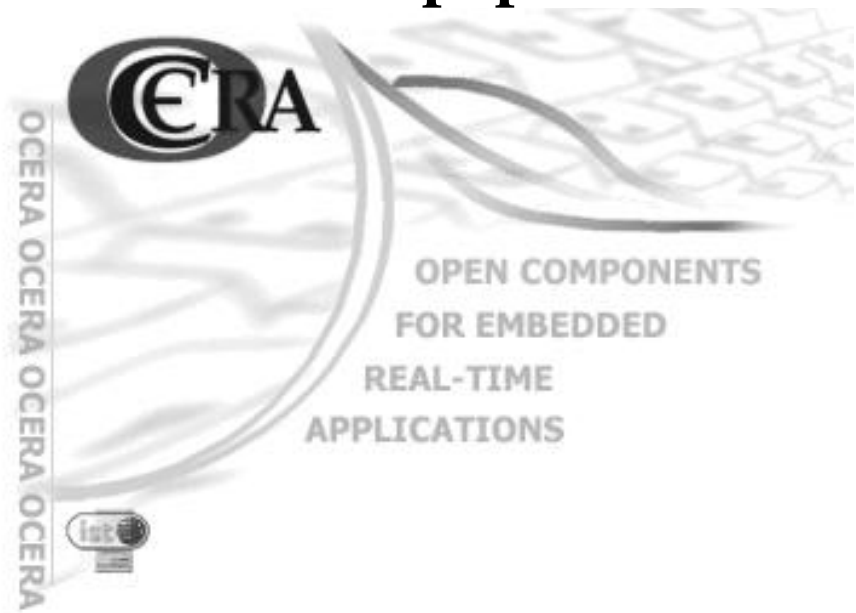
# White paper




# POSIX signals in a threaded environment

OCERA Legal Status.
by Alfons Crespo, Ismael Ripoll, Miguel Masmano, Josep Vidal.

Published 23. April 2004

# Table of Contents

# Document Presentation

### Project Coordinator


| | |
|---|---|
| Organisation: | UPVLC |
| Responsible person: | Alfons Crespo |
| Address: | Camino Vera, 14, 46022 Valencia, Spain |
| Phone: | +34 963877576 |
| Fax: | +34 963877576 |
| Email: | alfons@disca.upv.es |


### Participant List

| Role | Id. | Participant Name | Acronym | Country |
|---|---|---|---|---|
| CO | 1 | Universidad Politécnica de Valencia | UPVLC | E |
| CR | 2 | Scuola Superiore Santa Anna | SSSA | I |
| CR | 3 | Czech Technical University in Prague | CTU | CZ |
| CR | 4 | CEA/DRT/LIST/DTSI | CEA | FR |
| CR | 5 | Unicontrols | UC | CZ |
| CR | 6 | MNIS | MNIS | FR |
| CR | 7 | Visual Tools S.A. | VT | E |

### Document version

| Release | Date | Reason of change |
|---|---|---|
| 1_0 | 03/03/04 | First release |

# Chapter 1. Introduction

The signal mechanism is the method used in POSIX to deliver asynchronous events to a running process. Signals are similar in concept to a hardware interrupt: when a signal is delivered to a process, the normal flow of the process is interrupted and the signal handler function is called, once the handler function finishes the process continues with its original execution flow.

Signals were designed and developed to work in a UNIX heavy process environment, where each process has its own protected memory space, its priority (or round-robing quantum), a single process state, etc. In this execution environment, every process has its own set of signals handlers and blocking mask.

In a system where the execution entities are not processes but threads that share most of their state, the original definition (and operation) of signals is no longer valid. The POSIX standard has tried to extend the signal semantic for threads. The authors of this paper believe that the standard can be improved. A better combination of signals and threads can be defined.

# Chapter 2. POSIX signals with threads

Signals generated outside the process are delivered to the process as a whole. The external source of signals will not be aware whether the process is single threaded or a multi-thread. Signals generated by a specific thread shall be delivered to that particular thread.

Processes can defined which action will be done upon the arrival of each signal. The possible actions are:

- Block(`sigprocmask`): When signals are blocked, they are not immediately delivered to the process. The system stores the signal until the process unblocks it. Real-time signal specification requires that the system has to store all the signals sent to a process and once unblocked all the signals has to be delivered in chronological order.
- Ignore (`SIG_IGN`): Ignored signals will never be delivered to the process, that is, signals are lost.
- Caught (`sigaction`): The process executes the corresponding signal function handler upon the arrival of the signal.
- Default action (`SIG_DFL`): A signal the is not blocked, not ignored and is not caught by the process, kills the process (except some special signals that stops or continues the process).

In a multi-threaded process each thread has its own signal mask (set of blocked signals) that is managed with the function (`pthread_sigmask`). But on the other hand, the action done if the signal is not blocked is shared is global and shared by all threads. That is: all threads share the same signal handler functions; a ignored signal is never delivered to any thread; and the default action will kill the whole process.

When several threads has unblocked a given caught signal (a signal with a signal handler) the POSIX standard do not specify which thread should receive the signal. Most books and tutorials recommends to use one single threads to handle all signals by blocking all signals in all threads except in the dedicated thread that accepts (unblocks) all handled signals.

When a new thread is created (`pthread_create`) The signal state of the new thread shall

be initialised as follows: 1) The signal mask shall be inherited from the creating thread; 2) The set of signals pending for the new thread shall be empty.

If any of the `SIGFPE`, `SIGILL`, `SIGSEGV`, or `SIGBUS` signals are generated while they are blocked, the result is undefined.

The behaviour of a process is undefined after it ignores a `SIGFPE`, `SIGILL`, `SIGSEGV`, or `SIGBUS.`

# Chapter 3. Problems in the current POSIX standard definition

In order to deliver a signal the system has to find a suitable thread, a thread that has not blocked the given signal. This operation may involve a loop to check the state of all the thread masks.

Killing the whole process (all the thread) due to a not handled exception seems to be too drastic. A better choice may be to kill the conflicting thread rather than all the threads.

The default action done by the system when a signal is neither handled nor ignored is to kill the process. If we consider that most embedded and real-time programmers are not experts in POSIX programming[1] and that signals are one of the most obscure and less understand POSIX feature, then it is probable that most programmers will not care about signals and so the application will not contain code to ignore or handle any signal.

# Chapter 4. Implemented signal behavior

There are three different kind of signals depending on how they are generated and handled:

**Hardware triggered signals: `SIGFPE`, `SIGILL`, `SIGSEGV`, or `SIGBUS.`**

Most hardware generated signals are generated by the processor due to a programming error, like floating point exception, illegal instruction or memory access fault. These kind of hardware events should be immediately delivered and attended by the offending thread. **Hardware generated signals can neither be ignored nor blocked**. Signals triggered in response to a processor fault must be handled properly, otherwise the processor will enter into an endless loop which will hang the system. Next is an example illustrates the problem:

Suppose that a thread tries to execute a floating point division and the divisor is zero. Then the processor raises hardware exception and the signal handler is called. The signal handler must change the state of the thread to a safe state (for example changing the value of the divisor variable, changing the processor program counter register to go to a valid instruction, or via `longjmp()` to go to a previous safe state). If the signal would be ignored or the signal handler were an empty function (a function that did not solved the problem) then the same instruction with the same parameters would be called on return from exception handler, which would raise again the same hardware exception. At this time, the system looping executing the same faulty instruction.

1 The Mars Pathfinder programmers didn't know what is the priority inversion problem.

The default action is to kill offending thread. Please, note that POSIX requires to kill the whole process.

**User generated signals**: `SIGURS1`, `SIGUSR2`, `SIGRTMIN`...`SIGRTMAX`.

These are general purpose signals that can be freely used by user threads. They can be blocked, ignored and caught.

To avoid delivery randomness and to speedup implementation, **user generated signal handlers are "owned" by the thread that installed it last**. That is, when a thread installs a signal handler, only this thread can receive the signal. If the owner thread blocks the signal then the signals is blocked (regardless the blocking mask of other threads). The function pthread_kill will only send the signal if the target thread is the last one that installed the signal handler. The addition of the signal owner greatly simplifies the code to deliver threads because the sigaction structure directly identifies the receiving thread.

Initially, the default action is to ignore them (`SIG_IGN`). We think that this default action is a more safe behaviour.

The default action is to kill the thread that received the signal.

**System generated signals**: `WAKEUP`, `CANCEL`, `SUSPEND`, `TIMER`, `NULL`.

These signals are used to control the running state of the threads. These are used internally by the RTLinux scheduler, but also can be used by user threads. The action of this signals is defined by the system and can not be redefined by the user: installing a handler, ignoring or blocking them.

Newly created threads do not inherit the signal blocking mask of it creator thread. All threads start with an empty (no signal is blocked) signal mask.