Annex J
(informative)
Interrupt Control Considerations

J.1 Introduction

In the area of realtime systems there often exist devices with very simple interfaces, that can or should be operated without a full-fledged driver within the operating system: what is needed is the ability to access the control and status registers of the interface, and the ability to capture the interrupts that are generated and have the application program handle them.

The functions defined in this section allow a process or thread to capture an interrupt, to block awaiting the arrival of an interrupt, and to protect critical sections of code which are contended for by a user-written interrupt service routine. Capturing an interrupt involves registering a user-written interrupt service routine (ISR). The introduction of user-written ISRs does not make application programs completely portable, but at least establishes a reference model that allows programs to be rehosted without completely subverting their logic, and confines non-portable code to specified modules.

A single threaded process, or a process in an implementation which does not support threads, is considered to consist of a single thread of control; in this case, the term *thread* in the description of the interfaces in this section shall refer to this single thread of control.

J.2 Definitions

⇒ **2.2.2 General Terms** *Add the following definitions, in the right sorted order:*

J.2.0.1 interrupt:

(1) The suspension of a process to handle an event external to the process. *Syn:* **interruption**. *See also:* **interrupt latency; interrupt mask; interrupt priority; interrupt service routine**. (2) To cause the suspension of a process. (3) Loosely, a hardware interrupt request.

J.2.0.2 interruption:

See: **interrupt**.

J.2.0.3 interrupt latency:

The delay between a computer system's receipt of a hardware interrupt request and its handling of the request. *See also:* **interrupt priority**.

J.2.0.4 interrupt mask:

A mask used to enable or disable interrupts by retaining or suppressing bits that represent interrupt requests.

J.2.0.5 interrupt priority:

The importance assigned to a given interrupt request. This importance determines whether the request will cause suspension of the current instructions and, if there are several outstanding interrupt requests, which will be handled first.

J.2.0.6 interrupt request:

An external or other hardware input that requests the execution of the current instruction flow be suspended to permit execution of an ISR.

J.2.0.7 interrupt service routine:

A routine that responds to interrupt requests by storing the contents of critical registers, performing the processing required by the interrupt request, and then, if no higher priority process is eligible to run, restoring the register contents and restarting the interrupted process.

J.2.0.8 ISR:

Abbreviation for interrupt service routine.

J.3 Concepts

The following opaque data type is defined by the implementation in the header `<intr.h>`.

intr_t

> Identifies the source of a hardware interrupt in an implementation-defined manner. The implementation shall also supply some means of obtaining legal values of type intr_t, that represent supported interrupt sources a process may connect to.

The header `<intr.h>` shall define the following symbols:

`POSIX_INTR_PRI_LOWEST`
The lowest available interrupt priority.

`POSIX_INTR_PRI_LOW`
A low available interrupt priority.

`POSIX_INTR_PRI_MED_LOW`
A medium low available interrupt priority.

`POSIX_INTR_PRI_MEDIUM`
A medium available interrupt priority.

`POSIX_INTR_PRI_MED_HIGH`
A medium high available interrupt priority.

`POSIX_INTR_PRI_HIGH`
A high available interrupt priority.

`POSIX_INTR_PRI_HIGHEST`
The highest available interrupt priority.

J.4 Interrupt Control Functions

J.4.1 Associate a User-Written ISR with an Interrupt

Functions: *posix_intr_associate*(), *posix_intr_disassociate*(), *posix_intr_lock*(), *posix_intr_unlock*().

J.4.1.1 Synopsis

```
#include <intr.h>

int posix_intr_associate (intr_t intr,
                  int (*intr_handler)(void *area),
                  volatile void *area, size_t areasize);

int posix_intr_disassociate (intr_t intr,
                  int (*intr_handler)(void *area));

int posix_intr_lock (intr_t intr);

int posix_intr_unlock (intr_t intr);
```

J.4.1.2 Description

If the Interrupt Control option is supported: B

If the number of ISRs currently connected to *intr* is less than {_POSIX_-INTR_CONNECT_MAX} then the *posix_intr_associate*() function shall associate the given user-written ISR *intr_handler* with a given interrupt *intr*. The interrupt service routine shall conform to the following function prototype:

```
int int_handler (void *area);
```

After executing the *posix_intr_associate*() function, the issuing thread shall become connected to the given interrupt. The system shall call *intr_handler*s in the reverse order that the ISRs were registered until one of the *intr_handler*s returns a code signifying that the interrupt has been handled. The most recently registered ISR is thus called first.

Although an ISR is initially located in a process' address space, it executes in an implementation-defined context, subject to a number of implementation-defined restrictions. It is unspecified what restrictions may be imposed by an implementation.

The execution context of an ISR may have an address space different from the normal process' space, so any data areas accessed by the ISR must be clearly identified as such. The argument *area* identifies a communication region, whose size is *areasize*, where the ISR and the thread shall be able to exchange data; when the ISR is called, it shall receive the address of the communication region as its first argument. Implementations may have additional arguments of implementation-defined types.

The return code which is returned by the ISR determines whether the interrupt has been handled by this ISR, and whether the thread that registered the ISR should be notified of the successful handling of this interrupt.

An interrupt handler wakes up a thread by posting one of the *notify* return codes, which causes a thread waiting in a corresponding *posix_intr_timedwait*() to unblock. No other ISR-to-thread notification mechanism is specified.

Notification is described in clause C.3.2. Possible return codes (defined in `<intr.h>`) include:

`POSIX_INTR_HANDLED_NOTIFY`
: The ISR handled this interrupt, and the thread that registered the ISR should be notified that the interrupt occurred.

`POSIX_INTR_HANDLED_DO_NOT_NOTIFY`
: The ISR handled this interrupt, but the thread that registered the ISR should not be notified that the interrupt occurred.

`POSIX_INTR_NOT_HANDLED`
: The ISR did not handle this interrupt; if there are other ISRs connected to this interrupt, then the next ISR should be called.

The *posix_intr_disassociate*() function shall cancel any existing association between the interrupt *intr* and the ISR *interrupt_handler*.

If a thread calls *posix_intr_disassociate*() and the thread does not have the specified ISR registered for the specified interrupt, the *posix_intr_disassociate*() function shall fail.

If a thread has connected one or more user-written ISRs to a given interrupt *intr*, then that thread calling the *posix_intr_lock*() function shall prevent the system from calling those ISRs or notifying the connected thread until delivery is re-enabled by means of the *posix_intr_unlock*() function, thus allowing the thread to perform operations in an atomic way with respect to the ISR; if an ISR is executing when the thread that connected that ISR to an interrupt calls *posix_intr_lock*(), then the *posix_intr_lock*() function shall not return until that ISR has completed; the methods used by an implementation to obtain these results are implementation-defined. To allow implementation using a hardware disable-interrupts instruction, *posix_intr_lock*() need not be a cancellation point. It is implementation-defined whether locking an ISR causes other ISRs to be locked. It is implementation-defined whether interrupts that arrive while an interrupt is locked are queued or discarded. It is implementation-defined whether registration under lock is supported.

It is implementation-defined whether these functions require an appropriate privilege from the calling thread.

It is implementation-defined whether ISRs remain registered after the registering thread terminates. It is implementation-defined which POSIX operations, if any, may be executed from an interrupt handler.

Otherwise:

Either the implementation shall support the *posix_intr_associate*(), *posix_intr_disassociate*(), *posix_intr_lock*(), and *posix_intr_unlock*() functions as described above, or these functions shall not be provided. B

J.4.1.3 Returns

Upon successful completion, *posix_intr_associate*(), *posix_intr_disassociate*(), *posix_intr_lock*(), and *posix_intr_unlock*() shall return zero. Otherwise an error code shall be returned.

J.4.1.4 Errors

If any of the following conditions occur, the *posix_intr_associate*(), *posix_intr_disassociate*(), *posix_intr_lock*(), and *posix_intr_unlock*() function shall return the corresponding non-zero error code:

[EINVAL] The *intr* argument does not identify a supported interrupt that can be connected to a user-specified ISR. B

B

[EPERM] The calling thread does not have an appropriate privilege to call this function and the implementation requires such a privilege.

If the following condition occurs, it is implementation-defined whether the *posix_intr_associate*() function shall return the corresponding non-zero error code:

[EAGAIN] The interrupt identified by the *intr* argument currently has the implementation-defined maximum number of ISRs connected.

If the following condition occurs, the *posix_intr_disassociate*(), *posix_intr_lock*(), and *posix_intr_unlock*() functions shall shall return the corresponding non-zero error code:

[ENOISR] The thread has not registered an ISR for the given interrupt.

If the following condition is detected, the *posix_intr_associate*() function shall return the corresponding non-zero error code:

[EINVAL] The arguments *area* and/or *areasize* and/or *intr_priority* are invalid for the implementation.

J.4.1.5 Cross-References

J.4.2 Await Interrupt Notification

Function: *posix_intr_timedwait*().

J.4.2.1 Synopsis

```
#include <intr.h>
int posix_intr_timedwait (int flags, const struct timespec *timeout);
```

J.4.2.2 Description

If the Interrupt Control option is supported: B

The *posix_intr_timedwait*() function causes the calling thread to block until notified that an interrupt has occurred. If an interrupt notification was delivered to the calling thread prior to the call to the *posix_intr_timedwait*() function, and this notification has not previously caused a call to the *posix_intr_timedwait*() function to be unblocked, then the calling thread is not blocked and instead the *posix_intr_timedwait*() function returns immediately.

The input argument *flags* contains only implementation-defined input values.

If the value of the *timeout* input argument is non-null, the wait for an interrupt to occur shall be terminated when the specified timeout period expires.

If the *timeout* input argument is null, the wait is terminated only by the interrupt.

The timeout expires after the interval specified by *timeout* has elapsed since the wait began. If the Timers option is supported, the timeout is based on the CLOCK_REALTIME clock; if the Timers option is not supported, the timeout is based on the system clock as returned by the POSIX.1 function *time*(). The resolution of the timeout is determined by the resolution of the B B clock that it uses.

Under no circumstance will the function fail with a timeout if the interrupt notification occurred prior to the *posix_intr_timedwait*() call. The validity of the *timeout* argument need not be checked if the interrupt notification occurred prior to the *posix_intr_timedwait*() call. Invocation of *posix_intr_timedwait*() shall implicitly release an *posix_intr_lock*().

This function shall fail if no ISR is currently registered by the calling thread.

Otherwise:

Either the implementation shall support the *posix_intr_timedwait*() function as described above or this function shall not be provided. B

J.4.2.3 Returns

Upon successful completion, *posix_intr_timedwait*() shall return zero. Otherwise an error code shall be returned.

J.4.2.4 Errors

If any of the following conditions occur, the *posix_intr_timedwait*() function shall return the corresponding non-zero error code.

[EINVAL]	The thread would have blocked, but the timeout argument specified a nanoseconds value less than zero or greater than or equal to 1000 million.
[EINTR]	A signal interrupted this function.
[ENOISR]	The thread has not registered an ISR for the given interrupt. B
[ETIMEDOUT]	The interrupt notification was not received before the specified timeout expired.

J.4.2.5 Cross-References

J.5 Rationale for Interrupt Control

J.5.1 The Interrupt Model

J.5.1.1 Background

The purpose of this interface is to allow connection of non-standard interrupt-generating hardware in a standard way.

Such hardware, when enabled, may generate a continuous stream of interrupts, not following the request-response model typical of common I/O devices such as disks and tapes. A typical example would be a radar antenna generating a stream of azimuth-change and north-crossing interrupts as the antenna rotates. Another example would be the stream of angle resolver pulses from the joints of a robot.

Many hardware architectures have fewer interrupts than peripheral devices, requiring interrupts to be shared. In such cases, more than one ISR is invoked by a given interrupt, and the identity of the device or devices generating an interrupt must be established by polling the devices connected to that interrupt.

Some kinds of non-standard hardware generates multiple and related streams of interrupts which should all be handled by a single thread. Again, a radar antenna, with its two interrupt streams, provides a classic example.

J.5.1.2 The Model

To each suitable interrupt one may attach zero or more interrupt service routines (ISRs). When the interrupt is activated, these ISRs are executed in reverse order of registration; that is, last registered, first executed.

Each ISR must first poll its device to determine if that device is asserting the interrupt. If not, the ISR returns immediately with a return value signifying that the interrupt was not handled. If the ISR's device is asserting the interrupt, the ISR does whatever is needed (a matter of local design), and returns with a return value signifying that the interrupt has been handled, and further that the registering thread should or should not be notified (awakened). The decision to notify or not is made by the user-written ISR code, but notification is performed by the vendor supplied code which invokes these ISRs.

Note that there are no direct error returns from these ISRs to the invoking kernel; device error handling is performed within the ISRs and reported as needed in the communications area.

The first ISR to handle an interrupt consumes it, preventing execution of ISRs in the remainder of the list. If there are two devices simultaneously asserting the interrupt, the second device will continue to assert the interrupt, forcing re-traversal of the ISR chain, from the top.

If no ISR in an interrupt chain claims an interrupt, the behavior is unspecified. Typically, unclaimed interrupts are simply ignored.

Multiprocessors are implicitly handled, depending on the underlying hardware. In many systems, one associates each device with a processor that will handle its interrupts. In other systems, such as recent ones from Sun Microsystems, the ISR (a kernel thread) runs on any available processor.

Interrupt handling has no effect on scheduling queues, except that an interrupt can result in the unblocking of a process whose priority exceeds that of any that were executing when that interrupt arrived.

J.5.1.3 Registration

ISRs become connected to interrupts by registration, and disconnected by de-registration. A thread registering an ISR provides four pieces of information: the address of the ISR code, the address and size of the communication region of memory (used for data shared by ISR and the registering thread), the interrupt ID, and (implicitly) the thread ID of the registering thread.

A thread may have multiple ISRs registered, and each ISR may generate notification requests. The thread waits for any and all such notifications in one place, using the *posix_intr_timedwait*() function. In such cases, the ISRs must place whatever information is needed for the application to tell one device from another in their respective communication regions. It is necessary to have precisely one wait-point to prevent deadlocks due to interrupts arriving in an unexpected order.

To implement smooth and leakproof transfer of interrupt traffic from one ISR to another, it is sufficient to register the new ISR before deregistering the old ISR. For the short period of time that there are two ISRs for one device, the most recently registered ISR will consume all the traffic, allowing the old ISR to be deregistered at leisure.

The communication region is an area of memory that is visible both to the ISR during an interrupt and to the registering thread at all (other) times. It is the user's responsibility to allocate a suitable region, perhaps by the use of the Typed Memory facilities provided by the implementation. Shared access to the communications region by both thread and ISR is mediated using *posix_intr_lock*() and *posix_intr_unlock*().

J.5.2 Portability

Although interrupt handling isn't entirely portable, there is still profit in standardizing the interrupt control interface. First is the implicit standardization of core functionality. Second is programmer portability. Third is that interrupt handling code can follow the hardware device for which it was written. All of this is supported by a great deal of embedded and/or realtime (often non-UNIX) system practice. The resulting modularization and isolation of non-portable code also aids portability.

The model which repeatedly emerges from existing practice involves two facilities. First, the application should be able to arrange for an interrupt to notify a process or thread when the interrupt occurs. Second, the application should also be able to provide an interrupt handler or interrupt service routine (ISR) to immediately service each interrupt occurrence without a time-consuming process context switch. Users of this model may require one or the other or both of these facilities. Clause C.4.9 shows an example using the interfaces in this section to implement an application which conforms to this model.

J.5.3 Existing Practice

Most operating systems designed specifically to support realtime applications provide similar services for application interrupt handling and control. Ada language runtime environments which support interrupt handling also provide similar services. The fact that UNIX systems have not typically provided such services is indicative of the non-realtime heritage of UNIX. See also **Portability** above.

J.5.3.1 Interrupt Specification

There is no portable way to specify an interrupt. Therefore, these interfaces must rely on an opaque type, intr_t, to identify a specific interrupt. Each implementation which supports the Interrupt Control option must provide a mechanism for B obtaining objects of this type which identify all user accessible interrupts supported by the implementation. Such mechanisms may include constant objects of type intr_t, and functions, macros, typecasts which involve implementation specific interrupt identification procedures, returning objects of type intr_t. Once such an object has been associated with an interrupt of interest, this interrupt may be accessed via the portable interfaces in this section. The implementation-specific mechanisms cannot and will not be standardized herein.

J.5.4 Interrupt Latency

Connecting an interrupt to a POSIX realtime signal, while performing the necessary function of initiating application processing, often cannot alone guarantee adequate and timely response to rapid and/or time critical interrupts. There are two reasons for this. First, the sequence of execution of POSIX processes and/or threads is a function of the CPU scheduling policy, not the relative urgency of various interrupts; it is possible to work within the constraints of the scheduling policy, but interrupt response and handling time is still likely to be non-deterministic. Second, even if the notified process becomes the running process immediately upon interrupt occurrence, the overhead necessary for process context switching is unlikely to support interrupt latency requirements in the ten to hundred microseconds range, or for interrupts occurring at a rapid rate. Also, many interrupting devices require execution of special code to respond to and deassert each and every interrupt.

The purpose of *posix_intr_associate*(), therefore, is to provide a path to first level interrupt servicing code whose latency is a function only of other interrupts occurring at or near the same time. Such code is intended to deal with the high speed portion of the interrupt handling. It should perform only functions which cannot be postponed until they are handled by a normally scheduled process; it typically executes with at least the interrupt of interest locked out, and therefore need not be reentrant. To achieve this low latency, it is expected that an implementation will bind ISR code as closely as possible to the hardware interrupt mechanism. The issues of response time (timeliness) and mutual exclusion (ISR, non-reentrant) are independent.

J.5.5 Relationship To Realtime Profiles

Handling of interrupts by user written code is typical in applications conforming to the two smaller realtime profiles from IEEE 1003.13, the *Minimal* and *Control* profiles. Systems which require these profiles typically utilize neither virtual memory nor an architecture supporting separate user and kernel modes. In such systems, interrupt servicing via the ISR model is easily implemented as simple procedure call in the context of the single executing process.

For the two more complex realtime profiles in IEEE 1003.13, the *Dedicated* and *Multi-Purpose* profiles, process and kernel separation is of concern to most implementations, and the ISR model becomes somewhat more difficult to implement. Although, systems requiring these profiles are far more likely to utilize full scale device drivers integrated or loaded into the kernel in an implementation specified manner, some may require the interfaces of this section, especially *posix_intr_associate*().

J.5.6 Limitations Imposed on ISR Code

An ISR needs to be able to execute without incurring unpredictable delays; it must complete in a timely manner. For this reason an ISR is normally restricted to kernel level calls since the boundary of kernel level calls' behavior is strongest. Invo- B cation of an ISR has the implicit effect of a call to *posix_intr_lock*() to block further invocations of that same ISR. Consequently, ISRs need not be reentrant. There is, therefore, no free choice on the interfaces that can be used and caution needs to be exercised in selecting interfaces. Invocation of POSIX interfaces within an ISR cannot be considered consistent with the timeliness requirement of an ISR.

The notion of the "current process" is erroneous within an ISR since it may execute in the context of the kernel or of an arbitrary process. Since most POSIX interfaces may query or alter the state of the "current process", their use within ISR code is not only untimely, but erroneous.

It is expected that anyone wanting to write a user-written ISR is familiar with how to write I/O drivers for their particular system. Mistakes in an ISR can and will crash the system. Caution is advised.

J.5.7 Handler Specification

The working group is undecided on how an interrupt handler (not executing in process context) may be specified to the *posix_intr_associate*() function. Since the handler code may need to ultimately run in a kernel address mapping different from that of the process doing the registration, specifying the virtual address of non-relocatable code would appear problematic. The extent of the handler (i.e. what functions it can call or data it can access) is also unclear in the virtual memory case. Finally, at least one example of existing practice requires that a handler be dynamically loaded into the kernel from a file; in this case, a pathname would be required as the handler specification.

The working group opposes converting the handler specification argument of *posix_intr_associate*() to an opaque type. It has been suggested that a pathname be used for this argument, and this pathname could identify a memory resident code segment on systems such as those conforming to the minimal realtime profile (like the pathname interpretation for *exec*() or *posix_spawn*() on such systems). Unless the working group can agree on a workable method of handler specification, the original method; specifying a pointer to a function, will be retained.

J.5.8 posix_intr_timedwait() versus Sigwait()

Why doesn't *posix_intr_timedwait*() precisely follow the existing *sigwait*() model? This was discussed in the working group, which for simplicity decided not to attempt to map hardware interrupts onto signals, as the underlying models are only in general similar, differing greatly in the details. The great mass of existing code renders the signals model and *sigwait*() API essentially immutable, so it was decided to make *posix_intr_timedwait*() independent of *sigwait*(), and as simple as possible, to reduce the burden on implementors, and for performance and predictability.

J.5.9 Interrupt Specification

Several examples may serve to clarify the use of intr_t:

Figure J-1 – intr_t Examples

An implementation supplied constant:

```
posix_intr_associate (IVEC_240, &handler, &data, sizeof(data) );
```

Asking a device how it will interrupt:

```
posix_devctl (fd, GET_INTERRUPT_ID, &interrupt, sizeof(intr_t), NULL);
posix_intr_associate (interrupt, &handler, &data, sizeof(data) );
```

Simple cast of known interrupt vector:

```
posix_intr_associate ((intr_t)240, &handler, &data, sizeof(data) );
```

J.5.10 Application Example

The following C-Language program fragment demonstrates usage of the interfaces in this section:

Figure J-2 − An Interrupt Control Application Fragment

```
/* Collect digitized data to a file - The A to D converter runs */
/* at 30khz sampling rate, has a 256-sample circular buffer, */
/* and interrupts after each 128 samples.                     */

queue my_queue;  /* statically allocated */

main()
        {
        int my_handler(queue *my_queue);
        posix_intr_associate(INTR_240, &my_handler, &my_queue, sizeof(my_que

        start_A_to_D();
        while (TRUE)
                {
                int localbuffer[128];
                if (posix_intr_timedwait(0, A_to_D_timeout()) == 0)
                        {
                        posix_intr_lock(INTR_240);
                        while (dequeue(&my_queue, localbuffer))
                                {
                                posix_intr_unlock(INTR_240);
                                fwrite(localbuffer, 1, sizeof(localbuffer),
                                       stdout);
                                posix_intr_lock(INTR_240);
                                }
                        posix_intr_unlock(INTR_240);
                        }
                else
                        /*handle errors, including timeout*/
                }
        }

int my_handler(queue *my_queue)
        {
        int localbuffer[128];
        read_A_to_D(localbuffer);
        enqueue(my_queue, localbuffer);
        return POSIX_INTR_HANDLED_NOTIFY;
        }
```