

XtratuM Hardware Initialization

Nicholas Mc Guire

Distributed & Embedded Systems Lab
Lanzhou University, SISE, P.R. China January 29, 2006

Contents

1. XtratuM Initialization	1
1.1. XtratuM.o <code>init_module</code>	1
1.1.1. Code path in <code>init_module</code>	1
1.2. Kernel side hooks	3
1.2.1. u-kernel hardware takeover	4
1.2.2. virtualizing Interrupts	4
1.2.3. building interrupts	4
1.2.4. Interrupt takeover	6
1.2.5. Virtual Interrupt functions	9
1.3. u-kernel hardware release	10
2. List of Acronyms	11

Contents

Version	Author	Date	Comment
0.1	Nicholas Mc Guire	28th Jan 2006	first shot

1. XtratuM Initialization

XtratuM is loaded as a kernel module in Linux, in this section we describe the initialization sequence of the u-kernel in more detail, for further details see the well documented source code. This description assumes all options in make menuconfig off so the core code only is described for now.

1.1. XtratuM.o init_module

XtratuM is provided as a dynamically loadable kernel module. When the kernel is instructed to load a kernel module by invoking `insmod` it will initialize a module structure in the kernel address space and then load the module content into the kernel, after the module content was loaded the user provided `init_module` from the module is executed to perform the module specific initialization. For a detailed discussion of module initialization see [3]

1.1.1. Code path in `init_module`

The mandatory `init_module` function for the XtratuM kernel module is located in the architecture independent part of XtratuM in `xm_generic/XtratuM_generic.c`, the only function called here is `xm_init`. The following is the code flow of `xm_init` with all configuration options off.

- Disable interrupts:

As the initialization of XtratuM needs to modify the interrupt handling procedures and hardware handlers, no interrupts are allowed until the necessary handlers are set up in an accessible way again - so the entire XtratuM initialization must run with disabled interrupts.

```
hw_save_flags_and_cli (flags);
```

- Enable Global events:

Enable all events that will be processed by the pipeline. This is setup with disabled interrupts so no events will actually arrive before `arch_takeover` has completed.

```
xm_clear_bitmap (global_pending_events);
```

- Mark root domain active:

This does not yet actually activate the root domain, it simply makes sure that one domain will be ready to run when the hardware virtualization is completed.

```
xm_set_domain_state(root_domain, xm_DOMAIN_ACTIVE);
```

- Pipeline Config for root-domain:

Register the root domain to receive all hardware interrupts, by marking it in the domains intercept mask and by clearing the mask bit, finally the irq handler which replaces `do_IRQ`, `linux_irq_handler` is assigned as interrupt entry point into the Linux kernel.

```
for (event = 0; event < 16; event ++) {
    xm_set_bit (root_domain -> intercepted_events, event);
    xm_clear_bit (root_domain -> masked_events, event);
    root_domain -> event_handler [event] = linux_irq_handler;
}
```

- Pipeline Init:

Initialize the pipeline by putting the root domain into the domain list as the only entry.

```
current_domain = xm_domain_list = root_domain;
```

- Virtualize hardware:

The next step is to actually virtualize the hardware by calling

```
xm_arch_takeover ();
```

which is described in detail in the following sections.

- Enable Interrupts:

Finally enable virtual interrupts (events) and turn on hardware interrupts again. return 0 to indicated success for `init_module`).

```
xm_enable_events_flag (root_domain);

hw_restore_flags (flags);

printk ("<XtratuM> XtratuM initialized successfully\n");

return 0;
```

1.2. Kernel side hooks

In vanilla Linux the kernels most low level functions and mappings are not accessible to kernel modules, they are not part of the public or module interface. The patch that needs to be applied for XtratuM to work must due two things:

- export the necessary functions so that they are accessible to modules
- add a layer of indirection so that the functions can be remapped

The first is not too wild, it basically means adding a few `EXPORT_SYMBOL` and in some cases maybe removing that static inline (though not done in the current XtratuM patch). The functions that XtratuM needs to modify are packed into the `XtratuM_root_exp_func` structure (read it as XtratuM Root-domain Exported Functions).

The second is the more important one. In vanilla Linux the low level data structures like the IDT are at fixed addresses only known within the kernel, furthermore some critical functions like `cli` are hard coded at a number of places (i.e. `entry.S`), this must be replaced by pointers to the structures, respectively calls to functions accessed via pointers, so that they can later be replaced. For details see the patch documentation [\[4\]](#).

Note that beyond the technical need of these changes, there also are far reaching legal implications, all code that uses direct access to the low level kernel functions and data structures, not available by default in the vanilla kernel, are implicitly under GPL license and legally can't be published under proprietary license. We question the legality of only releasing the kernel patch in such cases and not the including sources of modules that utilize the so exported functionality (but as usual INAL).

1.2.1. u-kernel hardware takeover

After the host-system, GNU/Linux, booted, it has full control of the hardware. At this point this is a standard time-sharing GPOS. To now put the u-kernel between the hardware and the GPOS requires to virtualize the interrupt and traps, thus preventing the GPOS from directly accessing interrupt related functions. In XtratuM this `arch_takeover` is done from the `init_module` function in `XtratuM.o`. The code shown is ia32 code from `arch/i386/XtratuM_arch.c`:

To operate on the low level functions safely `arch_takeover` must run with disabled interrupts, interrupts are re-enabled after all the remapping has completed.

1.2.2. virtualizing Interrupts

The first part of `arch_takeover` is simply saving the real interrupt management functions (those that actually modify hardware settings) and replacing them with the virtual interrupt management functions:

```
__root_sti = XtratuM_root_exp_func.xm__sti;
__root_cli = XtratuM_root_exp_func.xm__cli;
__root_save_flags = XtratuM_root_exp_func.xm__save_flags;
__root_restore_flags = XtratuM_root_exp_func.xm__restore_flags;
XtratuM_root_exp_func.xm__sti = xm_root_sti;
XtratuM_root_exp_func.xm__cli = xm_root_cli;
XtratuM_root_exp_func.xm__save_flags = xm_root_save_flags;
XtratuM_root_exp_func.xm__restore_flags = xm_root_restore_flags;
```

The `xm_root_` functions are the virtual functions for the root domain (Linux).

1.2.3. building interrupts

The second step is the remapping of the interrupt and trap functions, this is a bit messy so we will describe it in more detail.

The building of the IDT is quite a messy business in the Linux kernel, it is done with a set of macros that generate the necessary hard-coded table.

```
/*
 * Build the entry stubs and pointer table with
```

```
* some assembler magic.
*/
.data
ENTRY(interrupt)
.text

vector=0
ENTRY irq_entries_start
.rept NR_IRQS
ALIGN
1: pushl $vector-256
   jmp common_interrupt
.data
   .long 1b
.text
vector=vector+1
.endr

ALIGN
common_interrupt:
SAVE_ALL
movl %esp,%eax
call do_IRQ
jmp ret_from_intr
```

This repeats `NR_IRQS` times the sequence of pushing the vector followed by a jump to the `common_interrupt` code. The `common_interrupt` is what then actually calls the interrupt handling function which is built of three parts:

- entry: `SAVE_ALL` - saves all the registers to the stack
- handle: `do_IRQ` - extract the vector number from `%eax` and passes control on to `__do_IRQ` which is a generic irq handler function. In `__do_IRQ` the irq specific functions are called via `desc->handler`. `desc` is a pointer to the IDT at offset `irq`, that is the irq number is also the offset into the IDT table.
- cleanup: `ret_from_intr`: wich fixes the EFLAGS and restores the registers from the stack.

The assembler representation of this is:

1. XtratuM Initialization

```
00000240 <irq_entries_start>:
240:  68 00 ff ff ff      push  $0xffffffff00
245:  e9 f6 00 00 00      jmp   340 <common_interrupt>
24a:  8d b6 00 00 00 00   lea   0x0(%esi),%esi

250:  68 01 ff ff ff      push  $0xffffffff01
255:  e9 e6 00 00 00      jmp   340 <common_interrupt>
25a:  8d b6 00 00 00 00   lea   0x0(%esi),%esi

260:  68 02 ff ff ff      push  $0xffffffff02
...
00000340 <common_interrupt>:
340:  fc                  cld
341:  06                  push  %es
...
349:  53                  push  %ebx
34a:  ba 7b 00 00 00      mov   $0x7b,%edx
34f:  8e da              mov   %edx,%ds
351:  8e c2              mov   %edx,%es
353:  89 e0              mov   %esp,%eax
355:  e8 fc ff ff ff      call  356 <common_interrupt+0x16>
                               356: R_386_PC32 do_IRQ
35a:  e9 fc ff ff ff      jmp   35b <common_interrupt+0x1b>
                               35b: R_386_PC32 ret_from_intr
35f:  90                  nop
```

The entry part, `SAVE_ALL`, is the long list of `push RegName` only partially shown here, followed by the call to `do_IRQ` to actually handle the interrupt and the cleanup by calling `ret_from_intr`.

1.2.4. Interrupt takeover

XtratuM's IDT version is used to hold the entry points to the real hardware functions, basically copied from what ever Linux provided, and Linux gets its entry points replaced by calls to virtual interrupt functions.

```
for (irq = 0; irq < NR_IRQS; irq++) {
    root_hw_irq_controller [irq] =
```

1. XtratuM Initialization

```
((irq_desc_t *)XtratuM_root_exp_func.xm__irq_desc)[irq].handler;  
  
((irq_desc_t *)XtratuM_root_exp_func.xm__irq_desc)[irq].handler =  
    &xm_generic_hw_irq_c;  
}
```

After this step the functions have been swapped, next the entry points must be swapped. The entry path for an interrupt on an X86 architecture is shown in the following figure:

The hardware software boundary thus is the IDT. As shown in the previous section XtratuM already prepared a private IDT. This private IDT `root_idt_table` is used within the root domain to replace the real idt. The replacement is simply done by assignment.

```
xm_irq_addr = (const void (**) (void))&__start_xm_irq_handlers_addr;
```

The real idt `real_idt_table` is then set up with calls to the `hw_set_irq_gate`, which assigns XtratuM's new idt table to the respective vectors.

```
for (irq = 0; irq < NR_IRQS; irq++) {  
    vector = irq + FIRST_EXTERNAL_VECTOR;  
  
    // Replacing all hw irq gates for XtratuM routines  
    hw_set_irq_gate(vector, xm_irq_addr [irq]);  
}
```

1. XtratuM Initialization

At this point a hardware interrupt would no longer reach the original Linux entries but would get the handler of the `xm_irq_addr[irq]` which is setup to use `xm_irq_handler_#`, i.e. :

```
00000050 <xm_irq_handler_0x02>:
    50:      68 02 ff ff ff      push   $0xffffffff02
    55:      e9 06 01 00 00      jmp    160 <common_xm_irq_body>
    5a:      8d b6 00 00 00 00   lea   0x0(%esi),%esi
```

for irq 2. The actual work is done in `common_xm_irq_body` which in turn simply calls on `xm_irq_handler`

```
static int xm_irq_handler (struct pt_regs regs) {
    int irq = regs.orig_eax & 0xff;
    int execute_root_ret_from_intr;

    // Checking whether interrupts are really disabled
    assert (!hw_are_interrupts_enabled ());
    hw_irq_ack (irq);

#ifdef XtratuM_TIMER_SUPPORT
    if (irq != HW_TIMER_EVENT) {
#endif
        xm_set_bit (global_pending_events, irq);
        xm_set_bit (xm_domain_list -> pending_events, irq);
#ifdef XtratuM_TIMER_SUPPORT
    } else {
        xm_timer_handler (HW_TIMER_EVENT, &xm_TSC_PIT_timer);
        hw_irq_enable (irq); // Ready to get another timer irq
    }
#endif

    // xm_sched is called to execute the suitable handlers
    // return 1 if the root irq handler has been executed
    execute_root_ret_from_intr =
        (xm_sched () && (current_domain == root_domain));

    return execute_root_ret_from_intr;
}
```

This is the actual interrupt emulation code, except for the timer which is managed by XtratuM it self (if configured), all other events are just pushed into the pipeline. Pushing into the pipeline means simply setting a pending event mask so the event is recorded (this can be seen as the virtualized PIC) and marking it in the domain lists pending events for processing. The processing happens in the call to `xm_sched` (in `xm_generic/XtratuM_generic.c`) which then loops of the list of registered domains and checks if the domain wants to receive the pending event or not.

1.2.5. Virtual Interrupt functions

XtratuM does not actually set up the hardware related low level functions for managing the particular interrupt controller it simply reuses what Linux had available. All that needs to be done is to clear/set the virtual PIC with `xm_clear/set_bit` and then execute the root domains irq controller function.

Note that virtual interrupts don't need to acknowledge interrupts at the hardware level because this was done by XtratuM with `hw_irq_ack (irq)` in `xm_irq_handler` all ready.

```
static unsigned int virtual_irq_hw_startup (unsigned int irq){
    unsigned int hw_flags;
    hw_save_flags_and_cli (hw_flags);
    xm_clear_bit (root_domain -> masked_events, irq);
    hw_restore_flags (hw_flags);
    return (root_hw_irq_controller [irq]->startup(irq));
}
```

```
static void virtual_irq_hw_shutdown (unsigned int irq) {
    xm_set_bit (root_domain -> masked_events, irq);
    if (root_hw_irq_controller [irq] &&
        root_hw_irq_controller [irq] -> shutdown)
        root_hw_irq_controller[irq] -> shutdown(irq);
}
```

```
static void virtual_irq_hw_ack (unsigned int irq){
    return;
}
```

```
static void virtual_irq_hw_enable (unsigned int irq) {
    unsigned int hw_flags;
    hw_save_flags_and_cli (hw_flags);
}
```

```
xm_clear_bit (root_domain -> masked_events, irq);
if (xm_is_bit_set (global_pending_events, irq)) {
    if (root_hw_irq_controller[irq])
        root_hw_irq_controller[irq] -> enable(irq);
    xm_clear_bit (global_pending_events, irq);
}
hw_restore_flags (hw_flags);
}

static void virtual_irq_hw_end (unsigned int irq) {
    virtual_irq_hw_enable (irq);
}

static void virtual_irq_hw_disable (unsigned int irq) {
    unsigned int flags;
    hw_save_flags_and_cli (flags);
    xm_set_bit (root_domain -> masked_events, irq);
    hw_restore_flags (flags);
}
```

Note that the virtual interrupt functions in XtratuM are directly using the low level root domain functions, so the virtualization is reduced to the minimum necessary - disabling of interrupts. Basically interrupts are simply never disabled aside from critical regions protected with `cli/sti`. By virtualizing disabling of interrupts through a domain event mask, the event dispatcher logically can provide a interrupt pipeline but does not require domains not interested in particular events to actually process them in any way.

The XtratuM u-kernel is thus bound to its root-domain on a functional level. The u-kernel utilizes interrupt control functions from the root domain and only provides the minimum virtualization functions necessary to allow deterministic processing of events and different levels of priority.

1.3. u-kernel hardware release

TODO

2. List of Acronyms

FDL - Free Documentation License
GPL - General Public License
GPOS - General Purpose Operating System
IA32 - Intel 32bit archtiecture
IDT - Interrupt Descriptor Table
INAL - I'm Not A Lawyer
OS - Operationg System
PIC - Programmable Interrupt Controller
RT - Real Time

References

- [1] Maintainer:Miguel Masmano, Ismael Ripoll, *XtratuM - Open Source nanokernel*,<http://www.ocera.org>
- [2] M. Masmano, I. Ripoll, *XtratuM Interface*,XtratuM release 0.1 (docs directory), 2003
- [3] Alessandro Rubini, Jonathan Corbet: *Linux Device Drivers, 3^{ed} Edition*, O'Reilly, 2004
- [4] Nicholas Mc Guire: *XtratuM Patch Documentation*,<http://dslab.lzu.edu.cn>, 2005