics of the members are expressed in the comments.

```

1  // The signal raised
2  int signo;
3
4  // The system specific error code for this signal
5  stdc_siginfo_error_code_t error_code;
6
7  // The memory location that caused the fault, if appropriate
8  void *addr;
9
10 // A user-defined value
11 union stdc_siginfo_value value;
12
13 // The system specific information. Can be a null pointer if no
14 // information is available.
15 stdc_siginfo_siginfo_t *raw_info;
16
17 // The system specific context. Can be a null pointer if no
18 // information is available.
19 stdc_siginfo_context_t *raw_context;

```

The sig_func_t type is a function type with a single argument of type `union stdc_siginfo_value` and a return type of `union stdc_siginfo_value`, used as the type of the guarded function of the `sigguarded` function (7.14.3.1).

The `sig_recover_t` type is a function type with a single argument of type `const struct stdc_siginfo *` and a return type of `union stdc_siginfo_value`, used as the type of the recovery function of the `sigguarded` function (7.14.3.1).

The `sig_decision` enumeration shall contain at least the following members, in any order. The semantics of the members are expressed in the comments:

```
1 // Defer to the next decider
2 sig_decision_next_decider
3
4 // Resume execution; the cause of the signal has been fixed
5 sig_decision_resume_execution
6
7 // Thread-local signal deciders only: restore the environment
8 // to what it was at entry to sigguarded(), and call the
9 // recovery function
10 sig_decision_call_recovery
```

The `sig_decide_t` type is the function type of a signal decider (7.14.1): a function with a single argument of type `struct stdc_siginfo *` and a return type of `enum sig_decision`.

`sig_handlers_t` which is an implementation-defined opaque pointer type that represents a set of installed signal handlers. It is the type of the value returned by the `siginstall` function (7.14.2.5) and of the argument of the `siguninstall` function (7.14.2.6); a null pointer value represents no installation.

`signal_decider_t` which is an implementation-defined opaque pointer type that represents an installed global signal decider. It is the type of the value returned by the `signal_decider_create` function (7.14.2.7) and of the argument of the `signal_decider_destroy` function (7.14.2.8); a null pointer value represents no decider.

`sig_atomic_t`

which is the (possibly volatile-qualified) integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

`sigset_t`

which is an implementation-defined complete object type able to represent a set of signals. Copying the representation of a `sigset_t` object, by assignment or otherwise, yields an object that represents the same set of signals as the original; the set represented does not depend on the address of the object or on the continued existence of the object from which it was copied. In particular, the following code is valid:

```
1 sigset_t a;
2 sigemptyset(&a);
3 sigset_t b = a;
```

After paragraph 19 (base paragraph 2), insert the following new paragraph:

`20 SIGGUARDED_FAILURE_VALUE`, which is a `constexpr` object of type `union stdc_siginfo_value` with an implementation-defined value, is the value returned by the `sigguarded` function (7.14.3.1) when it fails to install the guard.

21 (base paragraph 3) The macros defined are

```
1 SIG_DFL
2 SIG_ERR
3 SIG_IGN
```

which expand to constant expressions with distinct values that have type compatible with the second argument to, and the return value of, the `signal` function, and whose values compare unequal to the address of any declarable function; and the following, which expand to positive integer constant expressions with type `int` and distinct values that are the signal numbers, each corresponding to the specified condition:

`SIGABRT` abnormal termination, such as is initiated by the `abort` function, which is of the *synchronous* signal category

`SIGFPE` an erroneous arithmetic operation, such as zero divide or an operation resulting in overflow, which is of the *synchronous* signal category

`SIGILL` detection of an invalid function image, such as an invalid instruction, which is of the *synchronous* signal category

`SIGINT` receipt of an interactive attention signal, which is of the *asynchronous non-debug* signal category

`SIGSEGV` an invalid access to storage, which is of the *synchronous* signal category

`SIGTERM` a termination request sent to the program, which is of the *asynchronous non-debug* signal category

In paragraph 21 (base paragraph 3), insert the following after the `SIGTERM` entry:

and the following macro, which restricts, for the memory described below, the freedom of an implementation to deviate from the abstract machine (5.2.2.4):

`sigfence(...)` provides the guarantees specified in 5.2.2.4 for the following memory:

- the memory storing all objects with external or internal linkage;
- the memory storing the objects without linkage named by the arguments of the invocation.

`sigfence()` is async-signal-safe. The behavior is undefined if the `sigfence` macro is invoked with more than eight arguments. Each argument, if any, shall be an lvalue designating an object; the behavior is undefined if an argument is a bit-field or designates an object declared with the `register` storage-class specifier. An argument that designates an object with external or internal linkage is permitted but has no effect, because the memory storing such objects is described by any invocation of this macro.

After paragraph 21 (base paragraph 3), insert the following new paragraph:

22 NOTE The `atomic_signal_fence` function (7.17.4.3) provides weaker guarantees than the `sigfence` macro, and may be sufficient for performance-oriented use cases.

23 (base paragraph 4) An implementation is not required to generate any of these signals, except as a result of explicit calls to the `raise` function. Additional signals and pointers to undeclarable

functions, with macro definitions beginning, respectively, with the letters **SIG** and an uppercase letter or with **SIG_** and an uppercase letter, may also be specified by the implementation. The complete set of signals, their semantics, and their default handling is implementation-defined; all signal numbers shall be positive.

After paragraph 23 (base paragraph 4), insert the following new paragraph:

24 A *valid signal number* is a positive integer that is the signal number of one of the signals defined by the implementation.

Insert the signal set functions into clause 7.14.2, renumbering the subsections below accordingly

Insert the following new subsections at the beginning of clause 7.14.2 (Specify signal handling), before "The signal function" (7.14.2.1):

- 7.14.2.1 Signal set initialization
- 7.14.2.2 Signal set manipulation

"The signal function" (7.14.2.1) is renumbered 7.14.2.3, and "The raise function" (7.14.3.1), together with its section heading "7.14.3 Send signal", is moved into clause 7.14.2 and renumbered 7.14.2.4. The new functions specified below are numbered 7.14.2.5 through 7.14.2.10, and the new sections "7.14.3 Recover from signal" and "7.14.4 Requirements for signal handlers and signal deciders" are added after clause 7.14.2.

7.14.2.1 Signal set initialization

The `sigemptyset` function

1 Synopsis

```
1 #include <signal.h>
2 int sigemptyset(sigset_t *setp);
```

2 **Description** The `sigemptyset` function is async-signal-safe.

3 The set of signals pointed to by `setp` is set to the empty set. It is permitted for the memory pointed to by `setp` to be uninitialized.

4 **Returns** The `sigemptyset` function always returns zero.

The `sigfillset` function

5 Synopsis

```
1 #include <signal.h>
2 int sigfillset(sigset_t *setp);
```

6 **Description** The `sigfillset` function is async-signal-safe.

7 The set of signals pointed to by `setp` is set to the full set, the set of all signals defined by the implementation. It is permitted for the memory pointed to by `setp` to be uninitialized.

8 Returns The `sigfillset` function always returns zero.

The `sigfillset_synchronous` function

9 Synopsis

```
1 #include <signal.h>
2 int sigfillset_synchronous(sigset_t *setp);
```

10 Description The `sigfillset_synchronous` function is async-signal-safe.

11 The set of signals pointed to by `setp` is set to exactly the set of synchronous signals. It is permitted for the memory pointed to by `setp` to be uninitialized.

12 Synchronous signals are those caused by the execution of a thread (7.14.1). This set can include additional implementation-defined signals; however, at least the following standard signals are in this set: `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGSEGV`.

13 Returns The `sigfillset_synchronous` function always returns zero.

The `sigfillset_asynchronous_nondebug` function

14 Synopsis

```
1 #include <signal.h>
2 int sigfillset_asynchronous_nondebug(sigset_t *setp);
```

15 Description The `sigfillset_asynchronous_nondebug` function is async-signal-safe.

16 The set of signals pointed to by `setp` is set to exactly the set of asynchronous non-debug signals. It is permitted for the memory pointed to by `setp` to be uninitialized.

17 Asynchronous non-debug signals are those generated by the environment and not usually generated by testing (7.14.1). This set can include additional implementation-defined signals; however, at least the following standard signals are in this set: `SIGINT`, `SIGTERM`.

18 Returns The `sigfillset_asynchronous_nondebug` function always returns zero.

The `sigfillset_asynchronous_debug` function

19 Synopsis

```
1 #include <signal.h>
2 int sigfillset_asynchronous_debug(sigset_t *setp);
```

20 Description The `sigfillset_asynchronous_debug` function is async-signal-safe.

21 The set of signals pointed to by `setp` is set to exactly the set of asynchronous debug signals. It is permitted for the memory pointed to by `setp` to be uninitialized.

22 Asynchronous debug signals are those generated by the environment and usually generated by testing (7.14.1).

23 Returns The `sigfillset_asynchronous_debug` function always returns zero.

7.14.2.2 Signal set manipulation

The **sigaddset** function

1 Synopsis

```
1 #include <signal.h>
2 int sigaddset(sigset_t *setp, int signo);
```

2 Description The **sigaddset** function is async-signal-safe.

3 Signal number **signo** is added to the set of signals pointed to by **setp**; if **signo** is already a member of the set, nothing is done. If **signo** is not a valid signal number (7.14.1), the set is not modified.

4 If ***setp** was not previously initialized by one of the signal set initialization functions (7.14.2.1), the behavior is undefined.

5 Returns The **sigaddset** function returns zero if **signo** is a valid signal number (7.14.1) and a negative value otherwise.

The **sigdelset** function

6 Synopsis

```
1 #include <signal.h>
2 int sigdelset(sigset_t *setp, int signo);
```

7 Description The **sigdelset** function is async-signal-safe.

8 Signal number **signo** is removed from the set of signals pointed to by **setp**; if **signo** is not a member of the set, nothing is done. If **signo** is not a valid signal number (7.14.1), the set is not modified.

9 If ***setp** was not previously initialized by one of the signal set initialization functions (7.14.2.1), the behavior is undefined.

10 Returns The **sigdelset** function returns zero if **signo** is a valid signal number (7.14.1) and a negative value otherwise.

The **sigismember** function

11 Synopsis

```
1 #include <signal.h>
2 int sigismember(const sigset_t *setp, int signo);
```

12 Description The **sigismember** function is async-signal-safe.

13 If ***setp** was not previously initialized by one of the signal set initialization functions (7.14.2.1), the behavior is undefined.

14 Returns The **sigismember** function returns one if signal number **signo** is set within the set of signals pointed to by **setp**, zero if it is not set, and a negative value if **signo** is not a valid signal number (7.14.1).

Modifications in clause 7.14.2.3: The `signal` function

...

Description

...

4 If the signal occurs as the result of calling ~~the `abort` or `raise` function~~the `abort`, `raise`, or `stdc_raise` function, the signal handler shall not call ~~the `raise` function~~the `raise` or `stdc_raise` function.

5

~~If the signal occurs other than as the result of calling the `abort` or `raise` function, the behavior is undefined if the signal handler refers to any object with static or thread storage duration that is not a lock-free atomic object and that is not declared with the `constexpr` storage-class specifier other than by assigning a value to an object declared as `volatile sig_atomic_t`, or the signal handler calls any function in the standard library other than -- the `abort` function, -- the `_Exit` function, -- the `quick_exit` function, -- the functions in `<stdatomic.h>` (except where explicitly stated otherwise) when the atomic arguments are lock-free, -- the `atomic_is_lock_free` function with any atomic argument, or -- the `signal` function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler. Furthermore, if such a call to the signal function results in a `SIG_ERR` return, the object designated by `errno` has an indeterminate representation.~~

The signal handler shall meet the requirements of 7.14.4.

6 ...

7 Use of this function in a multi-threaded program results in undefined behavior. It is also undefined behavior to call this function for a signal number that is activated. ~~The implementation shall behave as if no library function calls the signal function.~~

Modifications in clause 7.14.2.4: The `raise` function

The text of "The `raise` function" (7.14.3.1), moved and renumbered 7.14.2.4, is amended as shown:

1 Synopsis

```
1 #include <signal.h>
2 int raise(int sig);
```

2 **Description** The `raise` function is async-signal-safe. The `raise` function carries out the actions described in ~~7.14.2.1~~7.14.1 for the signal `sig`. Signal deciders are invoked for `sig` only if `sig` is activated; if `sig` is not activated, no signal deciders are invoked. If a signal handler is called, the `raise` function shall not return until after the signal handler does; if signal deciders are called, the `raise` function shall not return until after the ordered sequence of signal deciders (7.14.1) completes.

3 **Returns** The `raise` function returns zero if successful, nonzero if unsuccessful.

Add the new signal handling functions to clause 7.14.2

Add 7.14.2.5: The `siginstall` function

1 Synopsis

```
1 #include <signal.h>
2 sig_handlers_t siginstall(const sigset_t *guarded);
```

2 Description If `*guarded` was not previously initialized by one of the signal set initialization functions (7.14.2.1), the behavior is undefined.

3 If `guarded` is a null pointer, it is treated as if it pointed to a signal set containing all signals.

4 All signal numbers in the signal set `guarded` shall be activated as specified in 7.14.1, except that a signal number that cannot be installed for reasons specific to the implementation is silently not installed and the call reports success. It is implementation-defined which signal numbers are not installed; an implementation should document the set of signal numbers for which it silently does not install.

5 Returns The `siginstall` function returns a handle that can be passed to the `siguninstall` function (7.14.2.6) to uninstall the installation, if the installation was successful; otherwise it returns a null pointer. The installation is successful if every signal number in the signal set was either installed or silently not installed as specified in paragraph 4.

Forward references: the `siguninstall` function (7.14.2.6).

Add 7.14.2.6: The `siguninstall` function

1 Synopsis

```
1 #include <signal.h>
2 int siguninstall(sig_handlers_t handle);
```

2 Description All signal numbers in the signal set installed by the `siginstall` call that returned `handle` shall be deactivated as specified in 7.14.1. Uninstallation does not remove any signal deciders, and no signal decider is associated with any particular call of `siginstall` or `siguninstall`.

3 The handle becomes invalid after a successful call to this function. The behavior is undefined if `handle` is not the value returned by a prior call of the `siginstall` function (7.14.2.5) that has not yet been uninstalled, or if `handle` is a null pointer.

4 Returns The `siguninstall` function returns zero if successful and a negative value if unsuccessful.

Add 7.14.2.7: The `signal_decider_create` function

1 Synopsis

```
1 #include <signal.h>
2 signal_decider_t signal_decider_create(const sigset_t *guarded, bool callfirst,
3                                     sig_decide_t decider,
4                                     union stdc_siginfo_value value);
```

2 Description If `*guarded` was not previously initialized by one of the signal set initialization functions (7.14.2.1), the behavior is undefined.

3 Installs a global signal decider function, which shall meet the requirements of 7.14.4. See 7.14.1 for how global signal decider functions are called.

4 If `callfirst` is true, the function is installed at the beginning of the ordered sequence of global signal deciders (7.14.1), to be called before any other global signal deciders currently installed; otherwise it is installed at the end of that ordered sequence, to be called after any other global signal deciders currently installed.

5 If a signal number in `guarded` is not activated at the time of the call, a decider installed for that signal is not invoked until the signal number becomes activated (7.14.1).

6 It is undefined behavior to call this function from within a signal decider function.

7 Returns The `signal_decider_create` function returns a non-null pointer that can be later passed to the `signal_decider_destroy` function (7.14.2.8) if the call is successful; otherwise it returns a null pointer.

Forward references: the `signal_decider_destroy` function (7.14.2.8), requirements for signal handlers and signal deciders (7.14.4).

Add 7.14.2.8: The `signal_decider_destroy` function

1 Synopsis

```
1 #include <signal.h>
2 int signal_decider_destroy(signal_decider_t handle);
```

2 Description The `signal_decider_destroy` function uninstalls a previously installed global signal decider function. The function is async-signal-safe only when it is called by a signal decider to destroy the storage of the decider that is currently executing; for any other call, the function is not async-signal-safe.

3 It is permitted to call this function from within a signal decider function only to destroy that decider's own storage. If the decider whose storage is being destroyed is currently executing, it shall complete its execution and the storage for its entry is released after that decider returns. A decider that is destroyed before it is called for the current occurrence of the signal is not called for that occurrence.

4 The behavior is undefined if `handle` is not the value returned by a prior call of the `signal_decider_create` function (7.14.2.7) that has not yet been destroyed, or if `handle` is a null pointer.

5 Returns The `signal_decider_destroy` function returns zero if successful and a negative value if unsuccessful.

Add 7.14.2.9: The `sighandle` function

1 Synopsis

```

1 #include <signal.h>
2 bool sighandle(int signo,
3                 stdc_siginfo_siginfo_t *raw_info,
4                 stdc_siginfo_context_t *raw_context);

```

2 Description The `sighandle` function is `async-signal-safe`.

3 The `sighandle` function does not call the `raise` function (7.14.2.4). It invokes the signal deciders whose signal sets include `signo`, in the order specified in 7.14.1, as if a signal with the signal number `signo` had been received by the calling thread. The signal deciders installed by the `sigguarded` function (7.14.3.1) on the calling thread are invoked whether or not `signo` is activated; the global signal deciders installed by the `signal_decider_create` function (7.14.2.7) for `signo` are invoked only if `signo` is activated.

4 If every signal decider in the ordered sequence returns `sig_decision_next_decider`, or if the ordered sequence is empty, then: if `signo` is not activated (7.14.1), this function returns false without passing the signal on; otherwise the signal is passed to the signal handler that was installed for `signo` before the signal number was activated, and if that signal handler was the default handling, the default handling for `signo` is performed.

5 If `signo` is zero, this function returns false after initializing any per-thread state required for the calling thread. If `signo` is not a valid signal number (7.14.1), this function returns false without raising a signal.

6 The `signo` member of the struct `stdc_siginfo` object passed to deciders is set to `signo`, and the `error_code` and `addr` members are implementation-defined.

7 If `raw_info` specified here is not a null pointer, the `raw_info` member in the struct `stdc_siginfo` object passed to deciders is set to `raw_info`. Otherwise, an object of type `stdc_siginfo_siginfo_t` containing an implementation-defined default signal information for `signo` is constructed, and the `raw_info` member in the struct `stdc_siginfo` object passed to deciders is set to a pointer to that object.

8 If `raw_context` specified here is not a null pointer, the `raw_context` member in the struct `stdc_siginfo` object passed to deciders is set to `raw_context`. Otherwise, an object of type `stdc_siginfo_context_t` that represents the calling stack frame is constructed, and the `raw_context` member in the struct `stdc_siginfo` object passed to deciders is set to a pointer to that object.

9 Objects constructed as described in the previous two paragraphs persist at least as long as the struct `stdc_siginfo` object whose `raw_info` or `raw_context` member points to it (7.14.1).

10 Returns It is unspecified whether the `sighandle` function ever returns. If it returns, it returns true if a signal decider did not return `sig_decision_next_decider`, or if the signal was passed to a previously installed signal handler as described in paragraph 4, and false otherwise.

Add 7.14.2.10: The `stdc_raise` function

1 Synopsis

```

1 #include <signal.h>
2 int stdc_raise(int signo,
3               stdc_siginfo_siginfo_t *raw_info,
4               stdc_siginfo_context_t *raw_context);

```

2 Description The `stdc_raise` function is async-signal-safe.

3 This function behaves as if it called the `raise` function (7.14.2.4) with the argument `signo`, except that the additional information `raw_info` and `raw_context` is made available to the signal deciders.

4 The `signo` member of the `struct stdc_siginfo` object passed to deciders is set to `signo`, and the `error_code` and `addr` members are implementation-defined.

5 If `raw_info` specified here is not a null pointer, the `raw_info` member in the `struct stdc_siginfo` object passed to deciders is set to `raw_info`. Otherwise, an object of type `stdc_siginfo_siginfo_t` containing an implementation-defined default signal information for `signo` is constructed, and the `raw_info` member in the `struct stdc_siginfo` object passed to deciders is set to a pointer to that object.

6 If `raw_context` specified here is not a null pointer, the `raw_context` member in the `struct stdc_siginfo` object passed to deciders is set to `raw_context`. Otherwise, an object of type `stdc_siginfo_context_t` that represents the calling stack frame is constructed, and the `raw_context` member in the `struct stdc_siginfo` object passed to deciders is set to a pointer to that object.

7 Objects constructed as described in the previous two paragraphs persist at least as long as the `struct stdc_siginfo` object whose `raw_info` or `raw_context` member points to it (7.14.1).

8 Returns The `stdc_raise` function returns zero if successful and nonzero if unsuccessful. If `signo` is not a valid signal number (7.14.1), this function is unsuccessful and no signal is raised.

Add 7.14.3 Recover from signal

Add 7.14.3.1: The `sigguarded` function

1 Synopsis

```

1 #include <signal.h>
2 union stdc_siginfo_value
3 sigguarded(const sigset_t *signals,
4            sig_func_t guarded,
5            sig_recover_t recovery,
6            sig_decide_t decider,
7            union stdc_siginfo_value value);

```

2 Description The `sigguarded` function is async-signal-safe. The behavior is undefined if this function is called during the handling of a signal whose signal number is not activated (7.14.1). The decider function shall meet the requirements of 7.14.4.

3 Installs a thread-local signal decider function on the calling thread, saving the calling environment such that it can be restored later by a non-local jump (5.2.2.4). If a decider installed by this call

returns `sig_decision_call_recovery`, a non-local jump (5.2.2.4) to the saved calling environment is performed, and the `recovery` function of this call is called, with a pointer to the `struct stdc_sinfo` object that was passed to the decider, to implement recovery from the signal raise. The thread-local signal decider installed by this call is removed when the `sigguarded` function returns or when the `recovery` function of this call begins, whichever is earlier. At that point, this `sigguarded` function call is no longer active, and any earlier `sigguarded` function call on the same thread takes over. See 7.14.1 for how thread-local signal decider functions are called.

4 The behavior is undefined if `signals`, `guarded`, or `decider` is a null pointer.

5 The behavior is undefined if `guarded` or `recovery` transfers control out of the function other than by returning, unless the `decider` abandons the guard (7.14.3.2).

6 If `recovery` is a null pointer, a decider returning `sig_decision_call_recovery` is treated as if it had returned `sig_decision_next_decider`.

7 **Returns** If the `guarded` function returns, the `sigguarded` function returns the value returned by `guarded`; if a signal decider initiated recovery, it returns the value returned by `recovery`. If this function is unable to install the guard, for example because of exhaustion of implementation resources, it returns `SIGGUARDED_FAILURE_VALUE`.

Forward references: the `sigdecider_abandon` function (7.14.3.2), requirements for signal handlers and signal deciders (7.14.4).

Add 7.14.3.2: The `sigdecider_abandon` function

1 Synopsis

```
1 #include <signal.h>
2 void sigdecider_abandon(struct stdc_sinfo *rsi);
```

2 **Description** The `sigdecider_abandon` function is async-signal-safe.

3 The `sigdecider_abandon` function declares that the calling decider function will never return through the signal decider machinery (7.14.1): the `decider` function of a call to `sigguarded` (7.14.3.1), or a globally installed decider function. The argument `rsi` shall be the `stdc_sinfo` pointer passed to the calling decider function; otherwise the behavior is undefined.

4 It is undefined behavior to call this function from within a `recovery` function, or from code that is not currently executing within a decider function.

5 It is undefined behavior for the calling decider function to return after calling this function, unless a call of the `sigdecider_abandon_resume` function (7.14.3.3) with the same argument has been made in the same decider function call.

6 If called within a thread-local decider, that decider shall be the decider of the topmost `sigguarded` (7.14.3.1) for the calling thread, and the current guard shall be abandoned. After a guard has been abandoned, a decider returning `sig_decision_call_recovery` for that guard results in undefined behavior. The execution of the interrupted thread is not resumed.

7 If called within a globally installed decider, no further signal deciders are called for the current occurrence of the signal, and the interrupted thread remains suspended.

8 Returns The `sigdecider_abandon` function returns no value.

Add 7.14.3.3: The `sigdecider_abandon_resume` function

1 Synopsis

```
1 #include <signal.h>
2 void sigdecider_abandon_resume(struct stdc_siginfo *rsi);
```

2 Description The `sigdecider_abandon_resume` function is `async-signal-safe`.

3 The `sigdecider_abandon_resume` function undoes a prior call of the `sigdecider_abandon` function (7.14.3.2): the argument `rsi` shall be the same `struct stdc_siginfo` pointer that was passed to a prior call of `sigdecider_abandon` made by the same decider function call on the calling thread; otherwise the behavior is undefined.

4 It is undefined behavior to call this function from a decider function other than the one in which the prior `sigdecider_abandon` call was made, or to undo an abandonment after the decider function from which the abandonment was called has exited.

5 Returns The `sigdecider_abandon_resume` function returns no value.

Add 7.14.4 Requirements for signal handlers and signal deciders

1 The requirements of this subclause apply to a signal handler (7.14.2.3) and to a signal decider (7.14.2.7, 7.14.3.1).

2 When the signal occurs other than as the result of calling the `abort`, `raise`, or `stdc_raise` function, the signal handler or signal decider shall not call any function in the standard library not described as `async-signal-safe` (7.14.1).

3 A signal decider may call the `raise` (7.14.2.4), `stdc_raise` (7.14.2.10), and `abort` (7.25.5.1) functions.

4 The rules of 5.2.2.4 paragraph 6 apply to any object modified by a signal handler or by a signal decider. In particular, the representation of an object modified by a signal decider becomes indeterminate when the decider exits unless the object is of one of the categories specified in 5.2.2.4 paragraph 6; a signal decider can ensure that the representation of an object it modifies does not become indeterminate by calling the `sigfence` macro (7.14.1) with the object as an argument, sequenced after the last modification of the object by the decider.

Modifications in clause 7.17.8.1

In 7.17.8.1: General

NOTE ... so the `atomic_flag` type is the minimum hardware-implemented type needed to conform to this document that is ~~asynchronous-signal-safe~~`async-signal-safe` (7.14.1) and that is expected to be compatible with implementation-specific extensions for shared objects between different program executions.

Modifications in clause 7.25.5.1

In 7.25.5.1: The **abort** function

The **abort** function causes abnormal program termination to occur, unless the signal **SIGABRT** is being caught and the signal handler does not return. Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined. An implementation-defined form of the status *unsuccessful termination* is returned to the host environment by means of the function call **raise(SIGABRT)**.

The **abort** function is async-signal-safe. If a signal decider invoked for the signal raised by this function returns **sig_decision_resume_execution**, abnormal program termination shall occur.

Modifications in clause 7.25.5.5

In 7.25.5.5: The **_Exit** function

The **_Exit** function causes normal program termination to occur and control to be returned to the host environment. No functions registered by the **atexit** function, the **at_quick_exit** function, or signal handlers registered by the **signal** function are called. The status returned to the host environment is determined in the same way as for the **exit** function (7.25.5.4). Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined.

The **_Exit** function is async-signal-safe.

Modifications in clause 7.25.5.7

In 7.25.5.7: The **quick_exit** function

The **quick_exit** function causes normal program termination to occur. No functions registered by the **atexit** function or signal handlers registered by the **signal** function are called. If a program calls the **quick_exit** function more than once, or calls the **exit** function in addition to the **quick_exit** function, the behavior is undefined. If a signal is raised while the **quick_exit** function is executing, the behavior is undefined.

The **quick_exit** function is async-signal-safe.

Modifications in clause 7.30 Threads <threads.h>

In 7.30.1 Introduction

In paragraph 4, after the entry for **tss_t**, insert:

tss_async_signal_safe_t which is a complete object type that holds an identifier for an async-signal-safe thread-specific storage pointer. The identifier is the value set by **tss_async_signal_safe_create** (7.30.6.5), and is not the thread-specific storage pointer itself;

After paragraph 4, insert the following new paragraphs; the existing paragraph 5 is renumbered 7:

5 The `tss_async_signal_safe_attr` structure shall contain at least the following members, in any order. The semantics of the members are expressed in the comments.

```
1 // Create an instance
2 int (*create)(void **dest);
3
4 // Destroy an instance
5 int (*destroy)(void *v);
```

The functions pointed to by the members of this structure shall be reentrant (3.23) and shall not introduce data races.

6 EXAMPLE 1 Use `tss_async_signal_safe_get` to retrieve the thread-specific storage pointer in a signal handler

```
1 #include <threads.h>
2 #include <stdlib.h>
3
4 struct thread_state
5 {
6     int signal_count;
7     ...
8 };
9
10 static int create_state(void **dest)
11 {
12     struct thread_state *state = malloc(sizeof(*state));
13     if (state == nullptr)
14     {
15         return -1;
16     }
17     *dest = state;
18     return 0;
19 }
20
21 static int destroy_state(void *v)
22 {
23     free(v);
24     return 0;
25 }
26
27 static const struct tss_async_signal_safe_attr attr = {
28     .create = create_state,
29     .destroy = destroy_state,
30 };
31
32 static tss_async_signal_safe_t tss;
33
34 /* Called near the beginning of each thread */
35 static void thread_setup(void)
36 {
37     if (tss_async_signal_safe_create(&tss, &attr) == thrd_error ||
38         tss_async_signal_safe_thread_init(tss) == thrd_error)
39     {
40         abort();
41     }
```

```

42 }
43
44 /* Called from within a signal handler or signal decider */
45 static void signal_handler(int signo)
46 {
47     struct thread_state *state = tss_async_signal_safe_get(tss);
48     state->signal_count++;
49     ...
50 }

```

Add 7.30.6.5: The `tss_async_signal_safe_create` function

1 Synopsis

```

1 #include <threads.h>
2 int tss_async_signal_safe_create(tss_async_signal_safe_t *val,
3                                const struct tss_async_signal_safe_attr *attr);

```

2 Description Creates an async-signal-safe thread-specific storage pointer. A copy of `attr` is taken; the copy contains function pointers that are later called to create and destroy instances of the thread-specific storage. The object pointed to by `val` is set to a value that uniquely identifies the newly created instance. Each instance has its own thread-specific storage pointer for each thread, created by `tss_async_signal_safe_thread_init` (7.30.6.7).

3 Returns The `tss_async_signal_safe_create` function returns `thrd_success` if successful; otherwise, `thrd_error` is returned and the object pointed to by `val` is set to an indeterminate representation.

Add 7.30.6.6: The `tss_async_signal_safe_destroy` function

1 Synopsis

```

1 #include <threads.h>
2 int tss_async_signal_safe_destroy(tss_async_signal_safe_t val);

```

2 Description Destroys a previously created async-signal-safe thread-specific storage pointer. All thread-specific storage pointers associated with this instance for which the `destroy` function pointer of the original `attr` argument has not already been called at thread exit (7.30.6.7) are destroyed using that `destroy` function pointer. The `destroy` function pointer of the original `attr` argument is called at most once for each thread-specific storage pointer, and calls of the `destroy` function pointer for different thread-specific storage pointers are not synchronized with one another.

3 It is undefined behavior if this function is called while any thread that has called the `tss_async_signal_safe_thread_init` function for this instance has not yet exited.

4 Returns The `tss_async_signal_safe_destroy` function returns `thrd_success` if successful and `thrd_error` if unsuccessful.

Add 7.30.6.7: The `tss_async_signal_safe_thread_init` function

1 Synopsis

```

1 #include <threads.h>
2 int tss_async_signal_safe_thread_init(tss_async_signal_safe_t val);

```

2 Description Creates the thread-specific storage pointer for the calling thread for the instance identified by `val`, by invoking the `create` function pointer of the original `attr` argument. It is permitted to call this function multiple times on the same thread for the same instance; subsequent calls do not invoke the `create` function pointer again, do not change the thread-specific storage pointer, and return `thrd_success`.

3 At thread exit, it is implementation-defined whether the `destroy` function pointer of the original `attr` argument is called with the thread-specific storage pointer as its argument.

4 The behavior is undefined if this function is called for an instance that has been destroyed by the `tss_async_signal_safe_destroy` function (7.30.6.6).

5 Returns The `tss_async_signal_safe_thread_init` function returns `thrd_success` if successful and `thrd_error` if unsuccessful.

Add 7.30.6.8: The `tss_async_signal_safe_get` function

1 Synopsis

```

1 #include <threads.h>
2 void *tss_async_signal_safe_get(tss_async_signal_safe_t val);

```

2 Description The `tss_async_signal_safe_get` function is `async-signal-safe`.

3 The `tss_async_signal_safe_thread_init` function (7.30.6.7) shall have been called on the same thread for the instance identified by `val`, in which case the thread-specific storage pointer created at that time for that instance is returned; otherwise the behavior is undefined.

4 Returns The `tss_async_signal_safe_get` function returns the thread-specific storage pointer for the calling thread for the instance identified by `val`.

Editorial tasks

- Delete footnote 259 (7.13.3.1): its text was moved to 5.2.2.4, where the floating-point environment and the state of open files are now enumerated in the paragraph itself.
- Delete footnote 262 (7.14.2.3 paragraph 5): its rule (a signal generated by an asynchronous signal handler is undefined behavior) is superseded — signal handlers and signal deciders may now generate signals, for example by calling the `raise`, `stdc_raise`, or `abort` functions, subject to the restrictions of 7.14.4 and paragraph 4 of 7.14.2.3 as amended.
- Footnote renumbering: after the deletions of footnotes 259 and 262, footnotes 260 and 261 are renumbered 259 and 260, and footnotes 262 onward are renumbered 261 onward.
- Paragraph numbering: the inserted paragraphs are numbered with their final numbers in this paper (5.2.2.4 paragraphs 7 through 10; 7.14.1 paragraphs 2 through 18, 20, 22, and 24; 7.30.1 paragraphs 5 and 6), and the displaced base paragraphs are renumbered accordingly: 5.2.2.4

paragraphs 7 through 17 become 11 through 21; 7.14.1 paragraphs 2 through 4 become 19, 21, and 23; 7.30.1 paragraph 5 becomes 7. The paragraphs of the new subsections 7.14.2.1, 7.14.2.2, 7.14.2.5 through 7.14.2.10, 7.14.3.1 through 7.14.3.3, 7.14.4, and 7.30.6.5 through 7.30.6.8 are numbered from 1, except that the forward references of 7.14.2.5 and 7.14.2.7 are unnumbered, as is the forward references line of the base text. The new clause 3 entry is numbered 3.23, and the entries 3.23 through 3.33 are renumbered 3.24 through 3.34.

- Annex B (summary of library functions): add entries for `sigfence`, `SIGGUARDED_FAILURE_VALUE`, `sigemptyset`, `sigfillset`, `sigfillset_synchronous`, `sigfillset_asynchronous_nondebug`, `sigfillset_asynchronous_debug`, `sigaddset`, `sigdelset`, `sigismember`, `siginstall`, `siguninstall`, `signal_decider_create`, `signal_decider_destroy`, `sighandle`, `stdc_raise`, `sigguarDED`, `sigdecider_abandon`, `sigdecider_abandon_resume`, `tss_async_signal_safe_create`, `tss_async_signal_safe_destroy`, `tss_async_signal_safe_thread_init`, and `tss_async_signal_safe_get`.
- Annex J.1 (unspecified behavior): update item (3) for the amended 5.2.2.4 paragraph 6; add the thread on which the signal handler is called (7.14.1), the thread on which signal deciders are invoked for asynchronous category signals (7.14.1), and whether the `sighandle` function ever returns (7.14.2.9).
- Annex J.2 (undefined behavior): renumber the references to the `signal` function and the `raise` function in items (102) through (109) from 7.14.2.1 and 7.14.3.1 to 7.14.2.3 and 7.14.2.4; update item (105) for the amended paragraph 4 of 7.14.2.3 (a signal occurs as the result of calling the `abort`, `raise`, or `stdc_raise` function, and the signal handler calls the `raise` or `stdc_raise` function); update item (107) for the `errno` rule relocated to 7.14.1 (drop the qualifier "other than as the result of calling the `abort` or `raise` function", and change the reference from 7.14.2.1 to 7.14.1); delete item (106), whose object-access rule is no longer undefined behavior (it is now specified by the amended 5.2.2.4 paragraph 6 and by 7.14.4), and add an item for the violation of 7.14.4 (a signal handler or signal decider calls a function in the standard library not described as `async-signal-safe`); delete item (108), whose rule was deleted with footnote 262; and add the undefined behaviors introduced by this paper: calling the `signal` function for a signal number that is activated (7.14.2.3); use of an uninitialized `sigset_t` object (7.14.2.1, 7.14.2.2); misuse of the `siginstall`, `siguninstall`, `signal_decider_create`, `signal_decider_destroy`, `sigguarDED`, `sigdecider_abandon`, and `sigdecider_abandon_resume` functions (7.14.2.5 through 7.14.2.8, 7.14.3.1 through 7.14.3.3); misuse of the `sigfence` macro (7.14.1); and misuse of the `tss_async_signal_safe_create`, `tss_async_signal_safe_destroy`, `tss_async_signal_safe_thread_init`, and `tss_async_signal_safe_get` functions (7.30.6.5 through 7.30.6.8).
- Annex J.3 (implementation-defined behavior): add the signal categories of the standard signals and of implementation-defined signals (7.14.1); which signal numbers are not installed by a call of `siginstall` (7.14.2.5); the default handling performed when a global signal decider returns `sig_decision_call_recovery` (7.14.1); the value of `SIGGUARDED_FAILURE_VALUE` (7.14.1); the additional signals in the sets produced by `sigfillset_synchronous`, `sigfillset_asynchronous_nondebug`, and `sigfillset_asynchronous_debug` (7.14.2.1); the values of the remaining members of the struct `stdc_siginfo` object, for signals other than those raised by the `stdc_raise` function (7.14.1); the values of the `error_code` and `addr` members when `raw_info` is not a null pointer (7.14.2.10); whether a later occurrence of a signal is blocked, discarded, or processed

immediately while an earlier occurrence is being processed (7.14.1); and whether the `destroy` function pointer of the original `attr` argument is called at thread exit (7.30.6.7); and renumber the references to the signal function in items (9), (10), and (16) from 7.14.2.1 to 7.14.2.3.

- Annex J.5.16 (extra arguments for signal handlers): renumber the reference from 7.14.2.1 to 7.14.2.3.
- Index: add the new types, enumeration constants, functions, macros, and the term `reentrant`.
- No change is needed to the future library directions: 7.35.10 already reserves macro names beginning with `SIG_` and an uppercase letter (which covers `SIGGUARDED_FAILURE_VALUE`), and 7.35.20 already reserves function names, type names, and enumeration constants beginning with `tss_` and a lowercase letter (which covers the `tss_async_signal_safe_*` names).

3 References

- [N2471] Douglas, Niall
WG14 N2471/WG21 P2069: Stackable, thread local, signal guards
<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2471.pdf>
- [N3540] Douglas, Niall
Modern signals handling
<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3540.pdf>
- [N3872] Douglas, Niall
Thread-safe signals handling rev 3
<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3872.pdf>
- [N3886] ISO/IEC JTC1/SC22/WG14
Working Draft
<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3886.pdf>
- [N3924] Douglas, Niall
Thread-safe signals handling rev 4
<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3924.htm>