Embedded System Design
Practice 6

TaeWook Kim & SeokHyun Hong
Hanyang University

TEST UART & TIMER SETTINGS

Preparation for the VPOS kernel porting

1. Implement Startup code
2. UART Settings
3. TIMER Settings
4. Implement Hardware Interrupt Handler
   (1) UART Interrupt
   (2) Timer Interrupt
5. Implement Software Interrupt Entering/Leaving Routine
6. Kernel compile + load kernel image in RAM

Contents

1. Compile the kernel
2. Test UART Settings
3. Test Timer Settings

COMPILE THE KERNEL

Edit Linker Script

- **Edit Linker Script**
  - >> vi vpos/hal/cpu/vpos_kernel-ld-script
  - Edit first line of 'SECTIONS' to ". = 0x20008000;"

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(vh_VPOS_STARTUP)
SECTIONS
{
        . = 0x20008000;

        . = ALIGN(4);
        .text : { *(.text) }

        . = ALIGN(4);
        .rodata : { *(.rodata) }

        . = ALIGN(4);
        .data : { *(.data) }

        . = ALIGN(4);
        .got : { *(.got) }

        . = ALIGN(4);
        .bss : { *(.bss) }

}
```

Kernel Compile

- **Copy image to /tftpboot after kernel compilation**
  1. make clean
  2. make
  3. cp images/vpos.bin /tftpboot

- **Check vpos.bin has been copied**
  1. ls /tftpboot

```
root@ubuntu:~/embedded/vpos/kernel# ls /tftpboot/
u-boot.bin  uBoot-S5PC100.bin  vpos.bin
```

TEST UART SETTINGS

Answer Code

```
void vh_serial_init(void)
{
    int i;
    // UART 1 Setting
    vh_GPA0CON = vh_vSERIAL_CON;
    vh_GPA0PUD = vh_vSERIAL_PUD;

    // UART 1 Configuration
    vh_ULCON    =
    vh_UCON     =
    vh_UFCON    =
    vh_UINTM1   =
    vh_UINTP1   =
    vh_UBRDIV   = ((66000000 / (115200 * 16)) - 1);

    push_idx = 0;
    pop_idx = 0;
    for(i=0;i<SERIAL_BUFF_SIZE;i++) serial_buff[i] = '\0';
}
```

```
vh_ULCON    = 0x3;
vh_UCON     = 0x245;
vh_UFCON    = 0xc7;
vh_UINTM1   = 0xe;
vh_UINTP1   = 0x1f;
vh_UBRDIV   = ((66000000 / (115200 * 16)) - 1);
```

vpos/hal/io/serial.c

UART Communication

- **Run the minicom and upload the vpos kernel**

```
$ tftp c0008000 vpos.bin
$ bootm c0008000
```

```
2017123456 # bootm c0008000
Boot with zImage

Starting kernel ...


*******************************************************************************
*  QURIX version 3.0    xx/10/2012                                            *
*******************************************************************************
vk_swi_classifier switch up

Race condition value = 0

Shell>
```

You can see the kernel output through UART

UART Communication

- **Type “ls” to see if we can send the string**

```
Shell> ls
```

```
2017123456 # bootm c0008000
Boot with zImage

Starting kernel ...


**********************************************
*  QURIX version 3.0    xx/10/2012
**********************************************
vk_swi_classifier switch up

Race condition value = 0

Shell>ls
Data abort exception. Location [0x20009558]
```

It’s not an error because we didn’t implement interrupt

Error Case

- **If the output suspends after “Starting kernel…”, you should recheck UART configuration**

```
2017123456 # bootm c0008000
Boot with zImage

Starting kernel ...
```

If the UART is misconfigured,
we cannot see the kernel’s output through UART

TEST TIMER SETTINGS

Answer Code

```
#define vh_TCFG0            (*(volatile unsigned *)0xea000000)
#define vh_TCFG1            (*(volatile unsigned *)0xea000004)
#define vh_TCON             (*(volatile unsigned *)0xea000008)
#define vh_TINT_CSTAT       (*(volatile unsigned *)0xea000044)
#define vh_TCNTB4           (*(volatile unsigned *)0xea00003c)
#define vh_TCNTO4           (*(volatile unsigned *)0xea000040)
```

vpos/hal/include/vh_io_hal.h

```
// Initialize Timer 4
vh_TINT_CSTAT   &= 0xffffffef;
vh_TINT_CSTAT   |= 0xfffffdff;
vh_TCFG0        |= 0x0000ff00;
vh_TCFG1        |= 0x00040000;
vh_TCNTB4       = 0x3e80;
vh_TINT_CSTAT   |= 0x00000010;
vh_TCNTO4       = 0xea000040;
```

vpos/hal/io/timer.c

Test the Timer using TCNTO0

- **Add Timer Test Function**
  - vpos/kernel/kernel_start.c

- **Timer4 Enable code (Start Timer4)**
  1. Clears the Timer4 field of TCON to 0 (bits 20 to 22)
  2. Auto Reload on and Manual Update (sets bit 21 and bit 22 to 1)
  3. Clear Timer4 part of TCON to all 0
  4. Auto Reload on and Timer4 Start (sets bit 20 and bit 22 to 1)

- **Check the value of the TCNTO4 register several times in the loop statement**
  - TCNTO4 value continues to change when timer works

Test the Timer using TCNTO0

- **Define Timer Test Function**
  - vpos/kernel/kernel_start.c
  - Define above the VPOS_kernel_main()

- **Code**

```
void TIMER_test(void)
{
        int timebuffer[20];
        int i;

        // Timer4 Starts
        vh_TCON = (vh_TCON & ~0x700000) | 0x600000;
        vh_TCON = (vh_TCON & ~0x700000) | 0x500000;

        for (i = 0; i < 20; i++) {
                timebuffer[i] = vh_TCNTO4;
                printk("timebuffer[%d] : %d\n", i, timebuffer[i]);
        }
}
```

Test the Timer using TCNTO0

- **Call the Timer test function**
  - Call the test function from the VPOS_kernel_main () function

- **Code**

```
void VPOS_kernel_main( void )
{
        pthread_t p_thread, p_thread_0, p_thread_1, p_thread_2;

        /* static and global variable initialization */
        vk_scheduler_unlock();
        init_thread_id();
        init_thread_pointer();
        vh_user_mode = USER_MODE;
        vk_init_kdata_struct();

        vk_machine_init();
        set_interrupt();


        printk("%s\n%s\n%s\n", top_line, version, bottom_line);

        // Timer4 Test
        TIMER_test();

        /* initialization for thread */
        race_var = 0;
        pthread_create(&p_thread, NULL, VPOS_SHELL, (void *)NULL);
        //pthread_create(&p_thread_0, NULL, race_ex_1, (void *)NULL);
        //pthread_create(&p_thread_1, NULL, race_ex_0, (void *)NULL);
        //pthread_create(&p_thread_2, NULL, race_ex_2, (void *)NULL);

        VPOS_start();
```

Test the Timer using TCNTO0

- **Run the minicom and upload the vpos kernel**

```
$ tftp c0008000 vpos.bin
$ bootm c0008000
```

Test the Timer using TCNTO0

- The counter decreases as time passes while executing
- The counter starts from the reload value (16000)

```
*****************************************************************************
*  QURIX version 3.0    xx/10/2012                                          *
*****************************************************************************
timebuffer[0] : 16000
timebuffer[1] : 15968
timebuffer[2] : 15935
timebuffer[3] : 15903
timebuffer[4] : 15870
timebuffer[5] : 15837
timebuffer[6] : 15805
timebuffer[7] : 15772
timebuffer[8] : 15739
timebuffer[9] : 15707
timebuffer[10] : 15674
timebuffer[11] : 15640
timebuffer[12] : 15606
timebuffer[13] : 15571
timebuffer[14] : 15537
timebuffer[15] : 15503
timebuffer[16] : 15469
timebuffer[17] : 15435
timebuffer[18] : 15401
timebuffer[19] : 15366
vk_swi_classifier switch up
```

UART INTERRUPT

Preparation for the VPOS kernel porting

1. Implement Startup code
2. UART Settings
3. TIMER Settings
4. Implement Hardware Interrupt Handler
   (1) UART Interrupt
   ~~(2) Timer Interrupt~~
5. Implement Software Interrupt Entering/Leaving Routine
6. Kernel compile + load kernel image in RAM

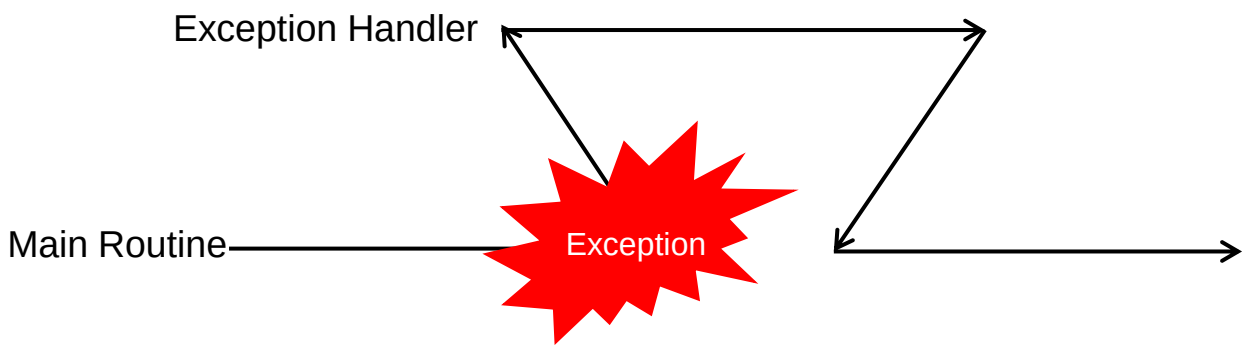
Contents

1. Exception
2. Interrupt
3. Vectored Interrupt Controller
4. UART Interrupt Handler
5. Interrupt Entering & leaving Routine

EXCEPTION

Exception

- **Exception?**
  - The state in which the sequential execution of commands should be stopped
  - The state in which an exception or interrupt occurred
    - Undefined instruction
    - Memory access failure
    - Software interrupt
    - External hardware interrupt
    - …
  - Move to the Exception Handler from the current routine



Types of Exception

Exception	Mode	Vector table offset
Reset	SVC	+0x00
Undefined Instruction	UND	+0x04
Software Interrupt(SWI)	SVC	+0x08
Prefetch Abort	ABT	+0x0c
Data Abort	ABT	+0x10
Not assigned	-	+0x14
IRQ	IRQ	+0x18
FIQ	FIQ	+0x1c

Entering Exception Handler

- **When an exception occurs, the CPU automatically,**
  1. Store CPSR value in SPSR in exception mode
  2. Save the PC value to the Link Register(lr) in exception mode
  3. Change mode bit of CPSR to enter corresponding exception mode
  4. Execute the exception handler by storing the address of the exception handler on the PC
     - Vector table base address + Vector table offset
     - Vector table base address stored in ARM Coprocessor

```
vh_vector_start:
        b       vh_VPOS_reset
        b       vk_undef
        b       vh_software_interrupt
        b       vh_pabort
        b       vk_dabort
        b       vk_not_used_handler
        b       vh_irq
        b       vk_fiq_handler
```

Vector Table

```
// change vector table base address (0x20008044)
ldr     r0, =vh_vector_base
mcr     p15, 0, r0, c12, c0, 0
```

Save Vector base address

Exception Handler

- **Exception Handler**
  - Find the reason of the exception and resolve the exception
  - If the exception is an interrupt, it must process the interrupt and return to the main routine.

- **How can program return to the original routine after completion of exception handling?**
  - movs pc, lr
    - Save the value of Link Register to PC
      - Link register value needs additional calculation according to exception

INTERRUPT

Polling vs. Interrupt

- **How do I know if I can use the device now?**
  - Polling & Interrupt

- **Polling**
  - The CPU checks the status of the device at regular intervals
  - Check the status bit value

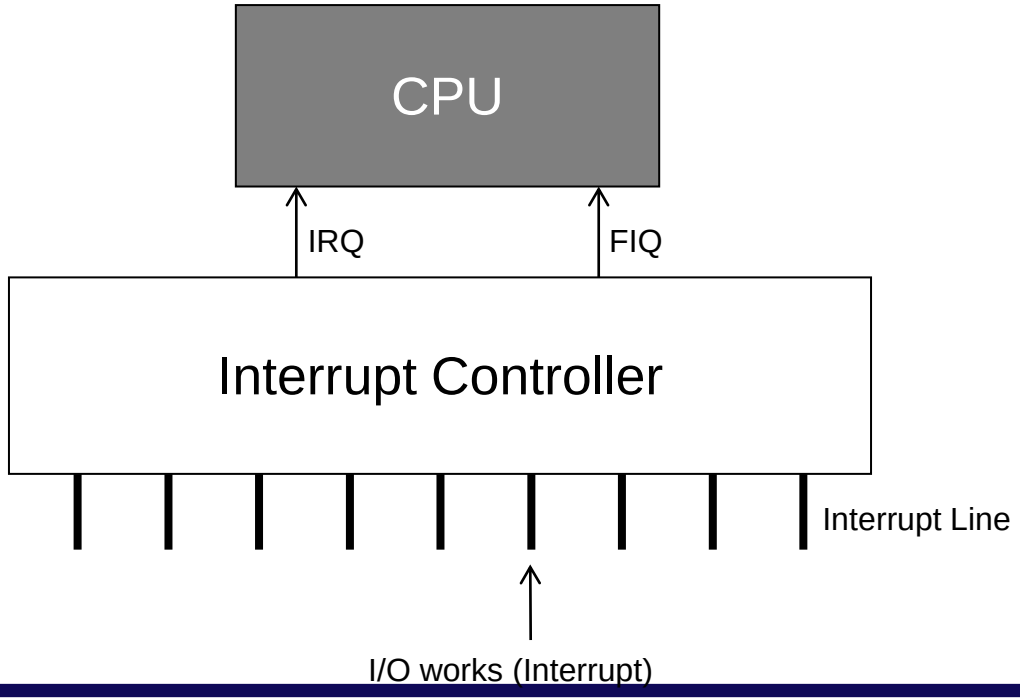
- **Interrupt**
  - Notify the CPU when the device can send data or receive data from the outside
  - CPU can do other things until interrupt occurs

Interrupt

- **ARM's Interrupt**
  - IRQ : Normal Interrupt
    - General purpose interrupt
    - Lower priority and higher priority delay time than FIQ
  - FIQ : Fast Interrupt
    - Used for interrupt sources that require fast response time
    - Use for specific applications only
  - SWI : Software Interrupt
    - Software interrupts to enter privileged mode
    - Used to call kernel functions

Interrupt Controller

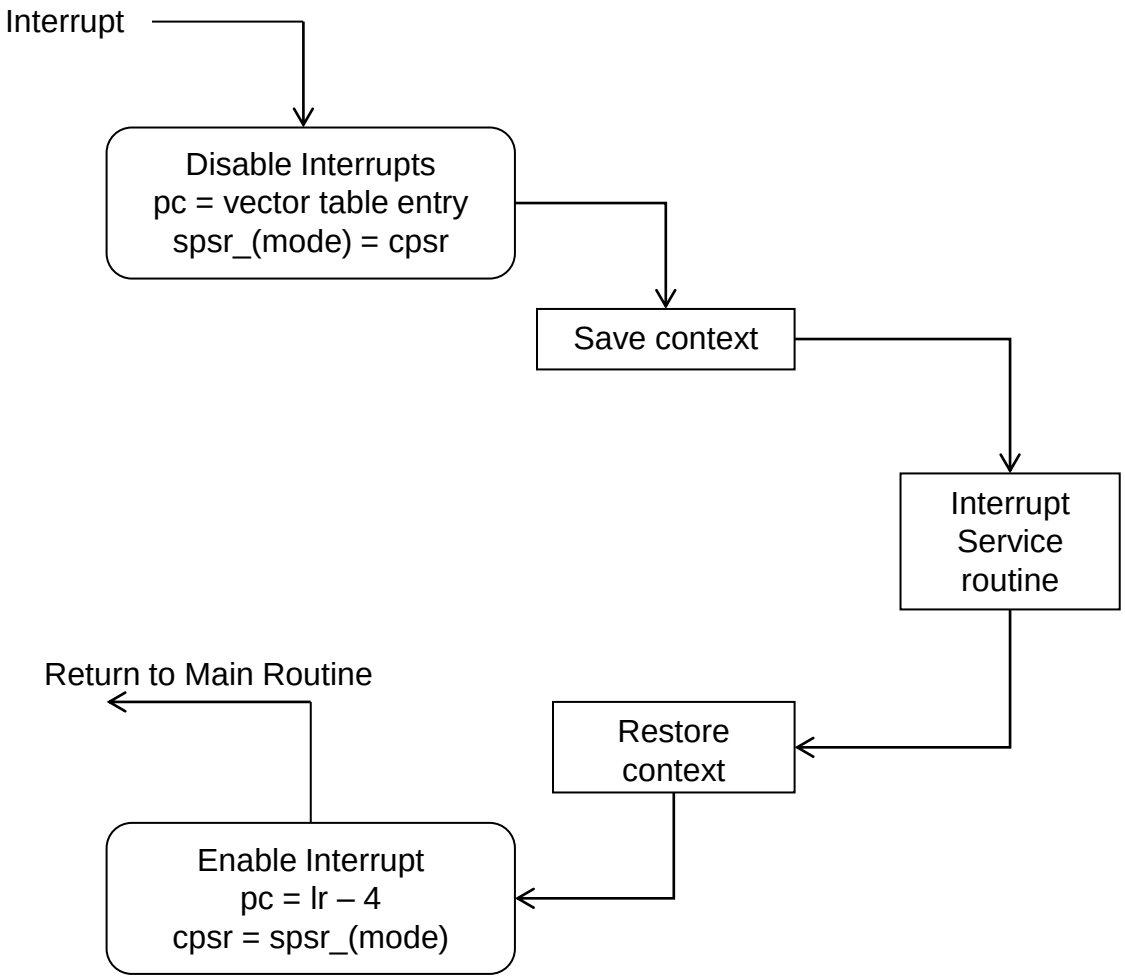
- **Definition**
  - A controller for connecting multiple external interrupts to one of the CPU's interrupt request ports



Interrupt Service Routine

- **Definition**
  - A routine for processing an interrupt when an interrupt occurs
    - Timer interrupt → Call scheduler
    - UART interrupt → Receiving data transmitted from outside

How to Process Interrupt



How to Process Interrupt

1. Interrupts occur on the device

2. The ARM CPU changes to IRQ mode (the CPU handles it automatically)

- Disable interrupt
- Store CPSR value in SPSR
- Go to vector table's interrupt handler

3. Save context

- Storing r0-r12, sp, lr values in the previous mode(User mode) on the stack (stmfd)

4. Locate the interrupt source in the interrupt handler and run its interrupt service routine (ISR)

5. Execute interrupt service routine to process interrupts

How to Process Interrupt

6. Restore context

- Loads the r0-r12, sp, lr values stored in the stack into each register in the previous mode (ldmfd)

7. Enable Interrupt

8. lr = lr - 4

- Pipeline
- Since the IRQ occurs after the current instruction is executed, the address to be returned must use the next instruction, that is, the value of lr-4, as the return address.

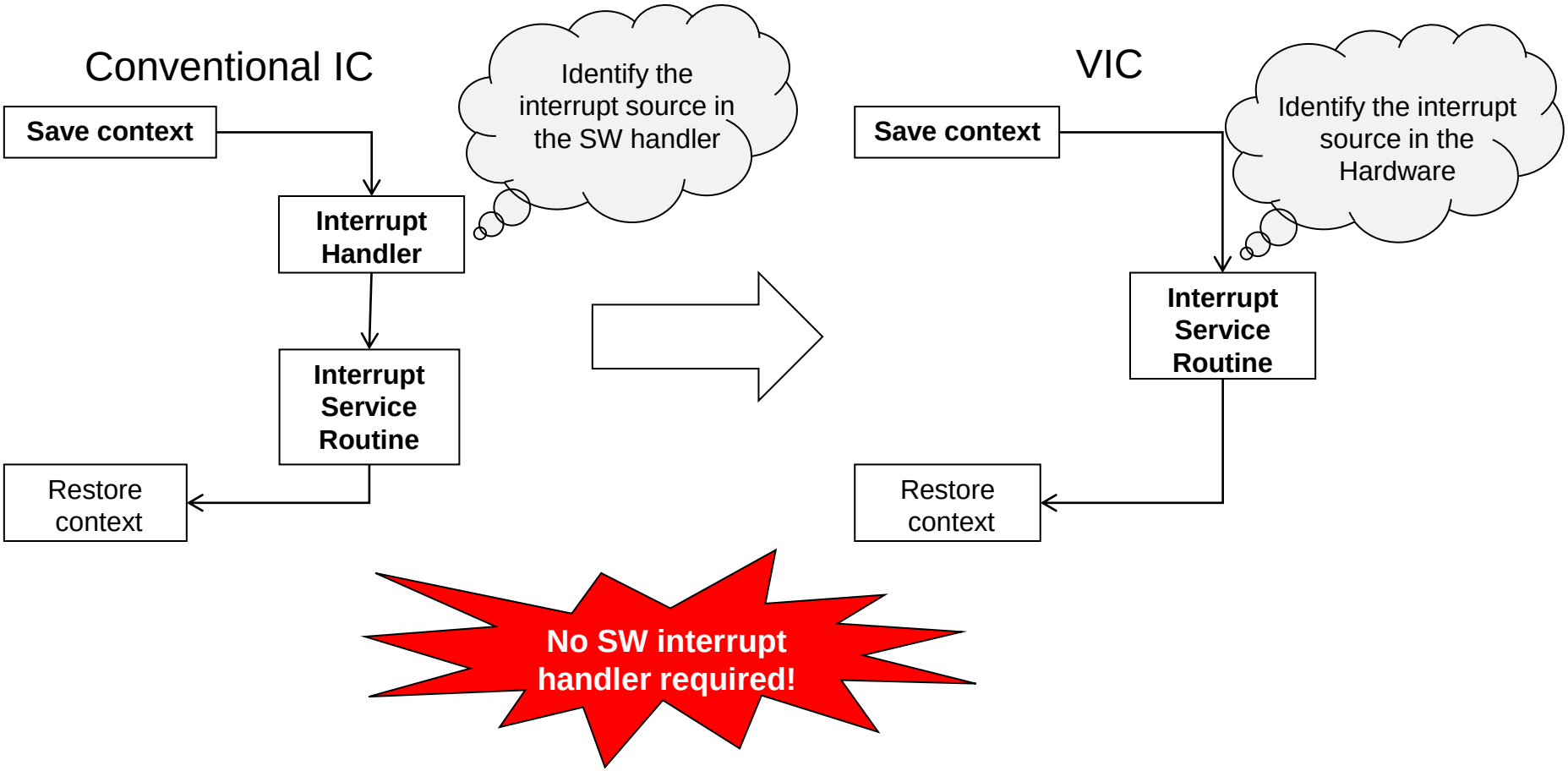
9. Return to original routine using 'movs pc, lr' command

VECTORED INTERRUPT CONTROLLER

Vectored Interrupt Controller

- **VIC (Vectored Interrupt Controller)**
  - Vector interrupt register set support
    - One register for each interrupt source
    - Each register stores the address of the interrupt service routine.
    - When an interrupt occurs, the controller automatically loads the service routine address of the interrupt into the VICADDRESS register
    - The CPU jumps to the interrupt service routine stored in the VICADDRESS register

Vectored Interrupt Controller



Vectored Interrupt Controller

- **Advantages**
  - Interrupt latency can be reduced
    - The other interrupt controller identifies the interrupt source in the software interrupt handler
    - No software interrupt handler required on VIC
      - Interrupt source can be known on hardware
      - Can run ISRs faster than other ICs

VIC of S5PC100

- **There are three VICs**
  - Supports 32 interrupt sources per VIC

- **Interrupt Source List**
  - See Datasheet Page. 360~362

VIC1

ARM, power, memory, Connectivity, Storage

50	**IrDA**	IrDA Interrupt
49	**SPI2**	SPI2 Interrupt
48	**SPI1**	SPI1 Interrupt
47	**SPI0**	SPI0 Interrupt
46	**I2C0**	I2C0 Interrupt
45	**UART3**	UART3 Interrupt
44	**UART2**	UART2 Interrupt
43	**UART1**	UART1 Interrupt
42	**UART0**	UART0 Interrupt
41	**CFC**	CFCON Interrupt
40	**NFC**	NFCON Interrupt

VIC total : 43rd
VIC 1 : 11th

Registers of VIC

- **Control registers**
  - VICINTSELECT
  - VICINTENABLE
  - VICINTENCLEAR
  - VICSWPRIORITYMASK
- **Status register**
  - VICIRQSTATUS
- **ISR address storage register**
  - VICVECTADDR[0-31]
  - VICADDRESS

VICINTSELECT

- **Register : VICINTSELECT**
  - Interrupt Select Register
    - Register to set Interrupt Source to IRQ/FIQ
    - One interrupt source corresponds to one bit

      ex) 0 interrupt source corresponds to bit 0

VICINTSELECT	Bit	Description	Reset Value
IntSelect	[31:0]	Selects interrupt type for interrupt request: 0 = IRQ interrupt 1 = FIQ interrupt There is one bit of the register for each interrupt source.	0x00000000

VICINTENABLE

- **Register : VICINTENABLE**
  - Interrupt Enable Register
    - Register to enable Interrupt Source
    - No function to disable

VICINTENABLE	Bit	Description	Reset Value
IntEnable	[31:0]	Enables the interrupt request lines, which allows the interrupts to reach the processor. Read: 0 = Disables Interrupt 1 = Enables Interrupt Use this register to enable interrupt. The VICINTENCLEAR Register must be used to disable the interrupt enable. Write: 0 = No effect 1 = Enables Interrupt. On reset, all interrupts are disabled. There is one bit of the register for each interrupt source.	0x00000000

VICINTENCLEAR

- **VICINTENCLEAR**
  - Interrupt Enabler Clear Register
    - A register that disables the interrupt source

VICINTENCLEAR	Bit	Description	Reset Value
IntEnable Clear	[31:0]	Clears corresponding bits in the VICINTENABLE Register: 0 = No effect 1 = Disables Interrupt in VICINTENABLE Register. There is one bit of the register for each interrupt source.	-

VICSWPRIORITYMASK

- **VICSWPRIORITYMASK**
  - Software Priority Mask Register
    - Registers that mask 16 priority levels of interrupts

VICSWPRIORITYMASK	Bit	Description	Reset Value
Reserved	[31:16]	Reserved, read as 0, do not modify	0x0
SWPriorityMask	[15:0]	Controls software masking of the 16 interrupt priority levels: 0 = Interrupt priority level is masked 1 = Interrupt priority level is not masked Each bit of the register is applied to each of the 16 interrupt priority levels.	0xFFFF

VICIRQSTATUS

- **Register : VICIRQSTATUS**
  - IRQ Status Register
    - Register indicating which interrupt occurred
    - When a specific interrupt occurs, the bit indicating the interrupt is set to 1

VICIRQSTATUS	Bit	Description	Reset Value
IRQStatus	[31:0]	Shows the status of the interrupts after masking by the VICINTENABLE and VICINTSELECT Registers: 0 = Interrupt is inactive 1 = Interrupt is active. There is one bit of the register for each interrupt source.	0x00000000

VICVECTADDR[0-31]

- **Register : VICVECTADDR[0-31]**
  - Vector Address Register
    - Registers that store the ISR address of each interrupt source
    - There are 32 VICVECTADDRs corresponding to 32 interrupt sources
      - There are 3 VICs in S5PC100, so there are 96 VICVECTADDR in total

VICVECTADDR[0-31]	Bit	Description	Reset Value
VectorAddr 0-31	[31:0]	Contains ISR vector addresses.	0x00000000

VICADDRESS

- **Register : VICADDRESS**
  - Vector Address Register
    - When the interrupt occurs, load the ISR address of the corresponding interrupt source
    - Controller automatically loads

  ex) Interrupt occurs at interrupt source 12,

  VICADDRESS ← VICVECTADDR[12]

VICADDRESS	Bit	Description	Reset Value
VectAddr	[31:0]	Contains the address of the currently active ISR, with reset value 0x00000000. A read of this register returns the address of the ISR and sets the current interrupt as being serviced. A read must be performed while there is an active interrupt. A write of any value to this register clears the current interrupt. A write must only be performed at the end of an interrupt service routine.	0x00000000

UART INTERRUPT CODE (C CODE)

VPOS_kernel_main()

- **Functions**
  - Initialize the VPOS kernel data structure
  - Initialize hardware such as serial device and timer
  - Enable interrupt
  - Print boot message
  - Create a shell thread
  - Enter scheduler calling VPOS_start routine

- **Source code location**
  - vpos/kernel/kernel.start.c

```
void VPOS_kernel_main( void )
{
        pthread_t p_thread, p_thread_0, p_thread_1, p_thread_2;

        /* static and global variable initialization */
        vk_scheduler_unlock();
        init_thread_id();
        init_thread_pointer();
        vh_user_mode = USER_MODE;
        vk_init_kdata_struct();

        vk_machine_init();
        set_interrupt();


        printk("%s₩n%s₩n%s₩n", top_line, version, bottom_line);

        /* initialization for thread */
        race_var = 0;
        pthread_create(&p_thread, NULL, VPOS_SHELL, (void *)NULL);
        //pthread_create(&p_thread_0, NULL, race_ex_1, (void *)NULL);
        //pthread_create(&p_thread_1, NULL, race_ex_0, (void *)NULL);
        //pthread_create(&p_thread_2, NULL, race_ex_2, (void *)NULL);

        VPOS_start();

        /* cannot reach here */
        printk("OS ERROR: VPOS_kernel_main( void )₩n");
        while(1){}
}
```

set_interrupt()

- **Location**
  - vpos/kernel/kernel_start.c

- **Add code**
  - Call vh_serial_irq_enable()

```
void set_interrupt(void)
{
        // interrupt setting
        vh_serial_irq_enable();
}
```

vh_serial_irq_enable()

- **Role**
  - Functions that enable UART1 Interrupt

- **Location**
  - vpos/hal/io/serial.c

vh_serial_irq_enable()

- **Order of execution**
  1. Save ISR address in VIC1VECTADDR11 register
     - Save vh_serial_interrupt_handler() function address
  2. Enable UART1 interrupt in VIC1INTENABLE register
     - Set bit 11 to 1
  3. Set the UART1 interrupt to IRQ in the VIC1INTSELECT register
     - Clear bit 11 to 0
  4. Mask all VIC1SWPRIORITYMASK register
     - Set all bits to 1

vh_serial_irq_enable()

- **Add code**

```
void vh_serial_irq_enable(void)
{
    /* enable UART1 Interrupt */
    vh_VIC1VECTADDR11 = (unsigned int)&vh_serial_interrupt_handler;
    vh_VIC1INTENABLE |= vh_VIC_UART1_bit;
    vh_VIC1INTSELECT &= ~vh_VIC_UART1_bit;
    vh_VIC1SWPRIORITYMASK = 0xffff;
}
```

vpos/hal/io/serial.c

vh_serial_irq_enable()

- **Add code**

```
/*************************************************************
    VIC - S5PC100
*************************************************************/
#define vh_VIC1INTENABLE            (*(volatile unsigned *)0xe4100010)
#define vh_VIC1INTSELECT            (*(volatile unsigned *)0xe410000c)
#define vh_VIC1INTENCLEAR           (*(volatile unsigned *)0xe4100014)
#define vh_VIC1VECTADDR11           (*(volatile unsigned *)0xe410012c)
#define vh_VIC1SWPRIORITYMASK       (*(volatile unsigned *)0xe4100024)
```

vpos/hal/include/vh_io_hal.h

vh_serial_interrupt_handler()

- **Description**
  - UART1 Interrupt Handler
  - Receives keyboard input and stores it in a buffer

- **Location**
  - vpos/hal/io/serial.c

vh_serial_interrupt_handler()

- **Order of execution**
  1. Store keyboard character data received in the URXH register in a buffer
     - Call vk_serial_push()
  2. Pending clear UART1 interrupt
     - Disable UART1 interrupt in VICINTENCLEAR register
       - Set bit 11 to 1
     - Enable UART1 interrupt in VIC1INTENABLE register
       - Set bit 11 to 1
     - Reset UART1 interrupt in UINTP1 register
       - Set all bits to 1
       - Set the UINTP1 register to 1 to clear the interrupt

vh_serial_interrupt_handler()

- **Add code**

```
void vh_serial_interrupt_handler(void)
{
        vk_serial_push();
        vh_VIC1INTENCLEAR |= vh_VIC_UART1_bit;
        vh_VIC1INTENABLE |= vh_VIC_UART1_bit;
        vh_VINTP1 = 0xf;
}
```

Store keyboard character data received in the URXH register in a buffer

Pending Clear UART1 Interrupt

Modify getc()

- **Edit code**
  - vpos/hal/io/serial.c

```
char getc(void)
{
    char c;
    while(pop_idx == push_idx){}

    c = serial_buff[pop_idx++];
    pop_idx %= SERIAL_BUFF_SIZE;

    return c;
}
```

Modify vk_serial_push()

- **Edit code**
  - vpos/hal/io/serial.c

```
void vk_serial_push(void)
{
    char c;
    int i=0;
    c = vh_URXH1;

    serial_buff[push_idx++] = c;
    push_idx %= SERIAL_BUFF_SIZE;
}
```

Polling vs Interrupt

- **Polling**

```
char getc(void)
{
    char c;
    unsigned long rxstat;

    while(!vh_SERIAL_CHAR_READY());

    c = vh_SERIAL_READ_CHAR();
    rxstat = vh_SERIAL_READ_STATUS();

    return c;
}
```

Keep spinning
until UART is ready

Read the value

Polling vs Interrupt

- **Interrupt**

```
void vh_serial_interrupt_handler(void)
{
    vk_serial_push();
    vh_VIC1INTENCLEAR |= vh_VIC_UART1_bit;
    vh_VIC1INTENABLE |= vh_VIC_UART1_bit;
    vh_UINTP1 = 0xf;
}
```

Write the value
when interrupt occurs

```
void vk_serial_push(void)
{
    char c;
    int i=0;
    c = vh_URXH1;

    serial_buff[push_idx++] = c;
    push_idx %= SERIAL_BUFF_SIZE;
}
```

Polling vs Interrupt

- **Interrupt**

```
char getc(void)
{
    char c;
    while(pop_idx == push_idx){}

    c = serial_buff[pop_idx++];
    pop_idx %= SERIAL_BUFF_SIZE;

    return c;
}
```

Read the value from the kernel buffer

UART INTERRUPT CODE (ASSEMBLY)

Interrupt entry routine

- **Routine flow**
  1. Link register adjust
  2. Store registers and SPSRs in the previous mode on the stack
  3. Check VICIRQSTATUS to see which of the three controllers has generated an interrupt
     - Use only VIC 0 and VIC 1 in this practice
  4. Store the value of VICADDRESS on the pc

Interrupt return routine

- **Routine flow**
  1. Change the CPSR to IRQ mode and set the IRQ Mask bit to 1
  2. Restore the registers and SPSR of the previous mode from the stack
  3. Return to original routine using ‘movs pc, lr’

Interrupt entry routine : vh_irq

- **Code**
  - vpos/hal/cpu/HAL_arch_startup.S

```
vh_irq:
        sub     lr,lr,#4
        str     sp, vk_save_irq_mode_stack_ptr
        stmfd   sp,{r14}^
        sub     sp, sp, #4
        stmfd   sp,{r13}^
        sub     sp, sp, #4
        stmfd   sp!,{r0-r12}
        mrs     r0, spsr_all
        stmfd   sp!,{r0, lr}
        str     sp, vk_save_irq_current_tcb_bottom
        ldr     r0, =0xe4000000
        ldr     r1, [r0]
        cmp     r1, #0x0
        bne     vh_irq_VIC0
        ldr     r0, =0xe4100000
        ldr     r1, [r0]
        cmp     r1, #0x0
        bne     vh_irq_VIC1
```

Save Context

^ : Registers in previous mode

Save SPSR and Link Register

VIC0IRQSTATUS

VIC1IRQSTATUS

Interrupt entry & return routine : vh_irq_VIC1

- **Code**
  - vpos/hal/cpu/HAL_arch_startup.S

```
vh_irq_VIC1:
        ldr     r0, =0xe4100f00
        ldr     r1, [r0]

        mov     r14, pc
        mov     pc, r1
```

Jump to the interrupt handler address stored in VIC0ADDRESS

```
        msr     cpsr_c, #vh_IRQMODE | 0x80

        ldr     r14, =0xe4100f00
        str     r14, [r14]
```

VIC interrupt service completed

Restore context

```
        ldmfd   sp!, {r0, lr}
        msr     spsr_cxsf, r0
        ldmfd   sp!, {r0-r12}
        ldmfd   sp, {r13}^
        add     sp, sp, #4
        ldmfd   sp, {r14}^
        add     sp, sp, #4
        movs    pc,lr
```

ASSIGNMENT

Assignment1 - Boot the Modified U-Boot

(only for the people who didn't upload the assignment)

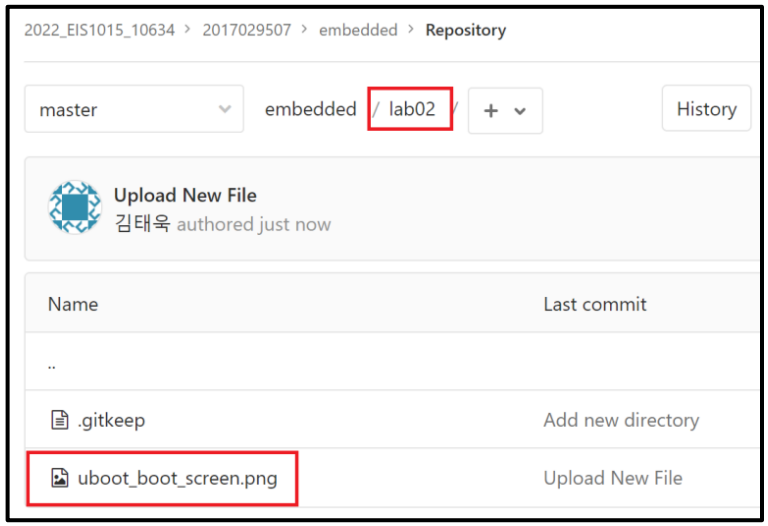
- Every teammate should upload the boot screen

```
U-Boot 1.3.4 (Mar  9 2022 - 19:36:05) for SL2_C100


CPU:      S5PC100@666MHz
          Fclk = 1332MHz, Hclk = 166MHz, Pclk = 66MHz, Serial = PCLK
Board:    SL2_C100
DRAM:     256 MB
Flash:    1 MB
NAND:     512 MB
In:       serial
Out:      serial
Err:      serial
Hit any key to stop autoboot:  0
2022xxxxxx #
2022xxxxxx #
2022xxxxxx #
2022xxxxxx #
2022xxxxxx #
```

example image file to update

(You or Teammate's Student ID must be shown)



Create "lab02" directory

and upload uboot screenshot file

Assignment1 - Boot the Modified U-Boot

(optional : who cannot use the embedded board)

1. Differences between NAND Flash and SDRAM
2. Why we use cross-compiler
3. What happens after “tftp c0008000 u-boot.bin”

- Upload the answer of the question above
  on the “lab02” directory of the embedded repository
- Accepted Format : md, txt, doc, docx

Final Deadline : 4/10 (Sun), 11:59pm

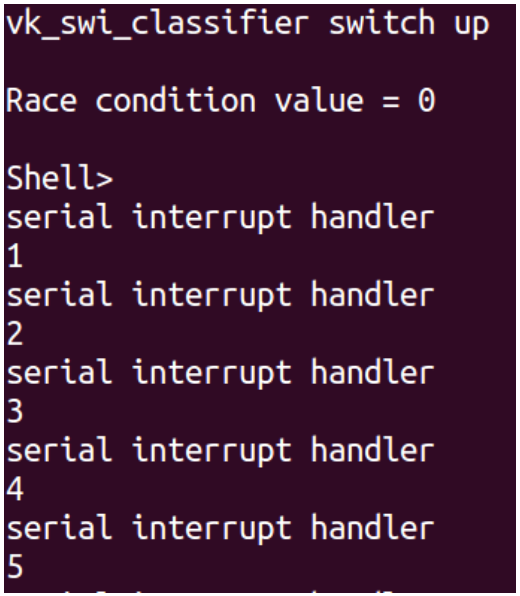
Assignment2 - UART Settings and Interrupt

- Modify vpos/hal/io/serial.c

```
void vh_serial_interrupt_handler(void)
{
        printk("\nserial interrupt handler\n");
        vk_serial_push();
        vh_VIC1INTENCLEAR |= vh_VIC_UART1_bit;
        vh_VIC1INTENABLE |= vh_VIC_UART1_bit;
        vh_UINTP1 = 0xf;
}
```

Assignment2 - UART Settings and Interrupt

- Upload the kernel screen showing that UART interrupt has occured
- When you press the keyboard, kernel prints “serial interrupt handler”
- Upload the image file on the “lab06” of embedded repository



example image file to upload

Assignment2 - UART Settings and Interrupt

(optional : who cannot use the embedded board)

1. What is “Exception”
2. Polling vs Interrupt
3. Explain the process after receiving UART interrupt

- Upload the answer of the question above
  on the “lab06” directory of the embedded repository
- Accepted Format : md, txt, doc, docx

Deadline : 4/15 (Fri), 11:59pm

Thank you