XMeRTL-4.0-rc1 Manual

Distributed & Embedded System Lab

January 15, 2013

Contents

1. Introduction

XM/eRTL is an embedded real-time system which comprises of XtratuM and PaRTiKle. XtratuM is a hypervisor and PaRTiKle is a new open source real-time kernel for embedded systems, especially the PaRTiKle is POSIX complied and support RTLinux/GPL applications, so the PaRTiKle is named eRTL in the XM/eRTL. XM/eRTL has the characteristics from XtratuM and PaRTiKle.
Before going on introduce XM/eRTL system, let's clear about the main concepts in XM/eRTL system again. XtratuM is a hypervisor which can run based on Linux, and PaRTiKle is another kind of POSIX threads embedded real-time system. XM/eRTL system is the combination of XtratuM and PaRTiKle or PaRTiKle is ported to XtratuM system and run as a XtratuM domain. Domain is the smallest task running on XtratuM. Linux is a domain too. But XtrautM should be based on Linux, so Linux is called root domain. PaRTiKle is real-time and in XM/eRTL system, PaRTiKle is a real-time domain. In the whole system, XtratuM runs in the lowest level which can direct hardware level, Linux and PaRTiKle are in the same level and run on XtratuM. So the programmer can write the application for XtratuM, PaRTiKle and LINUX separately. The applications running on XtratuM are domains which same as Linux and PaRTiKle. The application programmed for PaRTiKle is real-time thread. And applications running on Linux are normal process. In the released version, XM/eRTL system is consisted of XtratuM and PaRTiKle only, not including Linux.
XM/eRTL system is a nano-kernel system. Only few functions are supported by XM/eRTL kernel, interrupt management, timer management, and domain scheduler. The most system functions are done by peripheral components and domains. Linux is the root domain offering memory allocation and system booting service. Two IDC tools can be used to transfer data between domains, XM/FIFO and XM/SHM. One device driver model is built for XM/eRTL for porting and building new device driver. In the XM/eRTL, the XM/Serial device driver is the only real-time device driver implemented now. And programmers can use trace tool to trace the events triggered in the XM/eRTL kernel: XtratuM.
Different from RTLinux/GPL which is dual-core system, XM/eRTL is multi-core system. The purpose of XM/eRTL is to build more stable and reliable real time system and support the application running RTLinux/GPL. RTLinux/GPL which was one of most famous real-time Linux system in the open source area is a hard realtime RTOS runs the entire Linux operating system as a fully preemptable process. Wind River Systems acquired the RTLinux copyright in 2007 and now makes a version available as Wind River Real-Time Core for Wind River Linux. In order to support the RTLinux/GPL applications, XM/eRTL is built by the RTLinux/GPL community to replace RTLinux/GPL system. One of most important characteristics of XM/eRTL is RTLinux/GPL compatible.

The basic idea behind of XM/eRTL is hypervisor architecture. In the architecture, both real-time and non-real-time application can be supported. The XM/eRTL is the hard real-time and the system scheduler support runtime preemption. 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 real hypervisor. XtratuM focus on the interrupt management, timer management and domains scheduler which are critical components for real-time system. In XM/eRTL, Linux with XtratuM patch boots the machine first, initializing the whole system as normal, such as devices initialization, memory initialization, interrupt and timer initialization (not for XM/eRTL), etc. So Linux installation and booting are basic operation or preparation for building XM/eRTL. When the Linux boots successfully, XtratuM can be loaded. XtratuM start taking over the machine management permissions from Linux. The process starts from interrupt takeover, then entry timer takeover, and at last jumping in XtratuM scheduling era. When the process over, the old environment is changed too much. Linux is not the whole system running on the machine, and it is not the administrator of the system any more. Linux becomes the root domain whose interrupt and timer management permission have been take over by XtratuM. But Linux is the main roles offering device service and memory service. Linux can be scheduler by XtratuM. When there is no any domain whose priority is higher than Linux, the CPU will be assigned to Linux.
When XtratuM takeover Linux interrupt and timer successfully, the system can be used to build Linux applications and XtratuM domains. Linux applications programming is simple as usually. XtratuM domains programming introduction can be found in the 6.1 section which shows how to build and run a new domain. And the Appendix II shows XtratuM hypercalls which can be used by domains.
PaRTiKle is special embedded real-time system. It can run as a standalone operation system or as a Linux process. In XM/eRTL, PaRTiKle runs as a domain with higher priority. PaRTiKle is a POSIX THREAD compatible operating system. The application following POSIX THREAD can be ported to PaRTiKle with little or without source code modification. In order to support RTLinux/GPL applications, PaRTiKle is ported to XtratuM. And PaRTiKle and XtratuM are the two most important components in XM/eRTL system. Section 6.2 will show how to build real-time thread based on PaRTiKle.
XtratuM and PaRTiKle are core components in XM/eRTL. Excepting for the basic services, some important functions are offered by them too. Those functions include IDC (Inter-Domain Communication) function, device driver model and trace function. FIFO and SHM are both IDC tools offered by XtratuM. The IDC is used to transfer data between kinds of domains. Conventional, FIFO is used for stream data and SHM is used to transfer block data. Device driver model is used to build or add new device driver for XM/eRTL. The trace tool is used to trace the triggered events. In other sections, those functions will be explained in detail.

The whole manual is divided into six sections. This is the section one which showed what is XM/eRTL roughly. Section 2 is installation. In the section, XM/eRTL environment building will be explained in detail. From introduction section, it is clear that the installation process should include Linux Installation, XtratuM installation and PaRTiKle installation. In order to boot XtratuM and manage the domains, some useful tools offered by XM/eRTL. The section will introduce XM/eRTL tools and explain tools usage.
Section 3 is IDC tools. IDC tools are used to transfer data between domains. There are two IDC tools in XM/eRTL now, XM/FIFO and XM/SHM. XM/FIFO adopts memory mapping and lock free mechanism. The tool is fit for stream data transmission. Different from XM/FIFO, XM/SHM is fit for block data transmission.
Section 4 is Device Driver. The section includes device deriver model in XtratuM and serial device driver which is the only hardware device driver offered by XtratuM. Following XM/DEV (XtratuM DEVice Driver Model), the hardware device driver comprised of static component and dynamic component. The static component is located in Linux kernel. They are used to initiate the device, apply the source, and release source. The dynamic component is main body which will access and manage the device in run-time, and normally the dynamic component is a scheduled domain. The serial device driver is built with the XM/DEV model.
Section 5 is Trace tool. XM trace uses statically allocated shared memory to buer timestamp, address and type of events. Both hard-coded system events and user-dened events are supported. xmtrace provides adequate trace points in the XtratuM code base for critical functions like scheduling, enabling/disabling interrupts, and acquiring/ freeing spin locks. 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 is exercise. Beginners can learn how to write domain and thread applications in the XM/eRTL system. Domain is the scheduled unit on XtratuM and thread is for PaRTiKle system. Domains is nearer from XtratuM than the thread which above the PaRTiKle. So domains have lower kinds of latency, such as interrupt latency and timer latency. It will be complex to build a domain by hand from scratch and PaRTiKle is an appreciate choice. PaRTiKle is a specially domain which offers POSIX thread API to programmer. It is easier to write the thread for PaRTiKle. In order to resolve one problem, domain and thread may be resolution both. Having a think should be done before selection.

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 verions are available, but still have to be concidered development verions. Therefore this document will focus on linux-2.6.17.4 [1].
The operation system used in this manula 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 dependcies 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
```

[1]The Linux kernel can be download from: http://www.kernel.org.

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 verfiy 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, 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
# 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 sucessfully 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 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, there are XtratuM core in this kernel, while we need to add some modules and tools of XtratuM. The modules are load to make some XtratuM device can work, and these tools are used to operate or control the XtratuM system.
First we should compile to generate these modules and tools. So enter the subdirectory of xtratum. The file "Makefile" in the subdirectory of xtratum is used to realize this function. But you should make a modification to it. The place need to modify is the value of LINUX_PATH. The Linux kernel file is locate in the subdirectory of XtatuM and named Linux as default, just like xtratum/linux/. But it is like this at most time. So modify or even check it to make sure the path of LINUX_PATH is correct. The operation can be like this: LINUX_PATH=/path/to/linux-2.6.17.4

```
# vim Makefile
```

Then, we can compile them. Because the file "Makefile" support the rules about how to compile, so we just use the command "make". If you want to know how the work going on, just read the source of the file.

```
# 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
```

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 you 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 list. 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
```

Progress smoothly. Of course, only 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 load, 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 located at xtratum/user_tools/examples/trivial_domain. Then compile it.

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

After compile, a new domain named trivial.xmd is generated. We use the tools loader.xm to load it. You can learn more information from the Tools Usage. To convenient, we set the PATH contains the path of loader.xm. Then we can use it as loader.xm in place of /path/to/loader.xm.

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

If it returns a message like: "... Ok (Id: X)", it tell you that the domain load successfully. The return value of Id is assigned for this domain. Every domain loaded successfully can get a id. The first loaded domain gets the id 1, and the second get 2, and so on. The id is used to unload the domain by unloader.xm. Refer to Tools Usage. Remeber that id of the domain load just now is 1.
The domain loaded successfully, then we can display message about it by "dmseg". The command "dmseg" is very useful for display the message about the system and domains. The message of domain displayed is just the message output to stand out. More information about dsmeg can refer to [**?**].

```
# dmesg
```

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.xm -id 1
>> Unloading the domain "(1)" ... OK
```

There are some problems may be you will come cross.

```
# 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 cat 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: 0xffffffef
     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 successully, 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
>> 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 locate in xtratum/user_tools/xmloader/loader.xm:

```
#[XtratuM dir]/user_tools/xmloader/loader.xm 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.xm -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.xm -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 executabel 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.xm 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.xm dom1.
```

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

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

It is easy. But there is a limitation using this way. The PATH includes the loaderxm just effictive 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.xm (and unloader.xm) to the directory /usr/bin/ . It is once for all. Every loaded domain has its priority. The defualt 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.xm /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 lacal in the directory xtratum/user_tools/xmloader named unloader.xm. 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.xm -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

3.1. XM/FIFO

3.1.1. About XM/FIFO

Fifo stands for "first in first out". It is a tool for communication. The fifo have name and a fixed storage. We can think of it as such: there is pipe to transmission of data stream. We write data to one port, and read data from the other port. The data which have been read can not been read again. If the write port reached to the read port, no data can write to the fifo again. At this time, we call the fifo is full. If the read port reached to the write port, no data can read from the fifo. At this time, we call the fifo is empty. If you want to know more about fifo, you can refer to the wiki:http://en.wikipedia.org/wiki/FIFO_(computing).
We employ fifo in XtratuM, then we get the XtratuM/FIFO. Why the fifo is used in XtratuM or what is used for? If you have read the introduction about XtratuM, you should know the design of XtratuM. In order to raise the safety of the domains, XtratuM make the domains independent. Each domain has memory pages of itself. One domain can not access the the data of another domain directly, and these operations are forbid. However, some domains need to cooperate to finish some tasks. In order to improve the ability of coopertation, some comminucation tools should be used. The fifo can achieve these requirement and its realization is simple.
Take a simple example: domain d1 generates some data, and domain d2 deals with these data. The operations of d2 ought to be after the data generated in d1. How can d2 get the data from d1. It can realize simply with the help of fifo. When the data was generated, d1 write it to fifo; while d2 read these data from the fifo.

3.1.2. Characteristics

There are 16 fifos in XtratuM, named rtf0, rtf1, ... , rtf15. Fifo can be accessed by its name. These 16 fifos locale at /dev/, so they can be used by accessing /dev/rtf0, /dev/rtf1 and so on. In XtratuM, every domain can access these fifos: open, write, read, close. And these operations are similar to the operations of ordinary file in Linux.
The storage size of each fifo is PAGE_SIZE. The storages are assigned at the time of modules loaded, and they are in the memory all the time, unless some operations to unload these fifo modules. In order to manage or control these fifos, a struct fifo_struct was used. A struct occupies 20-bits, including the fifo data memory address, the fifo can be used or not, the read point and write point. XtratuM use an array to organize the 16 structs. The opertations of array are simple. And it is not need to assign the memory to the struct when the fifo or domain is running.

There are three parts in XtratuM/FIFO: FIFO core, Linux Fifo device drive, Partikle Fifo device drive.
The fifo core is belong to XtratuM, but separated from the XtratuM core. This part supports the fifo managment module. The fifo_struct is used here. At the time of initialization, it registers sixteen fifo devices whose sizes are all PAGE_SIZE. It also set the fifo_struct of each fifo. At the time of end, the core of XM is in charge of exit of fifos. It will free the pages applied by fifos, and unregister these fifo devices.
When the domain wants to use these fifos, it should access the page of the fifo. the fifo core supports a function to do the memory map. The liner address of the domain can map to the page of the fifo, so it access the fifo by the linear address to read and write. In XtratuM, the pages mapped by the domain will free when the domain unload. This is not suit to the fifo pages. So a SHARE flag is used to mark the fifo page. When the domain is unloaded, the page can free except the fifo page. Just make the index in pte is 0.
The core of XM support some function interfaces, the domain can use these fifos by these functions. While we konw that the core of XM support the fifo management, it registers some fifo devices. So the domain can use these devices directly by function of itself. Ok, now we can find that the different domains can use different methods to fifo. The Linux and Partikl are the examples.
Operatation of access to the fifo is not blocked. When try to read from the fifo while the fifo is null, return directly. In the same way, when try to write the fifo while it is full, return directly. Each domain can read and write the fifo. The mechine of Lock-Free is used by the fifo to realize this goal. It use CAS(Compare-and-Swap) operation which make the read and write as atomic operation. You can read or write data successfully, otherwise, the data will not be changed. In other words, the operation can stop at any time. So it can avoid the operation block the task with high priority and it can enure the consistency of th e data.
XM use two variant of fifo_struct to realize read and write: ff_top and ff_buttom. The variant ff_top points to the head of the fifo, used for write; and ff_buttom point to the tail of the fifo, used for read. The CAS operation checks the values of ff_top and ff_buttom to check whether the read and write are interrupted or not.
The CAS operation can avoid the competition of RR, RW,Wr. But unfortunately, the CAS operation can not avoid the competition of WW. So it is suggested that to reduce the number of task write to a fifo.
Linux fifo divece drive supports a way for the Linux domain to access the fifo. Linux is the root domain, and just use the function interfaces of the XM. The function of read is excutes in this flow: read Linux lfifo_read Linux rt_fifo_user_read XM and the write is same.
However, the Partikle domain can be loaded later, and can use the function of Partikle. The flow of function read executes like this:
Although the implements are different, Linux and Partikle support some similar func-

tions we can use, such as read, write, open, ioctl and so on.

3.1.3. Using XM/FIFO

In Partikle, when we create a domain using fifo, we should know the interrelated functions. As XM/FIFOs are devices too, so normal file operation routines can be used for FIFO devices, such open, close, read, write, etc..
There are three examples about fifo in the directory: partikle/user/example/c_example. Let's see their codes and results to know how the fifo works.

The first one is fifo.c . It uses varient fifo_dev to point out that its will the fifo device- /dev/rtf0. In the content of the function, the fifo is open by the open() function:

```
if((fd = open(fifo_dev, O_RDWR, O_EXCL)) < 0)
```

Now, the fd is the fifo descriptor of the fifo. We can write data to the fifo or read date from the fifo by it. In the example, it write a message to the fifo. The message is defined by varient msg.

```
char msg[] = "I am in the partikle\n";
... ...
for(i = 0; i < 10; i ++) {
    ret = write(fd, msg, sizeof(msg));
    printf("write fd %d ret = %d\n", fd, ret);
... ...
```

Compile this domain, load it and display its message using that XtratuM tools.

```
$loader.xm fifo.prtk
$dmesg
write fd 3 ret = 22
... ...
write fd 3 ret = 22
```

This is a simple example. The fifo is used for communicate inter-domain. Such example about communication between domain is supported, fwrite.c and fread.c. In this example, the fifo device /dev/rtf1 is used. So the two domain both open this fifo using function open. Fwrite domain write 50*2 digit string "1234567890" to rtf1, in the other port, fread domain read these 49 digits everytime. The digits written are uninterrupted, then the data read is interesting. The first time is 49 digits, just read to 9. The second time to read from 0, and end with 8. The code is this:

```
char message[] = {"123456789012345678901234567890"};
... ...
for(i = 0; i < 50; i ++) {
ret = write(fd, message, 20);

char message[50];
... ...
message[49] = '\0';
for(i = 0; i < 5; i ++) {
ret = read(fd, message, 49);
... ...
printf("%d %d %s\n", ret, i, message);
```

Compile the two domain, and load them. Load the fwrite domain first to write data to fifo in order to ensure there is data in the fifo. Then display the message, you can get this:

```
$dmesg:
... ...
49 0 1234567890123456789012345678901234567890123456789
49 1 0123456789012345678901234567890123456789012345678
49 2 9012345678901234567890123456789012345678901234567
49 3 8901234567890123456789012345678901234567890123456
49 4 7890123456789012345678901234567890123456789012345
```

After the introduce the functions and some examples, you can try to write a domain yourself. The domain can put into the directory of user/example/c_example/, and execute "make". Then you can get the domain end with .prtk. Of cource, you can use the new directory to make the test. For example, I mkdir a new name test_fifo.

```
$mkdir -p /path/to/test_fifo
$cd /path/to/test_fifo
$cp /path/to/rtlinux-4.0-rc1/partikle/config.mk .
$cp /path/to/rtlinux-4.0-rc1/partikle/mkfiles/mkfile-c .
$vim Makefile

$cat Makefile
SOURCES = $(wildcard *.c)
NAMES = $(basename $(SOURCES))
TARGETS = $(addsuffix .prtk,$(NAMES))
```

```
all: $(TARGETS)
include config.mk
include mkfiles/mkfile-c
clean:
        $(Q3)rm -f *.{prtk,dump,sym,size,hex,bin} *.ktr
```

Write your program, compile and load it:

```
$vim fifo_test.c        //write the program of your self
$make
$loader.xm fifo_test.prtk
```

Note: You should remember that we use "open()" function to open the fifo decives. We should make the name is correct: /dev/rtf0 , /dev/rtf2 , ..., /dev/rtf15. If the name is just rtf0, rtf1, or rtf15, it will be wrong. As we said above, there are just SIXTEEN fifo device supported by XtratuM. So the /dev/rtf16, /dev/rtf17 and so on do not exist. You don't need try to use them. In patikle of XtratuM, the "open" function just is useful for fifo devices. If you tried to use it open a ordinary, it is not success. The fifo is used for communication, so when you write a program, you had better plan which domain to write and which domain to read. The max data can write once is a PAGE_SIZE, which is setted in the Linux Kernel.

3.2. XM/SHM

Shared memory is a method of interprocess communication (IPC) 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, allowing for things like malloc debuggers to be written. It is one of efficeint ways of IPC. This partion include XM/SHM, TLSF, APIs and Using examples.

3.2.1. About XM/SHM

Shared Memory in Xtratum is like a segment of memory, which can be mapped into address space of different domains of XtratuM including Linux and Partikle. So users can access any part of the segment in both real-time domain and non-real-time domain. Shared memory management module is the important part in the share memory system, as all the high level virtual memory must depend on it. It manages the physical memory, allocate or release the physical memory for operating systems run above XtratuM and mapped it into their address space. In the XtratuM system, the physical memory

is managed by the Linux kernel, so we can use the Linux kernel APIs or some other functions excluding XtratuM APIs to allocate physical memory.
TLSF stands for Two Levels Segregate Fit memory allocator. It is designed for real-time system as a general purpose dynamic memory allocator. As the requirements of real-time system, the longest time to allocate memory and the conditions which will cause deadlock should be known and unrelated with the applications. This is the most important requirement for dynamic memory allocate in real-time system. Bounded response time is only a basic requirement, but not enough. According the application area of real-time, the response of time is required as short as possible. Besides the time requirement, real-time systems also need a guarantee of execution resource including memory. So TLSF should minimize the chances of exhausting the memory pool by minimizing fragmentation and wasted memory.
XM/SHM use two function malloc_ex() and free_ex() which are defined in xtratum/kernel/tlsf.c to alloc and free physical memory.
There are some features of SHM follow:

- Shared memory is typically used to hold a data structure whose members are read and written by two different processes.
- Unlike FIFOs, shared memory is not queued. Any data written to 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.
- Because it is a shared resource, care must be taken to ensure data consistency. In RT Linux, 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 usage is a single RT Linux task writes a data array to shared memory periodically, and a single Linux process reads this data periodically and prints the results. Various types of data consistency techniques are demonstrated, so usually SHM accompany with other IPC tools like semaphore for inter-processes commumication.

3.2.2. XtratuM/SHM APIs

You can access device of /dev/shm[0-15] after load XM/SHM modules. Every domain of XtratuM or Partikle can write and read the shm memory if it have access permission. So processes of Linux can communicate with processes of Partikle or other domain's.
The most efficient way to implement shared memory applications is to rely 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. Normally, the processes of XtratuM domain use functions mbuff_alloc() and mbuff_free() which are defined in ashm.h. mbuff_alloc() call mmap() to allocate memeory and mbuff_free() call munmap() to free memory.

- char *mbuff_alloc(const char *name, ssize_t size);
  DESCRIPTION: alloc physical memory in shm device, name is a const string means the shm device name; size means the size of memory wait to alloc.
  RETURN VALUE: On success, mbuff_alloc() return the pointer of alloced memory address. On error the NULL is returned.

- int mbuff_free(char *start, int size
  DESCRIPTION: free physical memory, start is pointer of string. size means the size of memory wait to free.
  RETURN VALUE: On success, mbuff_free() returns 0. On error -1 (and sets errno) is returned.

3.2.3. Usage Example

In XMeRTL system, XM/SHM can be used to transfer data between normal Linux tasks, between PaRTiKle tasks, or between PaRTiKle task and Linux task. In the subclause, an example which use XM/SHM to transfer data between Linux and PaRTiKle will be showed.
The example comprised of two programs, prtk_writer and linux_reader. prtk_writer is a PaRTiKle task which will write a string to a XM/SHM deivce. And linux_reader will read the string from Linux space. Let's check the prtk_writer.c file first.

```
# cat prtk_writer.c
```

In the program, the prtk_writer will use /dev/shm/0 to transfer data to linux_reader. The device /dev/shm/0 will be mapped to /dev/shm0 in Linux system. So you will find that linux_reader will open /dev/shm0 device.
explain the source code ......
In order to compile the prtk_writer successful, you should edit the Makefile like this.

```
# cat Makefile
```

The prtk_writer is ok now. Before compile the prtk_writer.c file, linux_reader.c should be done. The program can read the string from the shm device and output it to console.

```
# cat linux_reader.c
```

The linux_reader.c is very simple. From the source code, it is same as normal linux programm. And it's clear that the shm device name is different. When the shm device is used, the device suffix number should be cared. In the exmple, /dev/shm0 and dev/shm/0 will present same device as their suffixes are 0. The Makefile for linux_reader is simple.

```
# cat Makefile
```

By far, all the source code for the example are finished. Let's compile, execute the example and see the result.

```
# cd prtk_writer_path
# make
# make -C linux_reader_path
# loader.xm ...
# linux_reader_path/linux_reader
....
```

If you will see "Message from PaRTiKle!", it meanings your example is successful or failure.

Notes: before executing the example you should insure all XMeRTL has been installed successfully and shm devices have been created.

4. Device Driver

4.1. Device Driver Model

4.1.1. Introduction

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, appications 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.

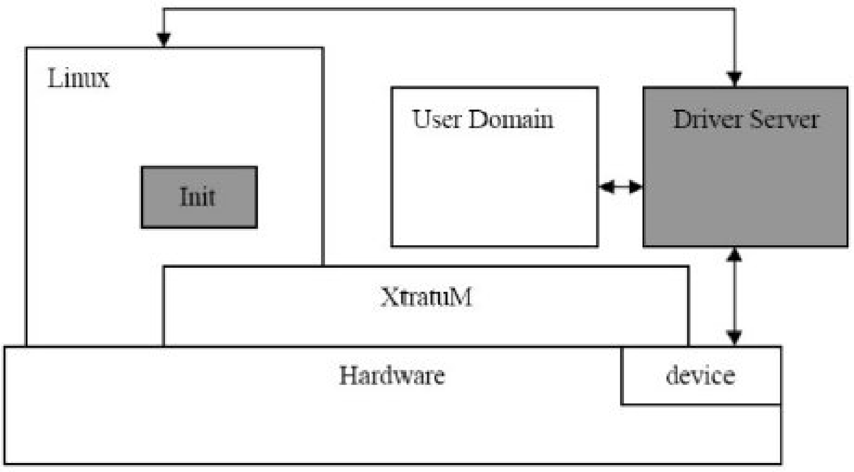

Fig1 : XtratuM Device Driver Model Arcihtecture

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:

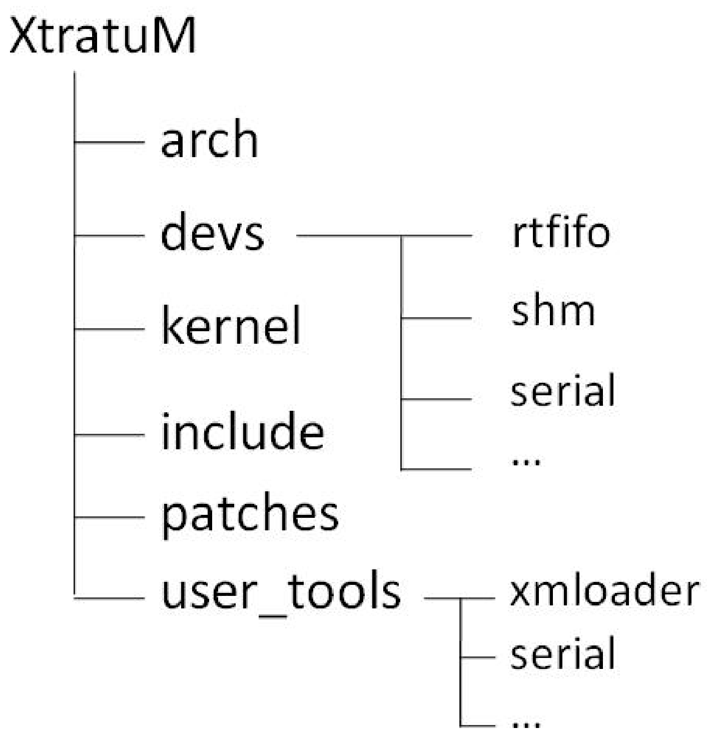

Fig2 : XtratuM source code structure

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);
```

– irq handler allocate
  The irq handler can be allocated by the functions in Linux kernel.

```
#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/n */
};
```

This is a reuse of Linux functions, but there is a different from the Linux in our device driver model. 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 has these values:

∗ 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.

```
// Virtual address to page directory entry
#define va2pd(vaddress) (vaddress >> PGDIR_SHIFT)

// Virtual address to page table entry
#define va2pt(vaddress) ((vaddress & 0x3FF000) >> PAGE_SHIFT)

// Page directory and page table to virtual address
#define pdpt2va(pd, pt) ((pd << PGDIR_SHIFT) | (pt << PAGE_SHIFT))
```

– 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;
```

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)
{
 ();
init_xm_dev_parameter();
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.

- irq handler

  Based on the different parameter passed by the function, you should implement different access method for your device.

- 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.

• Add the event handler
  Add the event handler in the structure of event_handling_struct in includeevents.h.

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 importandt 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 dada 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 dataÂ¡Â¯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 gived 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 init 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 init 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 SCCUCESSFULLY " 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 SCCUCESSFULLY\n"};

int kmain (void)
```

```
{
    struct xmitimerval 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 xmitimerval 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
$xmloader/loader.xm serial/serial.xmd   //load uart driver domain by loader.xm
>> Loading the domain "serial.xmd (serial.xmd)" ... Ok (Id: 1)
$dmesg
<XtratuM> VUART device loaded!
$xmloader/loader.xm 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
        $xmloader/loader.xm serial/serial.xmd
       $xmloader/loader.xm tests/server.xmd
on the client:
$dmesg
then will get the following information:
UART TEST SCCUCESSFULLY
```

The "UART TEST SCCUCESSFULLY " was transported by serial port, the result indicates that the driver domain runnig normaly and the client gets what it wants to get.
This is an 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) <7ffffffe> (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 befor this trace)
DUMP: cpos 2 (data_ready) <7ffffffc> (overwritten) 70 finalize 101
D 1 -181110149 xm-domain                        0x1 <0xce8213fe>
That was trace # 3 (lost 0 traces befor this trace)
DUMP: cpos 3 (data_ready) <7ffffff8> (overwritten) 70 finalize 101
D 1 -181109454 xm-domain                        0x2 <0xce8213fe>
....
```

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

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

and ends with a line like this:

```
That was trace # 3 (lost 0 traces befor 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 donates the which domain's context the trace event has happend. In our example it happend 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 resovled 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) <7ffffffe> (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) <7ffffffc> (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) <7ffffff8> (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.

```
#include <xm_syscalls.h>

void handler (int event, void *a) {
  //write_scr ("irq 0\n", 6);
  write_scr("hola\n", 6);
     //pass_event (event);
     //unmask_event (0);
}

int kmain (void) {
  struct xmitimerval req = {{0, 200000}, {0, 0}};
  struct xmtimespec t;
  int i;

  install_event_handler (0, handler);

  set_timer (&req, 0);

  unmask_event (0);

  get_time(&t);

  xmtrace_user(20,4711);
  xmtrace_user(18,4711);

  xprint("nsec_%lu_sec_%lu\n",t.tv_nsec, t.tv_sec);

  enable_events_flag();
  while (1){
          for(i=0;i< 100; i++){

                 xmtrace_user(20,i);   //RTL_TRACE_USER2
                 xmtrace_user(18,0);   //RTL_TRACE_FINALIZE
```

```
            }
  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();
        }
        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 hander and domain main body. ¡xm_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 ¡xm_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 envent bit and enable the related event again. If you use the routine, the interrupt will be responsed all the time. But in my progamm, it responces once only.
kmain() is the domain entry routine. There is no parameter for kmain() and the return value type is integer. The porpuses of the kmain routine are: 1) install a timer handler for the domain; 2) initilize 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 xmitimerval 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 envents */
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 XtratuM. 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: 0xffffffef
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.xm -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 mechnism 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 (&sleeptime , 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() rountine. 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 programms 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. List of Acronyms

`REDD` - Realtime Ethernet Device Drivers
`CVS` - Concurent Version Control
`GNU` - GNU Not UNIX (recursive acronym)
`TLSF` - TwoLevelSegregatedFit (Memory Storage allocator implementation)
`MAC` - Media Access Control
`RT` - RealTime

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, Quinggou 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.