Analysis of User Level Device Driver usability in embedded application

-Technique to achieve good real-time performance-

Katsuya Matsubara Takanari Hayama
Hitomi Takahashi Hisao Munakata

IGEL Co., Ltd
Renesas Solutions Corp.

Background

Device Driver Development in Embedded World is different in the following senses:

- Non-common New Devices
  - Due to newly developed devices, it is quite hard to re-use the device driver from previous development.
  - Some of the device are common in embedded world, but not in the Linux world, i.e. new in Linux.
- Closed relationship with Applications
  - Tends to be monolithic system architecture. Application requires to manage devices directly in fine-grain.
  - Only one application dominantly uses the device.
- Single-user, multi-task
- IPR issue
- Requires easiness of device driver development
  - Short development cycle

Objective

- Design of a framework for User-Level Device Driver
- Evaluation on Implementation Methodology and its Environment
- Related Work
  - Peter Chubb, “Get more device drivers out of kernel,” OLS2004.

Required Features to Realize ULDD

- Memory Access
  - I/O Memory
    - In many cases, devices are controlled over registers.
    - In some cases, access to device memory is required to perform I/O.
  - RAM
    - Large contiguous memory is needed for DMA transfer etc.
- Interrupt Handling
  - Communication between device and host CPU needs to be interrupt driven.
- Latency Guarantee
  - How quick can a user task run after reception of interrupt, needed to be guaranteed or presumable.
- Disabling Interrupt
  - Interrupt handler needs to be able to disable interrupts and run dominantly.

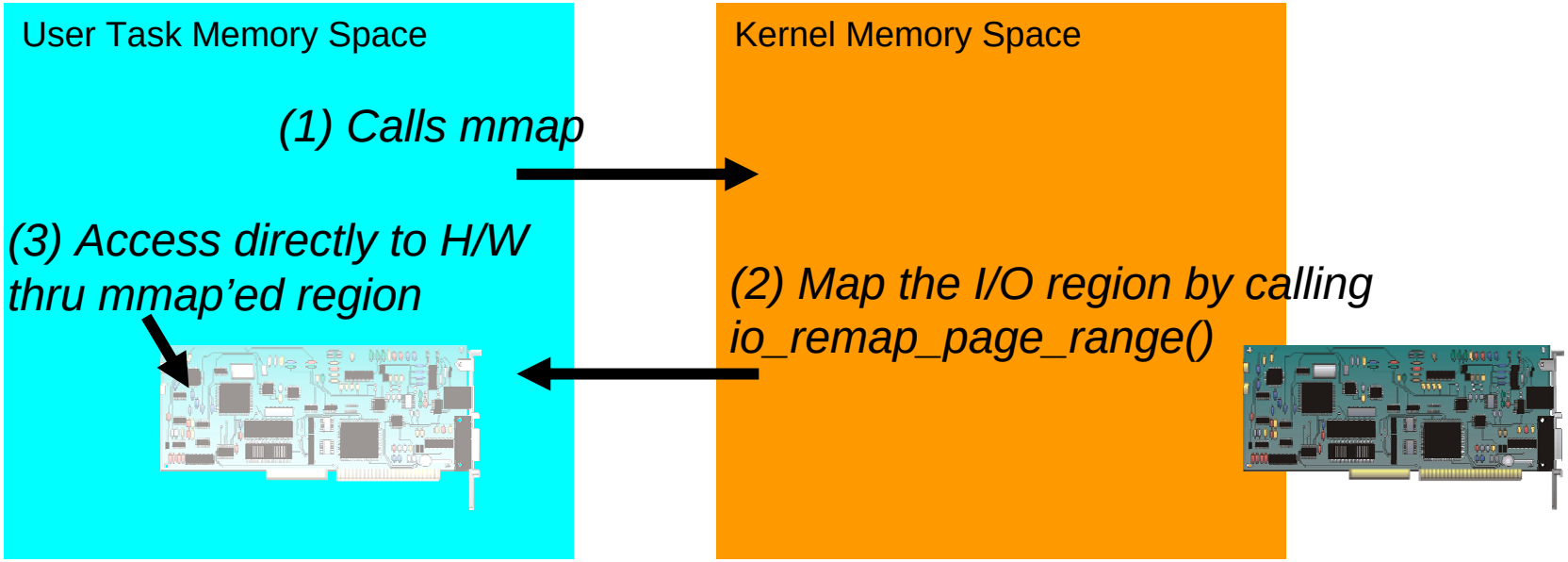
Design Principal

- Memory Access
  - Access to I/O Memory
    - Allow mmap(2) the I/O registers
  - Contiguous Memory Allocation
    - Allow mmap(2) the contiguous memory allocated by the kernel.
- Interrupt Handling
  - Two Methodologies to Awake User Task
    - Synchronous: Wake up task sleeping on I/O event
    - Asynchronous: Send UNIX signal
- Latency Guarantee and Disabling Interrupt
  - RT Task
  - NPTL
  - O(1) Scheduler
  - Kernel Preemption etc.

Memory Access

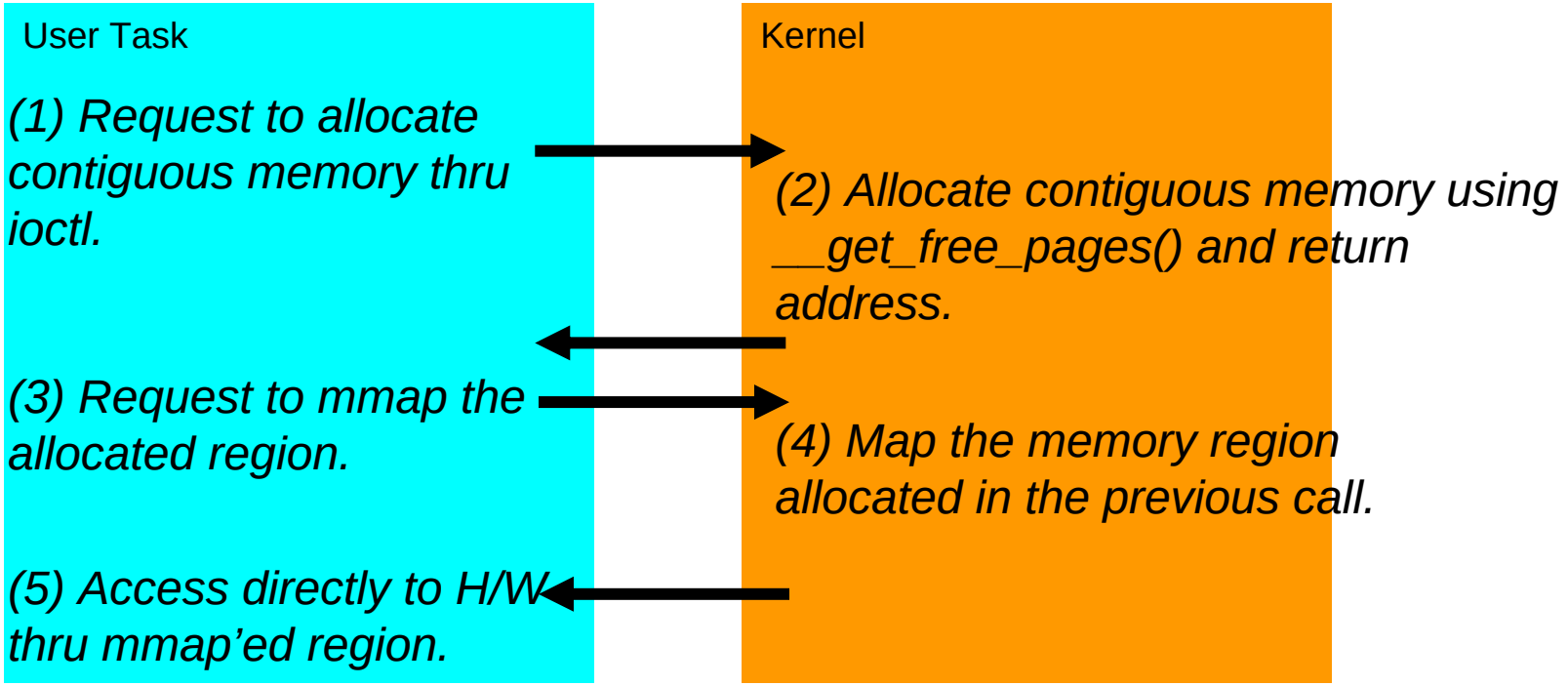
Memory Access

- Access to I/O memory such as RAM and registers.
  - Make accessible by memory mapped I/O (mmap) from the user task.



Contiguous Memory Allocation

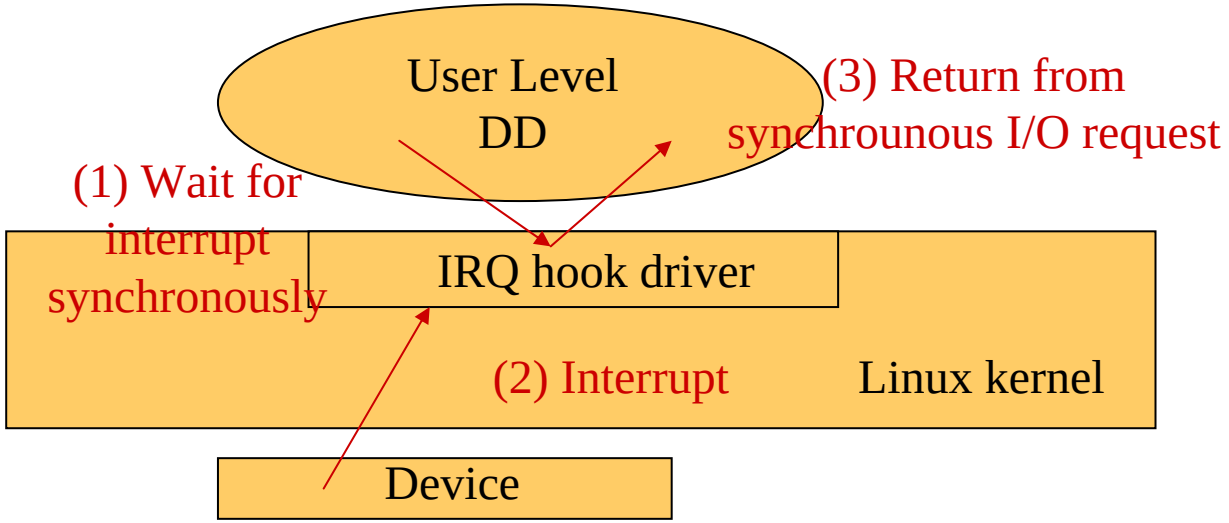
- Allocate contiguous memory in the kernel driver, and let user task to access through mmap(2).



Interrupt Handling

Synchronous Interrupt Handling

- Wake up the task from the kernel using synchronous file I/O.



API for Waiting Interrupt Synchronously

- Specify IRQ number to wait by `[irqno]`

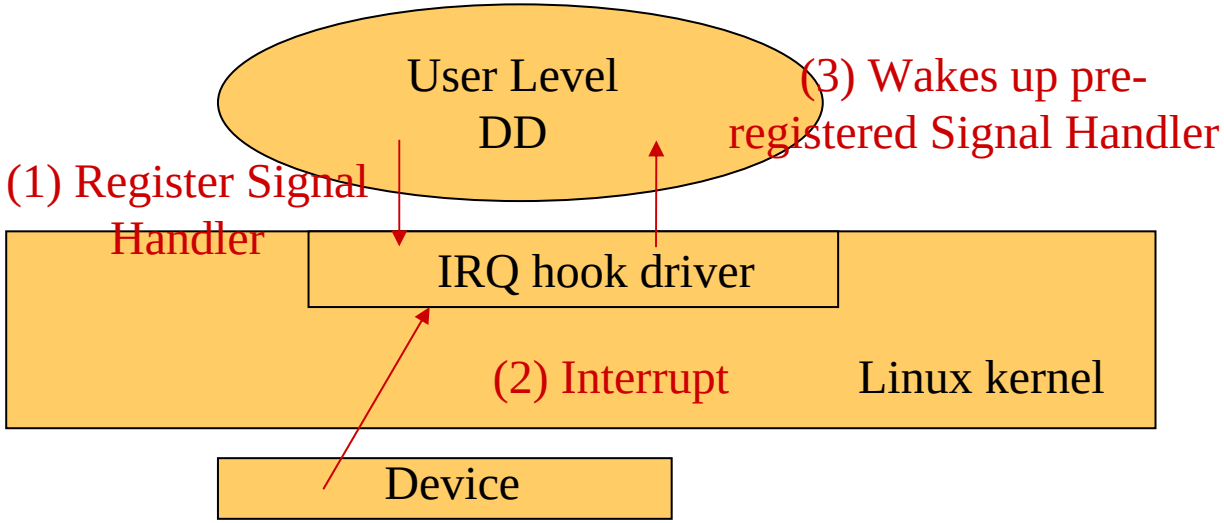
```
intr_fd = open("/proc/irqhook/[irqno]",
O_RDWR);
```

- Wait for Interrupt

```
read(intr_fd, &i, sizeof(int));
```

Asynchronous Interrupt Handling

- Send UNIX signal to pre-registered task when interruption occures.



API for Waiting Interrupt Asynchronously

- Registering Interrupt Handler

```
act_sig.sa_handler = input_handler;
sigaction(SIGIO, &act_sig, NULL);
```

- Specify IRQ Number to wait by [irqno]

```
intr_fd = open("/proc/irqhook/[irqno]",
O_RDONLY);
```

- Wait for Interrupt

```
oflags = fcntl(irqfd, F_GETFL);
fcntl(irqfd, F_SETFL, oflags | FASYNC);
read(intr_fd, &i, sizeof(int));
```

Latency Guarantee and Disabling Interrupt

Latency Guarantee and Disabling Interrupt

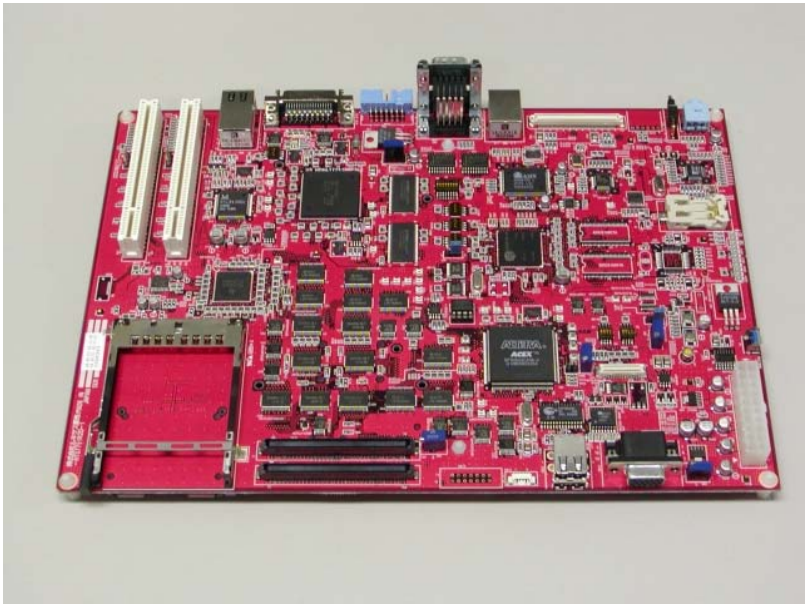
- To restrain context switch while device driver is processing interrupt, and to minimize the latency to wake up device driver, RT task shall be used.
- Linux 2.6 kernel that employs improved NPTL, O(1) Scheduler, Kernel Preemption etc should minimize the latency.

Evaluation to see how these work using real ULDD implementation!!

Prototype Implementation

To Evaluate, Implemented SM501 UART Device Driver on Renesas RTS7751R2D Evaluation Board as ULDD

- SM501 UART Device
  - 8250 Compatible
  - Supports Byte I/O Mode and FIFO Mode（For this experiment, we used byte I/O mode）



http://tree.celinuxforum.org/pubwiki/moin.cgi/RTS7751R2DHandlingManual

I/O Memory in the Kernel

```
int iommap_mmap(struct file *filp,
                struct vm_area_struct *vma) {
    size_t size = vma->vm_end - vma->vm_start;
    unsigned long offset =
                  vma->vm_pgoff << PAGE_SHIFT;

    ...
    if (io_remap_page_range(vma, vma->vm_start,
                            offset, size,
                            vma->vm_page_prot))
        return -EAGAIN;
    return 0;
}
```

I/O Memory Access in ULDD

```
/* mmap the IO memory */
addr = mmap(0, IOMEM_SIZE, PROT_READ|PROT_WRITE,
            MAP_SHARED, iomap_fd, IOMEM_ADDR);

/* wait for an interrupt and then receive data */
if (read(intr_fd, &i, sizeof(int)) == sizeof(int)) {
    if ((st = *(u_char *)(addr + STATREG_OFFSET)
         & 0x01) {
        do {
            /* get a byte from RX register */
            dt = *(u_long *)(addr + RXREG_OFFSET);
```

Interrupt Handling in the Kernel

```
irqreturn_t irq_handler(int irq, void *vidp,
                        struct pt_regs *regs) {
    ...
    if(idp->fasync){
        /* Notify by UNIX signal (SIGIO) */
        kill_fasync(&idp->fasync, SIGIO, POLL_IN);
    } else {
        /* Wakeup task by usual notification */
       wake_up(&idp->q);
    }
    return IRQ_HANDLED;
}
```

Interrupt Handling in ULDD (Synchronous)

```
/* mmap the IO memory */
addr = mmap(0, IOMEM_SIZE, PROT_READ|PROT_WRITE,
            MAP_SHARED, iomap_fd, IOMEM_ADDR);

/* wait for an interrupt and then receive data */
if (read(intr_fd, &i, sizeof(int)) == sizeof(int)){
    if ((st = *(u_char *)(addr + STATREG_OFFSET))
         & 0x01) {
        do {
            /* get a byte from RX register */
            dt = *(u_long *)(addr + RXREG_OFFSET);
```

Interrupt Handling in ULDD (Asynchronous)

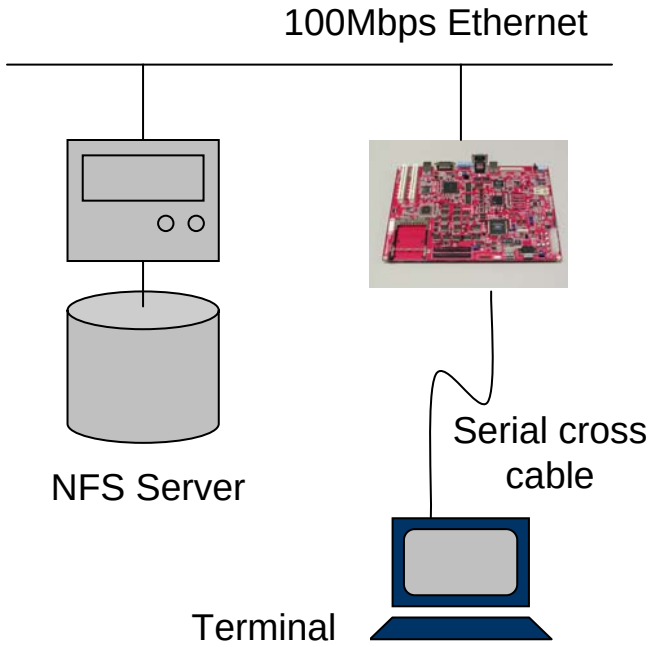
```
    oflags = fcntl(intr_fd, F_GETFL);
    fcntl(intr_fd, F_SETFL, oflags | FASYNC);
    s.sa_handler = sigio_handler;
    sigaction(SIGIO, &s, NULL);
    read(intr_fd, &i, sizeof(int));
     ...
}
void sigio_handler(void) {
  if((st = *(u_char *)(addr+STATREG_OFFSET)) & 0x01) {
        do {
            /* get a byte from RX register */
            st = *(u_long *)(addr+RXREG_OFFSET);
```

Experiments and Evaluation

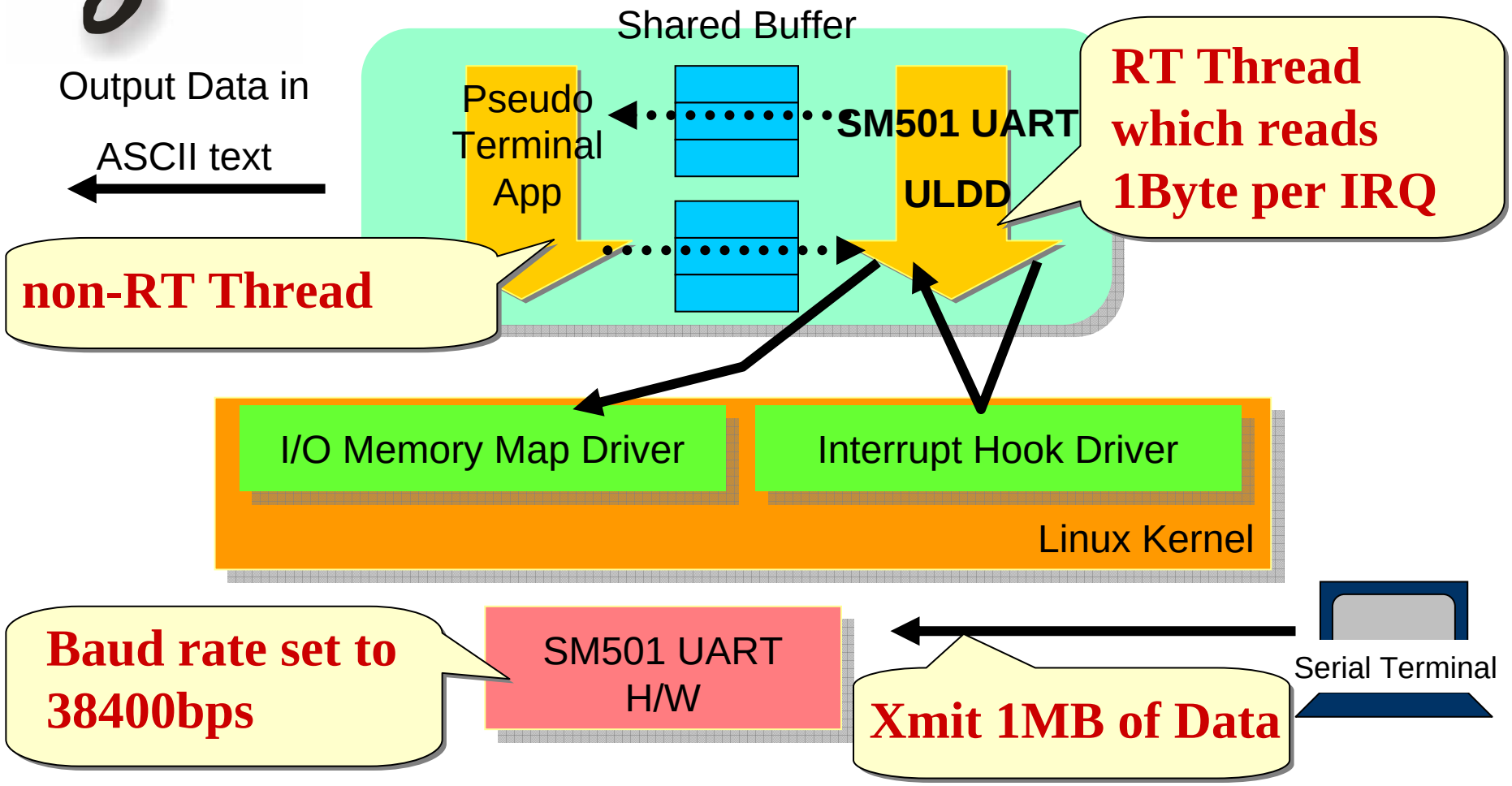
- Performed experiments on RTS7751R2D's SM501 UART ULDD device driver under the following condition:
  1. Evaluation on Interrupt Handling
     a. File I/O (Synchronous) vs UNIX Signal (Asynchronous)
  2. Evaluation on Latency
     a. RT Task vs non-RT Task
     b. Kernel Level D/D vs User Level D/D
     c. Linux 2.4 vs 2.6

Environment for Experiments

- H/W
  - Renesas RTS-7751R2D evaluation board
    - Renesas SH7751R(SH-4) 240MHz
    - 64MB RAM, 100Mbps Ethernet
  - NFS Server for rootfs
    - Intel Pentium4 2.8GHz
    - 512MB RAM, IDE HDD, 100Mbps Ethernet
  - Serial Terminal
    - Intel Pentium4 laptop
    - Connected with 32kbps serial
- S/W
  - Linux 2.6.13.4
  - glibc 2.3.3
  - Compile option: -O2 -g
  - Latency Measurement
    - Kernel Space: `current_kernel_timer()`
    - User Space: `clock_gettime(CLOCK_REALTIME)`



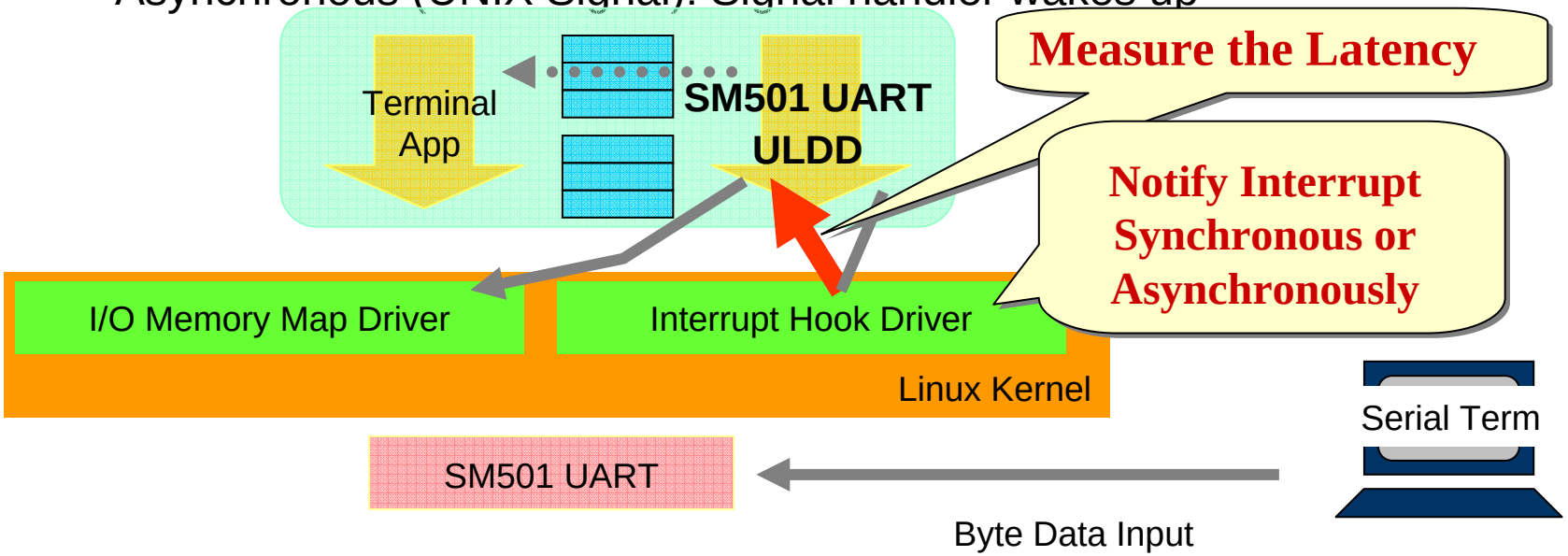
Architecture: SM501 UART ULDD and Terminal Application



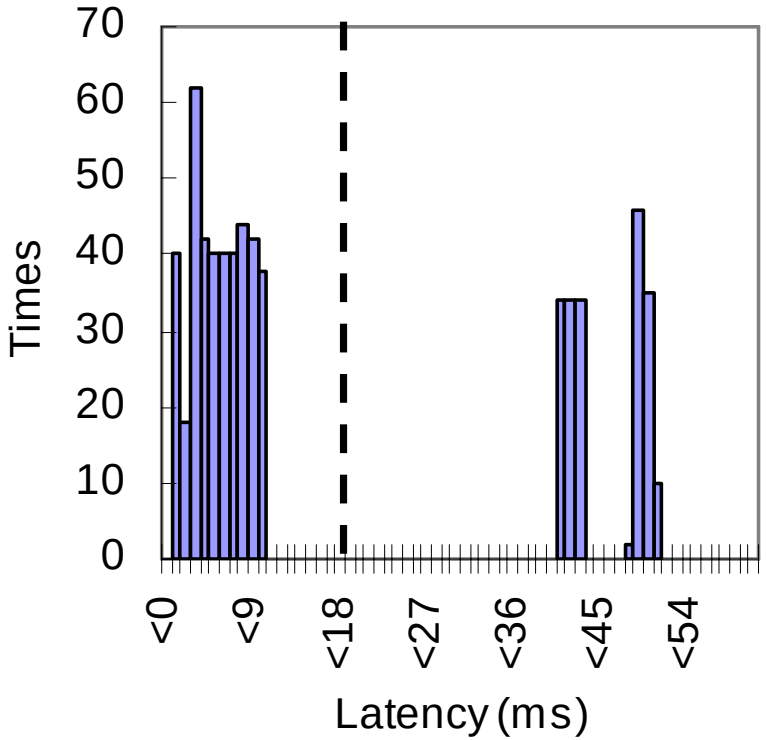
Evaluation on Interrupt Handling

Experiment 1-a: Comparison on Interrupt Reception Methodology

- Measure the latency between the time interrupt hook driver receipt the interrupt and ULDD wakes up, i.e. either one of the followings
  - Synchronous (File I/O): Return from the read() call
  - Asynchronous (UNIX Signal): Signal handler wakes up



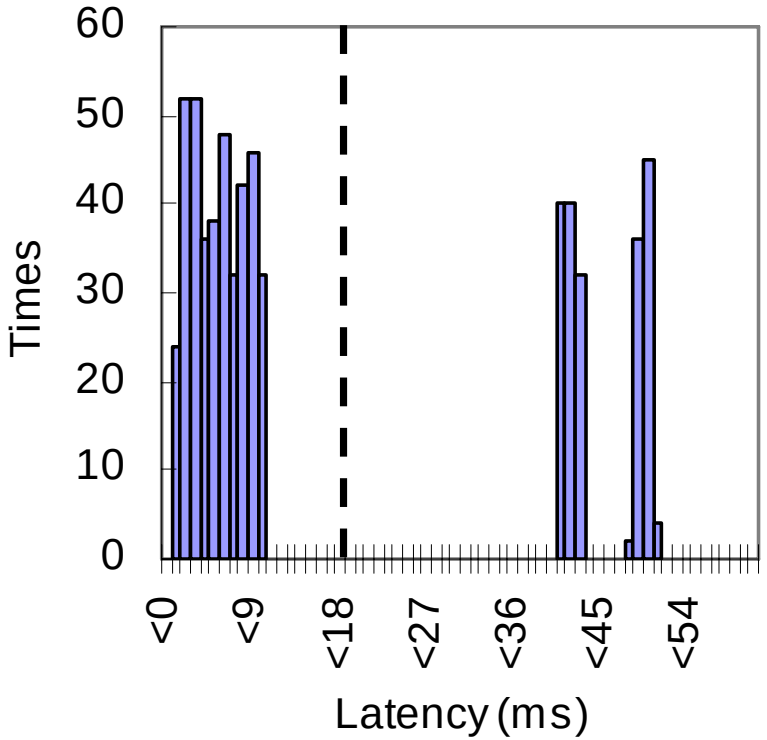
Experiment 1-a : Result (Sychronous)



Measurement	600
Average (ms)	18.06
Maximum (ms)	50.1
Minimum (ms)	0.076

- The worst case was 50ms.
- The worst case scenario is not the rare case.

Experiment 1-a : Result (Asychronous)



Measurement	600
Average (ms)	18.15
Maximum (ms)	50.02
Minimum (ms)	0.019

- The worst case was again 50ms.
- No big difference from synchronous one.

Experiment 1-a: Observations

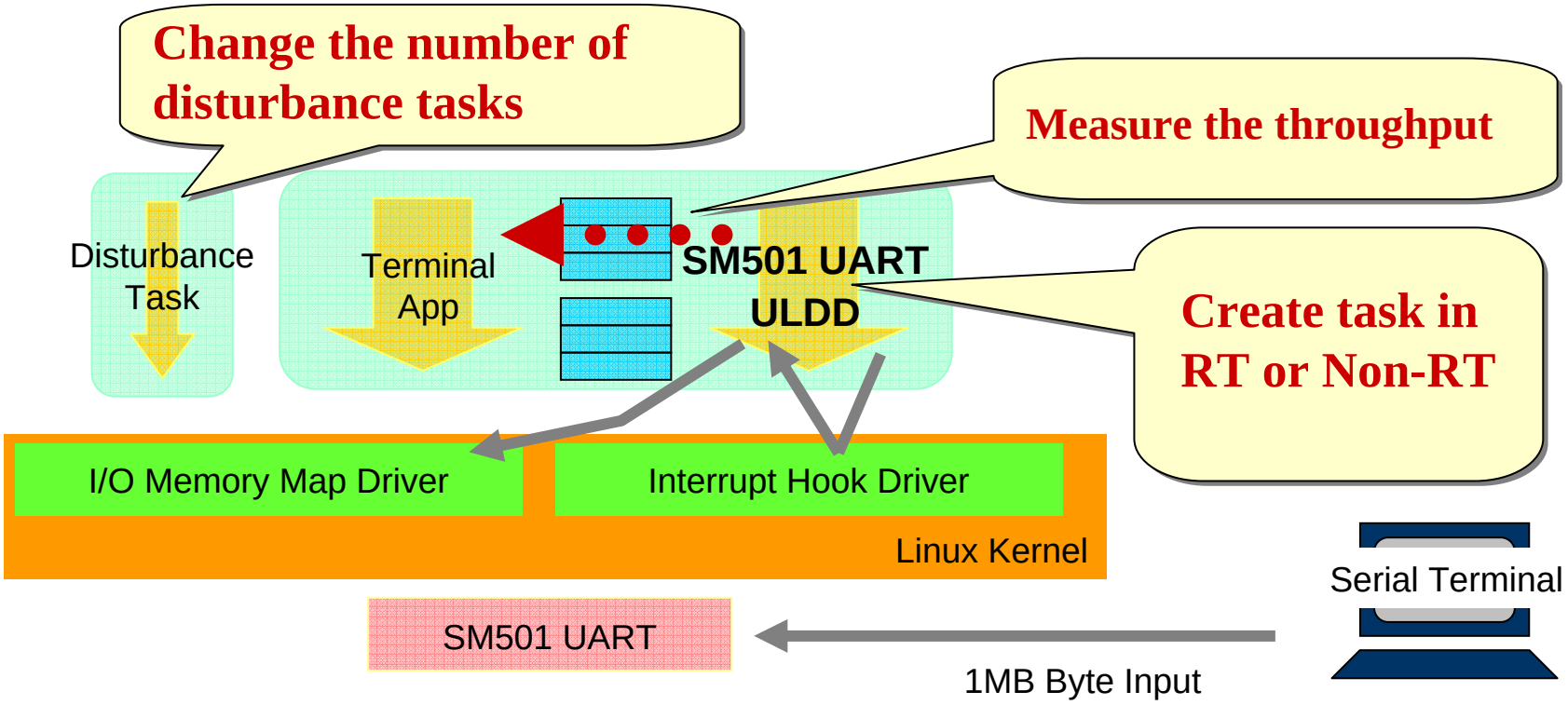
- No obvious differences have been observed between synchronous and asynchronous interrupt handling.
- However, when transmission speed is increased, i.e. increase interruption frequency, asynchronous method using UNIX signal couldn’t catch up and, with “I/O possible” error in glibc, application has terminated.

Evaluation on Latency

- RT Task vs non-RT Task -

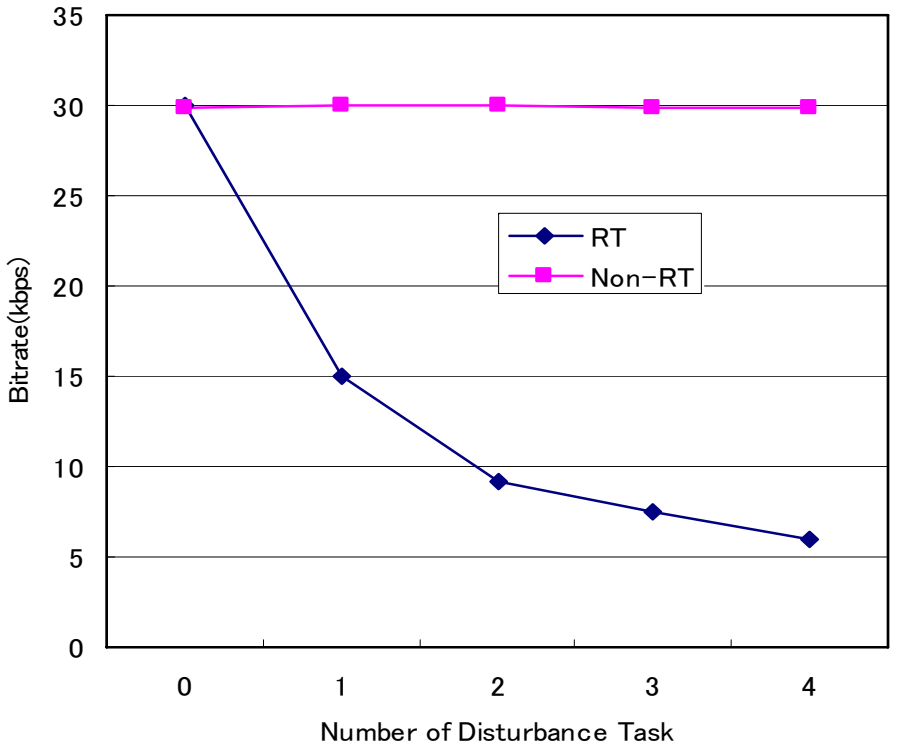
Experiment 2-a: Characteristic of RT Task

- By running CPU consuming disturbance in parallel, measure the impact on ULDD task running as either RT or Non-RT Task.



Experiment 2-a: Results and Observations

- No difference between non-RT and RT when there is no disturbance tasks.
- In existence of disturbance tasks, proportional to number of disturbance task the non-RT task has been delayed.
- In the case of RT task, no influence of disturbance task has been observed.

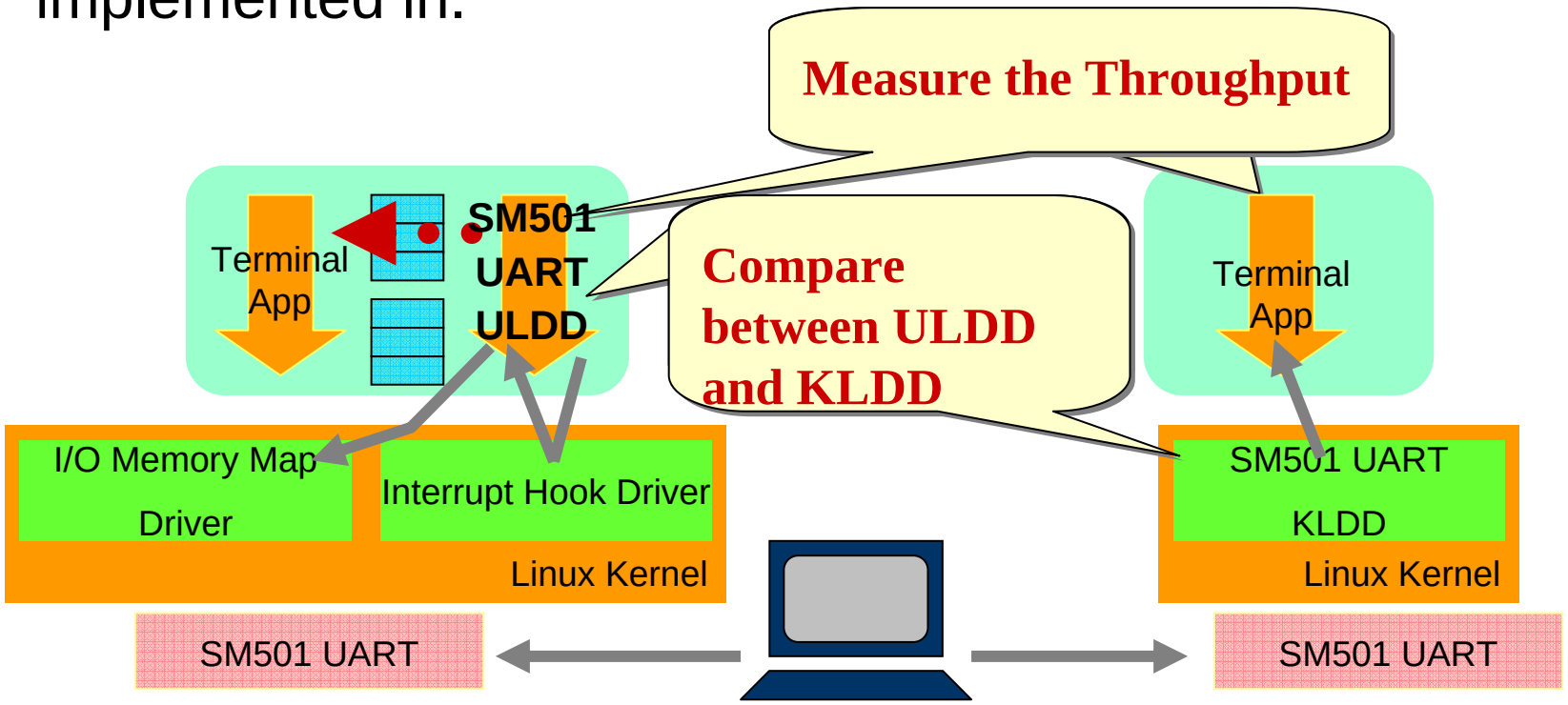


Evaluation on Latency

- Kernel Level D/D vs User Level D/D -

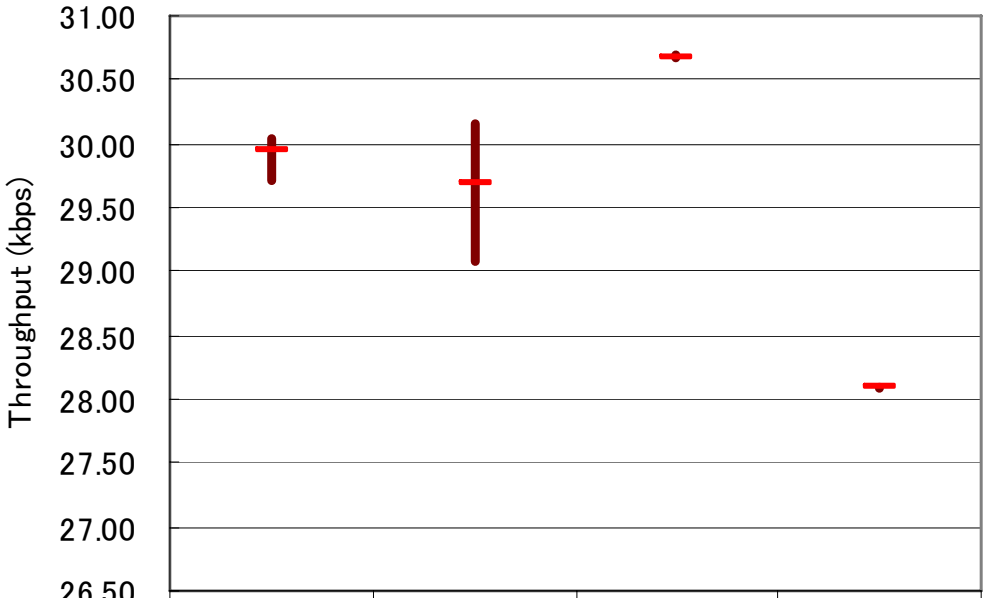
Experiment 2-b: Overhead by Implmenting ULDD

- Evaluate the influence of layers device driver is implemented in.



Experiment 2-b: Result and Observation

- ULDD got stable result than that of KLDD
- This is the result of unification of device driver and consumer application, i.e. resulting in lesser number context switch and memory copy.



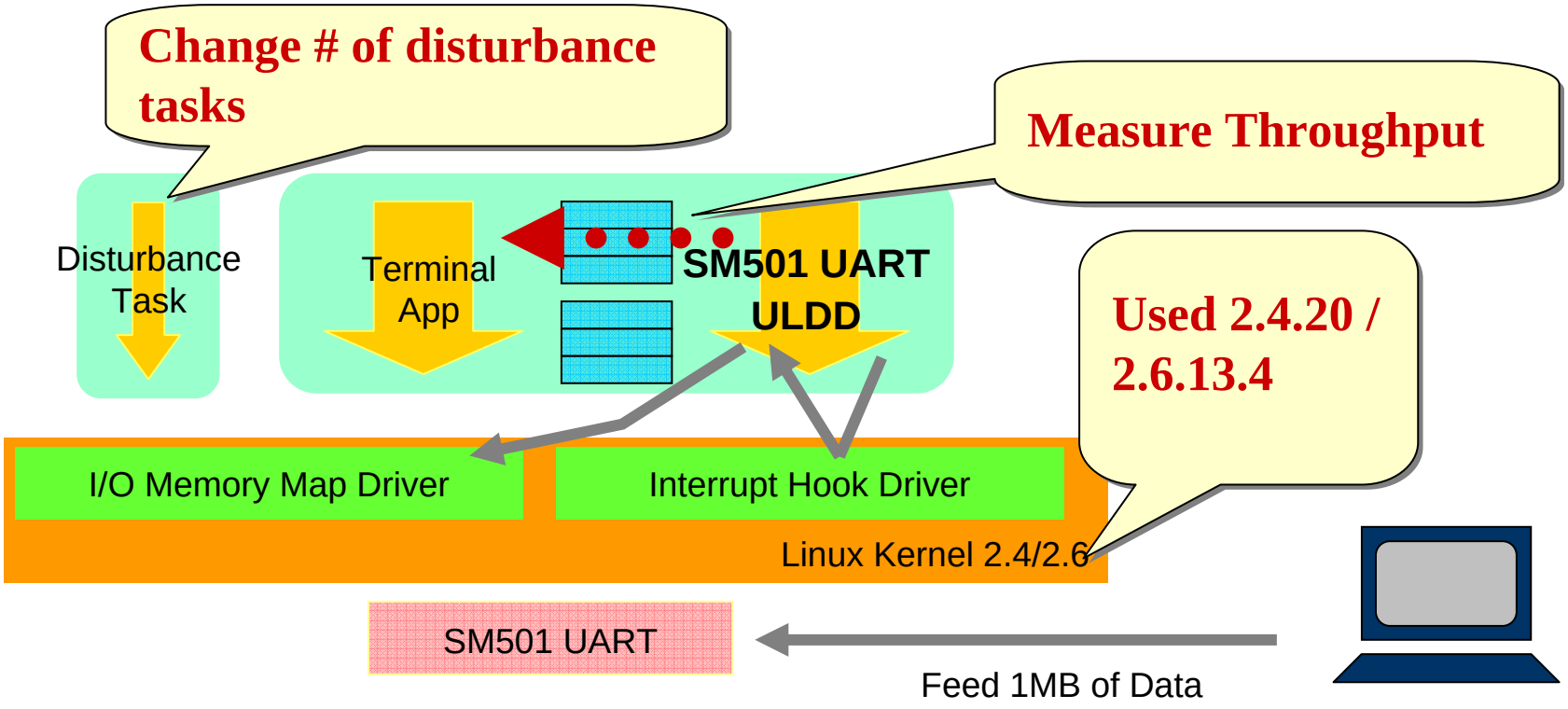
	ULDD (read)	KLDD (read)	ULDD (write)	KLDD (write)
MAX	30.04	30.15	30.69	28.08
MIN	29.71	29.08	30.68	28.08
AVG	29.95	29.69	30.68	28.08

Evaluation on Latency

- Linux 2.4 vs 2.6 -

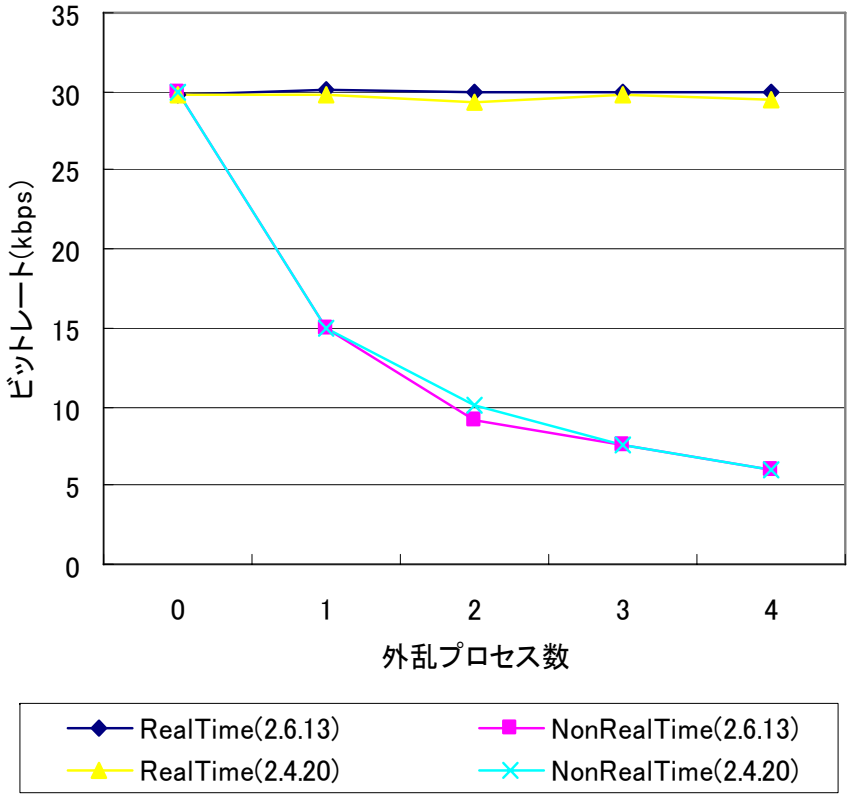
Experiment 2-c: Linux 2.6 vs 2.4

- Evaluate how new features in 2.6 helps ULDD



Experiment 2-c: Results

- None major difference has been observed.
- This may due to:
  - Too few tasks
  - Frequency of interruption is not high enough.



Conclusion

- Implemented re-using existing File I/O, interrupt notification mechanism. No new system call is added.
- Implemented real ULDD device driver using the functionality above.
- Evaluated the usability of ULDD under embedded environment.

Future Work

- Evaluate the feature like RT_PREEMPT to see the impact to ULDD.
- Evaluate ULDD implementation on more various device to see its characteristic.
- Promote the use of ULDD ☺