Computer Architecture

LAB3: Virtual Machine & Linux Device Driver

Simulation Scenario

- We have created an environment that booting Linux kernel on ARM Realview EB by QEMU.
- This LAB will design a virtual hardware running in ARM Realview EB and interacting with Linux device driver and application to achieve some specific work.
- We implement a simple calculator as example.
- Application write two operands to calculator and let it go.
- When calculator complete, it will notify CPU through interrupt. CPU then execute ISR and wake up application to receive the result.

Define Virtual Hardware

- Register file (address and effect value)
  - We have to define the register space and memory space.
  - The hardware will execute the particular behavior according to the related registers' value.
  - Local memory space holds the necessary data.
- 3 basic functions
  - Virtual hardware simulated by QEMU
  - Initial Function
    - When virtual hardware is booting.
  - Read Function
    - When we wish to get some data from the virtual hardware
  - Write Function
    - When we want to write some data to virtual hardware's register file or memory

Design A Virtual Hardware

- Define calculator's register and memory
  - Register
    - go
      - Address: 0x0
      - Effect value: 0 stop, 1 start
    - operator
      - Address: 0x4
      - Effect value: 0 plus, 1 minus, 2 multiply, 3 divide
  - Memory
    - Address 0x8~0X10
      - 2 operands and 1 result
- The addresses above are offsets.

LAB3 總說明(1/2)

- 延續lab2的模擬環境，差別在於新增一個虛擬硬體calculator給QEMU進行模擬。
- 教材包檔案(皆為修改好的檔案，可直接使用)如下：

教材包 qemu-0.15.1-cal	檔案放置位置	說明
calculator.c	/qemu-0.15.1/hw/calculator.c	QEMU所模擬的虛擬硬體都會放在 HW 資料夾之下，所以我們將設計好的虛擬硬體－ calculator.c 放到 HW 資料夾下。
Makefile.target	/qemu-0.15.1/Makefile.target	讓QEMU知道必須編譯calculator此虛擬硬體
realview.c	/qemu-0.15.1/hw/realview.c	新增了一行程式碼 sysbus_create_simple("calculator", 0x80000000, pic[30]); 意思是指定虛擬硬體的base為0x80000000，且使用IRQ阜號30

- 執行步驟：

1) 在qemu新增完虛擬硬體calculator按照LAB1方法重新建構qemu
   - qemu-0.15.1]# make clean
   - qemu-0.15.1]# ./configure --target-list=arm-softmmu,arm-linux-user --prefix=/home/{your username}/qemu-bin
   - qemu-0.15.1]# make && make install
2) 重新對kernel做設定與編譯(教材包內的.config其實已經設定好了)
   - linux-2.6.38]# make ARCH=arm menuconfig
   - 設定**Enable loadable module support**開啟
   - linux-2.6.38]# make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
3) 於calculator-driver-app資料夾內下make指令編譯device driver
4) 將calculator-driver-app整包資料夾複製到/busybox-1.20.2/_install

教材包	檔案放置位置
calculator-driver-app (整包資料夾)	/busybox-1.20.2/_install/calculator-driver-app

LAB3 總說明(2/2)

- 執行步驟：

5) 產出相對應的initrd

- cd _install
- /busybox-1.20.2/_install]#
- /busybox-1.20.2/_install]# find . | cpio –H newc –o > ../initrd
- /busybox-1.20.2/_install]# gzip ../initrd

6) 利用qemu-system-arm進行全系統模擬

- qemu-test]# ./qemu-system-arm -M realview-eb -kernel zImage -initrd initrd.gz -cpu arm1136

qemu-test	檔案位置
qemu-system-arm (步驟1)	qemu-bin/bin/qemu-system-arm
zImage (步驟2)	linux-2.6.38/arch/arm/boot/zImage
initrd.gz (步驟3~5)	/busybox-1.20.2/initrd.gz

7) QEMU 模擬器開啟(ctrl+alt+3)

- # cd calculator-driver-app
- # insmod cal_drv.ko

察看Linux kernel設定我們driver的掛載編號

- # cat /proc/devices

替此device宣告一個node，應用程式要讀寫的時候是對此node做讀寫

- # mknod /dev/calculator c 254 0

執行應用程式並觀察回傳值是否正確

- # ./calculate 115 5

- LAB3總說明為概略性描述，接下來的投影片為詳細說明，務必仔細閱讀

Design A Virtual Hardware (1/5)

- Virtual hardware state
  - 下圖中，第一個structure為virtual hardware內部自己使用的資料結構，第二個structure則是向QEMU註冊virtual hardware所需要使用的資料結構。
  - 之後我們將會採用這兩組資料結構來進行虛擬硬體的初始化函式(Initial Function)。

```
typedef struct {
    SysBusDevice busdev;
    qemu_irq irq;
    const unsigned char *id;
} calculator_state;

static const VMStateDescription vmstate_calculator = {
    .name = "calculator",
    .version_id = 1,
    .minimum_version_id = 1,
    .fields = (VMStateField[]) {
        VMSTATE_END_OF_LIST()
    }
};
```

Design A Virtual Hardware (2/5)

- Initial Function
  - 向QEMU所模擬的系統BUS註冊虛擬硬體
    - FROM_SYSBUS
  - 告知CPU，若要對虛擬硬體存取必須透過那些function
    - cpu_register_io_memory
    - 其中readfn與writefn為函示陣列，陣列中三個function為byte-RW, half-RW和word-RW。我們設定全部的RW皆以word為單位，所以全指向同一個RW函式
  - 註冊虛擬硬體的記憶體空間
    - sysbus_init_mmio
    - 註冊0x1000。假設我們將此虛擬硬體掛載在0x80000000，那CPU存取 0x80000000 ~ 0x80000FFF就會使用上面定義的RW函式。
  - 註冊虛擬硬體的IRQ阜號
    - sysbus_init_irq
  - 初始化所有register及memory

```
static CPUReadMemoryFunc * const calculator_readfn[] = {
   calculator_read,
   calculator_read,
   calculator_read
};

static CPUWriteMemoryFunc * const calculator_writefn[] = {
   calculator_write,
   calculator_write,
   calculator_write
};
static int calculator_init(SysBusDevice *dev, const unsigned char *id)
{
    int iomemtype;
    calculator_state *s = FROM_SYSBUS(calculator_state, dev);

    iomemtype = cpu_register_io_memory(calculator_readfn,
                                       calculator_writefn, s,
                                       DEVICE_NATIVE_ENDIAN);

    sysbus_init_mmio(dev, 0x1000,iomemtype);
    sysbus_init_irq(dev, &s->irq);
    s->id = id;

    operator    = 0;
    operand_1   = 0;
    operand_2   = 0;
    go          = 0;
    result      = 0;

    printf("Calculator Hardware Initialization! \n");

    return 0;
}

static int calculator_init_arm(SysBusDevice *dev)
{
    return calculator_init(dev, calculator_id_arm);
}

static SysBusDeviceInfo calculator_hw_info = {
    .init = calculator_init_arm,
    .qdev.name  = "calculator",
    .qdev.size  = sizeof(calculator_state),
    .qdev.vmsd = &vmstate_calculator,
};

static void calculator_register_devices(void)
{
    sysbus_register_withprop(&calculator_hw_info);
}

device_init(calculator_register_devices)
```

Design A Virtual Hardware (3/5)

- 初始函式實作完成之後，我們必須要讓QEMU知道此硬體的存在，讓其啟動時可以將此虛擬硬體初始化。
- 上頁圖中，structure calculator_hw_info即是向QEMU宣告此虛擬硬體的相關起始函式與名稱。
- 實作QEMU所要求的增設虛擬硬體必要function
  - calculator_register_devices
    - sysbus_register_withprop(&hw_info)
  - calculator_init_arm
    - 將初始函式指向我們一開始設計的初始函式並與前面向QEMU註冊來的device結合起來。
  - device_init
    - 整個虛擬硬體初始化的源頭
- 由於我們預計是在realview EB中增加，所以必須在hw/realview.c中新增我們的虛擬硬體
  - sysbus_create_simple("calculator", 0x80000000, pic[30]);
  - 指定虛擬硬體的base為0x80000000，且使用IRQ阜號30

Design A Virtual Hardware (4/5)

- Read Function
  - 前面有提過，CPU只要讀取base+0x1000之間的位址就會來執行此函式。
  - 我們採用switch來做一個簡單的decoder。
  - 由於我們是以ARM為開發環境，所以ARM Linux在註冊device driver時會先確認硬體編號，此硬體編碼設定在0XFE0~0XFFC。
  - 我們定義在calculator_id_arm[8]中，並且當CPU要求這段位址時，我們將id送回去。ARM device的ID末四位元組一樣。
  - 在我們設計的calculator中，我們設定成應用程式只會讀取最後的結果，故除了ID與result的address之外，皆是不可讀取的位址。

```
static const unsigned char calculator_id_arm[8] =
  { 0x11, 0x10, 0x88, 0x08, 0x0d, 0xf0, 0x05, 0xb1 };

static uint32_t calculator_read(void *opaque, target_phys_addr_t offset)
{
    printf("Calculator read from %d!\n", offset);
    switch(offset & 0xFFF){
    case 0XFE0:
        return calculator_id_arm[0];
        break;
    case 0XFE4:
        return calculator_id_arm[1];
        break;
    case 0XFE8:
        return calculator_id_arm[2];
        break;
    case 0XFEC:
        return calculator_id_arm[3];
        break;
    case 0XFF0:
        return calculator_id_arm[4];
        break;
    case 0XFF4:
        return calculator_id_arm[5];
        break;
    case 0XFF8:
        return calculator_id_arm[6];
        break;
    case 0XFFC:
        return calculator_id_arm[7];
        break;
    default:
        if((offset & 0xFFF) == 0x10)
            return result;
        else{
            printf(" non-readable address!!!\n");
            return 0;
        }
    }
}
```

Design A Virtual Hardware (5/5)

- Write Function
  - 與read function類似，但是input除了offset外還多了要寫入的value。
  - 我們一樣採用switch設計decoder，透過offset的判斷，我們依照一開始所訂下的register和memory位址來寫入value。
  - 這裡還多處理了interrupt的部分，我們可看到當go被設定為1時，write function會去觸發calculate並依據operator的定義完成對應的計算。計算完成之後將interrupt拉起。
  - CPU接收到interrupt之後會觸發ISR，我們設定ISR處理完成後會寫值到0X100的register，此時write function再將IRQ拉下以完成中斷處理。

```
static void calculator_write(void *opaque, target_phys_addr_t offset, uint32_t value)
{
    calculator_state *s = (calculator_state *)opaque;

    printf("Calculator write %d to %d!\n", value, offset);

    switch(offset & 0xFFF){
    case 0x0:
        go = value;
        if(value != 0)
            calculate(s);
        break;
    case 0x4:
        operator = value;
        break;
    case 0x8:
        operand_1 = value;
        break;
    case 0xC:
        operand_2 = value;
        break;
    case 0x100:
        qemu_irq_lower(s->irq);
        printf("ISR is triggered!!!\n");
        break;
    default:
        printf("invalid address!!!\n");
    }
}
```

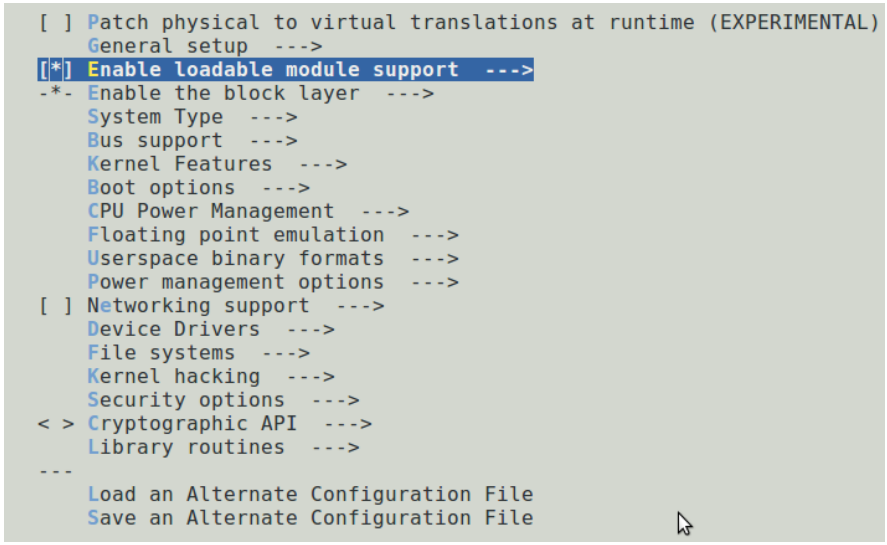
```
static void calculate(calculator_state *s)
{
    switch(operator){
    case 1:
        result = operand_1 + operand_2;
        break;
    case 2:
        result = operand_1 - operand_2;
        break;
    case 3:
        result = operand_1 * operand_2;
        break;
    case 4:
        if(operand_2 != 0)
            result = operand_1 / operand_2;
        else{
            result = 0;
            printf("invalid operand_2!!!\n");
        }
        break;
    default:
        printf("invalid operator!!!\n");
    }
    qemu_irq_raise(s->irq);
    go = 0;
}
```

Compile The Virtual Hardware

- QEMU所模擬的虛擬硬體都會放在hw資料夾之下，所以我們將設計好的虛擬硬體─calculator.c放到hw資料夾下。
- 為了讓QEMU知道必須編譯此虛擬硬體，在qemu-0.15.1-cal資料夾下有個Makefile.target檔案，找到obj-arm-y的群組，加入”obj-arm-y += calculator.o”，因為我們是針對arm系統平台撰寫此虛擬硬體。
- 依照LAB1的方法重新建構qemu，所得到的qemu-system-arm執行檔就有模擬calculator虛擬硬體的功能
- 之後我們將會利用此執行檔來進行純QEMU的全系統模擬。

Linux Device Driver - Setting

- Linux device driver和Linux kernel以及所使用的cross compiler具有強烈的相關性，為了正確且順利的開發驅動程式，我們將採用Linux-2.6.38的kernel搭配arm-none-linux-gnueabi-gcc 4.5.2。
- 由於我們所設計的driver為module類型，因此我們重新對kernel做設定與編譯。
- 設定Enable loadable module support開啟後重編kernel。



Linux Device Driver - Overview

- 驅動程式必須依照其所對應的硬體做設計，因此我們配合先前所設計的calculator，採取一樣的register位址以及effect value。
- 我們採用char device來設計calculator的驅動程式。和設計虛擬硬體時一樣，device driver也必須實現basic function。
- Linux在處理硬體存取時，採用類似檔案存取的方式來完成，因此我我們就必須完成讀寫檔案的basic function，定義如下圖。
- 裡面較特殊的是unlocked_ioctl，其被用來對特定register位址寫入特定值，因此從R/W function中獨立出來。
- 除了這五個針對file的處理函示之外，還必須實作init與exit兩個function，來處理driver的啟動與結束

```
static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .unlocked_ioctl = device_unlocked_ioctl,
    .open = device_open,
    .release = device_release
};
```

Linux Device Driver - Initial Function

- Initial function
  - 此函式處理所有driver初始化的工作，我們的driver是採用char device driver完成
  - 此函示定義了driver掛載上Linux後的識別碼包含Major與Minor number
  - 我們實作了allocate_memory_region來向kernel要求記憶體空間，ioremap會將實體空間映射至虛擬記憶體空間。
  - 最後是註冊IRQ的function: request_irq，必須注意Linux kernel所看到的IRQ阜號分為share和private，各有32個，而我們只使用share，因此在QEMU設定pic[30]，Linux看到的卻是30+32 = 62。這裡所有函式的input我們都採用macro define或structure替代，目的是為了良好的移植性。

```
static int __init cal_init(void)
{
    dev_t dev = 0;
    int result;

    /* register char device */
    if(CAL_MAJOR){
        dev = MKDEV(cal_major, cal_minor);
        result = register_chrdev_region( dev, 1, DRIVER_NAME);
    }
    else{
        result = alloc_chrdev_region( &dev, cal_minor, 1, DRIVER_NAME);
        cal_major = MAJOR(dev);
    }
    if(result < 0) {
        printk(KERN_ERR DRIVER_NAME "Can't allocate major number %d for %s\n", cal_major, DEVICE_NAME);
        return -EAGAIN;
    }
    cdev_init( &cal_cdev, &cal_fops);
    result = cdev_add( &cal_cdev, dev, 1);
    if(result < 0) {
        printk(KERN_ERR DRIVER_NAME "Can't add device %d\n", dev);
        return -EAGAIN;
    }

    /* allocate memory */
    allocate_memory_region();

    /* request irq and bind ISR*/
    result = request_irq(IRQ_VIC_BRIDGE+PIC_BASE, (void *)cal_interrupt, IRQF_DISABLED, DEVICE_NAME, NULL);
    if(result){
        printk(KERN_ERR DRIVER_NAME "Can't request IRQ\n");
    }

    return 0;
}
```

```
static void allocate_memory_region (void)
{
    base = ioremap_nocache(CALCULATOR_BASE , 0x100);
    printk("Physical address: %08X\n", CALCULATOR_BASE);
    printk("After ioremap: %08X\n", (int)base);
}
```

Linux Device Driver - ISR

- 註冊IRQ的function中，除了通知Kernel皐號之外，還必須告知Interrupt Service Routine(ISR)的函式，我們的例子為fpga_interrupt。
- 顧名思義，當CPU接受到IRQ皐號為62的時候，就會去執行由我們所定義的ISR，在此函式中，我們先將0寫入0x100的位址，正好就是我們先前虛擬硬體定義的將IRQ放掉所必須做的對應動作，之後我們用wake_up_interrupt將process從waiting queue釋放出來。這部份我們將會在後面與把process擺入waiting queue中一起來看，這裡我們只要懂ISR做了甚麼即可。
- 當接收到IRQ時，就知道硬體已經完成工作或是需要軟體執行相對應的行為讓硬體可以繼續作業下去。

```
static irqreturn_t fpga_interrupt (int irq, void *dev_id, struct pt_regs *regs)
{
    //printk("received interrupt from SC_LINK\n");
    if (irq == IRQ_VIC_BRIDGE+PIC_BASE) {
        printk("received interrupt from Caculator HW\n");
        writel(0, base+0x100);
        printk("Release interrupt \n");
        flag = 1;
        wake_up_interruptible(&wq);
    }
    return IRQ_HANDLED;
}
```

Linux Device Driver – Read/ Write

- 先前我們提到，kernel將device視為檔案，因此我們必須實作讀檔寫檔等基本操作，那麼應用程式就能用fread, fwrite去操作device。
- 這裡最重要的兩個function為：copy_to_user及copy_from_user。
- 因為關係到將data從user space轉換到kernel space，因此我們需要這兩隻函式幫我們把一整串data轉移到driver，driver在針對這整筆資料做相對應的動作。
- 從下兩張圖看到，device_read透過readl將值讀回，並透過copy_to_user送回應用程式(注意不是用return)，而device_write則以copy_from_user得到應用程式想寫入device的值，再以writel寫進device。

```
static ssize_t device_read(struct file *filp, char *buff,
        size_t len, loff_t *off)
{
    int i;
    //  printk("<SC_DRV>read len: %d\n", len);

    my_buffer[0] = readl(base + result_offset);

    //unsigned long copy_to_user (void __user * to, const void * from, unsigned long n);
    i=copy_to_user(buff, my_buffer, 0x4);

    if (*off == 0) {
        *off += 0x4;
        return 0x4;
    }
    else {
        return 0;
    }
}
```

```
static ssize_t device_write(struct file *filp, const char *buff,
        size_t len, loff_t *off)
{
    int i;
//  printk("<SC_DRV>write len: %d\n", len);
    if (len > 0x8)
        len = 0x8;

    i=copy_from_user(my_buffer, buff, len);

    for(i = 0; i < len/4; i++) {
        writel(my_buffer[i], base + operand_offset +(i*4)); // write all 8 values
    }

    //printk("<SC_DRV>\n%d bytes written\n", len);
    return len;
}
```

Linux Device Driver – IOCTL(1/2)

- 驅動程式設計，除了上述的對記憶體進行read/write之外，最重要的就是存取register file讓硬體可以做對應的工作，ioctl即是負責此部分的function。
- IOCTL必須提供應用程式一連串的commands,並且當應用程式使用這些command的時候可以做出相對應的動作。
- 讓我們回憶設計calculator時，我們設定operator的offset位址在0x4，而啟動的offset位址則在0x0
- 而我們設定提供給應用程式的command有兩個，一個是設定operator的“OP_SET”定義為1，另一個則是啟動calculator的“GO”定義為5。

Linux Device Driver – IOCTL(2/2)

- Ioctl除了file指標之外，還會有兩個input，分別就是command與一項value，通常此value是要寫入的值，端看如何設計。
- 這邊我們還結合了waiting queue的概念，因為當應用程式使用“GO”指令要求calculator進行運算時，我們預設應用程式應該是處在等待運算結果回來的狀態，因此將此程序塞入waiting queue，讓出CPU給其他程序得以進行。
- 而OP_SET則是單純進行operator設定，因此利用writel將value寫入operator就完成了。

```
static long device_unlocked_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    //printk("<SC_DRV> IOCTL!!cmd = %d, arg = %d\n", cmd, (int)arg);
    switch(cmd)
    {
    case OP_SET:
        writel((unsigned int)arg, base + operator_offset);
        break;
    case GO:
        writel((unsigned int)arg, base);
        wait_event_interruptible(wq, flag != 0);
        flag = 0;
        break;
    default:
        printk("Unknown COMMAND!!!\n");
    }
    return 0;
}
```

Linux Device Driver – Waiting Queue

- 在ISR與IOCTL中我們都提到了waiting queue的概念，也就是當應用程式必須等待硬體回應時，我們希望他先進入等待佇列而不是採用busy waiting。
- Linux Kernel本身就有提供此功能,標頭檔為 <linux/wait.h>
- 首先必須先宣告等待佇列以及喚醒條件旗標
  - static DECLARE_WAIT_QUEUE_HEAD(wq);
  - static int flag = 0;
- 當我們需要應用程式進去等待佇列時，採用
  - wait_event_interruptible(wq, flag!=0);
  - 則應用程式會進入wq等待被喚醒，且必要條件為flag!=0時。
- 而要喚醒程式時，
  - 會先設定喚醒條件旗標: flag = 1;
  - 在將程序從wq喚醒: wake_up_interruptible(&wq);

Compile Device Driver

- 我們先新增一個資料夾calculator-driver-app，並將設計完成的cal_drv.c放入此資料夾。
- Makefile
  - 由於device driver與kernel相關，因此必須設定kernel資料夾位址
    - KERNELDIR ?= /home/{user-name}/linux-kernel/linux-2.6.38
    - PWD := $(shell pwd)
  - 設定編譯環境
    - ARCH=arm
    - CROSS_COMPILE=arm-none-linux-gnueabi-
  - Compile
    - $(MAKE) ARCH=$(ARCH) CROSS_COMPILE=$(CROSS_COMPILE) -C $(KERNELDIR) M=$(PWD) modules
- 之後下make指令就可以成功編譯我們的device driver

Application

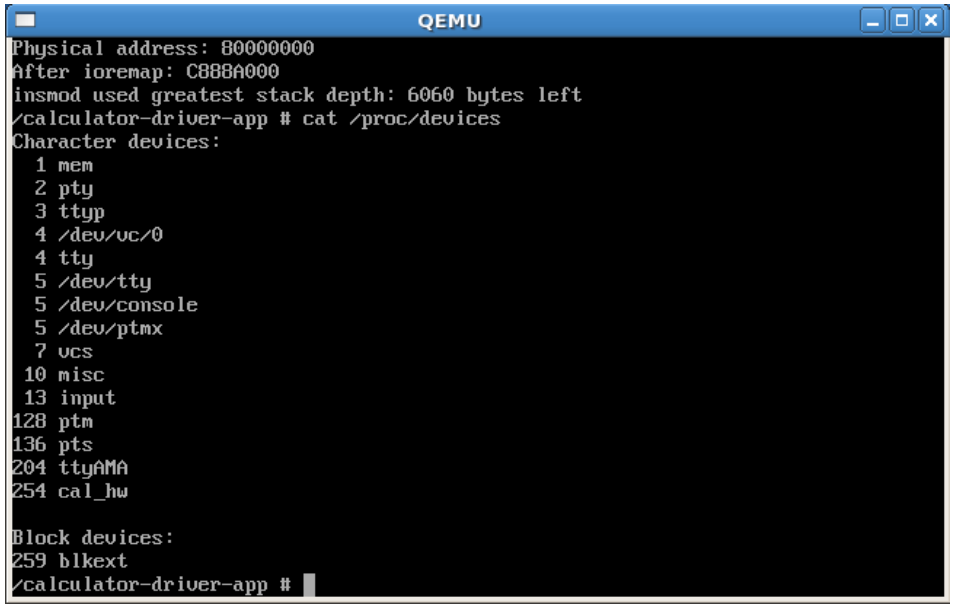
- 完成driver之後，應用程式就可以透過此driver去使用我們所設計的calculator。此應用程式我們亦放在calculator-driver-app資料夾中
- 開啟device
  - fin = open("/dev/calculator", O_RDWR);
  - 其中/dev/calculator是我們在linux中所新增的node，後續會提到
- 寫入operand
  - write(fin, operand, 8);
- 寫入operator
  - ioctl(fin, OP_SET, ADD);
- 通知硬體執行
  - ioctl(fin, GO, START);
- 讀回Result
  - read(fin, temp, 4);
- 最後我們一樣透過arm-none-linux-gnueabi-gcc編譯
  - `arm-none-linux-gnueabi-gcc calculate.c –o calculate`
  - 執行檔為`calculate`

Simulation Flow

- 首先我們先將calculator-driver-app資料夾放入先前busybox的_install中，並且產出相對應的initrd。
- Linux kernel則是2.6.38版本編譯完成的zImage。
- 之後就可以利用qemu-system-arm來進行全系統模擬。
  - ./qemu-system-arm -M realview-eb -kernel zImage-2.6.38 -initrd initrd-2.6.38.gz -cpu arm1136
- 開機完成後必須掛載device driver
  - #cd calculator-driver-app
  - #insmod cal_drv.ko

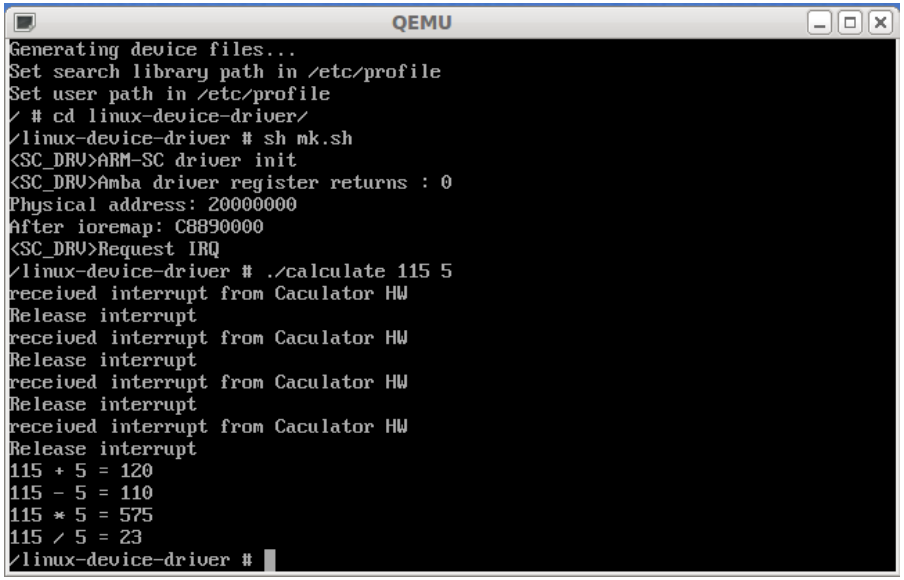
Simulation Flow

- 接著我們必須察看Linux kernel設定我們driver的掛載編號
  - # cat /proc/devices
  - 從圖我們看到掛載編號是254(因為我們的驅動程式採靜態登記法)。
- 之後我們就要替此device宣告一個node，應用程式要讀寫的時候是對此node做讀寫
  - # mknod /dev/calculator c 254 0
  - c表char device, 254為剛剛查到的編號(major number), 0為minor number。



Simulation Flow

- 此時應用程式開啟/dev/calculator才能成功並且可以順利操作driver來控制calculator。
- 執行應用程式並觀察回傳值是否正確
  - # ./calculate 115 5
  - 在範例程式中，後面接的參數為operand並且執行了加減乘除四項運算後把結果一次展現，途中會發現執行了四次ISR，以及觀察應用程式與硬體的互動為我們所預期。



Conclusion

- 在前面的LAB，我們以C語言設計了一套簡單的硬體calculator並且採用QEMU模擬。
- 為了讓應用程式可以順利使用此虛擬硬體，我們亦設計了對應的驅動程式。
- 但這些僅止於對硬體的行為(behavior)模擬，因為虛擬硬體完全不具有時間的概念，因此此種全系統模擬方法並無法滿足有時序的模擬硬體(cycle accurate)。
- 然而若將所有平台上的硬體(包含CPU)皆採用硬體模擬語言(HDL)開發，那其模擬速度將無法在可接受時間內啟動Linux OS。因此我們之後將會結合QEMU以及SystemC做分部模擬，讓QEMU全力執行Linux，而SystemC只模擬開發硬體(在此就是calculator)，冀望達到開發中硬體具有時間概念，並且可以做全系統模擬來發展應用程式與驅動程式，以此達到更高度的軟硬體整合。

Conclusion

- 為了使用上方便，我們可以建立靜態連結，如此一來不管是qemu重編或是linux kernel及initrd重編，都可以在我們建立靜態連結的資料夾中直接使用最新的檔案。
  - #mkdir ~/qemu-test
  - #cd ~/qemu-test
  - #ln -s {qemu-path}/arm-softmmu/qemu-system-arm .
  - #ln -s {busybox-path}/initrd.gz .
  - #ln -s {linux-kernel-path}/arch/arm/boot/zImage .
  - cd ~/qemu-test
  - ./qemu-system-arm -M realview-eb -kernel zImage -initrd initrd.gz -cpu arm1136