

Week 8. Memory Mapped I/O & Interrupts

18-342: Fundamentals of Embedded Systems

Rajeev Gandhi

INI & ECE
Carnegie Mellon University

Carnegie Mellon



Overview

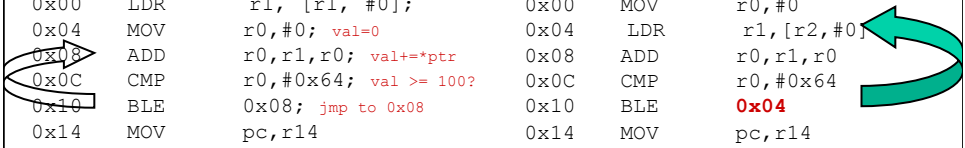
- I/O registers vs. memory locations
- Timers
 - Uses of Timers
 - Timers on Gumstix
 - Watchdog Timers
- Interrupts
 - IRQs vs. FIQs
 - Interrupt Controller
- Interrupt latency
 - Nested vs. Non-nested interrupt controllers
- Midterm on Thursday
 - Will cover everything
- Quiz 2 – min/avg/max were 21/63.83/93

I/O Register Basics

- I/O Registers are NOT like normal memory
 - Device events can change their values (e.g., status registers)
 - Reading a register can change its value (e.g., error condition reset)
 - so, for example, can't expect to get same value if read twice
 - Some are read-only (e.g., receive registers)
 - Some are write-only (e.g., transmit registers)
 - Sometimes multiple I/O registers are mapped to same address
 - selection of one based on other info (e.g., read vs. write or extra control bits)
- Cache must be disabled for memory-mapped addresses
- When polling I/O registers, should tell compiler that value can change on its own and therefore should not be stored in a register
 - `volatile int *ptr; (or int volatile *ptr;)`

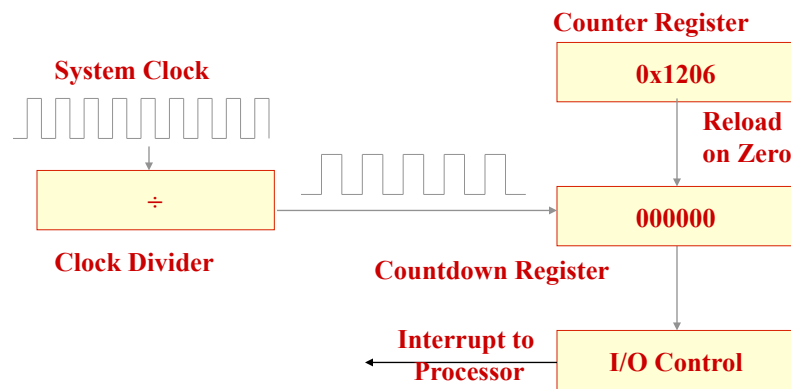
Volatile keyword

<pre>void test() { int *ptr = (int *) 0x20200000; int val=0; do { val+=*ptr; } while(val <= 100); }</pre>	<pre>void test() { volatile int *ptr=(int *) 0x2020...; int val=0; do { val+=*ptr; } while(val <= 100); }</pre>
Initial value: r1 contains 0x20200000 Assume r0 contains val	Initial value: <u>r2</u> contains 0x2020... Assume r0 contains val
<pre>0x00 LDR r1, [r1, #0]; 0x04 MOV r0, #0; val=0 0x08 ADD r0, r1, r0; val+=*ptr 0x0C CMP r0, #0x64; val >= 100? 0x10 BLE 0x08; jmp to 0x08 0x14 MOV pc, r14</pre>	<pre>0x00 MOV r0, #0 0x04 LDR r1, [r2, #0] 0x08 ADD r0, r1, r0 0x0C CMP r0, #0x64 0x10 BLE 0x04 0x14 MOV pc, r14</pre>



What is a Timer?

- A device that uses a high-speed clock input to provide a series of time or count-related events



Different Uses of Timers

- Pause Function
 - Suspends task for a specified amount of time
- One-shot timer
 - Single one-time-only timeout
- Periodic timer
 - Multiple renewable timeouts
- Time-slicing
 - Chunks of time to each task
- Watchdog timer

Controlling the Timer on Gumstix

- There are two hardware timers on the gumstix boards
 - Real-time timer
 - OS Timer (this timer has 8 independent channels on verdex-pro)
- The following memory-mapped registers can be used to control the OS timer from software
 - **OSCR0**: The OS Timer Count register
 - is a 32-bit count-up counter that increments on the rising edge of the clock (3.6864 MHz clock)
 - **OSMRx**: The OS Timer Match register(#) is a set of four registers
 - Each register is a 32-bit read-write register that holds a timer counter target and are matched against OSCR0
 - **OSSR**: The OS Timer Status register
 - holds bits indicating that a match had occurred between the OSCR and the corresponding OSMR
 - **OIER**: The OS Timer Interrupt Enable register has four non-reserved bits

OSCR & OSMR

Physical Address 0x40A0_0010										OS Timer Count Register (OSCR)										System Integration Unit												
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OSCV																																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bits		Name		Description																												
<31:0>		OSCV		OS Timer Counter Value. The current value of the OS timer counter.																												

Physical Address		OS Timer Match Register 0-3		System Integration Unit																															
0x40A0_0000		(OSMR3, OSMR2, OSMR1,																																	
0x40A0_0004		OSMR0)																																	
0x40A0_0008																																			
0x40A0_000C																																			
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1				
	OSMV																																		
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0				
	Bits		Name		Description																														
	<31:0>		OSMV		OS Timer Match Value. The value compared against the OS timer counter.																														

Other Uses of Timers – Watchdog Timers

- A piece of hardware that can be used to reset the processor in case of anomalies
- Typically a timer that counts to zero
 - Reboots the system if counter reaches zero
 - For normal operation – the software has to ensure that the counter never reaches zero (“kicking the dog”)



Source: Introduction to Watchdog Timers, M. Barr,

Embedded.com

Taking Care of Your (Watch)dog

- A watchdog can get the system out of many dangerous situations
- But be very careful
 - Bugs in the watch-dog timer could perform unnecessary resets
 - Bugs in the application code could perform resets
- Choosing the right kicking interval is important

Interrupt vs. Polled I/O

- **Polled I/O** requires the CPU to *ask* a device (e.g. Ethernet controller) if the device requires servicing
 - For example, if the Ethernet controller has changed status or received packets
 - Software plans for polling the devices and is written to know when a device will be serviced
- **Interrupt I/O** allows the device to *interrupt* the processor, announcing that the device requires attention
 - This allows the CPU to ignore devices unless they request servicing (via interrupts)
 - Software cannot plan for an interrupt because interrupts can happen at any time -- therefore, software has no idea when an interrupt will occur
- Processors can be programmed to ignore or mask interrupts
 - Different types of interrupts can be masked (IRQ vs. FIQ)

Polling Example

- Polling switches

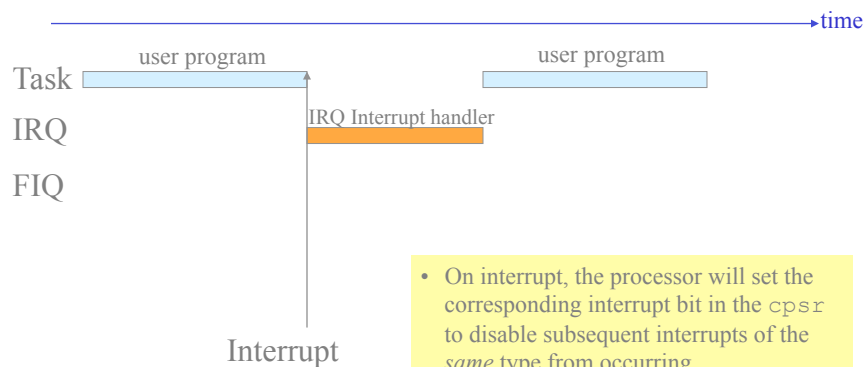
```
#define SWITCH_BASE          0x18200000
int main(int argc, char * argv[])
{
    volatile unsigned int *switchBank = (unsigned int *)
        SWITCH_BASE;
    unsigned int tmpSwitchState;
    unsigned int prevSwitchState;
    /* get the current state of the switches */
    prevSwitchState = *switchBank & 0xff;
    while (1) {
        /* loop until a switch is pressed */
        while (prevSwitchState ==
            (tmpSwitchState = (*switchBank & 0xff))) {}
    }
} /* end main() */
```

Polling vs. Interrupt-Driven I/O

- Polling requires code to loop until device is ready
 - Consumes *lots* of CPU cycles
 - Can provide quick response (guaranteed delay)
- Interrupts don't require code to loop until the device is ready
 - Device interrupts processor when it needs attention
 - Code can go off and do other things
 - Interrupts can happen at any time
 - Requires careful coding to make sure other programs (or your own) don't get messed up
- What do you think real-time embedded systems use?

Onto IRQs & FIQs: Interrupt Handlers

- When an interrupt occurs, the hardware will jump to an **interrupt handler** or **interrupt service routine (ISR)**



- On interrupt, the processor will set the corresponding interrupt bit in the `cpsr` to disable subsequent interrupts of the *same* type from occurring.
- However, interrupts of a *higher* priority can still occur.

cpsr & spsr for IRQs and FIQs



- Interrupt Disable bits
 - I = 1, *disables* the IRQ
 - F = 1, *disables* the FIQ
- Mode bits
 - Processor mode differs

M[4:0]	Mode
10000	User
10001	FIQ
10010	IRQ
10011	SVC
10111	Abort
11011	Undef
11111	System

Exception Priorities

Exceptions	Priority	I bit (1⇒IRQ Disabled)	F bit (1⇒FIQ Disabled)
Reset	1 (highest)	1	1
Data Abort	2	1	
Fast Interrupt Request (FIQ)	3	1	1
Interrupt Request (IRQ)	4	1	
Prefetch Abort	5	1	
Software Interrupt	6	1	
Undefined Instruction	6 (lowest)	1	

IRQ and FIQ ISR Handling

IRQ Handling

- When an IRQ occurs, the processor
 - Copies `cpsr` into `spsr_irq`
 - Sets appropriate `cpsr` bits
 - Sets mode field bits to 10010
 - Disables further IRQs
 - Maps in appropriate banked registers
 - Stores the “return address” in `lr_irq`
 - Sets `pc` to vector address 0x00000018
- To return, exception handler needs to:
 - Restore `cpsr` from `spsr_irq`
 - Restore `pc` from `lr_irq`
 - Return to user mode

FIQ Handling

- ◆ When an FIQ occurs, the processor
 - Copies `cpsr` into `spsr_fiq`
 - Sets appropriate `cpsr` bits
 - Sets mode field bits to 10001
 - Disables further IRQs and FIQs
 - Maps in appropriate banked registers
 - Stores the “return address” in `lr_fiq`
 - Sets `pc` to vector address 0x0000001c
- ◆ To return, exception handler needs to:
 - Restore `cpsr` from `spsr_fiq`
 - Restore `pc` from `lr_fiq`
 - Return to user mode

Recall: Controlling the Timer on Gumstix

- The following memory-mapped registers can be used to control the OS timer from software
 - **OSCR0**: The OS Timer Count register
 - is a 32-bit count-up counter that increments on the rising edge of the clock (3.25 MHz clock)
 - **OSMRx**: The OS Timer Match register(#) is a set of four registers
 - Each register is a 32-bit read-write register that holds a timer counter target and are matched against OSCR0
 - **OSSR**: The OS Timer Status register
 - holds bits indicating that a match had occurred between the OSCR and the corresponding OSMR
 - **OIER**: The OS Timer Interrupt Enable register has four non-reserved bits

Gumstix Interrupt Controller

- **Interrupt Controller Memory-mapped registers–**
 - There are 6 memory-mapped registers which can be used to determine the peripheral devices that have raised an interrupt, mask interrupts and configure which sources will cause an IRQ or an FIQ
- **Interrupt Controller Level Register** 0x40D00008
bit 29 if set indicates OSMR3=OSCR generates FIQ otherwise IRQ
bit 28 if set indicates OSMR2=OSCR generates FIQ otherwise IRQ
bit 27 if set indicates OSMR1=OSCR generates FIQ otherwise IRQ
bit 26 if set indicates OSMR0=OSCR generates FIQ otherwise IRQ
- **Interrupt Controller Mask Register**
Memory mapped address 0x40D00004
bit 29 if set=0 masks OSMR3=OSCR interrupts
bit 28 if set=0 masks OSMR2=OSCR interrupts
bit 27 if set=0 masks OSMR1=OSCR interrupts
bit 26 if set=0 masks OSMR0=OSCR interrupts

Registers on Gumstix Interrupt Controller

- **Interrupt Controller IRQ Pending Register** 0x40D00000
bit 29 if set indicates OSMR3=OSCR is not masked and caused IRQ
bit 28 if set indicates OSMR2=OSCR is not masked and caused IRQ
bit 27 if set indicates OSMR1=OSCR is not masked and caused IRQ
bit 26 if set indicates OSMR0=OSCR is not masked and caused IRQ
- **Interrupt Controller FIQ Pending Register**
Memory mapped address 0x40D0000C
bit 29 if set masks OSMR3=OSCR is not masked and caused FIQ
bit 28 if set masks OSMR2=OSCR is not masked and caused FIQ
bit 27 if set masks OSMR1=OSCR is not masked and caused FIQ
bit 26 if set masks OSMR0=OSCR is not masked and caused FIQ

Masking/Disabling Interrupts on ARM-Based Embedded Systems

- Three methods of masking/disabling interrupts
 - Disable Interrupts: set I/F bits in CPSR
 - Mask an Interrupt: unset the bits in Interrupt Controller Mask register
 - Don't use interrupts: configure the peripheral device to not generate interrupts
 - E.g. – Timer interrupt can be disabled through OIER

Physical Address 0x40A0_001C										OS Timer Interrupt Enable Register (OIER)										System Integration Unit																
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
	reserved																												E3	E2	E1	E0				
Reset	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?				
	Bits		Name		Description																															
	<31:4>		—		reserved																															
	<3>		E3		Interrupt enable channel 3. 0 – A match between OSMR3 and the OS Timer will NOT assert OSSR[M3]. 1 – A match between OSMR3 and the OS Timer asserts OSSR[M3].																															
	<2>		E2		Interrupt enable channel 2. 0 – A match between OSMR2 and the OS Timer will NOT assert OSSR[M2]. 1 – A match between OSMR2 and the OS Timer asserts OSSR[M2].																															
	<1>		E1		Interrupt enable channel 1. 0 – A match between OSMR1 and the OS Timer will NOT assert OSSR[M1]. 1 – A match between OSMR1 and the OS Timer asserts OSSR[M1].																															
	<0>		E0		Interrupt enable channel 0. 0 – A match between OSMR0 and the OS Timer will NOT assert OSSR[M0]. 1 – A match between OSMR0 and the OS Timer asserts OSSR[M0].																															

Interrupt Latency

- Interrupt latency
 - Interval of time from an external interrupt-request signal being raised to the first fetch of an instruction of a specific ISR
- Affected by the number of simultaneous interrupt sources to handle
- Contributors to interrupt latency
 - Time taken to recognize the interrupt
 - Time taken by the CPU to complete its current instruction (depends on whether the CPU is executing a multi-cycle or a single-cycle instruction)
 - Time taken by the CPU to perform a context switch (switching in banked registers, saving program counter, etc.)
 - Time taken to fetch the interrupt vector
 - Time taken to start the ISR executing
- For real-time systems, you need to compute worst-case interrupt latency

Minimizing Interrupt Latency

- Nested interrupt handler
 - Allows further interrupts to occur when a current interrupt is being serviced
 - Re-enable interrupts at a safe point in the current ISR
 - Ultimately, nested interrupts roll back to the original ISR
- Prioritization
 - Ignore interrupts of the same/lower priority than the current interrupt
 - Higher-priority tasks can interrupt the current ISR
 - Higher-priority interrupts have a lower average interrupt latency

Types of Interrupt Handlers

- Non-nested interrupt handler (simplest possible)
 - Services individual interrupts sequentially, one interrupt at a time
- Nested interrupt handler
 - Handles multiple interrupts without priority assignment
- Prioritized (re-entrant) interrupt handler
 - Handles multiple interrupts that can be prioritized

Non-Nested Interrupt Handler

- Does not handle any further interrupts until the current interrupt is serviced and control returns to the interrupted task
- Not suitable for embedded systems where interrupts have varying priorities and where interrupt latency matters
 - However, relatively easy to implement and debug
- Inside the ISR (**after** the processor has disabled interrupts, copied `cpsr` into `spsr_mode`, set the etc.)
 - Save context – subset of the current processor mode's nonbanked registers
 - Not necessary to save the `spsr_mode` – why?
 - ISR identifies the external interrupt source – how?
 - Service the interrupt source and reset the interrupt
 - Restore context
 - Restore `cpsr` and `pc`

Nested Interrupt Handler

- Allows for another interrupt to occur within the currently executing handler
 - By re-enabling interrupts at a *safe point* before ISR finishes servicing the current interrupt
- **Care needs to be taken in the implementation**
 - Protect context saving/restoration from interruption
 - Check stack
 - Increases code complexity, but improves interrupt latency
- Does not distinguish between high and low priority interrupts
 - Time taken to service an interrupt can be high for high-priority interrupts

Prioritized (Re-entrant) Interrupt Handler

- Allows for higher-priority interrupts to occur within the currently executing handler
 - By re-enabling higher-priority interrupts within the handler
 - By disabling all interrupts of lower priority within the handler
- Same care needs to be taken in the implementation
 - Protect context saving/restoration from interruption, check stack overflow
- Does distinguish between high and low priority interrupts
 - Interrupt latency can be better for high-priority interrupts

Interrupts and Stacks

- Stacks are important in interrupt handling
 - Especially in handling nested interrupts
 - Who sets up the IRQ and FIQ stacks and when?
- Stack size depends on the type of ISR
 - Nested ISRs require more memory space
 - Stack grows in size with the number of nested interrupts
- Good stack design avoids stack overflow (where stack extends beyond its allocated memory) – two common methods
 - Memory protection
 - Call stack-check function at the start of each routine
- Important in embedded systems to know the stack size ahead of time (as a part of the designing the application) – why?