# FIFO-based Event Channel ABI

David Vrabel <david.vrabel@citrix.com>

Draft A

## Contents

# 1 Introduction

## 1.1 Purpose

Xen uses event channels to signal events (interrupts) to (fully or partially) paravirtualized guests. The current event channel ABI provided by Xen only supports up-to 1024 (for 32-bit guests) or 4096 (for 64-bit guests) event channels. This is limiting scalability as support for more VMs, VCPUs and devices is required.

The existing ABI does not easily allow events to have different priorities. Current Linux kernels prioritize the timer event by special casing this but this is not generalizable to more events. Event priorities may be useful for prioritizing MMIO emulation requests over bulk data traffic (such as network or disk).

This design replaces the existing event channel ABI with one that:

- is scalable to more than 100,000 event channels, with scope for increasing this further with minimal ABI changes.
- allows guests to use up-to 16 different event priorities.

## 1.2 System Overview

[FIXME: diagram showing Xen and guest and shared memory block for events?]

## 1.3 Design Map

A new event channel ABI requires changes to Xen and the guest kernels.

### 1.4 References

[FIXME: link to alternate proposal?]

## 2 Design Considerations

### 2.1 Assumptions

- Atomic read-modify-write of 32-bit words is possible on all supported platforms. This can be with a linked-load / store-conditional (e.g., ARMv8's ldrx/strx) or a compare-and-swap (e.g., x86's cmpxchg).

### 2.2 Constraints

- The existing ABI must continue to be useable. Compatibilty with existing guests is mandatory.

### 2.3 Risks and Volatile Areas

- Should the 3-level proposal be merged into Xen then this design does not offer enough improvements to warrant the cost of maintaining three different event channel ABIs in Xen and guest kernels.
- The performance of some operations may be decreased. Specifically, re-triggering an event now always requires a hypercall.

## 3 Architecture

### 3.1 Overview

The event channel ABI uses a data structure that is shared between Xen and the guest. Access to the structure is done with lock-less operations (except for some less common operations where the guest must use a hypercall). The guest is responsible for allocating this structure and registering it with Xen during VCPU bring-up.

Events are reported to a guest's VCPU using a FIFO queue. There is a queue for each priority level and each VCPU.

Each event has a *pending* and a *masked* bit. The pending bit indicates the event has been raised. The masked bit is used by the guest to prevent delivery of that specific event.

# 4 High Level Design

## 4.1 Shared Event Data Structure

The shared event data structure has a per-domain *event array*, and a per-VCPU *control block*.

- *event array*: A logical array of event words (one per event channel) which contains the pending and mask bits and the link index for next event in the queue.

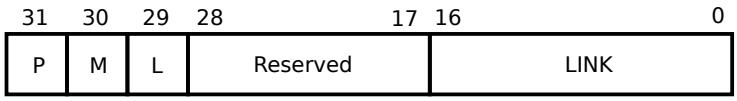


Figure 1: Event Array Word

- *control block*: This contains the head and tail indexes for each priority level. Each VCPU has its own control block and this is contained in the existing `struct vcpu_info`.

The pages within the event array need not be physically nor virtually contiguous, but the guest or Xen may make the virtually contiguous for ease of implementation. e.g., by using vmap() in Xen or vmalloc() in Linux. Pages are added by the guest as required by the number of bound event channels.

Only 17 bits are currently defined for the LINK field, allowing $2^{17}$ (131,072) events. This limit can be trivially increased without any other changes to the ABI. Bits [28:17] are reserved for future expansion or for other uses.

## 4.2 Event State Machine

Event channels are bound to a domain using the existing ABI.

A bound event may be in one of three main states.

| State | Abbrev. | Meaning |
|---|---|---|
| BOUND | B | The event is bound but not pending. |
| PENDING | P | The event has been raised and not yet acknowledged. |
| LINKED | L | The event is on an event queue. |

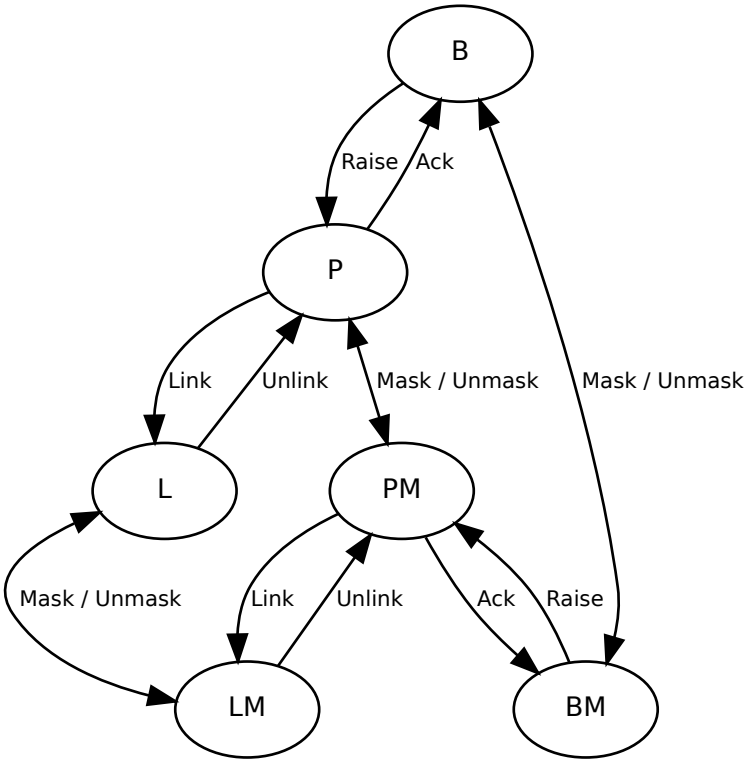


Figure 2: Event State Machine

Additionally, events may be UNMASKED or MASKED (M).

The state of an event is tracked using 3 bits within the event word. the M (masked), P (pending) and L (linked) bits. Only state transitions that change a single bit are valid.

### 4.3 Event Queues

The event queues use a singly-linked list of event array words (see figure 1 and 3).

Each event queue has a head and tail index stored in the control block. The head index is the index of the first element in the queue. The tail index is the last element in the queue. Every element within the queue has the L bit set.

The LINK field in the event word indexes the next event in the queue. LINK is zero for the last word in the queue.

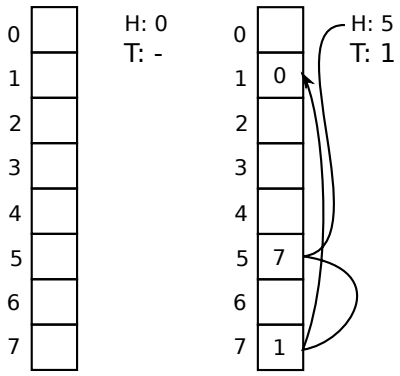


Figure 3: Empty and Non-empty Event Queues

The queue is empty when the head index is zero (zero is not a valid event channel).

## 4.4 Hypercalls

Four new EVTCHNOP hypercall sub-operations are added:

- `EVTCHNOP_init`
- `EVTCHNOP_expand`
- `EVTCHNOP_set_priority`
- `EVTCHNOP_set_limit`

### 4.4.1 `EVTCHNOP_init`

This call initializes a single VCPU's event channel data structures, adding one page for the event array.

A guest should call this during initial VCPU bring up (and on resume?).

```
struct evtchnop_init {
    uint32_t vcpu;
    uint64_t array_pfn;
};
```

| Field | Purpose |
|---|---|
| `vcpu` | [in] The VCPU number. |
| `array_pfn` | [in] The PFN or GMFN of a page to be used for the first page of the event array. |

| Error code | Reason |
|---|---|
| EINVAL | `vcpu` is invalid or already initialized. |
| EINVAL | `array_pfn` is not a valid frame for the domain. |
| ENOMEM | Insufficient memory to allocate internal structures. |

#### 4.4.2 `EVTCHNOP_expand`

This call expands the event array for a VCPU by appended an additional page.

A guest should call this for all VCPUs when a new event channel is required and there is insufficient space in the current event array.

It is not possible to shrink the event array once it has been expanded.

```
struct evtchnop_expand {
    uint32_t vcpu;
    uint64_t array_pfn;
};
```

| Field | Purpose |
|---|---|
| `vcpu` | [in] The VCPU number. |
| `array_pfn` | [in] The PFN or GMFN of a page to be used for the next page of the event array. |

| Error code | Reason |
|---|---|
| EINVAL | `vcpu` is invalid or already initialized. |
| EINVAL | `array_pfn` is not a valid frames for the domain. |
| EINVAL | The event array already has the maximum number of pages. |
| ENOMEM | Insufficient memory to allocate internal structures. |

#### 4.4.3 `EVTCHNOP_set_priority`

This call sets the priority for an event channel. The event must be unbound.

A guest may call this prior to binding an event channel. The meaning and the use of the priority are up to the guest. Valid priorities are 0 - 15 and the default is 7.

```
struct evtchnop_set_priority {
    uint32_t port;
    uint32_t priority;
};
```

| Field | Purpose |
|---|---|
| `port` | [in] The event channel. |
| `priority` | [in] The priority for the event channel. |

| Error code | Reason |
|---|---|
| EINVAL | `port` is invalid. |
| EINVAL | `port` is currently bound. |
| EINVAL | `priority` is outside the range 0 - 15. |

#### 4.4.4 `EVTCHNOP_set_limit`

This privileged call sets the maximum number of event channels a domain can bind. The default for dom0 is unlimited. Other domains default to 1024 events (requiring only a single page for their event array).

```
struct evtchnop_set_limit {
    uint32_t domid;
    uint32_t max_events;
};
```

| Field | Purpose |
|---|---|
| `domid` | [in] The domain ID. |
| `max_events` | [in] The maximum number of event channels that may be bound to the domain. |

| Error code | Reason |
|---|---|
| EINVAL | `domid` is invalid. |
| EPERM | The calling domain has insufficient privileges. |

### 4.5 Memory Usage

#### 4.5.1 Event Arrays

Xen needs to map every domains' event array into its address space. The space reserved for these global mappings is limited to 1 GiB on x86–64 (262144 pages) and is shared with other users.

It is non-trivial to calculate the maximum number of VMs that can be supported as this depends on the system configuration (how many driver domains etc.) and VM configuration. We can make some assuptions and derive an approximate limit.

Each page of the event array has space for 1024 events ($E_P$) so a regular domU will only require a single page. Since event channels typically come in pairs, the upper bound on the total number of pages is 2 × number of VMs.

If the guests are further restricted in the number of event channels ($E_V$) then this upper bound can be reduced further.

The number of VMs ($V$) with a limit of $P$ total event array pages is:

$$V = P \div \left(1 + \frac{E_V}{E_P}\right)$$

Using only half the available pages and limiting guests to only 64 events gives:

$$\begin{aligned} V &= (262144/2) \div (1 + 64/1024) \\ &= 123 \times 10^3 \text{ VMs} \end{aligned}$$

Alternatively, we can consider a system with $D$ driver domains, each of which requires $E_D$ events, and a dom0 using the maximum number of pages (128).

$$V = P - \left(128 + D \times \frac{E_D}{E_P}\right)$$

With, for example, 16 driver domains each using the maximum number of pages:

$$\begin{aligned} V &= (262144/2) - (128 + 16 \times \frac{2^{17}}{1024}) \\ &= 129 \times 10^3 \text{ VMs} \end{aligned}$$

In summary, there is space to map the event arrays for over 100,000 VMs. This is more than the limit imposed by the 16 bit domain ID (∼32,000 VMs).

#### 4.5.2 Control Block

With $L$ priority levels and two 32-bit words for the head and tail indexes, the amount of space ($S$) required in the `struct vcpu_info` for the control block is:

$$\begin{aligned} S &= L \times 2 \times 4 \\ &= 16 \times 2 \times 4 \\ &= 128 \text{ bytes} \end{aligned}$$

There is more than enough space within `struct vcpu_info` for the control block while still keeping plenty of space for future use.

# 5 Low Level Design

In the pseudo code in this section, all memory accesses are atomic, including those to bit-fields within the event word.

These variables have a standard meaning:

| Variable | Purpose |
|---|---|
| E | Event array. |
| p | A specific event. |
| H | The head index for a specific priority. |
| T | The tail index for a specific priority. |

These memory barriers are required:

| Function | Purpose |
|---|---|
| mb() | Full (read/write) memory barrier |

## 5.1 Raising an Event

When Xen raises an event it marks it pending and (if it is not masked) adds it tail of event queue.

```
E[p].pending = 1
if not E[p].linked and not E[n].masked
    E[p].linked = 1
    E[p].link = 0
    mb()
    if H == 0
        H = p
    else
        E[T].link = p
    T = p
```

Concurrent access by Xen to the event queue must be protected by a per-event queue spin lock.

## 5.2 Consuming Events

The guests consumes events starting at the head until it reaches the tail. Events in the queue that are not pending or are masked are consumed but not handled.

```
while H != 0
    p = H
    H = E[p].link
    if H == 0
        mb()
        H = E[p].link
    E[H].linked = 0
    if not E[p].masked
        handle(p)
```

handle() clears `E[p].pending` and EOIs level-triggered PIRQs.

> Note: When the event queue contains a single event, removing the event may race with Xen appending another event because the load of `E[p].link` and the store of `H` is not atomic. To avoid this race, the guest must recheck `E[p].link` if the list appeared empty.

### 5.3 Masking Events

Events are masked by setting the masked bit. If the event is pending and linked it does not need to be unlinked.

```
E[p].masked = 1
```

### 5.4 Unmasking Events

Events are unmasked by the guest by clearing the masked bit. If the event is pending the guest must call the event channel unmask hypercall so Xen can link the event into the correct event queue.

```
E[p].masked = 0
if E[p].pending
    hypercall(EVTCHN_unmask)
```

The expectation here is that unmasking a pending event will be rare, so the performance hit of the hypercall is minimal.

> Note: that after clearing the mask bit, the event may be raised and thus it may already be linked by the time the hypercall is done.