Lab3: IRQs and Hardware

Group 2: Shi Su & Mengjin Yan

November 8, 2015

Content

1 Overview

On the basis of lab2 to implement a small kernel - **Gravel**, adding the IRQ handling and timer driver.

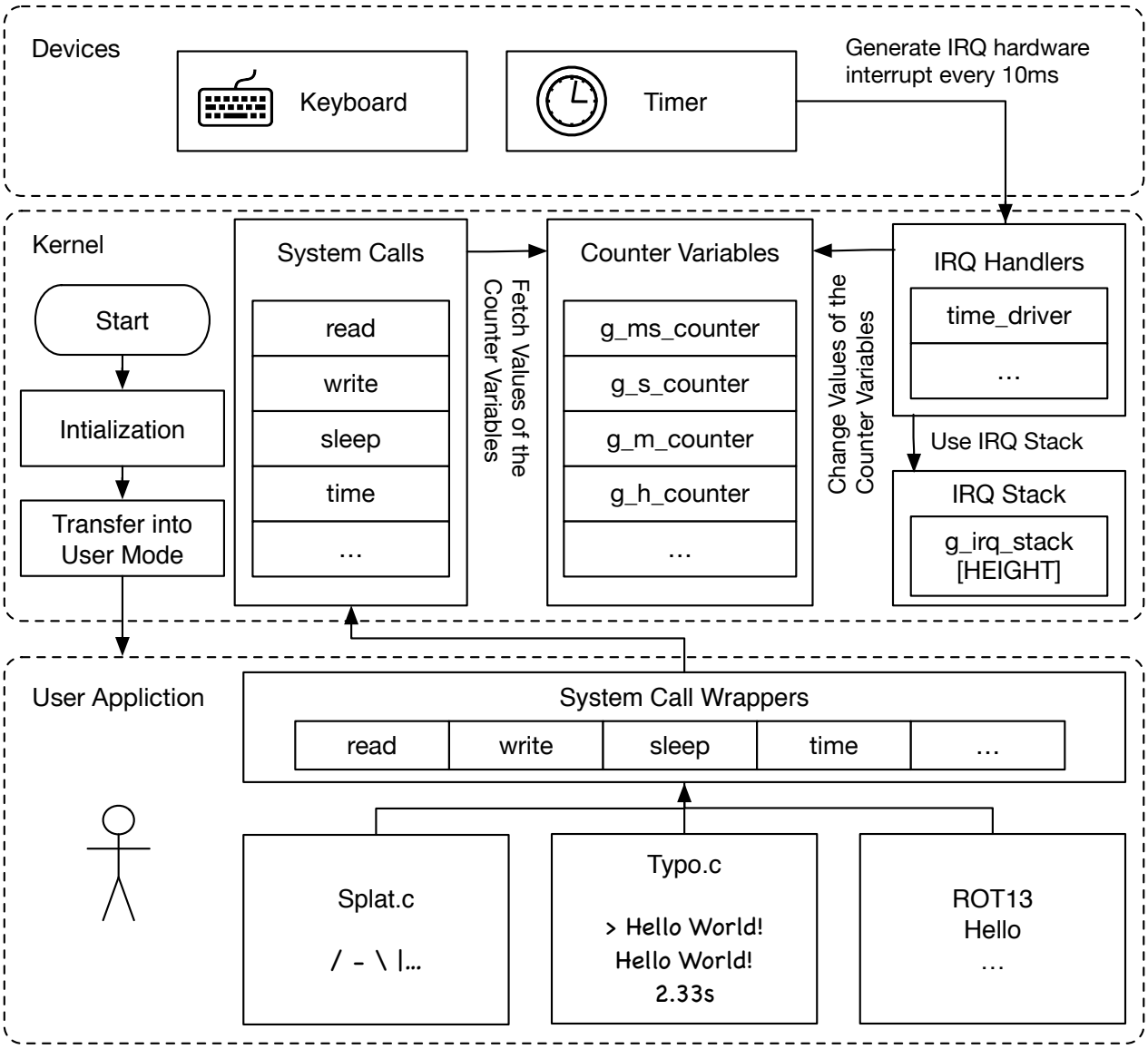

Figure 1: Modules of Gravel

2 Workflow

2.1 Kernel Initiation

- Store the value of r8 (uboot function table) to global variable *global_data*
- Wire in SWI/IRQ handler
- Set up IRQ stack
  A chuck of memory is allocated for IRQ stack in data section as the global variable array *g_irq_stack*, we don't need to specified its location explicitly and worry about it will be clobbered by code or stack. Since array grows from lower address to higher address, while the growth direction of stack is opposite. So sp is set to point at the highest address(end) of array *g_irq_stack*.

- Initiate interrupt controller
  Update ICMR to mask interrupt from all interrupt except OSMR0.
  Update ICLR to route interrupt from OSMR0 as IRQ.

- Initiate Timer Set all time counter variables to 0.
  Set the value of OSSR to 0
  Update OIER to enable the interrupt from OSMR0
  Set the value of OSMR0 to 32500, which corresponds to 10ms resolution

2.2 Entering User Mode

- Store the non-banked register on stack
- Store the pointer to supervisor mode stack to global variable *g_svc_stack*
  Follow the same rationale as IRQ stack, we store the pointer to supervisor stack in data section

- Update cpsr, change mode code to to user mode, unmask IRQ
- Set up full descending user mode stack at 0xa3000000
- Place the command line arguments from uboot on the user mode stack
- Branch to user program loaded at 0xa0000000

2.3 IRQ handling

When an IRQ interrupt is generated, the system automatically changes into IQR mode and the interrupt is handled by the IRQ handler which is consist of two parts, irq_handler.S and c_irq_handler.c.
In irq_handler.S:

- Store non-banked user/supervisor mode register on stack
- Branch into c_swi_handler

In c_irq_handler.c:

- Read the value of ICMR to determine which device generated the IRQ
- Branch to corresponding ISR

2.3.1 Time Driver

As shown in the graph, 4 global variables, *g_ms_counter*, *g_s_counter*, *g_m_counter* and *g_h_counter*, are maintained as time counter in our system. They records how much time has elapsed since kernel booted up.

- *g_ms_counter* represents the milliseconds
- *g_s_counter* represents the seconds
- *g_m_counter* represents the minutes

- *g_h_counter* represent the hours

For every 10ms, an IRQ interrupt is generated from the timer and the system dispatches the interrupt to be handled in the corresponding time_driver handler. In time_driver, the corresponding bit in OSSR is cleared and the value of the global variables are updated.

The design of our time counting system successfully separates the register register reading part and the task processing part of the system making it more flexible and convenient when implementing other timer related system calls.

2.4 SWI handling

2.4.1 SWI Handler

When user program makes a system call, swi wrapper initiate a swi instruction, which will change the control from user program (user mode) to SWI handler (supervisor mode)

In swi_handler.S:

- Store non-banked user mode register on stack
- Restore r8 from global variable *global_data*
- Calculate the swi number based on the swi intruction in swi wrapper, put in r0 (arg0)
- Put the pointer to the stored registers in r1 (arg1)
- With the swi number and register pointer as arguments, branch into c_swi_handler

In c_swi_handler.c:

- call the system call functions corresponding to the swi number
- put the return value on stack using register pointer, which will be picked up by swi_handler and passed along through swi wrapper back to user program

2.4.2 System Calls

- time
  unsigned long time();

  This system call read the value from global time counter: *g_ms_counter*, *g_s_counter*, *g_m_counter*, *g_h_counter* to calculate the milliseconds that have elapsed since the kernel booted up.

- sleep
  void sleep(unsigned long millis);

  This system call reads the start(current) system time using time() system call, and add it with param: millis to get a expected future time before which the program should sleep. Looping until the time() returns a value larger than the expected time.

- read
  ssize_t read(int fd, void *buf, size_t count);

  Read a requested number of bytes from the given file descriptor, implemented with uboot getc, putc API.

- write
  ssize_t write(int fd, const void *buf, size_t count);

  Write a requested number of bytes to the given file descriptor, implemented with uboot putc API

- exit
  void exit(int status);

  Branch into kernel return flow.

2.5 Test Programs

For the test application, we implemented the splat and typo application and make them running correctly on our system. Other applications can be implemented are stop watch and other timer related applications.

- splat

  User sleep system call to realize the delay of the state change.
  For every 200ms, print out backspace ("/b /b") and the next character in the sequence of "/ - "
  Use a while loop to repeat step 2.

- typo

  After print the prompt, use time system call to get the time for now.
  Read the user input and use time system call again to the the end time.
  Calculate the time difference and print out the input as well as the time difference.
  Use a while loop to repeat step 1 3.

2.6 Exiting Kernel

After return or exit from user mode, the kernel will:

- Restore supervisor mode stack from global variable *g_svc_stack*
- Restore supervisor mode registers
- Restore the native SWI and IRQ handlers
- Restore the default value in interrupt controller ICMR and ICLR
- Return to gumstix with user program's return value