Interrupts and Interrupt Handlers
Mark McDermott

Outline of This Lecture
- ARM Interrupts
- Interrupts on the iMX21 SoC
- Interrupt handlers
- Writing interrupt handler for iMX21 SoC

Review of ARM Exceptions
<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Occurs when the processor is first powered on. This exception is only expected to occur the first time a program executes.</td>
</tr>
<tr>
<td>Undefined/Reserved</td>
<td>Occurs if the processor, or any attached expansion, encounters an currently executing instruction.</td>
</tr>
<tr>
<td>Software-Interrupt (SWI)</td>
<td>This is a non-maskable interrupt instruction that allows a program running in User mode, for example, to request privileged operation in another environment, such as an RTOS environment.</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>Occurs when the processor attempts to execute an instruction that was not fetched. Because the address was illegal.</td>
</tr>
<tr>
<td>Data Abort</td>
<td>Occurs when the processor performs an illegal operation or loads an illegal address.</td>
</tr>
<tr>
<td>IRQ</td>
<td>Occurs when the processor receives an interrupt request from the hardware.</td>
</tr>
<tr>
<td>SVC</td>
<td>Occurs when the processor receives an interrupt request from a user-level (L1) core and the 0 bit in the CPSR is clear.</td>
</tr>
</tbody>
</table>

Review of ARM Interrupts
- Vector table
  - Reserved area of 32 bytes at the end of the memory map
    - One word of space for each exception type
    - Contains a Branch or Load PC instruction for the exception handler
  
- Exception modes and registers
  - Handling exceptions changes program from user to non-user mode
  - Each exception handler has access to its own set of registers
    - Its own r13 = stack pointer
    - Its own SP (Saved Program Status Register)
    - Exception handlers must save (restore) other register on entry (exit)

Register Organization Summary

What if Exceptions Happen Simultaneously?

Vector addresses | Exception type | Exception mode | Priority (highest, lowest) |
-----------------|---------------|----------------|---------------------------|
0x00             | Reset         | Supervisor (SVC) | 6 |
0x10             | Undefined/Reserved | Executive | 5 |
0x12             | Software-Interrupt (SWI) | Supervisor (SVC) | 4 |
0x14             | Prefetch Abort | Abort | 3 |
0x16             | Data Abort | Abort | 2 |
0x17             | Not applicable | Not applicable | Not applicable |
0x90             | User Interrupt (UI) | Interrupt (IRQ) | 4 |
0x91             | FIQ Interrupt (FIQ) | FIQ Interrupt (FIQ) | 3 |

Note: System mode uses the User mode register set.
Enabling IRQ and FIQ

- Program Status Register
  - W, Z, C, V
  - I
  - F, M4, M3, M2, M1, M0

- To disable interrupts, set corresponding "F" or "I" bit to 1
- On interrupt, processor does the following
  - Switches register banks
  - Copies CPSR to SPSR (saves mode, interrupt flags, etc.)
  - Changes the CPSR mode bits (M[4:0])
  - Disables interrupts
  - Copies PC to R14 (to provide return address)
  - Sets the PC to the vector address of the exception handler
- Interrupt handlers must contain code to clear the source of the interrupt

Interrupt Details

- On an IRQ interrupt, the ARM processor will ...
  - If the "I" bit in the CPSR is clear, the current instruction is completed and then the processor will
  - Save the address of the next instruction plus 4 in r14_irq
  -Save the CPSR in the SPSR_irq
  - Force the CPSR mode bits M[4:0] to 10010 (binary)
  - This switches the CPU to IRQ mode and then sets the "I" flag to disable further IRQ interrupts
- On an FIQ interrupt, the processor will ...
  - If the "F" bit in the CPSR is clear and the current instruction is completed, the ARM will
  - Save the address of the next instruction plus 4 in r14_irq
  - Force the CPSR mode bits M[4:0] to 10010 (binary)
  - This switches the CPU to FIQ mode and then sets the "I" and "F" flags to disable further IRQ or FIQ interrupts

Types of Interrupts

Synchronous
- Produced by the processor while executing instructions.
- Issues only after finishing execution of an instruction.
- Often called exceptions.
- Example: SWI, page faults, system calls, divide by zero

Asynchronous
- Generated by other hardware devices.
- Occur at arbitrary times, including while CPU is busy executing an instruction. Ex. I/O, timer interrupts

IRQ vs. FIQ

- FIQs have higher priority than IRQs
  - When multiple interrupts occur, FIQs get serviced before IRQs
  - Servicing an FIQ causes IRQs to be disabled until the FIQ handler re-enables them
  - CPSR restored from the SPSR at the end of the FIQ handler
- How are FIQs made faster?
  - They have five extra registers at their disposal, allowing them to store status between calls to the handler
  - FIQ vector is the last entry in the vector table
  - The FIQ handler can be placed directly at the vector location and run sequentially after the location
  - Cache-based systems: Vector table + FIQ handler all locked down into one block

IMX21 ARM Interrupt Controller (AITC)

- The AITC performs the following functions:
  - Supports up to 16 software controlled priority levels for normal interrupts and priority masking
- IMX21 Interrupts
  - The AITC performs the following functions:
    - Supports up to 64 interrupt sources
    - Supports fast and normal interrupts
    - Selects normal or fast interrupt request from any interrupt source
    - Indicates pending interrupt sources via a register for normal and fast interrupts
    - Indicates highest priority interrupt number via register (can be used as a table index)
    - Independently enable or disable any interrupt source
    - Provides a mechanism for software to schedule an interrupt
    - Supports up to 16 software controlled priority levels for normal interrupts and priority masking

IMX21 Interrupts
There are 192 additional interrupts via the GPIO ports. There are five 32-bit GPIO:
- Port A: bits 0–31
- Port B: bits 32–63
- Port C: bits 64–95
- Port D: bits 96–127
- Port E: bits 128–159
- Port F: bits 160–191

Details of the AITC Operation
- The interrupt controller consists of a set of control registers and associated logic to perform interrupt masking, and priority support of normal interrupts.
- The interrupt source registers (INTSRCH / INTSRCL) are a pair of 32-bit status registers with a single interrupt source associated with each of the 64 bits.
- An interrupt line or set of interrupt lines are routed from each interrupt source to the interrupt controller. This allows up to 64 distinct interrupt sources in an implementation. Interrupt requests may be forced to be asserted by way of the interrupt force registers (INTFRCH / INTFRCL).
- Each bit in this register is logically "OR-ed" with the corresponding hardware request line prior to feeding the INTSRCH or INTSRCL register inputs.

Details of the AITC Operation (cont)
- There is a corresponding set of interrupt enable registers (INTENABLEH / INTENABLEL), also 32-bits wide which allow individual bit masking of the INTSRCH / INTSRCL registers. There is also a corresponding set of interrupt type register (INTTYPEH / INTTYPETL) which selects whether an interrupt source will generate a normal or fast interrupt to the ARM926EJ-S core.

GPIO Interrupts on the iMX21
- Every general purpose input can be configured as an interrupt and each interrupt can be defined as either:
  - rising-edge triggered
  - falling-edge triggered
  - level sensitive
- The interrupts can be masked using a 32-bit mask register.
- Two levels of interrupt masking are provided. Interrupts can be individually masked at the bit level or at the port level.
- The interrupt status register bits corresponding to the interrupts waiting for service are stored as a value of 1. The interrupt status register is Write 1 to Clear (w1c).
Assigning Interrupt Number on the IMX21

- The 64 interrupt sources are assigned from 0-63 respectively.
- INT-8 (GPIO interrupt) is assigned interrupt numbers 64-255.
- To determine which interrupt sources Linux recognizes type:
  - more /proc/interrupts
- On the TLL-6219 you will see the following 4 interrupt sources:
  - INT-8 (GPIO interrupt)
  - INT-9 (IOMMU
  - INT-10 (interrupt)
  - INT-11 (interrupt)
- The first 3 sources are internal interrupts. INT-224 is an external interrupt from the Ethernet Controller.
  - The ENET interrupt is connected to Port-F Pin-0 (PF0).
  - This corresponds to bit #160 in the GPIO bit ordering.
  - INT Number = 64 + 160 = 224

Interrupt Handlers

- Kernel routine that runs in response to interrupt.
  - More than one handler can exist per IRQ.
  - Must run quickly.
    - How to deal with high work interrupts?
    - Ex: network, hard disk
- Must run quickly.
  - How to deal with high work interrupts?
  - Ex: network, hard disk
- Must run quickly.
  - How to deal with high work interrupts?
  - Ex: network, hard disk
- Must run quickly.
  - How to deal with high work interrupts?
  - Ex: network, hard disk

Interrupt Handlers

- When an interrupt occurs, the hardware will jump to an "interrupt handler"
- Must run quickly.
  - How to deal with high work interrupts?
  - Ex: network, hard disk

Interrupt Handlers

- Must run quickly.
  - How to deal with high work interrupts?
  - Ex: network, hard disk

Nested/Re-entrant Interrupts

- Interrupts can occur within interrupt handlers
- Must run quickly.
  - How to deal with high work interrupts?
  - Ex: network, hard disk

Jumping to the Interrupt Handler

- Auto-vectored
  - Processor-determined address of interrupt handler based on type of interrupt
  - This is what the ARM does
- V vectored
  - Device supplies processor with address of interrupt handler
- Why the different methods?
  - If multiple devices use the same interrupt type (IRQ vs. FIQ), in an Auto-vector system the processor must poll each device to determine which device interrupted the processor
  - This can be time-consuming if there is a lot of devices
  - In a vectored system, the processor would just take the address from the device (which dumps the interrupt vector onto a special bus).
Timing of Interrupts

- Before an interrupt handler can do anything, it must save away the current program’s registers (if it touches those registers)
- That’s why the FIQ has lots of extra registers - to minimize CPU context saving overhead

Interrupt Context

- Not associated with a process.
  - Cannot sleep: no task to reschedule.
  - Current macro points to interrupted process.
- Shares kernel stack of interrupted process.
  - Be very frugal in stack usage.

Registering a Handler

request_irq()
- Register an interrupt handler for a given interrupt input pin.

free_irq()
- Unregister a given interrupt handler.
- Disable interrupt line if all handlers unregistered.

Dos and Don’ts of Interrupt Handlers

- It’s a programming offense if your interrupt context code goes to sleep.
- Interrupt handlers cannot relinquish the processor by calling sleep functions such as schedule_timeout().
- For protecting critical sections inside interrupt handlers, you can’t use mutexes because they may go to sleep. Use spinlocks instead, and use them only if you must.
- Interrupt handlers are supposed to get out of the way quickly but are expected to get the job done. To circumvent this Catch-22, interrupt handlers split their work into two halves: top (slice) and bottom (fat).
- You do NOT need to design interrupt handlers to be reentrant. When an interrupt handler is running, the corresponding IRQ is disabled until the handler returns.
- Interrupt handlers can be interrupted by handlers associated with IRQs that have higher priority. You can prevent this nested interruption by specifically requesting the kernel to treat your interrupt handler as a fast handler.

Top and Bottom Halves

- Interrupt handling sometimes needs to perform lengthy tasks.
- This problem is resolved by splitting the interrupt handler into two halves:
  - Top half responds to the interrupt
    - The one registered to request_irq
    - Saves data to device-specific buffer and schedules the bottom half
    - Current interrupt disabled, possibly all disabled.
    - Runs in interrupt context, not process context. Can’t sleep.
    - Acknowledges receipt of interrupt.
    - Schedules bottom half to run later.
  - Bottom half is scheduled by the top half to execute later
    - With all interrupts enabled
    - Wakes up processes, starts I/O operations, etc.
    - Runs in process context with interrupts enabled.
    - Performs most work required. Can sleep.
    - Ex: copies network data to memory buffers.

Three mechanisms may be used to implement bottom halves
- SoftIRQs
  - Have strong locking requirements
  - Only used of performance sensitive subsystems – networking, SCSI, etc.
  - Reentrant
- Tasklets
  - Built on top of SoftIRQs
  - Should not sleep
  - Cannot run in parallel with itself
  - Can run in parallel with other tasklets on SMP systems
  - Guaranteed to run on the same CPU that first scheduled them
- Workqueues
  - Can sleep
  - Cannot copy data to and from user space
Writing an Interrupt Handler for PF16 on iMX21

The first task to do is the request the driver and associate an interrupt handler with it. This is done as part of init()

```c
static int __init init_interrupt_arm(void)
{
    /* request interrupt */
    request_irq(PF16_INT, interrupt_interrupt_arm, SA_TRIGGER_RISING | SA_DISABLED, "interrupt_int", NULL);
    return 0;
}
```

The SA_DISABLED flag specifies that this interrupt handler has treated as a fast handler, so the kernel has to disable interrupts while invoking the handler.

```c
#define MODULE_NAME "fpga_int"
```

The character device that is used by user application needs to be registered when the kernel driver is initialized.

```c
static void __init mknod(void)
{
    char *filename = "./dev/fpga_int";
    int major = 254, minor = 0;
    if (mknod(filename, major, minor, S_IFCHR | 0666))
        printk("fpga_int: /dev/fpga_int: failed to create.
```

The /dev/fpga_int device is assigned to 245,0

Use `mknod /dev/fpga_int 2450` to generate the node.

The &fpga_ops pointer is used to point to the routines that are called when the device is accessed from the user application.
### fpga_fasync() routine

This is invoked by the kernel when the user program opens the /dev/fpga_int device and issues fcntl(F_SETFL) on the associated file descriptor.

- **fasync_helper()** ensures that if the driver issues a kill_fasync(), a SIGIO is dispatched to the owning application.

```c
int fpga_fasync(int fd, void *vfa, int cmd)
{
  fasync_fpga_queue *faq;
  faq = vfa;
  return fasync_helper(fd, faq, cmd);
}
```

### kill_fasync() routine

- **kill_fasync()** is used to signal the interested process(es) when data arrives. "kill" is actually a misnomer. This function asynchronously delivers the SIGIO signal to the processes which requested it. Since the default action performed when receiving a signal is to terminate.

  - The arguments are the signal to send (usually SIGIO) and the band, which is almost always POLL_IN plus a pointer to the queue with the list of processes to be notified "fasync_fpga_queue"

  - **Usage:**
    ```c
    kill_fasync(&fasync_fpga_queue, SIGIO, POLL_IN);
    ```

### fcntl() routine

- **fcntl(int fd, int cmd)** manipulate open file descriptors. It performs one of various miscellaneous operations on fd. The operation in question is determined by cmd:
  - **F_GETFL:** Read the file descriptor's flags.
  - **F_SETFL:** Set the file status flags part of the descriptor's flags to the value specified by arg. Remaining bits (access mode, file creation flags) in arg are ignored. On Linux this command can only change the O_APPEND, O_NONBLOCK, O_ASYNC, and O_DIRECT flags.
  - **F_SETOWN** Set the process ID or process group that will receive SIGIO and SIGURG signals for events on file descriptor fd.

```c
int fcntl(int fd, int cmd, long arg)
{
  struct fasync *faq = fasync_fpga_queue;
  if (cmd == F_GETFL)
    return arg;
  return fasync_helper(fd, faq, cmd, arg);
}
```
main_loop() routine

- This while loop emulates a program running the main loop i.e. sleep(). The main loop is interrupted when the SIGIO signal is received.

```c
while(1) {
    /* this only returns if a signal arrives */
    sleep(86400); /* one day */
    if (!det_int)
        continue;
    num_int++;   // Count the number of interrupts – DEBUG ONLY
    printf("mon_interrupt: Number of interrupts detected: %d\n", num_int);
    det_int=0;   // Reset flag for next loop
}
```