Systematical Case Study
Zephyr Inside Shell

- Author: Ying-Jheng Chen (yingjheng.chen@gmail.com)

- Keywords: Interrupt, ISR, UART Driver, console driver, ISR, IDT, Callback.

- Abstract
  This case study comprehensively surveys zephyr through understanding of shell's internal implementation. It includes shell functions, console driver, UART driver, ISR set-up, hardware configuration, system initialization, and related IDT mechansims.

- Motivation
  To understand the interactivities between task, fiber, driver, interrupts and hardware, case study of Shell implementation is representative. It contains prompt, console driver, UART driver, and related hardware configurations. When I was programming the final assignment, due to my adoption of the Nanokernel, I must ensure that it is achievable to use shell, task, and fibers without related scheduling problems. Through examination of shell implementation, I understood how Zephyr works.
- Overview: Architecture and Files' Usage
  - This case study is one kind of integrational survey from highest level software to hardware. I will include shell initialization, console driver, UART driver, interrupts settings, hardware configuration. Specifically, I will emphasize the bridges between two neighboring layers. From the table, Description of File Usages, we can clearly understand the following thins:
    - The kinds of essential layers are involved in shell implementation.
    - Number of major and specific files are in each layer.
    - Usage of related files.

Description of File Usages		
Layers	Files	Usage
Shell	misc/shell.h	APIs include registration interface
	console_handler_shell.c:	Implementation of shell.h API
Console Driver	driver/console/uart_console.c:	Implementation of serial console driver
UART Driver	include/uart.h	Serial console driver
	uart_ns16550.c:	Implementation of UART Driver API
ISR Setup	inlcude/init.h	The registration interface for IRQ and its own handler
	system_apic.c	General APIC implementation
	ioapic_intr.c	Implementation of APIC interface for Galileo Gen 2.
	arch/x86/core/crt0.S	Fundamental booting procedures
Hardware definition	arch/x86/soc/quark_x1000/soc.h	

System initialization	kernel/nanokernel/nano_init.c

- Design
  Shell’s design is mostly intuitive. It contains four layers: Shell, Console Driver, UART Driver, Hardware. Shell’s internal mechanism uses the first callback, completion(), to communicate shell and console driver. Then it uses the second callback, the ISR: uart_console_isr(), to achieve communication between console driver, UART driver, and UART hardware. We will also look at how ISR works through trace IRQ_CONNECT and IDT table.

- Implementation
  - From Shell to Console driver
    - If developers want to use shell in Zephyr, they should use shell_init to initialize the shell for use. This API will create the fiber to execute shell’s functionalities as well as register the first callback funciton, completion(char *line, uint8_t len), to complete commands, through uart_register_input supported by console driver.
    - In shell_init API, the Zephyr will create a fiber to execute commands with their own corresponding handlers supported by the data structure, struct shell_cmd, in misc/shell.h.
    - Reference:

```
void shell_init(const char *str, const struct shell_cmd *cmds) {
  …
  task_fiber_start(stack, STACKSIZE, shell, 0, 0, 7, 0);
  uart_register_input(&avail_queue, &cmds_queue, completion);
 …
}

struct shell_cmd {
  const char *cmd_name;
  shell_cmd_function_t cb;
  const char *help;
};
```

  - From Console Driver to UART Driver
    - The API, uart_register_input(), is the bridge between shell and console. Depending on developers’ requirement, this API will be responsible for initializing the console through **console_input_init**.
    - To achieve interactions between user and system, the internal function, console_input_init, will reset the URAT RX/TX interface as well as register the second callback function, uart_console_isr, into the corresponding URAT device through uart_irq_callback_set(). A minor note is that before re-enabling the RX interface, the driver must clean the RX buffer first.

```
static void console_input_init(void){
        …
        uart_irq_rx_disable(uart_console_dev);
        uart_irq_tx_disable(uart_console_dev);
```

```
    uart_irq_callback_set(uart_console_dev, uart_console_isr);
    /* clean the RX buffer */
    while (uart_irq_rx_ready(uart_console_dev))
        uart_fifo_read(uart_console_dev, &c, 1);

   uart_irq_rx_enable(uart_console_dev)
}
```

- Uart_irq_callback_set is just a wrapper function in include/uart.h; moreover, the real implementation of Zephyr on Galileo Gen 2 is uart_ns16550_irq_callback_set belonged to uart_ns16550 public APIs. After console driver registered its own callback into UART, uart driver will call this cabllback through each UART ISR.

```
static inline void uart_irq_callback_set(struct device *dev,
                                          uart_irq_callback_t cb)
…
api->irq_callback_set(dev, cb);
…
}

// {.irq_callback_set = uart_ns16550_irq_callback_set}
static void uart_ns16550_irq_callback_set(struct device *dev,
                                         uart_irq_callback_t cb){
    struct uart_ns16550_dev_data_t * const dev_data =
                                            DEV_DATA(dev);
    dev_data->cb = cb;
}
```

- Clarification of Two callbacks:
  1. The first callback: static uint8_t completion(char *line, uint8_t len){…}
  This callback is responsible for communication between shell's prompt and the console driver. It provides the auto completion for commands based the current input. Completion(..) is one of steps in uart_console_isr(…), the second callback.
  2. The second callback: void uart_console_isr(struct device *unused)
  This callback is responsible for communication between console driver and UART driver as well as provided as ISR for UART hardware. Since that the first callback, completion_cb, is just one of steps in the second callback; therefore, the second callback is the essential role for interactivities between shell and UART hardware.

```
void uart_console_isr(struct device *unused) {
…
case '\t':
    if (completion_cb && !end) {
        cur += completion_cb(cmd->line, cur);
    }
    break;
…
}
```

- From UART Driver to Hardware (UART Driver’s ISR Registration and Initialization)
    - In Galileo Gen 2, UART Chip is NS16550. Based on arch/x86/soc/quark_x1000/soc.h, this UART chip uses two IRQs UART_NS16550_PORT_0_IRQ, 0,and UART_NS16550_PORT_1_IRQ, 17. For the usage of Shell, console driver adopts the UART_NS16550_PORT_1_IRQ. Through IRQ_CONNECT, this driver will build the connection between the device, IRQ number and ISR, uart_ns16550_isr.

```
static void irq_config_func_1(struct device *dev) {
      IRQ_CONNECT(UART_NS16550_PORT_1_IRQ,
                        CONFIG_UART_NS16550_PORT_1_IRQ_PRI,
                        uart_ns16550_isr,
      DEVICE_GET(uart_ns16550_1), UART_IRQ_FLAGS);
      irq_enable(UART_NS16550_PORT_1_IRQ);
}

struct uart_device_config uart_ns16550_dev_cfg_1 = {
…
#ifdef CONFIG_UART_INTERRUPT_DRIVEN
                .irq_config_func = irq_config_func_1,
#endif
}

static int uart_ns16550_init(struct device *dev) {
…
#ifdef CONFIG_UART_INTERRUPT_DRIVEN
      DEV_CFG(dev)->irq_config_func(dev);
#endif
}
```

    - To understand when to initialize the UART driver, we should check its initialization levels such as _SYS_INIT_LEVEL_PRIMARY and _SYS_INIT_LEVEL_SECONDARY defined in include/init.h. From the called API, DEVICE_AND_API_INIT, we can know that its level is _SYS_INIT_LEVEL_PRIMARY. It should be initialized in the _Cstart before the execution of the first fiber executing _main ().

```
FUNC_NORETURN void _Cstart(void) {
    …
    sys_device_do_config_level(_SYS_INIT_LEVEL_PRIMARY);
    …
}

DEVICE_AND_API_INIT(uart_ns16550_1,…, PRIMARY,…);*/
```

- In Galileo Gen 2, How does ISR Work?

- IRQ_CONNECT will trigger _ioapic_apic_irq_set through _SysIntVecProgram. The _ioapic_irq_set will finish setting ISR through ioApicRedSetHi() and ioApicRedSetLo() based on the _idt_base_address.

```
#define IRQ_CONNECT(irq_p, priority_p, isr_p, isr_param_p,
flags_p) \
                _ARCH_IRQ_CONNECT(irq_p, priority_p, isr_p,
isr_param_p, flags_p)

#define _ARCH_IRQ_CONNECT(irq_p, priority_p, isr_p,
isr_param_p, flags_p) \
          ({ \
             __asm__ __volatile__(                    \
                 ...
          _SysIntVecProgram(_IRQ_TO_INTERRUPT_VECTOR(irq_p),
          (irq_p), (flags_p));

          }}

void _ioapic_irq_set(unsigned int irq, unsigned int vector, uint32_t
flags)
{
      ...
      ioApicRedSetHi(irq, 0);
      ioApicRedSetLo(irq, rteValue);
}
```

- After pushing the ISRs into the corresponding entries in the IDT table, if each hardware interrupt happened, it will load the ISR in the corresponding entry in IDT table for execution.

```
/* arch/x86/core/crt0.S */
...
GDATA(_idt_base_address)
GDATA(_interrupt_stack)
GDATA(_Idt)
...
lidt    _Idt          /* load 32-bit operand size IDT */
...
_Idt:
.word   (CONFIG_IDT_NUM_VECTORS * 8) - 1/* limit: size of IDT-1*/
.long   _idt_base_address            /* physical start address */
```

- Conclusion
  From the full study of shell's internal implementation from highest level software to lowest level hardware, we can comprehend Zephyr's internal mechanisms such as driver model, ISR set-up, hardware configuration, and system initialization. Deep study of Shell implementation is critically helpful for students to learn Zephyr.
- Reference

- https://www.zephyrproject.org/
- https://github.com/zephyrproject-rtos/zephyr
- https://www.youtube.com/watch?v=fkfhWVOO8-w