Applying User-level Drivers on DTV System

Gunho Lee, Senior Research Engineer,
LG Electronics
ELC, April 18, 2007
Content

- Background
- Requirements of DTV Device Drivers
- Design of LG DTV User-level Drivers
  - Kernel Module (UDM)
  - Application SDK (UDD-SDK)
- Implementation of User-level Drivers
- Performance Evaluation
- Conclusion
Background
Background

◆ What is problem in developing Kernel-level device drivers in embedded Linux systems?
  
  - Hard to debug
    * Most of driver developers are not expert on Linux system.
    ➔ Most of driver developers use only “printk”.
  
  - Unstable
    * Bugs in the Kernel-level driver are critical for the system stability. (Kernel panic may occur)

◆ User-level drivers can be a good choice…
Background

◆ Merits of user-level drivers
  – Easy to develop…
  – Easy to debug…(GDB? or others)

◆ Risk: Real-time performance degradation
  – Real-time performance is very important in the DTV system.
  – What are the time constraints required by the DTV drivers?
  – Recent improvements on real-time performance in Linux Kernel 2.6 provide good environments for user-level drivers.
Requirements of DTV Device Drivers
Device Drivers on DTV System

- Basic DTV system consists of devices below...
  - SDEC : System Decoder (or demux)
  - VDEC : Video Decoder
  - ADEC : Audio Decoder
  - VDP : Video Display Processor (scaler)
  - OSD : On Screen Display
  - GFX : Graphic acceleration engine
  - I2C : Inter-Integrated Circuit
  - GPIO : General Purpose I/O
  - and Etc…

- Each device has control registers.
- Some devices have large buffer memory.
Features of Kernel-level DTV Drivers

◆ Memory access
  – Provide accessibility to the registers.
  – Provide accessibility to the large buffer memory.

◆ Interrupt handling
  – Provide ISR. (interrupt service routine)
  – Provide control over IRQ. (enabling/disabling)

◆ Real-time responsibility
  – Some DTV drivers require real-time responsibility.
    • ISR should finish job within a guaranteed latency.
# Real-time Requirements

- Representative real-time requirements of DTV drivers

<table>
<thead>
<tr>
<th>Drivers</th>
<th>Requirements</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDEC</td>
<td>PAT (Program Association Table) section filtering (interval of PAT section is 10msec)</td>
<td>10 msec</td>
</tr>
<tr>
<td></td>
<td>PCR (Program Clock Reference) recovery</td>
<td>&lt; 1 msec</td>
</tr>
<tr>
<td>VDEC</td>
<td>Caption data processing (interval of V-sync is 16msec)</td>
<td>16 msec</td>
</tr>
<tr>
<td>VDP</td>
<td>Frame error correction (complete during blank lines of frame)</td>
<td>1.39 msec</td>
</tr>
<tr>
<td>For all</td>
<td>Interrupt pending clear</td>
<td>At the Kernel ISR</td>
</tr>
</tbody>
</table>
Design of LG DTV
User-level Drivers
How Kernel-level Drivers are Used in LG DTV

- All drivers existed in the Kernel.
- Event (= outcome of ISR) was delivered to the event handler task.
- Each driver has an ISR, event queue and event handler task.
- Interrupt pending clear and status clear are done in the ISR.
Principles in Converting to User-level Drivers

◆ Minimize Kernel-level codes
  – Implement drivers in user-level, except some time critical codes.

◆ Minimize overhead
  – Simple and compact structure to reduce performance degradation.

◆ Easy to develop
  – DTV SW developers should adapt to new environments easily.
Requirements

◆ Memory access: same as Kernel-level drivers…
  – Provide accessibility to the registers.
  – Provide accessibility to the large buffer memory.

◆ Interrupt handling: ISR in the user-level
  – Provide interface to deliver Kernel IRQ to user task (U-IRQ)
  – Provide interface for user-level ISR (U-ISR, awaken by U-IRQ)
  – Provide control over IRQ & UIRQ (enabling/disabling)

◆ Real-time performance
  – Minimal time critical codes in the Kernel-level.
  – Minimize and guarantee the U-ISR latency.
Memory Access

User-level drivers can access control registers and buffer memory by mapping the physical memory.

User-level Driver

1. Request mapping by calling “mmap()”
2. Map the physical memory region by calling “io_reamp_pfn_range()”
3. Get a virtual address mapped
4. Can access registers and buffer

Kernel Module for the User-level Driver
Interrupt Handling: U-IRQ

- Methodology to deliver Kernel IRQ to user task (U-IRQ)
  - Use synchronous file I/O (system call “read()”)

1. Wait for interrupt by calling “read()”
4. Task wakes up (return from “read()”)
5. Run U-ISR

2. Wait for interrupt by calling “wait_event_interruptible()”
4. Wake up the blocked “read()” by calling “wake_up_interruptible()”

3. Receive an interrupt (IRQ)
Interrupt Handling: U-ISR

◆ Implementation of U-ISR (waken up by U-IRQ)
  – U-IRQ handler task is a real-time thread with maximum priority. It will run dominantly.

1. Receive an U-IRQ
2. Check vector table
3. Run U-ISR
4. Wait for next U-IRQ

Kernel Module for the User-level Driver
Interrupt Handling: Controlling IRQ

◆ To enable and disable IRQ
  – Use the file I/O (system call “ioctl()”)
  – This controls the HW interrupt in the Kernel module (using api “enable_irq()” and “disable_irq()”)

◆ To enable and disable U-IRQ
  – Also use the file I/O (system call “ioctl()”)
  – This controls the U-IRQ queue (FIFO) in the Kernel module (using flags “UIRQ_ENABLED” and “UIRQ_DISABLED”)

U-IRQ Handler Task

Kernel Module for the User-level Driver

U-IRQ Queue (FIFO)

ISR

H/W Interrupt

UIRQ_ENABLED / UIRQ_DISABLED

enable_irq() / disable_irq()
Real-time

◆ Following time critical codes should be implemented in the Kernel-level.

<table>
<thead>
<tr>
<th>Drivers</th>
<th>Time critical codes</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDEC</td>
<td>PCR recovery</td>
<td>&lt; 1 msec</td>
</tr>
<tr>
<td>For all</td>
<td>Interrupt pending clear</td>
<td>At the Kernel ISR</td>
</tr>
</tbody>
</table>

◆ Minimize and guarantee the IRQ delivery latency.
  – Use linux 2.6 Kernel.
  – Use real-time thread with maximum priority.
Implementation of User-level Drivers
Kernel Module & SDK for User-level Drivers

◆ User-level Driver Module (UDM)
  - Kernel module to provide…
    ● map physical memory region to user memory space.
    ● enable/disable IRQ & U-IRQ.
    ● deliver Kernel IRQ to user handler task.
    ● run time critical codes in Kernel-level.

◆ User-level Driver SDK (UDD-SDK)
  - Provides user-level APIs by calling UDM.
    ● get user memory space mapped with physical memory region.
    ● request U-IRQ and register U-ISR for it.
    ● enable/disable IRQ & U-IRQ.
Structure of User-level Drivers

User-level DTV Drivers

Middleware

Device Driver Interface

Driver-1

Driver-2

Driver-n

Application

Send event (asynchronous message)

User-level DTV Drivers

UDD-SDK

U-IRQ Handler Task

Vector Table

U-ISR-1

U-ISR-2

U-ISR-n

UDM

U-IRQ Queue (FIFO)

Pending Clear

Time Critical

Extra

ISR

H/W Interrupt

H/W Interrupt

H/W Interrupt

Kernel Module

Kernel

Hardware

Send event (asynchronous message)

Clear interrupt pending register here...

Time critical codes are here...

CE Linux Forum

ELC 2007, April 18

LG Electronics
typedef void (*UserISRFunc)(int uirqn);

typedef void (UserISRFunc)(int uirqn, unsigned int irq_status_h, unsigned int irq_status_l, const char *dev_name, unsigned int latency, unsigned int count);

typedef struct _UDM_INT_INFO {
    unsigned int uirq;
    unsigned int uirq_status_h;
    unsigned int uirq_status_l;
    unsigned int uirq_wait_count;
    char name[16];
} UDM_INT_INFO;

typedef struct _UDM_INT_INFO_T {
    int uirqn;
    int usage;
    unsigned int irq_status_h_addr;
    unsigned int irq_status_l_addr;
    char name[16];
} UIRQ_INFO_T;

typedef struct {
    int irqn;
    unsigned int irq_status_h;
    unsigned int irq_status_l;
    unsigned int calibration;
    unsigned int usage;
    char name[16];
} UIRQ_INFO_T;

Interrupt Handling Flow: SDEC Driver

1. Request U-IRQ by calling “UDD_RequestUIRQ()”

2. Request IRQ by calling “request_irq()”

3. Receive interrupt

4. Call

5. Call

PCR recovery

No extra codes

6. put an U-IRQ

7. Read U-IRQ (UDM_INT_INFO)

8. Run U-ISR (UserISRFunc)

9. Send event

1. Request U-IRQ by calling “UDD_RequestUIRQ()”

2. Request IRQ by calling “request_irq()”

3. Receive interrupt

4. Call

Clear SDEC’s interrupt pending register

5. Call

6. put an U-IRQ

7. Read U-IRQ (UDM_INT_INFO)

8. Run U-ISR (UserISRFunc)

9. Send event
**UDD-SDK APIs**

- **UDD_SDK_STATE UDD_SDK_Init**(void)
  - Initialization function
- **UDD_SDK_STATE UDD_SDK_Release**(void)
  - Release function
- **UDD_SDK_STATE UDD_SetLogLevel**(UDD_LOG_LEVEL loglevel)
  - Set logging level (run-time changeable)
- **UDD_SDK_STATE UDD_RequestUIRQ**(UIRQ_INFO_T uirqInfo, UserISRFunc uisrFunc)
  - Request IRQ and register U-ISR
- **UDD_SDK_STATE UDD_EnableIRQ**(unsigned int uirq)
  - Enable IRQ in Kernel
- **UDD_SDK_STATE UDD_DisableIRQ**(unsigned int uirq)
  - Disable IRQ in Kernel
- **UDD_SDK_STATE UDD_EnableUIRQ**(unsigned int uirq)
  - Enable U-IRQ in user-level
- **UDD_SDK_STATE UDD_DisableUIRQ**(unsigned int uirq)
  - Disable U-IRQ in user-level
- **UDD_SDK_STATE UDD_MemMap**(int nLength, int nProt, int nFlags, unsigned int PhysAddr, int * pVirtAddr)
  - Request mapping of physical memory region
Memory Access

◆ In the user-level driver

```c
{
...
    UDD_MemMap(SDEC_SIZE, PROT_READ | PROT_WRITE,
                MAP_SHARED, SDEC_BASE_ADDR, &SdecBase)
...
/* can access physical memory directly through SdecBase */
}
```

◆ In the UDD-SDK (user-level)

```c
int UDD_MemMap(int nLength, int nProt, int nFlags, unsigned int PhysAddr, int * pVirtAddr)
{
...
    nMemMapped = (int) mmap(0, nLength, nProt, nFlags, g_fdMem, PhysAddr);
...
    * pVirtAddr = nMemMapped;
...
    return UDDSDK_OK;
}
```
Memory Access

◆ In the UDM (Kernel-level)

static int udm_mmap(struct file *file, struct vm_area_struct *vma)
{
    ...
    if (io_remap_pfn_range(vma,
                            vma->vm_start,
                            vma->vm_start,
                            vma->vm_pgoff,        /* Physical address */
                            vma->vm_end - vma->vm_start,  /* Size */
                            vma->vm_page_prot))
    {
        return -EAGAIN;
    }
    return 0;
}
Requesting IRQ & U-IRQ

◆ In the UDD-SDK (user-level)

```c
int UDD_RequestUIRQ(UIRQ_INFO_T *puirqInfo, UserISRFunc uisrFunc)
{
    ...
    /* request IRQ & U-IRQ from UDM (Kernel) */
    ioctl(g_fdUDM, CMD_REQUEST_IRQ, (unsigned int) puirqInfo)
    ...
    /* register U-ISR function in the U-IRQ vector table*/
    UIRQVectT.uirq[uirqn].UISRFunc = uisrFunc;
    ...
```

◆ In the UDM (Kernel-level)

```c
static int udm_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long param)
{
    switch (cmd) {
    case CMD_REQUEST_IRQ: {
        ...copy_from_user(&uinfo, (void *) param, sizeof(UIRQ_INFO_T));
        irqn = uinfo.uirqn;
        /* request IRQ */
        request_irq(irqn, udm_isr, IRQF_DISABLED, uinfo.name, NULL);
        ...
        /* enable U-IRQ */
        uirqInfo[irqn].usage = UIRQ_ENABLED;
    }
    ```
Controlling IRQ & U-IRQ

◆ In the UDM (Kernel)

static int udm_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long param) {
    switch (cmd) {
    ...{  
    case CMD_ENABLE_IRQ: {  
        ...  
        enable_irq(irqn);  
    }  
    case CMD_DISABLE_IRQ: {  
        ...  
        disable_irq(irqn);  
    }  
    ...  
    case CMD_ENABLE_UIRQ: {  
        ...  
        uirqInfo[irqn].usage = UIRQ_ENABLED;  
    }  
    case CMD_DISABLE_UIRQ: {  
        ...  
        uirqInfo[irqn].usage = UIRQ_DISABLED;  
    }  
    ...  
}
**U-IRQ Handler Task (1)**

◆ In the UDD-SDK (user-level)

```c
#define UISR_HANDLER_TASK_PRIORITY 99

int CreateUIRQHandlerTask
{
    ... 
    pthread_attr_getschedparam(&attr, &sched);
    sched.sched_priority = UISR_HANDLER_TASK_PRIORITY; /* set priority */
    pthread_attr_setschedparam(&attr, &sched);

    /* create RT task */
    if ((ret = pthread_create(pthd, &attr, (void *) UIRQ_HandlerTask, NULL)) != 0)
    ...
}

/* This is RT task with maximum priority. This task will run dominantly. */
int UIRQ_HandlerTask(void)
{
    while (1)
    {
        /* wait for interrupt. */
        ret = read(g_fdUDM, &g_UIRQ, sizeof(UDM_INT_INFO));
        ...
        UIRQVectT.uirq[irqn].UISRFunc(...); /* run U-ISR */
        ...
    }
```
In the UDM (Kernel)

```c
ssize_t udm_read(struct file *file, char __user *buffer, size_t count, loff_t *offset)
{
    ...
    /* blocked here */
    ret = wait_event_interruptible(&udm_int_waitq, udm_fifo_count > 0);

    if (ret == 0) /* success, condition (udm_fifo_cound > 0) is true */
        return fifo_copy_to_user(buffer);
    ...
}
```

```c
irqreturn_t udm_isr(int irq, void* dev_id, struct pt_regs *regs)
{
    ...
    if (uirqlInfo[irqn].usage == UIRQ_ENABLED) /* check U-IRQ usage */
        fifo_put(&uint_info); /* add U-IRQ to FIFO */
    ...
    /* wake up the blocked udm_read() */
    wake_up_interruptible(&udm_int_waitq);
    ...
}
```
Performance Evaluation
Environments

- Implement and test user-level drivers on LG’s own DTV chipset board.
  - H/W
    - 333 MHz core
    - 128MB DDR2 & 32MB flash
  - Kernel-level Drivers
    - Ethernet, uart, pci, sata, usb,…
  - User-level Drivers (8 drivers)
    - SDEC, VDEC, ADEC, VDP, OSD, GFX, I2C, GPIO

- Bootloader & Kernel & rootfs
  - U-boot-1.1.4
  - Linux 2.6.20.2 Kernel
  - uClibc 0.9.28
  - Squashfs-3.2
Measurement

- Measured the …
  1. **IRQ delivery latency** from Kernel interrupt to the U-IRQ handler task.
  2. **Elapsed time until the U-ISR is completed** from the Kernel interrupt occur.

- Test conditions
  - Kernel : **Non-preemptible** Kernel
  - Stress : With lightweight stress (channel change)

- Functions to get time
  - Kernel : “**do_gettimeofday()**”
  - User-level : “**gettimeofday()**”

- Test targets
  - SDEC, VDEC and VDP driver (they have real-time requirements)

- Test time
  - For **10 minutes**
**SDEC**

**Statistics**
- Average = 69.1 usec
- Minimum = 19 usec
- Maximum = 1,183 usec

**Statistics**
- Average = 150.4 usec
- Minimum = 33 usec
- Maximum = 1,199 usec

**Real-time requirement of SDEC**
- Under of **10,000 usec**
**VDEC**

**Statistics**
- Average = 270.6 usec
- Minimum = 20 usec
- Maximum = 1,255 usec

**Real-time requirement of VDEC**
- Under of 16,000 usec

**Statistics**
- Average = 312.4 usec
- Minimum = 35 usec
- Maximum = 1,331 usec
VDP

◆ Statistics
  – Average = 67.4 usec
  – Minimum = 18 usec
  – Maximum = 716 usec

◆ Statistics
  – Average = 229.3 usec
  – Minimum = 157 usec
  – Maximum = 993 usec

◆ Real-time requirement of VDP
  – Under of 1,390 usec
Conclusion

- Implemented all DTV drivers in user-level.
- User-level drivers satisfied the requirement of LG DTV.
- Built general architecture of user-level drivers (UDM, UDD-SDK)
Future Works

◆ Evaluate trade-offs between real-time performance and throughput.
◆ Evaluate the Ingo Molnar’s “Real-Time Preemption” Kernel.
◆ Extend UDM and UDD-SDK to apply on other embedded Linux systems.
Reference


◆ Real-time resources of CE Linux Forum, (http://tree.ceLinuxforum.org/CelfPubWiki/RealTimeResources)

◆ Real-time preemption patches (http://redhat.com/~mingo/realtime-preempt/)
Thank you !