

XMeRTL-4.0-rc1 Manual

Distributed & Embedded System Lab

September 18, 2009

Acknowledgments

XMeRTL release XM-eRTL-4.0 as replacement for the legacy RTLinux/GPL 3.2 is now ready for download as release candidate 1 (XM-eRTL-4.0-rc1). This work based on the hypervisor XtratuM 1.0 as well as PaRTiKLe developed at the Universitat Politecnica de Valencia is now maintained by DSLab at Lanzhou University and has been extended to be a full featured replacement of RTLinux/GPL.

The DSLab team will continue to develop and enhance XM-eRTL in the future in tight coordination with the Universitat Politecnica de Valencia, DISCA, based on strong POSIX binding and compatibility to the vanilla Linux kernel as its root domain.

These people who write the document are listed bellow:

Andreas Platschek	<andi@opentech.at>
Bai Shuwei	<baishuwei@{dslab.lzu.edu.cn, gmail.com}>
Lv Qingquan	<lvqq05@dslab.lzu.edu.cn>
Jin Guojun	<jingj1986@dslab.lzu.edu.cn>
Pu Yiqiao	<puyq@dslab.lzu.edu.cn>
Zhang Chuande	<zhangchd@dslab.lzu.edu.cn>

Also thanks for Prof. Nicholas Mc Guire and Associate Researcher Zhou Qingguo who review the manual and give many important suggestion.

Contents

1. Introduction	1
2. Installation	6
2.1. Linux Installation	6
2.2. XtratuM Installation	9
2.3. PaRTiKle Installation	13
2.3.1. About Partikle	13
2.3.2. PaRTiKle in XM/eRTL	13
2.3.3. Requirements for installing partikle	14
2.3.4. Installation	14
2.4. Tools Usage	16
3. Inter-Domain Communication Tools	18
3.1. XM/FIFO	18
3.1.1. Characteristics	18
3.1.2. Using XM/FIFO	19
3.2. XM/SHM	21
3.2.1. About XM/SHM	21
3.2.2. XtratuM/SHM API	22
3.2.3. Usage Example	23
4. Device Driver	24
4.1. XM/DEV	24
4.1.1. About XM/DEV	24
4.1.2. Development	25
4.2. XM/Serial Driver	37
4.2.1. About XM/Serial Driver	37
4.2.2. Using XM/Serial Driver	38
5. xmtrace - Trace Tool	41
5.1. Run an Example	41
5.2. Using the Tracer	46
6. Examples	48
6.1. Domain	48
6.2. Thread	51
A. List of Hypercalls	53

B. Simple example of XtratuM device driver	53
B.1. Head file	54
B.2. Initialization module	55
B.3. Device driver server domain	65
B.4. Code modified in other files	69
3. List of Acronyms	72

1. Introduction

XM/eRTL is an integration project building on the open-source XtratuM hypervisor and the PaRTiKle embedded real-time kernel from the Departamento de Informtica de Sistemas y Computadores (DISCA) of the Universidad Politcnica de Valencia in Spain. The DSLab XtratuM team has been integrating and extending these components. In this manual we describe the installation and use of XM/eRTL. For further reference please visit xtratum.dslab.lzu.edu.cn, www.xtratum.org as well as the home-page of DISCA gii.disca.upv.es.

XM/eRTL is targeting embedded real-time systems. It is comprised of XtratuM and PaRTiKle. XtratuM is a hypervisor (also referred to as nanokernel) and PaRTiKle is a new open source real-time kernel for embedded systems, which is used to provide real-time domains under XtratuM. PaRTiKle is anticipating full POSIX compliance while at the same time supporting the non-posix extensions of legacy RTLinux/GPL applications, so which is why PaRTiKle is named eRTL in the XM/eRTL project. XM/eRTL exhibits the characteristics from the XtratuM nanokernel and PaRTiKle.

Before going into the details of the XM/eRTL system, let's clarify the main concepts in the XM/eRTL system. XtratuM is a hypervisor which runs on top of Linux, and PaRTiKle is a kernel following the POSIX PSE 51 profile, single process multiple threads, which is quite common for embedded real-time systems. XM/eRTL system is the combination of XtratuM and PaRTiKle or one could describe it as the PaRTiKle kernel being ported to the XtratuM system and it runs as a XtratuM domain. Domain is the smallest unit of execution controlled directly by XtratuM. Linux is also only a domain too in this model - with XtratuM scheduling the GNU/Linux OS as a single entity, that is it has no knowledge of the internal processes or threads of a domain. But XtratuM is based on Linux insofar as Linux is the root domain of XtratuM, providing special services to XtratuM. PaRTiKle is the real-time kernel in the XM/eRTL system, PaRTiKle can be seen as a real-time POSIX runtime environment that is under control of XtratuM. In the whole system, XtratuM runs at the lowest level which allows it direct hardware access, Linux and PaRTiKle are at the same level and access hardware resources via calls to XtratuM. So the programmer can write the application for XtratuM, PaRTiKle and LINUX separately (whereby the "programs" for XtratuM generally are operating systems and rarely does one write XtratuM "applications" directly). The applications running on XtratuM are domains like Linux and PaRTiKle, a domain is the schedulable entity in XtratuM and is associated with a set of resources as well as a priority. The application programmed for PaRTiKle are real-time thread (as PaRTiKle is a single process multiple threads model). Applications running on Linux can be process or multithreaded - basically anything that normal GNU/Linux permits. In the released version, XM/eRTL system consists of XtratuM and PaRTiKle only, and a patch to the Linux kernel to enable the necessary paravirtualization - the GNU/Linux itself though is a standard

Linux distribution (Debian in our case).

The XM/eRTL system is a nano-kernel based approach. Only the bare minimum functions are supported by XM/eRTL core - XtratuM:

- interrupt management
- timer virtualization and management
- address space management
- and domain scheduling

Note - that conversely to microkernels that provide IPC primitives in the core, a nano-kernel provides no such primitives at all - IPC is out side the core.

The systems actual functionality is provided by the domains. Linux is the root domain offering memory allocation and system boot services aswell as non real-time general purpose servers (i.e. sshd,httpd,etc.).

Two Inter-domain Communication (IDC) extensions are provided to transfer data between domains, XM/FIFO and XM/SHM. Further device drivers are external to the XtratuM core, the device driver model of XM/eRTL XM/DEV is provided as prototype for porting and building device driver. In the XM/eRTL, the XM/Serial device driver is the only real-time device driver implemented at present - this is an area where work on XM/eRTL is not yet completed. Finally programmers need basic debugging support, notably temporal debugging as this is a real-time system, so a trace tool, XM/TRACE, to trace the events triggered in the XM/eRTL kernel is provided..

Different from RTLinux/GPL which is dual-kernel system, XM/eRTL is a full multi-kenrel system. The purpose of XM/eRTL is to build more stable and reliable real time system while supporting existing application for RTLinux/GPL. RTLinux/GPL which was one of the early real-time Linux extensions in the open source area is a hard realtime OS which runs the entire Linux operating system as a single process and thus is fully pre-emptible. Wind River Systems acquired the RTLinux copyright in 2007 from FSMLabs and now makes a version available as Wind River Real-Time Core for Wind River Linux, this realtime extension though is based on the original dual kernel concept. In order to support the RTLinux/GPL applications, XM/eRTL is built by the RTLinux/GPL community to replace RTLinux/GPL system so providing RTLinux/GPL compatible was a main goal of this effort.

The basic idea behind of XM/eRTL is hypervisor architecture where a software layer to manage all hardware access is introduced between the hardware and the high-level OSs. In the architecture, both real-time and non-real-time application can be supported cuncurrently. The XM/eRTL is thus a hard real-time system and the system

scheduler support full preemption of domains thus making GNU/Linux as a whole fully preemptible. So the systems or domains running on XM/eRTL can inherit the real-time property from the original operating system, such as Linux and PaRTiKle. The core kernel of XM/eRTL is XtratuM, which is the hypervisor providing the minimum resource management. XtratuM focus on the interrupt management, timer management and domains scheduler and associated address area management which are the critical components for a real-time system.

In XM/eRTL, Linux with the XtratuM patch applied, boots the machine first, initializing the whole system as usual, such as devices initialization, memory initialization, interrupt and timer initialization (not for XM/eRTL just for GNU/Linux), etc. So Linux installation and booting are basic operation or preparation for building XM/eRTL. When the GNU/Linux has booted successfully, XtratuM can be loaded as a kernel module. XtratuM start taking over the machine management from Linux byx . Loading of XtratuM starts by interrupt takeover, then timer takeover, and at last jumping to XtratuM scheduler which at this time is initialized with a single domain, the root domain, GNU/Linux. From a users perspective not much changed in the system. Linux is only a partition or domain on the system, and it is no longer in control of time-critical resources on the system. In the Linux root-domain interrupt and timer management now operates on virtual timers and interrupts via calls to XtratuM. Linux' main roles is offering general purpose non-realtime services including device service and memory allocation service. Linux will be scheduled by XtratuM according to its priority - which is by default the lowest in the system. When there is no real-time domain, whose priority is higher than Linux, the CPU will be assigned to Linux as its "idle" task.

When XtratuM took over Linux interrupt and timers successfully, the system can be used to build and load further XtratuM domains. Building and running an XtratuM domains will be introduced in section 6.1, Linux application programming is not affected by the loading of XtratuM though. The hypercalls available for developing XtratuM domains are presented in Appendix II.

PaRTiKle is special embedded real-time system which can operate stand-alone [?], but it also can run as a guest-OS on XtratuM. Further it can run as a Linux process, primarily intended for ease of development. In XM/eRTL, PaRTiKle runs as a domain with higher priority than GNU/Linux. PaRTiKle is a POSIX compatible operating system, providing pthreads as its prime abstraction. Application compliant with POSIX threads model can thus be ported to PaRTiKle with little effort and often without source code modification all together. In order to support legacy RTLinux/GPL applications, PaRTiKle was extended with non-POSIX calls found in RTLinux/GPL, though we advise not to use any `_np` functions in new projects. PaRTiKle and XtratuM are thus the two most important components in XM/eRTL system. Section 6.2 will show how to build real-time application based on PaRTiKle.

XtratuM and PaRTiKle are core components in XM/eRTL offering the core OS services. Extensions, provided in sepearte loadable modules, include IDC (Inter-Domain Commu-

nication) function, device driver model and trace functionality. Further extensions are work, notably a GDB-stubs implementation ports to further hardware architectures (MIPS and PowerPC) aswell as safety related extensions. XMFIFO and XMSHM are the two most basic IDC modules offered for XtratuM. The IDC is used to transfer data between all kinds of domains. Typically FIFOs are used for streaming data and SHM is used to transfer sparsely modified data blocks. The XMDEV Device driver model is used to build or add new device driver for XM/eRTL. The trace tool is used to trace the core events relevant for the analysis of real-time systems, like scheduling, interrupts, or synchronisation events. These functions will be covered in later sections in more detail. The whole manual is divided into six sections. This section, section 1, is only the brief overview. Section 2 covers installation, in which building an XM/eRTL environment will be explained in detail. From introduction section, it should be clear that the installation process mandates a Linux Installation, XtratuM installation and PaRTiKle installation. In order to boot XtratuM and manage the domains, basic management tools are included in XM/eRTL. The section will introduce XM/eRTL tools and explain tools usage. Section 3 is dedicated to IDC modules. The two XM/eRTL IDC offerings, XM/FIFO and XM/SHM are covered in detail. XM/FIFO and XM/SHM adopts memory mapping and lock free mechanisms as there prime features.

Section 4 is introduces Device Driver, though this section is still preliminary. The section includes device deriver model in XtratuM and serial device driver which is the only hardware device driver offered by XtratuM at this point. Following XM/DEV (XtratuM DEvIce Driver Model), the hardware device driver comprised of static component and dynamic component. The static component is located in the Linux kernel and provides initialisation of the device related resources (allocate and release resources). The dynamic component (run time) is the main body which will access and manage the device at run-time, and normally the dynamic component is a schedulable domain. The serial device driver is built with the XM/DEV model.

Section 5 is provides guidance on the Trace tool. XM trace was designed based on the rtl_tracer found in RTLlinux/GPL. XM trace uses statically allocated shared memory to record trace data (timestamp, address and type of events). Both hard-coded system events and user-defined events are supported. xmtrace provides adequate trace points in the XtratuM code base for critical functions like scheduling, enabling/disabling interrupts, and acquireing/ freeing of spin locks, etc. Custom events can be added and triggered by the user via a simple interface. Events are grouped into event classes. Single events and event classes can be enabled and disabled as required, which makes the Tracer very flexible.

Section 6 contains exercises. Beginners can learn how to write domains and threaded applications in the XM/eRTL system. A domain is the scheduled unit of XtratuM and threads are the units of execution in the PaRTiKle kernel. As domains are directly scheduled entities they have lower interrupt and timer latency, but the non-POSIX API for domain programming is very limited, generally making it hard to write applications

directly as an XtratuM domain. PaRTiKle is an example of a complex XtratuM domain providing a full POSIX kernel tailored to the minimum POSIX System Environment (PSE) 51 specifications, it offers the basic POSIX thread API to programmers and thus is much easier to handle than native XtratuM domains.

2. Installation

The Linux operating system is the main operating environment for XM/eRTL, since it is used to boot the system and initialize the hardware. After initialization is complete, XtratuM (which is the core of XM/eRTL) is loaded and takes control over the interrupt and timer management permissions, and registers Linux as root domain. At last, PaR-TiKle is loaded by Linux tools. These are the milestones of the XM/eRTL installation process, which will be explained in a step-by-step description in detail in the following section. At the end of the section, the user-space tools used to manage XtratuM and its partitions are introduced.

2.1. Linux Installation

At the moment XMeRTL runs stable on Linux version 2.6.17.4, but other subsequent linux kernel versions are available, but still have to be considered development versions. Therefore this document will focus on linux-2.6.17.4 ¹.

The operation system used in this manual is GNU/debian etch, but the process of configuring and building XM/eRTL should be very similar on other linux distributions.

The first thing you need to be able to build a patched linux kernel and XM/eRTL, is a toolchain. For this purpose, debian offers us a dummy package, called *build-essentials* which has dependencies to almost everything we will need during the build process. One additional package we will need is the ncurses library, used to display the kernel configuration menu. So let's install these two packages:

```
#apt-get install build-essential libncurses5
```

[*NOTE* :] If you use a newer distribution (e.g. debian lenny), you might need to install the package *gcc-3.4* and add a `CC=gcc-3.4` to every make, since XtratuM requires gcc version 3.4.6 or 4.1.2. This is important to keep in mind, when using a newer distribution, since they come with newer compilers, and those might lead to unpredictable behaviour.

Ok, so we have all tools we will need to install XM/eRTL, next we need to fetch XM/eRTL itself. In the next step, we will download XM/eRTL from the ftp server at the DSLAB at Lanzhou University.

```
# wget http://URL/xmertl-4.0-rc1.tar.gz
# tar xzvf xmertl-4.0-rc1.tar.gz
```

¹The Linux kernel can be download from: <http://www.kernel.org>.

2. Installation

So let's have a look at the content of this tar ball. Below you can see the content in a tree structure. The main parts are the documentation in **docs**, which contains this document, **partikle** and **xtratum** which will be configured and installed in the following sections, and **tests** which contains a collection of simple test applications which check the different subsystems of XtratuM, to verify that the installation succeeded.

```
xmertl-4.0-rc1/
|-- docs
|   '-- manual
|-- partikle
|   |-- core
|   |-- docs
|   |-- mkfiles
|   |-- scripts
|   '-- user
|-- tests
|   |-- fifo_tests
|   |-- partikle_tests
|   |-- shm_tests
|   |-- trace_tests
|   '-- xtratum_tests
'-- xtratum
    |-- arch
    |-- devs
    |-- include
    |-- kernel
    |-- patches
    |-- user_tools
    '-- xmtrace
```

Next we download the linux kernel and patch it with the XtratuM patch. Among other things, this patch adds the hooks into the kernel, which will later be used by XtratuM to run Linux as root domain (to paravirtualize it). To check whether the patch will succeed, we will first run it with **--dry-run**, and after this test succeeded, we really patch the kernel.

```
# cd /usr/src
# wget -v http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.17.4.tar.gz
# tar xjvf linux-2.6.17.4.tar.bz2
# cd linux-2.6.17.4
```

2. Installation

```
# patch --dry-run -p1 < ../rtlinux-4.0-rc1/xtratum/patches/xtratum_linux_2.6.17.4.patch
# patch -p1 < ../rtlinux-4.0-rc1/xtratum/patches/xtratum_linux_2.6.17.4.patch
```

After the patching succeeded, we will now configure the linux kernel. Unfortunately, there are some options, we have to be careful about to successfully install XtratuM, and get the best performance:

```
#make menuconfig (or xconfig or config)
-> General setup -> XtratuM support

-> Processor type and features -> Symmetric multi-processing support [OFF]
-> Processor type and features -> Preemption Model (No Forced Preemption (Server))
-> Processor type and features -> Local APIC support on uniprocessors [OFF]
-> Processor type and features -> High Memory Support (off)
-> Processor type and features -> Memory model (Flat Memory)
-> Processor type and features -> Use register arguments [OFF]
-> General setup -> Power Management support [OFF]
-> General setup -> ACPI [OFF all options]
-> General setup -> CPU Frequency scaling [OFF all options]
```

Since the kernel is configured now, all we have left to do is to fire up the compiler, install the kernel and configure the bootloader:

```
#make && make modules_install
#cp arch/i386/boot/bzImage /boot/bzImage-2.6.17.4-xm
#mkinitramfs -o /boot/initrd.img-2.6.17.4-xm 2.6.17.4-xm.1.0
#vim /boot/grub/menu.lst
```

Now add the follow entry to the grub configuration in menu.lst

```
title                debian, kernel 2.6.17.4 Xtratum
root                 (hd0,0)
kernel               /boot/bzImage-2.6.17.4-xm root=/dev/hda1 ro
initrd               /boot/initrd.img-2.6.17.4-xm
boot
```

We assume that the /boot directory is located in the /dev/hda1 partition (being hd0 and hda1 your hard disk). If you are not sure what you are doing: lookout for the entries already in menu.lst and use the same entry (e.g. /dev/sda1, or whatever) as in the other entries.

Now we are ready to reboot the machine into the newly built XtratuM kernel.

2.2. XtratuM Installation

If everything went as planned, you enter the new system using the kernel with XtratuM support. Now we have a linux kernel running which has the hooks included which are needed by XtratuM to take over the system. But before this is possible, we have to build XtratuM.

In the next step, we enter the `xtratum` subdirectory which contains the source code of XtratuM. Of course XtratuM possesses a `make` buildsystem, making the compile process very easy. In order to be able to build XtratuM, we have to tell it, where the linux source code is located. This can be done by making a link to the linux kernel source directory called `linux` in the `xtratum` directory.

```
# cd ~/xmertl-4.0-rc1/xtratum
# ln -s /usr/src/linux-2.6.17.4 linux
```

After this last preparations, we are ready to compile XtratuM and all its components. Since XtratuM has a make system, we just have to type in `make` to start the build process.

```
# make
```

After the process of compile, many modules and executable files are generated. Of course, you can refer to them by the command of `"ls"` or `"ls -l"`. These new primary files are located in this directory: `xtratum/`, `xtratum/scripts/xmloader`, `xtratum/devs/*/`. In addition, there are some executable files we will use. These executable files are the tools we use to operate and control the system. You can refer to Tools Usage to know how to use these tools. Here, we check the file in directory `xtratum/`

```
# ls
arch  Changelog  COPYING  devs      include  INSTALL  kernel  Makefile
Modules.symvers  patches  README  user\_tools  xm.ko   xm.mod.c  xm.mod.o  xm.o
```

We can tell you that these files are generated just now: `Modules.symvers`, `xm.ko`, `xm.mod.o`, `xm.o`. The file `xm.ko` is useful, because it is a module of the XtratuM core. Using `insmod` to insert this module, then we can get the supporting of new characteristic of XtratuM.

```
# insmod xm.ko
```

2. Installation

However, this command is used to insert the modules of the XtratuM core. These are some device modules and useful modules have been generated just now. These modules will be used later, so we should insert these modules. If you insert them one by one using `insmod`, it would waste a long time. Don't worry, you can use the executable file `xmcmd.sh` insert these modules easily. The executable file is a tool to load and unload the XtratuM. Here we just load the XtratuM.

```
# ./user_tools/scripts/xmcmd.sh -l
<XtratuM> Detected 666.643 MHz processor
<XtratuM> XM Serial Dev(vttyS0~vttyS15)!
<XtratuM> XM FIFO Dev(rtf0~rtf15)!
<XtratuM> XtratuM SHM devices build successfully!
<XtratuM> LSHM DEVICE REGISTER SUCCESSFUL
<XtratuM> Linux UART register Suggested!
<XtratuM> Linux Register FIFO device successful
```

Now, the modules have been installed and the devices have been created. We can use them. Before use, we can check the work to ensure that is all right. Just as discussed above, some modules have been inserted. Use the command `lsmod` to see whether these modules are overall.

```
# lsmod
Module Size  Used by
lfifo 2720  0
luart 3368  0
lshm 3252  0
shm 4736  1 lshm
rtfifo 3968  1 lfifo
serial 5124  1 luart
xm 38824  3 shm,rtfifo,serial
```

If these seven modules are listed in your screen, you can think that these modules inserting step is all right. At the same time, some devices are created. These devices are these: `rtf0`, `rtf1`, ... , `rtf15`, `shm0`, `shm1`, ... , `shm15`, `VttyS`. All of them are located in the directory `/dev/`, so you can check them use the command `ls`. Here list the `rt-fifo`'s devices. You can list the others like this:

```
# ls /dev/rtf*
rtf0  rtf1  rtf10  rtf11  rtf12  rtf13  rtf14  rtf15  rtf2
rtf3  rtf4  rtf5  rtf6  rtf7  rtf8  rtf9
```

2. Installation

Progress smoothly. Of course, only if it can work for some task, we can say it is successful. So here is an application to test whether it can work or not. The application is used to create a simple domain. When it is loaded, it will print a string "hola" to tell us that it can run successfully.

Ok, let's compile the domain and load it. First enter the directory of the application, which is located at `xtratum/user_tools/examples/trivial_domain`. Then compile it.

```
# cd /path/to/trivial\_domain
# make
```

After compilation, a new domain named `trivial.xmd` is generated. We use the tool `loader.xml` to load it. You can learn more information from the Tools Usage. To be convenient, we set the `PATH` to contain the path of `loader.xml`. Then we can use it as `loader.xml` in place of `/path/to/loader.xml`.

```
# export PATH=$PATH:/path/to/rtrlinux-4.0-rc1/xtratum/user_tools/xmlloader
# loader.xml trivial.xmd
>> Loading the domain "trivial.xmd (trivial.xmd)" ... Ok (Id: 1)
```

If it returns a message like: `"... Ok (Id: X)"`, it tells you that the domain loaded successfully. The return value of `Id` is assigned for this domain. Every domain loaded successfully can get an `id`. The first loaded domain gets the `id` 1, and the second gets 2, and so on. The `id` is used to unload the domain by `unloader.xml`. Refer to Tools Usage. Remember that the `id` of the domain loaded just now is 1.

The domain loaded successfully, then we can display a message about it by `"dmseg"`. The command `"dmseg"` is very useful for displaying the message about the system and domains. The message of the domain displayed is just the message output to stand out. More information about `dmseg` can refer to [?].

```
# dmseg
```

If it went as planned, you can see the string "hola" at the bottom of the displayed message. If so, we just tell you it can work. The following work about the test is to unload the domain

```
# unloader.xml -id 1
>> Unloading the domain "(1)" ... OK
```

There are some problems that may be encountered by you.

2. Installation

```
# loader.xm trivial.xm
>> Loading the domain "trivial.xmd (trivial.xmd)" ... Failed (Error: -19)
```

This is because you insmod xm.ko, but not load XtratuM, so you should execute /path/to/rtlinux-4.0-rc1/xtratum/user_tools/scripts/xmcmd.sh -l

```
# loader.xm trivial.xm
>> Loading the domain "trivial.xmd (trivial.xmd)" ... Segmentation fault
```

At most time, you did not execut xmcmd.sh -l. So you had better check the modules by lsmod before executing loader.xm.

In addition, another domain can be used for test and serial operation. This domain is located at xtratum/user_tools/serial. As said top, you can enter this directory, compile to generate the domain and load it. When load we support a priority to the domain, which is 1. If you have read the introduce document, you would know that there are 1024 priorities from 0 to 1023 in XtratuM. The bigger the value is, the lower its priority is. If you know these, you can support the other priority to this domain.

```
# cd /path/to/serial
# make
# loader.xm serial.xmd -prio 1
>> Loading the domain "serial.xmd (serial.xmd)" ... Ok (Id: 2)
```

This domain does not output message to the stand out. However, we can get information about the domains loaded and not unloaded by /proc/xtratum.

```
# cat /proc/xtratum
XtratuM domains:
-----
(2) serial.xmd:
  Priority: 1
  Intercepted events: 0x10
  Masked events: 0xffffffff
  Pending events: 0x0
  Events' flag is enabled
  Current state: SUSPENDED
  Domain's state word: 0x1
(0) Linux:
  Priority: 1023 (MIN. PRIORITY)
  Intercepted events: 0xffff
```

```
Masked events: 0xffff0000
Pending events: 0x0
Events' flag is enabled
Current state: ACTIVE
Domain's state word: 0x0
```

"(2) serial.xmd:" is the message about the domain serial.xm. The digit 2 ahead is the domain id. And we can get more information about this domain. Its Priority is 1, and its state is SUSPENDED. "(0) Linux:" is the message about the root domain Linux. We can also know the related information. If you look carefully, you can find that there is no domain whose id is 1. Do not forget, we have unloaded it above. In additional, if you load the domain with setting the priority, then its priority is 30.

2.3. PaRTiKle Installation

2.3.1. About Partikle

PaRTiKle is a real-time kernel and it is open source. It is under GNU Public License, PaRTiKle was designed to be POSIX compatible. The native API is "C". It can run on a operating system as a thread or a domain which will be introduced next.

PaRTiKle is very small, maybe 890kb is the largest. It is a full-featured, flexible, configurable, real time embedded kernel. The kernel provides thread scheduling, synchronization, timer, and communication primitives. It handles hardware resources such as interrupts, exceptions, memory, timers, etc. You can get more information from:[?].

2.3.2. PaRTiKle in XM/eRTL

XtratuM is a hypervisor that it is used to embedded real-time system. It can concurrently run several OSes on XtratuM. If there is no real-time operating system on XtratuM, it is useless. So, PaRTiKle, as a good embedded real-time operating system, can meet the needs well and it is very small. PaRTiKle is used to run as a domain on XtratuM.

It supports running several PaRTiKle domains concurrently, so you may see a PaRTiKle domain as a thread, Each PaRTiKle domain has its own independ space to run application. You can build several PaRTiKle domains and they can communicate by FIFO. If you load several PaRTiKle domains with different priorities , it can meet the real-time needs well because of the scheduling of XtratuM.

2.3.3. Requirements for installing partikle

The development environment requires intel architecture pc running linux. Disk space are minimal(less than 32MB). We have tested it successfully on debian 4.0 and slackware 11.0. Partikle has been compiled successfully on the following tools : - Gnu C: 3.4.6 or 4.1.2 - Gnu make: 3.81beta4 - binutils: 2.16.91 or 2.17-3 - util-linux: 2.12r - module-init-tools: 3.2.2 - Linux C Library: 2.3.6 or 2.5 - Dynamic linker (ldd): 2.3.6 or 2.5 - Procps: 3.2.6 or 3.2.7-3 - Net-tools: 1.60 - Console-tools: 0.2.3 - ncurses-lib: 5.0 or 5.6

2.3.4. Installation

If you have download the PaRTiKle package successfully, you can compile PaRTiKle now. Before compiling, configure PaRTiKle should be done.

```
#cd [PaRTiKle dir]
#make menuconfig (or make config)
checking the following options:
Architecture (XtratuM domain) --->
Devices      --->  [*] Enable the serial uart device (i8250 Based)
                  [*] Enable XtratuM fifos
                  [*] Enable XtratuM SHM
Core Options  --->  (2097152) Kernel dynamic memory
                  (1047552) User dynamic memory
Ulibc Options --->  [*] Include error messages for the codes in errno.h
```

The meaning of the above options: The architecture is XtratuM domain,because we want PaRTiKle to run as a domain on XtratuM. The three options in devices make PaRTiKle can execute communication by serial ports and XtratuM fifo and shm. FIFO means first in first out. SHM means share memory. FIFO and SHM are both inter-domain-communication tools on XtratuM. 2MB for kernel dynamic memory and 1MB for user dynamic memory for PaRTiKle domain is enough. The Ulibc Options support debugging.

Compiling PaRTiKle: Because the compiles rules are defined in Makefile,you just need to make it:

```
#make
>> Detected PaRTiKle path: /home/lvqq/rtlinux-4.0-rc1/PaRTiKle
>> Building PaRTiKle utils: done
```

2. Installation

```
>> Building PaRTiKle kernel [xm_i386]: done
>> Building PaRTiKle user libraries: done
>> Include these in your profile environment:
PRTK=/home/lvqq/rtlinux-4.0-rc1/PaRTiKle export PRTK
PATH=$PATH:$PRTK/user/bin export PATH
```

When PaRTiKle is compiled successfully, the PRTK and PATH will be set in your system. How about your installation? The test can be done now. There are many examples in user/examples/c_examples which can be used to test the whole system.

```
#cd [PaRTiKle dir]/user/examples/c_examples
#make
```

then you will get many *.prtk files which contain PaRTiKle kernel and the application code compiled jointly. you can write a prtk domain by yourself as following:

```
#vi hello.c
#include<stdio.h>
int main(int argc,char **argv)
{
printf("hello\n");
return 0;
}
```

then compiling it:

```
#make
```

you will get the hello.prtk, load it by the command which is located in xtratum/user_tools/xmloader/loader.xml:

```
#[XtratuM dir]/user_tools/xmloader/loader.xml hello.prtk
>> Loading the domain "hello.prtk (hello.prtk)" ... Ok (Id: 1)
```

The Id 1 is your domain id number, you can use it to unload the domain just like `unloader.xml -id ;Id number;`

```
#dmesg
>> PaRTiKle Core <<
Detected 2800.456 MHz processor.
Setting up the dynamic memory manager (2048 kbytes at 0x200400c)
Free system memory 3071 Kb
PaRTiKle (59 Kb [.text=38 .data=7 .rodata=2 .bss=7] Kb)
App. (30 Kb [.text=17 .data=1 .rodata=6 .bss=0] Kb)
- init console: ok
- init root: ok
- init rtf: ok
- init shm: ok
```

unload the test domain:

```
#[XtratuM dir]/user_tools/xmloader/unloader.xml -id 1
>> Unloading the domain "(1)" ... OK
```

If the result showed is same as above, congratulation, you have installed XM/eRTL successfully.

2.4. Tools Usage

XtratuM support some tools to manage and control the system. The tools are executable script file and executable file generated by compile. `xtratum/user_tools/scripts` has some executable script files. The most useful file is `xmcmd.sh`. It is used to load the XtratuM modules, device drive and moduls. There are two options can be uses, `????`, to load these modules. `?????`, is used to unload these modules. The usage is like this:

```
$/path/to/user_tools_scripts/xmcmd.sh -l
$/path/to/user_tools_scripts/xmcmd.sh -u
```

`xtratum/user_tools/xmloader/loader.xml` is used to load a domain. For example you have a domain name "dom1", you can load it like this:

```
$/path/to/xtratum/user_tools/xmloader/loader.xml dom1 .
```

The path is too long and hard to operate. You can add the path to global varient `PATH`, and use `loader.xml` directly.

2. Installation

```
$export PATH=$PATH:/path/to/user_tools/xmloader
$loader.xml /path/to/dom1
```

It is easy. But there is a limitation using this way. The PATH includes the loaderxm just effective in this dialogue. If you restart it, or use another dialogue, you need to set it again. To avoid these trouble, you can copy the loader.xml (and unloader.xml) to the directory /usr/bin/. It is once for all. Every loaded domain has its priority. The default priority is 30, but you can set it by yourself just using the argument "-prio". The n is the priority, which can be 0 to 1023.

```
$loader.xml /path/to/dom1 -prio n
```

When the domain loads successfully, a id will return. This id is used to unload the domain. The tool of unload is also local in the directory xtratum/user_tools/xmloader named unloader.xml. If you want to operate easily, you can use the ways above to set it. For example, we load the domain dom1 and get its id 2. Then we can unload it by this:

```
$unloader.xml -id 2
```

After load a domain, we would want to know its information. The first tool is command "dmesg", which means display message. The message displayed on the screen is the standard output of the domain. You can see some message list above. The other tool is to look over "/proc/xmtratum" to get the information about the load domain. There is a example in the XtratuM Installation.

3. Inter-Domain Communication Tools

After we successfully installed XM/eRTL in the previous section, we will now discuss the options XtratuM offers to establish a communication between two domains. There are two inter-domain communication tools available in XtratuM: XM/FIFO and XM/SHM.

So why do we need inter-domain communication? As described in the introduction, several widely independent domains are loaded on top of XtratuM. Each of these partitions has its own memory region, and is not allowed to access the memory of other domains. But although we want the domains to be independent, and keep them from directly influencing each other (possibly in a bad way), some communication between them might be necessary. This communication is conducted via the inter-domain communication mechanisms.

3.1. XM/FIFO

FIFO stands for "First In First Out". XM/FIFO is used for the communication between two XtratuM domains. A FIFO is defined by its name and its size. You can think of a fifo as a pipe to transmit a data stream. One domain writes the data to one port, and a second domain reads the data from the other port (the other side of the pipe). The data which has been read can not be read again (consuming read). The buffers used in XM/FIFO are ring buffers. Therefore it is possible for a FIFO to be *full*, and no data can be written to the fifo until the reader consumes data already in the FIFO. Of course it is also possible, that the FIFO is *empty*, this means, that the reader tries to fetch a message in the FIFO, but since the writer did not put any data into the FIFO, there is none available. If you want to know more about FIFO's, you can refer to the wiki:[http://en.wikipedia.org/wiki/FIFO_\(computing\)](http://en.wikipedia.org/wiki/FIFO_(computing)).

One simple use case of XM/FIFO is described in the following simple example: domain d1 collects data from some sensors, and domain d2 processes the data. The operations of d2 ought to be after the data has been collected by d1. So how can d2 get the data from d1? This can be realized simply with the help of XM/FIFO. After the data has been collected, d1 writes it into a fifo, and d2 reads the data from the fifo.

3.1.1. Characteristics

By default, there are 16 (the number is defined by `XM_FIFO_COUNT` in the file `xtratum/include/config.h`) fifos in XtratuM, named `rtf0`, `rtf1`, ..., `rtf15`. The FIFOs can be accessed by their name. In Linux (the root domain), these 16 fifos are made

accessible by the linux driver called `lffifo` in `/dev/`, so they can be used by accessing `/dev/rtf0`, `/dev/rtf1` and so on. In `XtratuM`, every domain can access these fifos with simple open, write, read and close operations. These operations are similar to the operations of ordinary files in Linux.

The storage size of each fifo is `PAGE_SIZE`. The memory is allocated at initialization time (when the modules are loaded), and they are in the memory all the time, until the `XM/FIFO` modules are unloaded again. In order to manage or control the FIFOs, a struct `fifo_struct` is used. This struct occupies 20-byte, including the fifo data memory address, flags that indicate whether the fifo can be used or not, the pointer to the next free address in the fifo (point to write the next address) and a pointer to the next message to be read. `XtratuM` uses an array to organize the 16 structs.

`XM/FIFO` consists of three parts: FIFO core, Linux Fifo device drive, Partikle Fifo device drivers. In the following, we will see how and where these 3 parts are used. If you are interested in the internals of `XM/FIFO`, you are welcome to read [3].

3.1.2. Using XM/FIFO

Now, that we know what `XM/FIFO` is, we will have a look at some examples, and learn how to use `XM/FIFOs`. As mentioned above, `XM/FIFOs` are accessible as devicefiles in linux and Partikle, therefore normal file operation routines can be used (open, close, read, write).

Ok, so lets have a quick look at a simple example. What we do know will be the typical case, a real-time domain (in our case a partikle domain) generates some data, which we want to read from a non-real-time domain (linux). The sent message contains a string of numbers (20byte in length), The content of the FIFO is then read in chunks of 39bytes. Figure 3.1.2 shows the way of the package.

The writer part is located in `xmertl-4.0-rc1/partikle/user/examples/c_examples/fwrite.c`:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

/* we are going to use rtf1 in this exmaple */
#define fifo_dev "/dev/rtf1"

int main(int argc, char *argvs[])
{
    /* our first message: */
    char message[] = {"123456789012345678901234567890"};
```

```
int fd;          /* file descriptor for our fifo */
int i, ret;      /* counter variable, return value */

/* open the FIFO, and check the return value for errors. */
if((fd = open(fifo_dev, ORDWR, O_EXCL)) < 0) {
    printf("open device error\n");
    return 0;
}

/* write the message 50 times into the FIFO, or until
 * the buffer is full, whatever comes first:
 */
for(i = 0; i < 50; i++) {
    ret = write(fd, message, 20);
    printf(" i= %d\n", i);
    if(ret <= 0) break; /* FIFO full !!! */
}

return 0;
}
```

I do not want to loose too much words about the code, I think it is very simple. Now let's build the example (and all the other partikle examples), and load it:

```
# cd xmertl-4.0-rc1/partikle/user/examples/
# make
# cd c_examples
# loader.xm fwrite.prtk
```

Now the partikle domain has written it's message into the FIFO. It stays there, until someone reads it:

```
# cd xmertl-4.0-rc1/xtratum/devs/rtfifo/linux/
# insmod lfifo
# cd xmertl-4.0-rc1/xtratum/devs/rtfifo/linux/examples/
# make
# ./fread
```

What we have done in the last few lines is the following: we inserted the XM/FIFO support into the linux kernel. This module allows us to open /dev/rtf1 from our linux user-mode example. Then we build the user-mode example and run it (./fread). The output of this example looks like this:

```

123456789012345678901234567890123456789
012345678901234567890123456789012345678
901234567890123456789012345678901234567
890123456789012345678901234567890123456
789012345678901234567890123456789012345
678901234567890123456789012345678901234
567890123456789012345678901234567890123
456789012345678901234567890123456789012
345678901234567890123456789012345678901
234567890123456789012345678901234567890
total = 390

```

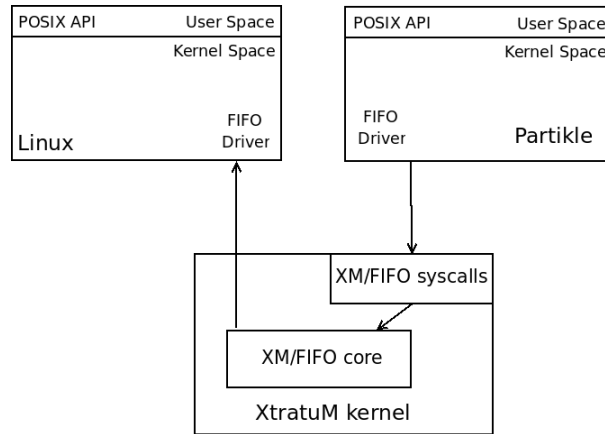


Figure 1: The way of our example messages.

3.2. XM/SHM

The second mechanism for inter-domain communication in XtratuM is XM/SHM, a shared memory system, where two or more processes share a single chunk of memory to communicate. The shared memory system can also be used to set permissions on memory regions, allowing e.g. malloc debuggers write into those regions. Shared memories are the most efficient way of IDC.

3.2.1. About XM/SHM

The shared memory in XtratuM maps pages into the address space of two (or more) domains of XtratuM (e.g. Linux and Partikle). This allows users to access these shared

memory regions in both, real-time domain and non-real-time domains.

The shared memory management module is the central part in the shared memory system, it uses the TLSF system, to allocate memory for the shared memory and maps the allocated space into the domain's memory region. Details about how this is done, can be found in [8]

The most important features of XM/SHM are:

- The shared memory system allows two (or more) domains to exchange data.
- Unlike FIFOs, SHM does not queue the data. This means, that any data written to the shared memory overwrites the previous data.
- Shared memory can also be accessed piecewise. Individual members of a large shared memory structure can be read and written independently.
- Reading data from the shared memory does not consume the data, that means, that several readers can read the same information.
- Because XM/SHM manages a shared resource, it has to ensure data consistency. In XtratuM, a real-time task can interrupt a Linux processes, but not vice-versa. There are two cases for a reader-writer pair:
 - the RT task is the reader, and it interrupts the Linux process during its write. Here, the RT task will see an inconsistent data structure, with fresh data at the beginning and stale data at the end.
 - The RT task is the writer, and it interrupts the Linux process during its read. Here, the Linux process will see an inconsistent data structure, with stale data at the beginning and fresh data at the end.

A typical application of XM/SHM is a single RT Linux task that writes a data array to the shared memory periodically, and a single Linux process reads this data periodically and prints the results. We will examine an example like this later.

3.2.2. XtratuM/SHM API

From linux and Partikle, XM/SHM can be accessed via device files (`/dev/shm[0-15]`) after the XM/SHM modules are loaded. Every domain of XtratuM can write and read the shm memory if it has access permission. This way, processes in different domains can communicate with each other.

The most efficient way to implement shared memory applications is to use on the `mmap()`

function and on the system's native virtual memory facility. The process of Partikle domain use `mmap()` to alloc memory from device `/dev/shm`.

open

mmap

close

3.2.3. Usage Example

In XM/eRTL, XM/SHM is used to allow communication between two (or more) domains (real-time or non-real-time), running on top of XtratuM.

The example we are going to run in the following will do almost the same as the FIFO example above, but this time we are going to send our message into the opposite direction.

```
# cd xmertl-4.0-rc1/xtratum/devs/shm/linux/examples
# make
# ./demo-writer
```

The result of this is, that the `demo-writer` program writes *"Hello World!"* into the beginning of the shared memory, and fills up the rest with the character `'c'`. Now we can start the `demo-reader` in partikle, and read all this.

```
# cd xmertl-4.0-rc1/partikle/user/examples/c_examples/shm
# make
# ./demo-reader
```

The result of this should be, that our message is displayed on the screen:

```
Hello World!cccccccccccccccccccccccccccccccccccccccccccccccccccccccc
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

4. Device Driver

One powerful function offered by embedded system is device driver. The system can get device status and send the response command to the device through device drivers. In order to make it easy to add real-time device driver for XMeRTL system, a device driver model named XM/DEV is design and implemented. The first subsection will show XM/DEV. Based on XM/DEV, an UART/Serial device driver is built. The UART/SERIAL device driver usage will be explained in the second subsection.

4.1. XM/DEV

XM/DEV is XtratuM device driver model. In the model, the device driver is divided into three parts: reused parts in linux, device runtime part which is an independant server domain, and driver interfaces for linux/domain tasks.

4.1.1. About XM/DEV

Nowadays most of operating systems support various of devices, and these device drivers run in kernel space which can response to the applications immediately. But there is a big problem in this model, device drivers and kernel run in the same address space will make the kernel unstable. So in some real-time embedded operating system, device drivers run as threads which are activated by interrupts. L4 and QNX is using this device driver model. In this kind of device driver model, applications must send request through the IPCs to the device driver sevice routine if they want to use the device. So it is thought less efficiency than the first model, but it is proved that this viewpoint is incorrect.

This model is mainly used in micro-kernel systems, but XtratuM is a nano-kernel system. Instead of the threads, domains are running above the core. Domains have their separated kernel space and user space. So the device drive is running as a domain in the system, and other domains use IDC (Inter-Domain Communication) tools to communicate with the device driver domain. There is the architecture of XtratuM device driver model.

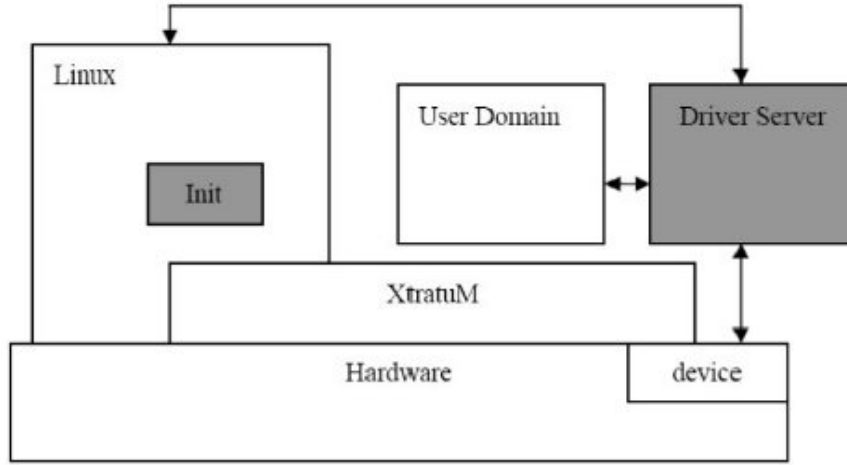


Figure 2 : XtratuM Device Driver Model Architecture

As we see in the graph, there are two parts of a XtratuM device driver. One is the initialization component in Linux and another is the real-time device driver run as a domain in the system. The initialization component in Linux continue to use the code provided by Linux which increase the reuse of the code. In the real-time device driver component, there are two functions: interrupt handle and the real-time device driver. The domain of device driver will be idle in the system until it is triggered by the interrupt.

4.1.2. Development

When we start to development a device driver in the system, there are two things to consider: resource management in Operating system and hardware. The first part of the device driver communicates with the operating system, it prepares the resources for the hardware management part. This part includes the resource requirement and management from the operating system, such as memory, interrupt handle and I/O ports etc. As we known, XtratuM only take care of the critical resource from Linux, such as memory, timer and interrupt. So this part uses the functions provided both by Linux and XtratuM. It will run as a module in Linux at the Linux kernel space. The second part communicates with a particular hardware, it takes care of device access and provides interface to the user in the system. This part runs as a domain in XtratuM and communicates with other domains by inter-domain communication tools, such as fifo and shared memory.

Before we start to explain how to add a device driver in the system, we should know the structure of XtratuM source tree, it is shown in the figure as following:

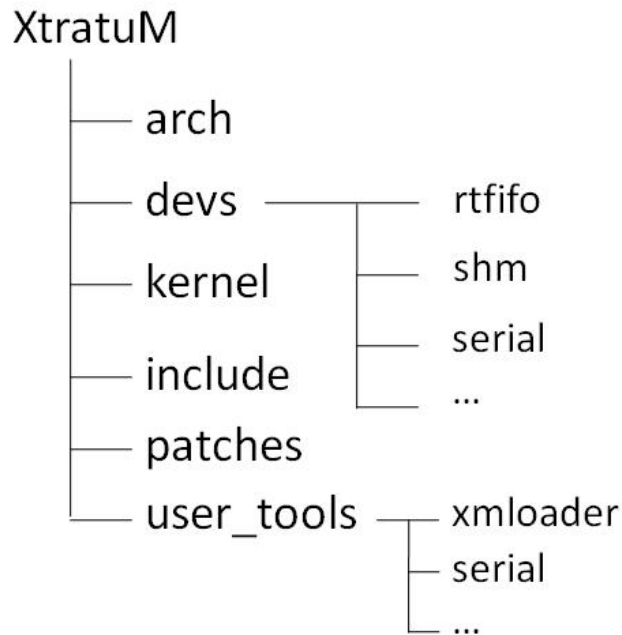


Figure 3 : XtratuM source code structure

There are two import directories when you add a device driver in XtratuM: `devs` and `user_tools`. We put the first part of device driver under the directory `devs` and called it initialization component, and the second part under the directory `user_tools` called device driver server.

- Initialization component

We put different device drivers in different directories as shown in the above figure. The access interface from Linux side is also implemented under the directory following the device class given by Linux kernel, such as character devices and block devices. The access interface make it is possible to access the device from Linux user space. So in the example of serial port device driver, there are two parts in this directory: initialization component and Linux interface.

As we mentioned above, the initialization component should request three kinds of resources from the system: IO port allocate, smemory management, irq handler and device register. And we also need to register the device to XtratuM in this componenet.

- IO port allocate

IO port is used by device driver to communicate with the hardware. It is managed by Linux kernel. Here is the functions provided by Linux kernel.

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long first, unsigned long n,
const char *name);
```

The parameters of this function tell the kernel that the device driver wants to use *n* ports from the first, and the name is the symbol of your device driver. It will return a address of resource structure in the normal condition. If it returns a NULL, the device driver will not use the ports it is expected. So check the return value of `request_region()` in your device driver.

There is also a function to check the use status of a set of ports.

```
int check_region(unsigned long first, unsigned long n);
```

It will check the use status of *n* ports from first. But there is a possibility that the ports is allocated before the device driver request the ports after checking it. This function is not commended to use.

And when everything is done, you should release the resource to system by using the function `release_region()`. The parameter start is corresponding with first.

```
void release_region(unsigned long start, unsigned long n);
```

– Interrupt allocate

An interrupt is always used to call the attention of the system to handle the event happened on devices. An interrupt is consisted by two parts: low-level part and high-level part. The high-level part is about the registration in the system which will implement in this component. And the low-level part is about how to deal with the interrupt, in another word it is a interrupt handler which will implement in the device driver server. There is a pointer in the interrupt structure which is pointed to the interrupt handler. But in our device driver model, we don't use the routine in Linux to handler the interrupt. We will implement the server domain for devices, so the interrupt handler here will be a blank routine which only return 0. As we said above this component only focus on the high-level parts of the interrupt: register and trigger the interrupt in the system.

We can use the functions provided by Linux kernel to register and release an interrupt.

```
#include <linux/interrupt.h>
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
                unsigned long irqflags,
                const char *devname,
```

```
void *dev_id);
```

This function is used to fill the `irq_action` structure in `irq` description array. The structure is as following. The parameters are corresponded with the items in the structure.

```
struct irqaction {
    irq_handler_t handler; /* point to the irq server routine */
    unsigned long flags; /* interrupt flags */
    unsigned long mask; /* interrupt mask */
    const char *name; /* name of the I/O device */
    void *dev_id; /* device identification mark */
    struct irqaction *next; /* point to the next structure */
    int irq; /* the irq number required */
    struct proc_dir_entry *dir; /* point to the directory of /proc/irq/ */
};
```

The purpose of using this function is just to register and trigger the interrupt in the system. So we use `irq` number to identify the device driver instead the `dev id` in Linux. So in your device driver you can fill the last parameter with `NULL`. And the flag can decide the access model for your device driver, and there are several options:

* `SA_INTERRUPT`

This kind of interrupt server routine will be executed with interrupts disabled .

* `SA_SHIRQ`

This interrupt can be shared between devices.

* `SA_SAMPLE_RANDOM`

This means the devices must can return random numbers which can be use in the entropy pool used by `/dev/random` and other device like it. This bit indicates that the generated interrupts can contribute to the entropy pool.

As the IO ports, after you use the `irq_handler` you should release the resource by `free_irq()`. Here use the `irq` number to identify which `irq_handler` should be released.

```
void free_irq(unsigned int irq, void *dev_id);
```

– Memory allocate

The memory allocate is using the functions provided by Linux to get a

block of physical memory and using the function provided by XtratuM to implement the memory map from the separate domain space. It can map the domain space to the IDC, such as fifo and shared memory, which will provide the communication mechanism between device driver and domains using the device.

```
#include <linux/mm.h>
```

```
unsigned long __get_free_page(unsigned int flags);
```

```
void free_page(unsigned long addr);
```

`__get_free_page()` can allocate a whole page of physical memory for device driver, and the flags in `__get_free_page` are as following:

- * `GFP_KERNEL`

This means the memory allocated is a normal for kernel which can sleep.

- * `GFP_ATOMIC`

This kind of memory is allocated for something outside of the process context just like interrupt handlers, it can not sleep.

- * `__GFP_DMA`

Choice this one if your device can access memory through ISA directly.

- * `__GFP_HIGHMEM`

This flag means the page is allocated for high memory.

After get the physical memory, you still need to build the memory map from the hardware to the device driver server domain.

```
/* You can get a page directory entry from virtual address*/
```

```
#define va2pd(vaddress) (vaddress >> PGDIR_SHIFT)
```

```
/* You can get a page table entry from virtual address*/
```

```
#define va2pt(vaddress) ((vaddress & 0x3FF000) >> PAGE_SHIFT)
```

```
/* You can get a virtual address with the page directory and page table*/
```

```
#define pdpt2va(pd, pt) ((pd << PGDIR_SHIFT) | (pt << PAGE_SHIFT))
```

So with these three functions you can implement the transform between virtual address and physical address which makes it is easier to map the physical address to a domain. And with these functions you can implement your own map handler and unmap handler which is used in the device register. The map handler function is used to allocate a page for the application. If the pd

of page required is not in the memory, it will allocate a new page and set the flags of this page, or it will just get the page. Then it check the pt value of the required page and fill it. It is like the example.

```
static unsigned long xm_map_page(unsigned long pd,
                                unsigned long addr, unsigned long vaddr, unsigned long flag)
{
    unsigned long pd_entry = va2pd(vaddr);
    unsigned long pt_entry = va2pt(vaddr);
    unsigned long nflag;
    unsigned long pt;

    if( !((((unsigned long *)pd)[pd_entry]) & _PAGE_PRESENT) ) {
        pt = __get_free_page(GFP_ATOMIC);
        if( !pt ) return -ENOMEM;
        memset((unsigned char *)pt, 0, PAGE_SIZE);
        nflag = flag & (~_PAGE_UNUSED1);
        fill_pd_entry (pd, pd_entry, pt, nflag);
    } else {
        pt = get_pd_addr(pd, pd_entry);
    }

    nflag = flag;
    if ( !((((unsigned long *)pt)[pt_entry] & _PAGE_PRESENT)) {
        fill_pt_entry (pt, pt_entry, addr, nflag);
    }

    return 0;
}
```

About the flags meanings, you can check here:

```
#define _PAGE_PRESENT 0x001 /*If the page is in memory*/
#define _PAGE_RW 0x002 /*read/write*/
#define _PAGE_USER 0x004
#define _PAGE_PWT 0x008
#define _PAGE_PCD 0x010
#define _PAGE_ACCESSED 0x020
#define _PAGE_DIRTY 0x040
#define _PAGE_PSE 0x080/*4 MB (or 2MB) page, Pentium+, if present*/
#define _PAGE_GLOBAL 0x100/*Global TLB entry PPro+*/
#define _PAGE_UNUSED1 0x200//available for programmer
```

```
#define _PAGE_UNUSED2 0x400
#define _PAGE_UNUSED3 0x800
```

It is defined in pgtable.h under the architecture directory.

The memory unmap handler is more easier than the map handler. It only get the page and then fill the pt_entry with 0 which means the page is not used.

```
static int xm_unmap_page(unsigned long pd,
                        unsigned long vaddr)
{
    unsigned long pd_entry = va2pd(vaddr);
    unsigned long pt_entry = va2pt(vaddr);
    unsigned long pt;

    if (((unsigned long *)pd)[pd_entry] & _PAGE_PRESENT) {
        pt = get_pd_addr(pd, pd_entry);
        if (((unsigned long *)pt)[pt_entry] & _PAGE_PRESENT) {
            ((unsigned long *)pt)[pt_entry] = 0UL;
        }
    }

    return 0;
}
```

– Device register

The device driver register function in XtratuM is implemented in xmdev.c under directory XtratuM_PATHkernel.

```
#include <xmdev.h>
int xm_dev_register(int devid, xm_dev_t *dev);
int xm_dev_unregister(int devid);
```

The devid here is the device driver number defined in the xmdev.h, you should add it when you implement your own device driver. dev is a structure of your device driver it defined the operations and parameters of your device drive. You should initialize it before you use it.

```
#include <xmdev.h>
typedef struct
xm_dev_struct
{
    int devid;
    int (*dev_map_handler)(unsigned long,
                          unsigned long, unsigned long, void *);
```

```
int (*dev_unmap_handler)(unsigned long ,
                          unsigned long , unsigned long);
int (*dev_ioctl_handler)(unsigned long minor ,
                          unsigned long cmd, void *);
int client;
} xm_dev_t;
```

The `xm_serial_xm_ioctl` is used to set the page flag for the memory your device gets. So you must pass several parameters to it such as: the device number you want to use(minor), the operations you want to do(cmd) and so on. The client here is a number of applications which is using this device. It should be updated when the device is required and released. It is used to judge if there are applications is using this device and can the server domain exit normally.

Take the serial port driver as an example:

```
/* Don't change the value, unless you know what they are */
#define XM_FIFO_DEV 0
#define XM_SHM_DEV 1
#define XM_SERIAL_DEV 2
```

This is a piece of code from `dev.h` in directory `include` of XtratuM source tree. We define the devid of XtratuM in this file. Then you can use it in your own device driver.

```
xm_dev_t xm_serial_dev = {
    XM_SERIAL_DEV,      /*the device id definid in dev.h*/
    xm_serial_xm_map ,
    xm_serial_xm_unmap ,
    xm_serial_xm_ioctl ,/*operation routine of device driver*/
    0                   /*the initialization value is 0*/
};
```

Then you can register or unregister the structure to or from the system with the functions just like this:

```
xm_dev_register(XM_SERIAL_DEV, &xm_serial_dev);
xm_dev_unregister(XM_SERIAL_DEV);
```

This part only requires the resources from the system, and the handler routine will implement in device driver server domain.

So the pseudocode is like this:

```

xm_dev_t xm_dev = {};
struct resource *dev_res;
int xm_devl_xm_init(void)
{
    dev_res = request_region();
    init_xm_dev_parameter(); /* initialization some value in
                               * structure xm_dev as it needs*/
    xm_mem_map();
    xm_dev_register();
    xm_request_irq();
}
void xm_serial_xm_exit(void)
{
    xm_free_irq();
    if(xm_dev.client > 0) {
        XMBUG();
        return;
    }
    xm_dev_unregister();
    xm_mem_free();
    release_region();
}

```

- Device driver server

Device driver server is run as a domain in XtratuM, we put this under the XtratuM_PATHuser_tools. In this component we will implement the access functions, such as readwrite functions through the IDC, irq handler routines and the scheduler functions. You can set up your own scheduler queues with your scheduler method in this component.

- IO ports

The access functions is as following:

```

#include <xmdev.h>
#include <asm/io.h>
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);

```

The parameter port is the port which is required in the initialization component, and the byte means the size you want to write to the port.

- interrupt handler

An interrupt handler is a routine which is triggered by an interrupt. The high-level counterpart of an interrupt handler is an event handler in the device driver.

The interrupt handler is declared in the initialization component which is shown in the above part, and the interrupt handler routine is implemented here. According the different condition of the hardware, you can implement it in different ways. Take the serial port driver as an example. There are several methods to access the hardware, so we get the access method by the parameter passed by the function that user is used. Then based on this flag, you should run the corresponding function to communicate with the hardware.

- Memory access

You should choose an inter-domain communication tool to support your server domain to communicate with the user domains. Now there are two IDCs in the system: fifo and shared memory. The first one is suitable for the short messages and the second one is suitable for big block of data which will be modified partly.

And you also need to initialize the memory resource in the system call: load domain sys in kernel/syscalls.c. It makes it possible to allocate the memory resource for a domain when it is loaded.

```
#if XM_SERIAL_DEV_SUPPORT
fvaddr = DEFAULT_SERIAL_ADDR;
if ((dev = xm_get_dev(XM_SERIAL_DEV)) == NULL) {
    destroy_domain(d);
    id = -ENODEV;
    goto exit_load_domain;
}
if (is_serial_dev(d)) {
    dev->dev_map_handler(d->pd, fvaddr,
        XM_SERIAL_COUNT*2*PAGE_SIZE, NULL);
    d->events->serialaddr = fvaddr;
}
#endif
```

The value of XM_SERIAL_DEV_SUPPORT is defined in include/config.h. And can be modified by your configuration file.

- Add the event handler

As we said above, the high-level of an interrupter handler is event handler. An event handler in XtratuM mainly used to register the interrupt handler routine to the event, and provide some sharing resources among independent domains. In our case, we register the interrupt handler to the device event and provide the defined memory address to the domains. The event handler structure in `include/events.h` is as following:

```
typedef struct event_handling_struct {
    volatile bitmap_t intercepted_events;
    volatile bitmap_t masked_events;
    volatile bitmap_t pending_events;
    handler_t event_handler [MAXEVENTS];
    handler_t trap_handler [MAXTRAPS];

    volatile unsigned long event_flags;
    unsigned long *heap;
    unsigned long heap_size;

    /* XtratuM FIFOi */
    unsigned long fctladdr;
    unsigned long fdataddr[XM_FIFO_COUNT];

    /*Share Memory*/
    unsigned long shmctladdr;
    unsigned long shmdataddr[XM_SHM_COUNT];

    /* Serial memory */
    unsigned long serialaddr;
} event_handling_t;
```

We can see this structure defined serveral event status, hold on the event_handler array and record some shared resource information. The value of `DEFAULT_SERIAL_ADDR` is defined in `include/config.h`. The `DEFAULT_SERIAL_ADDR` is defined as `0x34000000`, and the `XM_SERIAL_COUNT` is 16. So we default to set the memory space using by serial port device is from `0x34000000` to `38000000`, it is about 4M in our implement. And you can define yours in the same file.

The function of register the event handler to this structure is:

```
#include <xm_syscalls.h>
int install_event_handler (int event, handler_t h);
```

This function will add the event to the event_handler array, associate the handler routine to the event and set the bit for your event in this structure.

Because there are many options for a device, our advice is put them all in a structure. This structure includes the irq number of your device and other import options of your device. In the serial port example, we put this in the device structure:

```
typedef struct
uart_dev_struct
{
    int irq;
    int base;
    int parity;
    int speed;
} uart_dev_t;
```

The pseudocode of the device driver server domain should like this:

```
int kmain(void)
{
    uart_dev_init(); /* get the sharing resource by
                     * event_handling_struct, and
                     * set other parameters according your device*/

    struct xmitimerval req = {};
    set_timer (&req, 0); /* initialization a timer
                          * for the domain schedule*/

    install_event_handler(event, event_handler);

    unmask_event(event);
    enable_events_flag(); /* set the bit of event_handling_struct
                          * and make this event activate
                          * in the system */

    for (;;) {
        suspend_domain(0, 0);
    }
```

```
        return 0;
    }
```

4.2. XM/Serial Driver

4.2.1. About XM/Serial Driver

1) WHAT IS XM/SERIAL DRIVER

Xm/serial means that do communication through serial ports based on XtratuM and the xm/serial driver is the thing that can support user to communicate by serial port. It meets the real-time requirement. In XtratuM, one part of XM/SERIAL DRIVER runs as a XtratuM domain. so the developer can use the XM/SERIAL DRIVER to do communication through serial ports to meet their own needs by modify the driver domain.

2) XM/SERIAL DRIVER CHARACTERISTICS

UART is used to connect a number of standards to support the Universal Serial communications equipment ,so the driver developer only needs to write driver for UART . In Xtratum, the uart device is divided into two parts, one is loaded into kernel as module which is used to request resources for serial device, such as locating memory for FIFO, the UART's address space and so on, the other is loaded as a domain which initialize the device driver. There is only one device domain, reads from serial fifo or write data to serial fifo, so there is no concurrent problem.

There are 16 pairs of Lock-Free FIFO for IDC(inter-domain-communication), There are two fifos for each pair and the fifo is un-direction, one fifo send data from guest domain to server domain and the other fifo is opposite. Each pair of fifo can not be shared by multiple guest domains.

An scheduler based priority is provided in the uart device driver server. There will be an interrupt when Transmitter Holding Register is empty and this interrupt will trigger uart device driver domain which will choose the nonempty fifo which has highest priority an dada source by scheduler. If there is no data in all FIFOs, the Transmitter Holding Register's null value interrupt will be shielded and the null value interrupt can start when guest domain write data to fifo.

The serial driver domain was defined in serial.c in which the serial port address and baudrate defined, such as "static uart_dev_t uart={4,0x3f8,0,115200}" ,the meaning of the four parameter is irq, port address, the parity and baudrate. Another important function is vuart_device_init(void). In this function, the address for 16 fifos which used to make communication are located.

3) THE USE OF XM/SERIAL DRIVER

The UART driver domain can initialize the hardware when device running and support the implementation of the special operation for the device. One of its important use is

to serve the guest domains to meet their communication needs and developers can use it easily to communicate with other domain by serial ports.

One of its benefits for the driver domain is that it can decrease the threat to the kernel, it runs as domain just like a thread and the developer can modify the driver domain by modifying the serial.c, such as modifying baudrate in "static uart_dev_t uart={4,0x3f8,0,115200}" and another parameters.

4.2.2. Using XM/Serial Driver

1) About XMSerial Driver

Here will introduce how to build guest domains and use them to do communication by serial ports. The maximal device number and minimal device number is fixed in the guest domains. The maximal port device number is 2, the minimal number is to represent different fifos,

When the guest domain is loaded, it will call `suspend_domain()` to initialize device, then the guest domain will set the attribute for serial port, such as priority and speed and so on.

There are functions `dom_serial_fifo_write()` and `dom_serial_fifo_read()` in uart domain and they realize how to write and read data detailed. so you have to add these two function in your guest domain. The function `rt_serial_write(const char *src, int size)` which calls `dom_serial_fifo_write()` is used to write data whose address is `src` to the serial port and its length equals to `size`. Function `rt_serial_read(char *dst, int size)` which calls `dom_serial_fifo_read()` is used to receive data from the serial port, the data's length equals to `size`. So the developer only have to know how to use `rt_serial_write(const char *src, int size)` and `rt_serial_read(char *dst, int size)` and that is enough.

As given example on this version XM/eRTL, there are two fundamental examples, developer can change the baudrate by the parameter for function `serial_device_init(unsigned int ttys, long baud)`, the `ttys` represents fifo which `0<=ttys<16`. In the `kmain()`, developers can do what they want to do, such as sending "Hello, XM/eRTL" to another computer by serial ports, but XM/eRTL has to be installed on the other computer and the uart driver domain has been loaded.

Generally speaking, what developers have to do is to give parameters to `serial_device_init(unsigned int ttys, long baud)`, `rt_serial_read(char *dst, int size)` and `rt_serial_write(const char *src, int size)`.

So if you want to use the UART driver to do serial port communication, firstly, the UART Driver domain should be loaded,

2) XM/SERIAL DRIVER DOMAIN EXAMPLE

To do communication through serial ports, you should first make sure you have installed XM/eRTL and have loaded the uart driver domain "serial.xmd". Next, you can build your own guest domain.,

Here is an simple example will be introduced. The server will send the message " UART TEST SCCUCCESSFULLY " to the client and the client will display what it has received(Both server.c ,client.c and some comments you can see them at Appendix III). In the server domain, the function kmain() which is in server.c is as following: ‘

```
static char string[40] = {"UART_TEST_SCCUCCESSFULLY\n"};

int kmain (void)
{
    struct xmitinterval req = {{0, 200000}, {0, 0}};
    long baud =115200;
    int ret;
    ret = serial_device_init(my_Console_ttyS, baud);
        //initialize device with
        // your fifo number and baudrate
    set_timer(&req, 0);
    install_event_handler(0, timer_handler);
    unmask_event(0);
    enable_events_flag();
    suspend_domain (0, 0);
    rt_serial_write(string,40);
    return 0;
}
```

In the client domain, the function kmain() which is in client.c is as following:

```
static char string[100];

int kmain (void)
{
    struct xmitinterval req = {{0, 200000}, {0, 0}};
    long baud =115200;
    int ret;
    ret = serial_device_init(my_Console_ttyS, baud);
    set_timer(&req, 0);
    install_event_handler(0, timer_handler);
    unmask_event(0);
    enable_events_flag();

    while (1) {
        suspend_domain (0, 0);
    }
```

```
        ret = rt_serial_read(string, 40);
            //receive data to string
        write_scr(string, ret);
            //output content of string
    }

    return 0;
}
```

Then we can load these two domains on two computers(both computers must be installed XM/eRTL, and connected by serial port). one computer is server, and the other is client. The client computer:

```
$cd [XM/eRTL directory]/xtratum/user_tools
$./scripts/xmcmd.sh -l //load modules such as xm., serial, rtfifo and so on
$xmlloader/loader.xml serial/serial.xmd //load uart driver domain by loader.xml
>> Loading the domain "serial.xmd (serial.xmd)" ... Ok (Id: 1)
$dmesg
<XtratuM> VUART device loaded!
$xmlloader/loader.xml tests/client.xmd
```

Now, the client is waiting for data which will be sent by the server.

The server computer:

```
$cd [XM/eRTL directory]/xtratum/user_tools
$./scripts/xmcmd.sh -l
$xmlloader/loader.xml serial/serial.xmd
$xmlloader/loader.xml tests/server.xmd
```

on the client:

```
$dmesg
then will get the following information:
UART TEST SCCUCESFULLY
```

The "UART TEST SCCUCESFULLY " was transported by serial port, the result indicates that the driver domain running normally and the client gets what it wants to get.

This is a simple example that the developer can use the UART driver domain to meet their own needs, then can modify the baudrate such as 9600, 38400 and

5. *xmtrace* - Trace Tool

xmtrace is a tracing tool based on the *rtl_tracer* included in RTLinux/GPL, written by Michael Barabanov, and has been reimplemented for the use in XtratuM. This manual only covers the use of *xmtrace*, detailed information on how *xmtrace* works internally can be found in [6]. This section gives a step by step introduction on how to run the simple example, and after that the code of the simple example is explained to allow you to use *xmtrace* in your application.

5.1. Run an Example

The first thing what we have to do, is to load the *xmtrace_shm.ko* Linux kernel module. This module gets the address of the shared memory devices in use and passes the address to the part of the tracer that is located in the core. At the end of the initialization, it starts the tracer. From this point in time the tracer is running, and tracing events can be retrieved.

```
$ insmod xmtrace/xmtrace_shm.ko
```

```
$ cd xmtrace
$ ./tracer > data.raw
```

switch to a second console

```
$ cd user_tools/examples/tracer
$ ../../xmloader/./load.xm tracer.xmd
```

If you switch back to the first console now, you can stop *./tracer*, and have a look at *data.raw*. In it you will see the output of the tracer that looks something like this:

```
....
sorry no trace ready at cpos 0 (data_ready) <00000000> (overwritten) 0 finalize 0
sorry no trace ready at cpos 0 (data_ready) <00000000> (overwritten) 0 finalize 0
sorry no trace ready at cpos 0 (data_ready) <00000000> (overwritten) 0 finalize 0
DUMP: cpos 0 (data_ready) <7fffffff> (overwritten) 70 finalize 101
D 0 -181177411 hw_save_flags          0x82 <0xce8200cc>
D 0   165 hard cli                    0 <0xce8200a6>
```

```

D 0    177 hw_restore_flags      0x82 <0xce8200e9>
D 0  13435 hw_save_flags         0x82 <0xce8200cc>
D 0    206 hw_save_flags         0x96 <0xce8200cc>
D 0    327 hard cli              0 <0xce8200a6>
D 0    178 hw_restore_flags      0x96 <0xce8200e9>
D 0  15054 hw_save_flags         0x96 <0xce8200cc>
D 0    164 hard cli              0 <0xce8200a6>
D 0   3257 hw_save_flags         0x92 <0xce8200cc>
D 0    165 hard cli              0 <0xce8200a6>
D 0   3801 hw_restore_flags      0x92 <0xce8200e9>
D 0    181 hw_restore_flags      0x96 <0xce8200e9>
D-1   4013 scheduler in          0 <0xce82245b>
D-1    328 hw_save_flags         0x82 <0xce8200cc>
D-1    183 hard cli              0 <0xce8200a6>
D-1   5906 hw_save_flags         0x82 <0xce8200cc>
D-1    440 hard cli              0 <0xce8200a6>
D-1    714 hw_restore_flags      0x82 <0xce8200e9>
D-1    977 hw_save_flags         0x92 <0xce8200cc>
D-1    163 hard cli              0 <0xce8200a6>
D-1    393 hw_restore_flags      0x92 <0xce8200e9>
D-1    170 hw_save_flags         0x96 <0xce8200cc>
D-1    368 hard cli              0 <0xce8200a6>
D-1    455 hw_save_flags         0x92 <0xce8200cc>
D-1    163 hard cli              0 <0xce8200a6>
D-1   3762 hw_restore_flags      0x92 <0xce8200e9>
D-1    171 hw_restore_flags      0x96 <0xce8200e9>
D-1    173 scheduler in          0 <0xce82245b>
D-1    162 hw_save_flags         0x92 <0xce8200cc>
D-1    162 hard cli              0 <0xce8200a6>
D-1    246 hw_save_flags         0x96 <0xce8200cc>
D-1    163 hard cli              0 <0xce8200a6>
D-1    627 hw_save_flags         0x82 <0xce8200cc>
D-1    406 hard cli              0 <0xce8200a6>
D-1    501 hw_restore_flags      0x82 <0xce8200e9>
D-1    177 hw_restore_flags      0x96 <0xce8200e9>
D-1    177 hw_restore_flags      0x92 <0xce8200e9>
D-1    164 scheduler out         0x1 <0xce8225ed>
D 1   2920 xm-domain             0x1267 <0xce8213fe>
That was trace # 1 (lost 0 traces befor this trace)
DUMP: cpos 1 (data_ready) <7fffffff> (overwritten) 70 finalize 101
D 1 -181114183 hw_save_flags     0x92 <0xce8200cc>

```

```

D 1    193 hard cli                0 <0xce8200a6>
D 1    259 hw_save_flags           0x86 <0xce8200cc>
D 1    354 hard cli                0 <0xce8200a6>
D 1    489 hard sti                0 <0xce820086>
D 1    303 hw_save_flags           0x296 <0xce8200cc>
D 1    169 hard cli                0 <0xce8200a6>
D 1    541 hw_restore_flags        0x296 <0xce8200e9>
D 1    395 hard cli                0 <0xce8200a6>
D 1    205 hw_restore_flags        0x86 <0xce8200e9>
D 1    200 hw_restore_flags        0x92 <0xce8200e9>
D 1    425 xm-domain               0 <0xce8213fe>
That was trace # 2 (lost 0 traces before this trace)
DUMP: cpos 2 (data_ready) <7fffffff> (overwritten) 70 finalize 101
D 1 -181110149 xm-domain           0x1 <0xce8213fe>
That was trace # 3 (lost 0 traces before this trace)
DUMP: cpos 3 (data_ready) <7fffffff8> (overwritten) 70 finalize 101
D 1 -181109454 xm-domain           0x2 <0xce8213fe>
....

```

First you see some lines where no trace events have happened, followed by the traces. Each trace starts with a line similar to this:

```
DUMP: cpos 2 (data_ready) <7fffffff> (overwritten) 70 finalize 101
```

and ends with a line like this:

```
That was trace # 3 (lost 0 traces before this trace)
```

Between these two lines, the trace events in the last trace are listed. Lets see what a trace event looks like:

```
D-1    164 scheduler out           0x1 <0xce8225ed>
```

D-1 denotes the which domain's context the trace event has happened. In our example it happened in the XtratuM kernel context, therefore it is D-1. If you look at the trace, you will also find trace events that happened in domain 0 and domain 1.

Next there is the relative time to the last trace event in nanoseconds (e.g. 164ns), followed by the name of the trace event (e.g. scheduler out), an optional value that

is passed to the tracer (e.g. 0x1 - the number of the domain scheduled during this scheduler call), and the address of the tracepoint.

In the next step, we are going to resolve the addresses of the tracepoints into symbols. To do this, we first have to create a system map, and then send the tracing data into the symbol resolve script.

```
$ cd xmtrace
$ ./createmap.sh
$ cat data.raw | ./addr2sym.sh -m MySystem.map > data.lst
```

This creates a version of the trace from above, where the addresses are resolved into symbol names + offsets:

```
....
sorry no trace ready at cpos 0 (data_ready) <00000000> (overwritten) 0 finalize 0
sorry no trace ready at cpos 0 (data_ready) <00000000> (overwritten) 0 finalize 0
sorry no trace ready at cpos 0 (data_ready) <00000000> (overwritten) 0 finalize 0
DUMP: cpos 0 (data_ready) <7fffffff> (overwritten) 70 finalize 101
D 0 -181177411 hw_save_flags                                0x82 <hw_save_flags+0x1c
D 0    165 hard cli                                         0 <hw_cli+0x16
D 0    177 hw_restore_flags                                0x82 <hw_restore_flags+0x19
D 0 13435 hw_save_flags                                    0x82 <hw_save_flags+0x1c
D 0    206 hw_save_flags                                    0x96 <hw_save_flags+0x1c
D 0    327 hard cli                                         0 <hw_cli+0x16
D 0    178 hw_restore_flags                                0x96 <hw_restore_flags+0x19
D 0 15054 hw_save_flags                                    0x96 <hw_save_flags+0x1c
D 0    164 hard cli                                         0 <hw_cli+0x16
D 0   3257 hw_save_flags                                    0x92 <hw_save_flags+0x1c
D 0    165 hard cli                                         0 <hw_cli+0x16
D 0   3801 hw_restore_flags                                0x92 <hw_restore_flags+0x19
D 0    181 hw_restore_flags                                0x96 <hw_restore_flags+0x19
D-1  4013 scheduler in                                     0 <xm_sched+0x1b
D-1    328 hw_save_flags                                    0x82 <hw_save_flags+0x1c
D-1    183 hard cli                                         0 <hw_cli+0x16
D-1   5906 hw_save_flags                                    0x82 <hw_save_flags+0x1c
D-1    440 hard cli                                         0 <hw_cli+0x16
D-1    714 hw_restore_flags                                0x82 <hw_restore_flags+0x19
```

```

D-1    977 hw_save_flags          0x92 <hw_save_flags+0x1c
D-1    163 hard cli              0 <hw_cli+0x16
D-1    393 hw_restore_flags      0x92 <hw_restore_flags+0x19
D-1    170 hw_save_flags        0x96 <hw_save_flags+0x1c
D-1    368 hard cli              0 <hw_cli+0x16
D-1    455 hw_save_flags        0x92 <hw_save_flags+0x1c
D-1    163 hard cli              0 <hw_cli+0x16
D-1   3762 hw_restore_flags      0x92 <hw_restore_flags+0x19
D-1    171 hw_restore_flags      0x96 <hw_restore_flags+0x19
D-1    173 scheduler in          0 <xm_sched+0x1b
D-1    162 hw_save_flags        0x92 <hw_save_flags+0x1c
D-1    162 hard cli              0 <hw_cli+0x16
D-1    246 hw_save_flags        0x96 <hw_save_flags+0x1c
D-1    163 hard cli              0 <hw_cli+0x16
D-1    627 hw_save_flags        0x82 <hw_save_flags+0x1c
D-1    406 hard cli              0 <hw_cli+0x16
D-1    501 hw_restore_flags      0x82 <hw_restore_flags+0x19
D-1    177 hw_restore_flags      0x96 <hw_restore_flags+0x19
D-1    177 hw_restore_flags      0x92 <hw_restore_flags+0x19
D-1    164 scheduler out         0x1 <xm_sched+0x1ad
D 1    2920 xm-domain            0x1267 <system_call_handler_asm_0x82+0x26
That was trace # 1 (lost 0 traces befor this trace)
DUMP: cpos 1 (data_ready) <7fffffff> (overwritten) 70 finalize 101
D 1 -181114183 hw_save_flags    0x92 <hw_save_flags+0x1c
D 1    193 hard cli              0 <hw_cli+0x16
D 1    259 hw_save_flags        0x86 <hw_save_flags+0x1c
D 1    354 hard cli              0 <hw_cli+0x16
D 1    489 hard sti              0 <hw_sti+0x16
D 1    303 hw_save_flags        0x296 <hw_save_flags+0x1c
D 1    169 hard cli              0 <hw_cli+0x16
D 1    541 hw_restore_flags      0x296 <hw_restore_flags+0x19
D 1    395 hard cli              0 <hw_cli+0x16
D 1    205 hw_restore_flags      0x86 <hw_restore_flags+0x19
D 1    200 hw_restore_flags      0x92 <hw_restore_flags+0x19
D 1    425 xm-domain            0 <system_call_handler_asm_0x82+0x26
That was trace # 2 (lost 0 traces befor this trace)
DUMP: cpos 2 (data_ready) <7fffffff> (overwritten) 70 finalize 101
D 1 -181110149 xm-domain        0x1 <system_call_handler_asm_0x82+0x26
That was trace # 3 (lost 0 traces befor this trace)
DUMP: cpos 3 (data_ready) <7fffffff> (overwritten) 70 finalize 101
D 1 -181109454 xm-domain        0x2 <system_call_handler_asm_0x82+0x26

```

....

5.2. Using the Tracer

This section analyzes the example that has been run in section 5.1, and shows you how to add tracing events into your application, and how to create new events and event-classes. This simple example contains everything you need to trace your own applications. The example installs a simple timer interrupt handler, which prints a message to the screen. The timer is configured to be trigger an interrupt every 200 μ s.

The first thing we do, is to define the two tracing events we will need: `XMTRACE_USER` is the event from the users domain and `XMTRACE_FINALIZE` is used to trigger a finalize event, which - if detected - causes xmtrace to dump the collected data. Then we write the interrupt handler function. All it does is to print our message *hola* on the screen:

```
#include <xm_syscalls.h>

#define XMTRACE_USER      20
#define XMTRACE_FINALIZE  18

void handler (int event, void *a) {
    write_scr("hola\n", 6);
}
```

Next thing to do, is to write our main function. It contains all the code for configuration and initialization of the timer, as well as the while (1) loop.

```
int kmain (void) {
    /* requirements for the timer */
    struct xmitimerval req = {{0, 200000}, {0, 0}};
    /* variable to hold the timestamp */
    struct xmtimespec t;
    /* counting variable */
    int i;

    /* install the handler we want XtratuM to
       call in case of an timer interrupt: */
    install_event_handler (0, handler);
    /* set the timer to our requirements (200us period): */
    set_timer (&req, 0);
    /* unmask the timer event, so it can be triggered: */
    unmask_event (0);
```

```
    /* get the current time: */  
    get_time(&t);
```

After everything is ready to go, we generate a first tracer event. Since we want all the traces that happen before our tracer event, we directly after our user event call the finalize event:

```
    /* trigger a tracer event.  
       since the second argument is shown in the list of trace events,  
       we pass it some arbitrary value, just to see something... */  
    xmtrace_user(XMTRACE_USER, 4711);  
    xmtrace_user(XMTRACE_FINALIZE, 0);  
  
    enable_events_flag();
```

To trigger more than this one trace event, we put the whole thing into a loop and let it generate trace events periodically (since the domain is waked up, every time a timer interrupt occurs).

```
/* Do this forever (or until I tell you to stop :- ) */  
    while (1){  
        /* generate a hundred trace events, give the loop counter as  
           an argument to xmtrace.  
        */  
        for (i=0; i< 100; i++){  
  
            xmtrace_user(XMTRACE_USER, i);  
            xmtrace_user(XMTRACE_FINALIZE, 0);  
        }  
        /* suspend the domain until the next timer interrupt occurs: */  
        suspend_domain (0,0);  
    }  
    return 0;  
}
```

6. Examples

By far, I think you have understood what XM/eRTL is, how install XM/eRTL, the main components in XM/eRTL and how to usage those components or functions. In the section, two case studies will be showed. From the cases, hoping readers or learners can understand XM/eRTL more clearly. In the first example, the new domain will be built. It will show how to build the new domain, how to use the hypercalls offered by XtratuM, and how to operate (including compile, load, unload, etc.) the domain. The second example will show how to write a real-time thread based on resource offered by PaRTiKle.

6.1. Domain

Domain is the object running on XtratuM. Same as process is scheduled by Linux, domain is schedule by XtratuM. In XtratuM, domain can be an operating system, device driver server, an independent normal task. If it is an operating system, it could have all functions besides hardware timer and interrupt management. Let's see a simple domain program first.

```
#include <xm_syscalls.h>

void timer_handler (int event, void *a)
{
    write_scr ("In_IRQ_0\n", 9);
    /* unmask_event(0); */
}

int kmain(void)
{
    struct xmitimerval irqtime = {{0, 200000}, {0, 0}};

    install_event_handler (0, timer_handler);
    set_timer(&irqtime, 0);
    unmask_event(0);
    enable_events_flag();

    while(1) {
        suspend_domain();
    }
}
```

6. Examples

```
    return 0;
}
```

This is a very simple domain. The domain includes timer interrupt response handler. When timer is triggered in the system, the event will entry timer_handler routine at last.

The domain comprises of three parts: head files, timer interrupt handler and domain main body. `ixm_syscalls.h` is the only header file for domain now. There are more than ten hypercalls and other routines in the header file. And they are what APIs you can use for you domain too.

The timer_handler() can be taken place by any other routine name, but the routine's parameters and return value you cannot changed. So the interrupt handler should be:

```
void your_irq_handler_name(int irq_number, void *point_to_handler_arguments);
```

In the above timer_handler() routine, one(Or two, another is commented) routine is called. write_scr() routine is a hypercall defined in `ixm_syscalls.h`. It will output the string into console. "In IRQ 0" is the output string and 9 is the size. The output result can be get by dmesg command. The unmask_event() routine is used to unmask the event bit and enable the related event again. If you use the routine, the interrupt will be responded all the time. But in my program, it responses once only.

kmain() is the domain entry routine. There is no parameter for kmain() and the return value type is integer. The purposes of the kmain routine are: 1) install a timer handler for the domain; 2) initialize the domain timer which will trigger the system timer in period; 3) suspend the domain and avoid the domain exit.

```
/* Define the period, 200000ns */
struct xmitinterval irqtime = {{0, 200000}, {0, 0}};

/* hypercall, install the timer handler for domain */
install_event_handler (0, timer_handler);
/* hypercall, set the domain timer period value*/
set_timer(&irqtime, 0);
/* unmask the timer interrupt */
unmask_event(0);
/* enable the events */
enable_events_flag();

/* avoiding the domain exit */
while(1) {
```

```

suspend_domain ();
}

```

When you understand the domain source code. You should know how to write the Makefile for compiling the domain.

```

xmertl:~$ cat Makefile
SOURCES = $(wildcard *.c)
NAMES = $(basename $(SOURCES))
TARGETS = $(addsuffix .xmd,$(NAMES))
#change to you user_tools path.
USER_TOOL_PATH=/home/xmertl/xtratum/user_tools
MKDOMAIN=${USER_TOOL_PATH}/scripts/mkdomain

all: $(TARGETS)

include ${USER_TOOL_PATH}/Rules.mk

${USER_TOOL_PATH}/Rules.mk:
    make -C ${USER_TOOL_PATH}/ Rules.mk
%.xmd: %.o
    $(MKDOMAIN) -o $@ $<
%o: %c
    $(CC) $(CFLAGS) -c -o $@ $<
clean:
    $(RM) -f *~ *.o *.xmd

```

One value should be changed based on your enviroment in Makefile: USER_TOOL_PATH. It should specify the user_tools directory offered by Xtra-tuM. Those tools are used to compile the souce code, link the library, load and unload the domain.

When the Makefile is written, you can compile the source code, load the domain, check the result and unload the domain.

```

# make
# /home/xmertl/xtratum/user_tools/loader.xm timer.xmd
>> Loading the domain "timer.xmd (timer.xmd)" ... Ok (Id: 2)
# cat /proc/xtratum
(2) timer.xmd:
Priority: 1

```

```
Intercepted events: 0x10
Masked events: 0xffffffffef
Pending events: 0x0
Events' flag is enabled
Current state: SUSPENDED
Domain's state word: 0x1
(0) Linux:
Priority: 1023 (MIN. PRIORITY)
Intercepted events: 0xffff
Masked events: 0xffff0000
Pending events: 0x0
Events' flag is enabled
Current state: ACTIVE
Domain's state word: 0x0
# dmesg
In IRQ 0
# /home/xmertl/xtratum/user_tools/unloader.xml -id 2
>> Unloading the domain "(2)" ... OK
```

By far, you have known how to write a simple domain for XtratuM. If you want to know much on it, you should study 1) hypercalls offered by XtratuM (see Appendix A. List of Hypercalls), 2) tools offered in user_tools (see section 2.4. Tools Usage).

6.2. Thread

Thread mechanism is supported by PaRTiKle system which is a special domain running on XtratuM. You can learn more on PaRTiKle from section 2.3.1 About PaRTiKle. PaRTiKle is POSIX compatible, and many APIs are offered by PaRTiKle. Let's see the bellow example.

```
xmertl:~$ cat thread.c
#include <stdio.h>
#include <time.h>
#include <pthread.h>

static pthread_t thread;

void *func(void *args)
{
    struct timespec sleeptime = {2, 0};
```

```

        nanosleep (&sleep_time, 0);

        return (void *)23;
    }

    int main (int argc, char **argv)
    {
        int ret_v;

        pthread_create (&thread, 0, func, 0);
        pthread_join (thread, (void *) &ret_v);
        printf ("Returned value: %d\n", ret_v);

        return 0;
    }

```

The main() routine is PaRTiKle domain entry routine which same as the Kmain() routine in the above section. When the system enter the main() routine, PaRTiKle domain start running but without thread now. The first thread is created when pthread_create() routine is called. From here, two tasks will be secheduled seperately: the main process from pthread_create to pthread_join and thread body func() routine. The thread will sleep two seconds and then exit. The main process will wait thread by pthread_join() routine. When thread exit, the main process will catch the return value from thread and output the value.

The thread is same as any POSIX thread. So you can call POSIX API to handle threads. Signal, condition, timer, mutex are supported by PaRTiKle for thread. So you can read POSIX manual to learn more how to write thread programmes for PaRTiKle. But you should remember the difference of running enviroment between XtratuM thread and Linux thread:

XtratuM	Linux
Domain {thread1, ...}	Process {thread1, ...}

A. List of Hypercalls

- `extern void exit_domain (int status);`
- `extern int load_domain (char *dname, int prio, domain_image_t *dimg);`
- `extern int unload_domain (int id);`
- `extern int enable_hwirq_sys (int irq);`
- `extern void suspend_domain (unsigned long dev, unsigned long cmd);`
- `extern void sync_events (void);`
- `extern void pass_event (int event);`
- `extern int write_scr (char *buffer, int length);`
- `extern int get_time (struct xmtimespec *t);`
- `extern int set_timer (struct xmitimerval *, struct xmitimerval *);`
- `extern unsigned long get_cpu_khz (void);`
- `extern int shm_unlink(const char *name);`
- `extern int shm_open(const char *name, int flag, unsigned int mode);`
- `extern int dev_ctl(unsigned long dev, unsigned long cmd, void *arg);`

B. Simple example of XtratuM device driver

This is a simple example of XtratuM device driver. This device driver is used for domains to make a alert by using the speaker in the board. It will make a alert in the frequency calculating by the domain id. And this example is just used to show the structure of XtratuM device driver, and it can not run in your machine because of some blank functions.

B.1. Head file

There are several structure and value will be used both by initialization module and device driver server domain. So we put them together in a head file named beep.h, and put it under the path XtratuM_PATH/include.

```
#define RT_BEEP_unUsed      0 /* */
#define RT_BEEP_dev_INIT   1 /* */
#define RT_BEEP_dom_INIT   2 /* */
#define RT_BEEP_dom_USED   3 /* */
#define RT_BEEP_dom_READ   4 /* device read permission */
#define RT_BEEP_dom_WRITE  5 /* device write permission */

#define BEEP_BASE_ADDR    0x61
#define BEEP_CONTROL_ADDR 0x43
#define BEEP_COUNT_ADDR 0x42
#define BEEP_SIZE         8

#define XM_BEEP_DEV_USED  1

typedef struct
beep_fifo_struct
{
    unsigned int flag; /* FIFO state flag */
    long top;          /* point to the first free address */
    long bom;          /* point to the first data address */
    long size;         /* fifo size */
    long baud;         /* baudrate for the device */
    long prio;         /* priority */
    char *addr;        /* fifo data memory address,
                        * global address, don't change it */
} beep_fifo_t;

typedef struct
dom_fifo_struct
{
    beep_fifo_t *beepf;
    char *addr; /* specify the fifo data memory, local address */
} dom_fifo_t;
```

```
#ifdef __XM_KERNEL__

typedef struct
xm_beep_struct
{
    unsigned int flag;
    domain_t *dom;
    dom_fifo_t in;
    dom_fifo_t out;
} xm_beep_t;

#else
typedef struct
rt_beep_struct
{
    unsigned int flag;
    dom_fifo_t in;
    dom_fifo_t out;
} rt_beep_t;

#endif
```

B.2. Initialization module

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

#include <linux/gfp.h>
#include <linux/vmalloc.h>
#include <linux/mm.h>
#include <linux/interrupt.h>
#include <linux/ioport.h>

#include <asm/string.h>
#include <arch/mm.h>

#include <log.h>
#include <domain.h>
#include <config.h>
```

```
#ifndef __XM_KERNEL__
#define __XM_KERNEL__
#endif

#include <xmdev.h>
#include <beep.h>

static xm_beep_t xm_beep;

static domain_t *bdom;

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Beep_init");

static int beep_irq = 5;

module_param(beep_irq, int, 0);

/* Functions implement for xm_beep_dev structure */
static unsigned long xm_map_page(unsigned long pd,
    unsigned long addr, unsigned long vaddr, unsigned long flag)
{
    unsigned long pd_entry = va2pd(vaddr);
    unsigned long pt_entry = va2pt(vaddr);
    unsigned long nflag;
    unsigned long pt;

    if( !((((unsigned long *)pd)[pd_entry]) & _PAGE_PRESENT) ) {
        pt = __get_free_page(GFP_ATOMIC);
        if( !pt ) return -ENOMEM;

        memset((unsigned char *)pt, 0, PAGE_SIZE);

        nflag = flag & (~_PAGE_UNUSED1);
        fill_pd_entry (pd, pd_entry, pt, nflag);
    } else {
        pt = get_pd_addr(pd, pd_entry);
    }
}
```

```
    nflag = flag;
    if ( !(((unsigned long *)pt)[pt_entry] & _PAGE_PRESENT)) {
        fill_pt_entry (pt, pt_entry, addr, nflag);
    }

    return 0;
}

static int xm_unmap_page(unsigned long pd,
                        unsigned long vaddr)
{
    unsigned long pd_entry = va2pd(vaddr);
    unsigned long pt_entry = va2pt(vaddr);
    unsigned long pt;

    if (((unsigned long *)pd)[pd_entry] & _PAGE_PRESENT) {
        pt = get_pd_addr(pd, pd_entry);
        if (((unsigned long *)pt)[pt_entry] & _PAGE_PRESENT) {
            ((unsigned long *)pt)[pt_entry] = 0UL;
        }
    }

    return 0;
}

int xm_beep_xm_map(unsigned long pd,
                  unsigned long vaddr, unsigned long size,
                  void *arg)
{
    unsigned long nflag = _PAGE_UNUSED1 |
        _PAGE_PRESENT | _PAGE_RW | _PAGE_USER;
    unsigned long nvaddr = DEFAULTBEEP_ADDR;
    unsigned long addr;
    unsigned long i = (unsigned long)arg;
    int ret;

    /* the domain is normal domain */
    if (size == 0) {
        if (i >= XM_BEEP_COUNT) {
            return -ENODEV;
        }
    }
}
```

```
    if( is_bit_set(xm_beep.in.beepf->flag , RT_BEEP_dom_INIT) ||
        is_bit_set(xm_beep.out.beepf->flag , RT_BEEP_dom_INIT) )
    {
        XMLOG("Beep_device_has_been_used\n");
        return -EBUSY;
    }
    addr = (unsigned long)xm_beep.in.beepf;
    ret = xm_map_page(pd, addr, nvaddr, nflag);

    if(ret) {
        XMLOG("[a]_xm_map_page_failed\n");
        return ret;
    }

    addr = (unsigned long)xm_beep.out.beepf;
    ret = xm_map_page(pd, addr, nvaddr+PAGE_SIZE, nflag);
    if(ret) {
        XMLOG("[b]_xm_map_page_failed\n");
        return ret;
    }

    set_bit(xm_beep.in.beepf->flag , RT_BEEP_dom_INIT);
    set_bit(xm_beep.out.beepf->flag , RT_BEEP_dom_INIT);

} else {
    /* beep device driver */

    addr = (unsigned long)xm_beep.in.beepf;
    ret = xm_map_page(pd, addr, nvaddr, nflag);
    if(ret) {
        XMLOG("[c]_xm_map_page_failed\n");
        return ret;
    }

    addr = (unsigned long)xm_beep.out.beepf;
    ret = xm_map_page(pd, addr, nvaddr+PAGE_SIZE, nflag);
    if(ret) {
        XMLOG("[d]_xm_map_page_failed\n");
        return ret;
    }
}
```

```
    }

    nvaddr = nvaddr + 2*PAGE_SIZE;

    set_bit(xm_beep.in.beepf->flag, RT_BEEP_dev_INIT);
    set_bit(xm_beep.out.beepf->flag, RT_BEEP_dev_INIT);

}
return 0;
}

int xm_beep_xm_unmap(unsigned long pd,
                    unsigned long vaddr, unsigned long size)
{
    unsigned long pd_entry;
    unsigned long nvaddr;
    unsigned long pt;
    int i;

    /* address should be aligned with page size */
    if (vaddr & (PAGE_SIZE - 1)) {
        return -EINVAL;
    }

    nvaddr = DEFAULT_BEEP_ADDR;

    /* non-beep driver domain */
    if (size == 0) {
        xm_unmap_page(pd, nvaddr);
        xm_unmap_page(pd, nvaddr+PAGE_SIZE);
    } else {
        /* beep driver domain */
        xm_unmap_page(pd, nvaddr);
        xm_unmap_page(pd, nvaddr+PAGE_SIZE);
        nvaddr = nvaddr + 2*PAGE_SIZE;

        clear_bit(xm_beep.in.beepf->flag, RT_BEEP_dev_INIT);
        clear_bit(xm_beep.out.beepf->flag, RT_BEEP_dev_INIT);
    }
}
```

```
    pd_entry = va2pd(vaddr);
    pt = get_pd_addr(pd, pd_entry);

    for ( i = 0; i < 1024; i++ ) {
        if(((unsigned long *)pt)[i]) {
            break;
        }
    }

    if (i == 1024) {
        ((unsigned long *)pd)[pd_entry] = 0UL;
        free_page(pt);
    }

    return 0;
}

static int xm_beep_xm_ioctl(unsigned long minor,
                           unsigned long cmd, void *arg)
{
    domain_t *dom;
    int ret = 0;

    if (minor == XM_BEEP_COUNT) {
        if (cmd & XMDEV_IO_INIT) {
            if (bdom == NULL) bdom = (domain_t*)arg;
            else if (bdom != (domain_t *)arg) {
                return -EBADF;
            }
        }
        return 0;
    }

    if(cmd & XMDEV_IO_CLOSE) {
        clear_bit(xm_beep.flag, XM_BEEP_DEV_USED);
        clear_bit(xm_beep.in.beepf->flag, RT_BEEP_dev_INIT);
        clear_bit(xm_beep.out.beepf->flag, RT_BEEP_dev_INIT);
        xm_beep.dom = NULL;
    }
}
```

```
/* special command to initialize the serial domain */
if (cmd & XMDEV_IO_OPEN) {
    dom = (domain_t *)arg;
    if ((xm_beep.flag & ( 1 << XM_BEEP_DEV_USED)) &&
        (xm_beep.dom != dom) ) {
        return -EBUSY;
    }

    set_bit(xm_beep.flag, XM_BEEP_DEV_USED);
    xm_beep.dom = dom;

    ret = xm_beep_xm_map(dom->pd,
                        DEFAULT_BEEP_ADDR, 0, (void *)minor);
    if( ! ret ) {
        if( dom->events ) {
            dom->events->beepaddr = DEFAULT_BEEP_ADDR;
        } else {
            XMBUG();
        }
    } else {
        clear_bit(xm_beep.flag, XM_BEEP_DEV_USED);
        xm_beep.dom = NULL;
    }
} else if (cmd & XMDEV_IO_WKUP) {
    /* wake up the serial device domain */
    if (bdom != NULL) set_domain_state(bdom, DOMAIN_ACTIVE);
}

return ret;
}

/* Functions using in the init and exit functions of the moduler */
static int xm_beep_mem_alloc(void)
{
    printk("in_mem_alloc\n");
    unsigned long page1;
    unsigned long page2;
    int i;
```

```
page1 = __get_free_page(GFP_ATOMIC);
page2 = __get_free_page(GFP_ATOMIC);

if (!page1) {
    goto beep_mem_error;
} else if (!page2) {
    free_page(page1);
    goto beep_mem_error;
}

memset ((unsigned char *)page1, 0, PAGE_SIZE);
memset ((unsigned char *)page2, 0, PAGE_SIZE);

xm_beep.in.beepf = (beep_fifo_t *) page1;
xm_beep.in.beepf->addr = (char *) (page1 + sizeof(beep_fifo_t));
xm_beep.in.beepf->size = PAGE_SIZE - sizeof(beep_fifo_t);
xm_beep.in.beepf->flag = 0;

xm_beep.out.beepf = (beep_fifo_t *) page2;
xm_beep.out.beepf->addr = (char *) (page2 + sizeof(beep_fifo_t));
xm_beep.out.beepf->size = PAGE_SIZE - sizeof(beep_fifo_t);
xm_beep.out.beepf->flag = 0;

return 0;

beep_mem_error:
    if(xm_beep.in.beepf)
        free_page((unsigned long)xm_beep.in.beepf);
    if(xm_beep.out.beepf)
        free_page((unsigned long)xm_beep.out.beepf);

return -ENOMEM;
}

static void xm_beep_mem_free(void)
{
    if(xm_beep.in.beepf) free_page((unsigned long)xm_beep.in.beepf);
    if(xm_beep.out.beepf) free_page((unsigned long)xm_beep.out.beepf);
}
```

```
}

typedef irqreturn_t (*irq_handler_t)(int, void *, struct pt_regs *);

static irqreturn_t beep_irq_handler(int irq, void *arg, struct pt_regs *regs)
{
    return 0;
}

xm_dev_t xm_beep_dev = {
    XM_BEEP_DEV,
    xm_beep_xm_map,
    xm_beep_xm_unmap,
    xm_beep_xm_ioctl,
    0
};
struct resource *beep_res[3];

int xm_beep_xm_init(void)
{
    int ret, irq, tmp;

    release_region(BEEP_BASE_ADDR, BEEP_SIZE);
    release_region(BEEP_CONTROL_ADDR, BEEP_SIZE);
    release_region(BEEP_COUNT_ADDR, BEEP_SIZE);

    beep_res[0] = request_region(BEEP_BASE_ADDR, BEEP_SIZE, "beep");
    beep_res[1] = request_region(BEEP_CONTROL_ADDR, BEEP_SIZE, "beep");
    beep_res[2] = request_region(BEEP_COUNT_ADDR, BEEP_SIZE, "beep");
    ret = check_region(BEEP_BASE_ADDR, BEEP_SIZE);
    printk("ret_from_check_is_%d\n", ret);
    printk("resource_is_OK\n");

    for (tmp = 0; tmp < 3; tmp++){
        if (beep_res[tmp] == NULL) {
            printk("can_not_request_the_port_%d\n", tmp);
            return -1;
        }
    }
}
```

```
xm_beep.flag = 0;
printk("the_init_of_xm_beep_is_OK\n");
xm_beep.dom = NULL;
printk("the_init_of_xm_beep_is_OK\n");

ret = xm_beep_mem_alloc();
printk("memory_is_OK\n");

if ( ret ) return ret;

ret = xm_dev_register(XM_BEEP_DEV, &xm_beep_dev);

if ( ret ) return ret;

irq = request_irq( beep_irq , beep_irq_handler ,
                  SA_INTERRUPT, "beep", NULL);

if(irq < 0) {
    return -1;
}

XMLOG("XM_BEEP_is_ready!\n");

return 0;
}

void xm_beep_xm_exit(void)
{
    int i;

    free_irq( beep_irq , NULL);

    if(xm_beep_dev.client > 0) {
        XMBUG();
        return;
    }

    i = xm_dev_unregister(XM_BEEP_DEV);

    xm_beep_mem_free();
}
```

```
    xm_beep.flag = 0;
    xm_beep.dom = NULL;

    release_region(BEEP_BASE_ADDR, BEEP_SIZE);
    release_region(BEEP_CONTROL_ADDR, BEEP_SIZE);
    release_region(BEEP_COUNT_ADDR, BEEP_SIZE);

    XMLOG("XM_BEEP_Exit!\n");
}

module_init(xm_beep_xm_init);
module_exit(xm_beep_xm_exit);
```

B.3. Device driver server domain

```
#include <doms/stdlib.h>
#include <doms/string.h>
#include <doms/errno.h>
#include <devbug.h>
#include <io.h>
#include <xm_syscalls.h>
#include <xm/beep.h>

#define PAGE_SIZE 4096

typedef unsigned char uint8_t;
typedef struct
beep_dev_struct
{
    int irq;
    int base;
    int ctl;
    int count;
} beep_dev_t;

typedef struct
beep_buff_struct
{
    int top;
    int size;
```

```
        uint8_t data[2];
    } beep_buff_t;

#define CAS(adr, ov, nv) ({ \
    __typeof__(ov) ret; \
    __asm__ __volatile__( \
        "cmpxchg_%3,%1" \
        : "=a"(ret), "+m" (*(volatile unsigned long *) (adr)) \
        : "a"(ov), "r"(nv)); \
    ret; \
    })

static rt_beep_t rt_beep;
static beep_buff_t uin = {0, 2};

static beep_dev_t beep = {5, 0x61, 0x43, 0x42};

int dom_beep_fifo_read(dom_fifo_t *, uint8_t *, int);

static uint8_t beep_domain_schedule(void)
{
    return 0;
}

static void beep_irq_handler(int irq, void *arg)
{
    uint8_t data;
    struct xmitinterval req = {{0, 200000}, {0, 0}};

    do {
        beep_domain_schedule();
        dom_beep_fifo_read(&rt_beep.in, &uin.data[0], 8);
        dom_beep_fifo_read(&rt_beep.in, &uin.data[1], 8);

        outb(0xb6, 0x43);

        outb(uin.data[0], 0x42);
        outb(uin.data[1], 0x42);
    } while (1);
}
```

```
        data = inb(0x61);

        outb(data|0x3,0x61);

        outb(0xfc,0x61);
    } while (1);

    unmask_event(beep.irq);
}

static int print(char *string)
{
    int i = 0;
    while(string[i++]);
    write_scr(string, i);

    return i;
}

static int beep_dev_init(void)
{
    int ret;

    unsigned long beepaddr;
    int i;

    beepaddr = event_handling->beepaddr;

    print("beepaddr_is_of\n");
    if (beepaddr == 0UL) return -1;

    rt_beep.in.beepf = (beep_fifo_t *) beepaddr;
    rt_beep.in.addr =
        (char *) (beepaddr + sizeof(beep_fifo_t));
    beepaddr += PAGE_SIZE;

    rt_beep.in.beepf->top = 0;
    rt_beep.in.beepf->bom = 0;
    rt_beep.in.beepf->size = PAGE_SIZE - sizeof(beep_fifo_t);
```

```
//      rt_beep.out = NULL;

/* atomic set state flag */
set_bit(rt_beep.in.beepf->flag, RT_BEEP_dom.READ);
set_bit(rt_beep.in.beepf->flag, RT_BEEP_dom.WRITE);

return 0;
}

int kmain(void)
{
    struct xmitinterval req = {{10000, 200000}, {0, 0}};

    int ret;

    print("beep_dev_init_begin\n");
    ret = beep_dev_init();
    print("beep_dev_init_OK\n");

    if(ret) {
        print("beep_device_initialization_failed!\n");
        return -1;
    }

    set_timer (&req, 0);

    install_event_handler (beep.irq, beep_irq_handler);

    unmask_event (beep.irq);
    enable_events_flag();

    print("<XtratuM>_BEEP_device_loaded!\n");

    for (;;) {
        print("beep_server_is_looping\n");
        suspend_domain(0, 0);
    }

    return 0;
}
```

```
int dom_beep_fifo_read(dom_fifo_t *df, uint8_t *dest, int size)
{
    beep_fifo_t *ff;
    long bom, offset, nbom;
    int rsize;

    if(df == NULL || dest == NULL || size < 0) {
        return -EINVAL;
    }

    ff = df->beepf;

    if(!is_bit_set(ff->flag, RT_BEEP_dom_READ))
        return -EPERM;

    do {
        rsize = (ff->top + ff->size - ff->bom) % ff->size;

        if(rsize > size) rsize = size;

        bom = ff->bom;
        nbom = (bom + rsize) % ff->size;
        if (ff->top < bom && rsize > ff->size - bom) {
            offset = ff->size - bom;
            memcpy(dest, df->addr + bom, offset);
            memcpy(dest + offset, df->addr, rsize - offset);
        } else {
            memcpy((void *)dest, (void *)(df->addr + bom), rsize)
        }
    } while (CAS(&ff->bom, bom, nbom) != bom);

    return rsize;
}
```

B.4. Code modified in other files

The device address configured in include/config.h

```
#define XM_BEEP_COUNT 1
#define DEFAULT_BEEP_ADDR 0x3800000
```

```
#define XM_BEEP_DEV_SUPPORT      1

#if (XM_FIFO_DEV_SUPPORT | XM_SHM_DEV_SUPPORT | \
     XM_SERIAL_DEV_SUPPORT | XM_BEEP_DEV_SUPPORT)
#define XM_DEV_SUPPORT
#endif
```

The device address which is used to pass to domains in the `event_handling_t` structure in `include/events.h`

```
/* Beep memory */
unsigned long beepaddr;
} event_handling_t ;
```

Some parameter will set in `include/xmdev.h`

```
#define XM_BEEP_DEV      3
#define XM_BEEP_DEV_NAME "beep.xmd"
#define is_beep_dev(d) (strcmp(XM_BEEP_DEV_NAME, d->name) == 0)
```

Last but not least, add the resource initialization in `kernel/syscalls.c` which is used to initialize the resource when domain loaded in `load_domain_sys`, and the destroy function when domain unloaded in function `unload_domain_sys()`

```
#if XM_BEEP_DEV_SUPPORT
    fvaddr = DEFAULT_BEEP_ADDR;

    if ((dev = xm_get_dev(XM_BEEP_DEV)) == NULL) {
        destroy_domain(d);
        id = -ENODEV;
        goto exit_load_domain;
    }

    if (is_beep_dev(d)) {
        dev->dev_map_handler(d->pd, fvaddr,
                             XM_BEEP_COUNT*2*PAGE_SIZE, NULL);
        d->events->beepaddr = fvaddr;
    }

    dev->client++;
#endif
```

```
#if XM_BEEP_DEV_SUPPORT
    if ((dev = xm_get_dev(XM_BEEP_DEV)) != NULL) {
        if ( is_beep_dev( d ) ) {
            dev->dev_unmap_handler(d->pd,
                                   DEFAULT_BEEP_ADDR,
                                   XM_BEEP_COUNT*2*PAGE_SIZE);
        } else {
            dev->dev_unmap_handler(d->pd, DEFAULT_BEEP_ADDR, 0);
        }
        dev->client --;
    }
#endif
```

3. List of Acronyms

API - Application Programming Interface
CAS - Compare and Swap
FIFO - First In First Out
GCC - GNU Compiler Collection
GNU - GNU's Not Unix (recursive acronym)
GPL - General Public License
POSIX - Portable Operating System Interface
PTE - Page Table Entry
IDC - Inter-Domain Communication
RTOS - Real-Time Operating System
SHM - Shared Memory
TLSF - TwoLevelSegregatedFit (Memory Storage allocator implementation)
XM - XtratuM

References

- [1] Introduction to XtratuM, M. Masmano, I. Ripoll, A. Crespo, 2005
- [2] Introducing RTLinux/GPL, Der Herr Hofrat, 2002,
- [3] XM-FIFO: Interdomain Communication for XtratuM, Shuwei Bai, Yiqiao Pu, Kairui She, Qingguo Zhou, Nicholas MC Guire, Lian Li
- [4] RTLinux Version TWO Design documentation about RTLinux in FSMLabs, Victor Yodaiken and Michael Barabanov, 1997, <http://www.fsmlabs.com>
- [5] Device Driver Model for XtratuM, Shuwei Bai, Xingwen Huang, Quingou Zhou, and Nicholas McGuire. 10th Real-Time Linux Workshop.
- [6] XMTrace, Thomas Hisch, Georg Schiesser, and Andreas Platschek. 10th Real-Time Linux Workshop.
- [7] Shared Memory in Xtratum/PaRTiKle, Pu Yiqiao, Bai Shuwei, She Kairui, Zhou Qingguo, and Nicholas McGuire. 10th Real-Time Linux Workshop.
- [8] Linux device driver, third edition, Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. 2005