Lecture 8: Timers

Thomas Schmid
thomas.schmid@utah.edu

February 9, 2012
Announcements

• One point that sticks, one that’s fuzzy
  – Really helps me!
  – Website form or on paper after class?

• Some issues raised
  – Graduate student presentation vs undergraduate project?
  – Interrupts were clear, but Bus lecture was confusing
Issues with Interrupts

- Priority and masking!
- Main vs sub priority
- How does priority setup work

- Multiple interrupt examples
- Unclear how to treat interrupts (what is automagic, what is manual)
- What are the exact procedures that get invoked for an interrupt
- Cleanup of interrupts

- how do we create an interrupt (draw a block diagram)
- what peripherals typically call interrupts, why?
- Different request on the same interrupt line

- How do I register my own interrupt (is interrupt1 defined and I overwrite it’s function)

- How are interrupts different from typical function calls?

- Different interrupt configuration registers

- Is there a way to limit concurrent interrupts?
- How to prevent stack problems
If more bits are implemented, more priority levels will be available (see Figure 7.3). However, more priority bits can also increase gate counts and hence the power consumption. For the Cortex-M3, the minimum number of implemented priority register widths is 3 bits (eight levels).

The reason for removing the LSB of the register instead of the Most Significant Bit (MSB) is to make it easier to port software from one Cortex-M3 device to another. In this way, a program written for devices with 4-bit priority configuration registers is likely to be able to run on devices with 3-bit priority configuration registers. If the MSB is removed instead of the LSB, you might get an inversion of priority arrangement when porting an application from one Cortex-M3 chip to another. For example, if an application uses priority level 0x05 for IRQ #0 and level 0x03 for IRQ #1, IRQ #1 should have higher priority. But when MSB bit 2 is removed, IRQ #0 will become level 0x01 and have a higher priority than IRQ #1.
Today

• Background on time and clocking on a processor
• SmartFusion clocks and features
• How we use timers
• Design of a memory-mapped Pulse-Width Modulation (PWM) circuit.
Time in Embedded Systems: Where do we need accurate time?

- Scheduling of computation
  - Scheduler in operating systems
  - Real time operating systems
- Signal sampling and generation
  - Audio sampling at 44.1 kHz
  - TV/video generation (sync, vsync)
  - Pulse Width Modulated (PWM) signals
- Communication
  - Media Access Control (MAC) protocols
  - Modulation
- Navigation
  - GPS
Clock generation and use

- **Resonating element/Driver:**
  - Quartz crystal can be made to resonate due to Piezoelectric effect.
    - Resonate frequency depends on length, thickness, and angle of cut.
    - Issues: Very stable (<100ppm) but not all frequencies possible.
  - MEMS Resonator
    - Arbitrary frequency, potentially cheap, susceptible to temperature variations.
  - Others:
    - Inverter Ring, LC/RC circuits, Atomic clock, and many more.
Clock Driver

- **Barkhausen Criteria:**
  - For a positive feedback system, oscillation will occur when loop gain (product of forward gain and feedback gain) has zero phase shift and a magnitude greater than unity.

- **Performance Metrics**
  - Quality or Q factor: measure of energy loss within resonating structure.
  - Frequency Stability: How much the center of the peak moves (longer term).
  - Phase Noise: Energy around the peak (short term).
• How do we distribute and generate different clock signals?
This section describes the clocking resources available to the SmartFusion™ FPGA fabric. Some of the resources are embedded within the SmartFusion microcontroller subsystem (MSS), but provide the FPGA fabric with access to internal and external clock signals.

The SmartFusion device family has a robust collection of clocking peripherals, some of which are shared between the SmartFusion FPGA fabric and the microcontroller subsystem.

Figure 2-1 provides a top-level representation of the clocking resources available to the SmartFusion FPGA fabric. As shown in Figure 2-1, there is an MSS clock conditioning circuit (CCC) that contains a PLL. This MSS CCC is primarily configured via firmware running on the ARM® Cortex™-M3 processor and is shared between the MSS and FPGA fabric. Users have the option of using Actel's Libero® Integrated Design Environment (IDE) MSS configurator to configure the MSS CCC and Actel System Boot Firmware. Alternatively, users can create custom firmware to setup the MSS CCC Configuration Registers. For more information about configuring the MSS clocking resources, refer to the “PLLs, Clock Conditioning Circuitry, and On-Chip Crystal Oscillators” section of the SmartFusion Microcontroller Subsystem User’s Guide. Additionally, there are five standard CCs dedicated to the FPGA fabric. In the A2F200 device, the standard CCs do not integrate a PLL.
The GLA0 output of the MSS_CCC block drives the input clock to the microcontroller subsystem (MSS). The clock source for the 10/100 Ethernet MAC can be sourced from an external pin or the GLC output of the MSS_CCC block, and the GLA1 and GLB outputs are dedicated to the FPGA fabric.

As depicted in Figure 8-2, the MSS_CCC block consists of the following main components: input clock multiplexers, PLL, dividers and delays. There are three main paths through the MSS_CCC block: the CLKA, CLKB, and the CLKC paths, which output clocks onto the global buffers GLA, GLB, and GLC. As can be seen in more detail in Figure 8-3, there are actually two more outputs from the PLL/CCC block. The YB and YC outputs can drive additional local routing resources in the FPGA fabric.

Figure 8-6 depicts a simplified view of the CCC blocks without a PLL.
MSS Clock(s) Configurator

Input

Output

Clock Conditioning
• How do we keep time?
Features of Timers

- Time is kept in a hardware counter
  - Resolution: How often the hardware counter is updated.
  - Precision: The smallest increment the software can read the counter.
  - Accuracy: How close we are to UTC
  - Range: The counter reads a value of \( (f^*t) \mod 2^n \). Range is the biggest value we can read.

UTC is Coordinated Universal Time (French is Temps Universel Coordonné). I just work here...
• **Watchdog Timer**
  - 32-bit down counter
  - Either reset system or NMI Interrupt if it reaches 0!
Timers on the SmartFusion

- SysTick Timer
  - ARM requires every Cortex-M3 to have this timer
  - Essentially a 24-bit down-counter to generate system ticks
  - Has its own interrupt
  - Clocked by FCLK with optional programmable divider

- See Actel SmartFusion MSS User Guide for register definitions
Timers on the SmartFusion

- Real-Time Counter (RTC) System
  - Clocked from 32 kHz low-power crystal
  - Automatic switching to battery power if necessary
  - Can put rest of the SmartFusion to standby or sleep to reduce power
  - 40-bit match register clocked by 32.768 kHz divided by 128 (256 Hz)

http://www.actel.com/documents/SmartFusion_MSS_UG.pdf
Timers on the SmartFusion

• System timer
  – “The System Timer consists of two programmable 32-bit decrementing counters that generate interrupts to the ARM® Cortex™-M3 and FPGA fabric. Each counter has two possible modes of operation: Periodic mode or One-Shot mode. The two timers can be concatenated to create a 64-bit timer with Periodic and One-Shot modes. The two 32-bit timers are identical”

http://www.actel.com/documents/SmartFusion_MSS_UG.pdf
There are two basic activities one wants timers for:

- Measure how long something takes
  - "Capture"
- Have something happen every X time period.
  - "Compare"
Example #1 -- Capture

• FAN
  – Say you have a fan spinning and you want to know how fast it is spinning. One way to do that is to have it throw an interrupt every time it completes a rotation.
    • Right idea, but might take a while to process the interrupt, heavily loaded system might see slower fan than actually exists.
    • This could be bad.
  – Solution? Have the timer note immediately how long it took and then generate the interrupt. Also restart timer immediately.

• Same issue would exist in a car when measuring speed of a wheel turning (for speedometer or anti-lock brakes).
Example #2 -- Compare

• Driving a DC motor via PWM.
  – Motors turn at a speed determined by the voltage applied.
    • Doing this in analog land can be hard.
      – Need to get analog out of our processor
      – Need to amplify signal in a linear way (op-amp?)
        • Generally prefer just switching between “Max” and “Off” quickly.
          – Average is good enough.
          – Now don’t need linear amplifier—just “on” and “off”. (transistor)
  – Need a signal with a certain duty cycle and frequency.
    • That is % of time high.
Virtual timers

• You never have enough timers.
  – Never.

• So what are we going to do about it?
  – How about we handle in software?
Virtual Timers

• Simple idea.
  – Maybe we have 10 events we might want to generate.
    • Just make a list of them and set the timer to go off for the *first* one.
      – Do that first task, change the timer to interrupt for the next task.
Problems?

- Only works for “compare” timer uses.
- Will result in slower ISR response time
  - May not care, could just schedule sooner…
Implementation issues

- Shared user-space/ISR data structure.
  - Insertion happens at least some of the time in user code.
  - Deletion happens in ISR.
    - We need critical section (disable interrupt)
- How do we deal with our modulo counter?
  - That is, the timer wraps around.
  - Why is that an issue?
- What functionality would be nice?
  - Generally one-shot vs. repeating events
  - Might be other things desired though
- What if two events are to happen at the same time?
  - Pick an order, do both…
Implementation issues (continued)

• What data structure?
  – Data needs to be sorted
    • Inserting one thing at a time
  – We always pop from one end
  – But we add in sorted order.
Timer Virtualization

- What if we don’t have enough hardware timers?
- Virtual timer library interface

```c
typedef void (*timer_handler_t)(void);

/* initialize the virtual timer */
void initTimer();

/* start a timer that fires at time t */
error_t startTimerOneShot(timer_handler_t handler, uint32_t t);

/* start a timer that fires every dt time interval*/
error_t startTimerContinuous(timer_handler_t handler, uint32_t dt);

/* stop timer with given handler */
error_t stopTimer(timer_handler_t handler);
```
typedef struct timer {
    timer_handler_t handler;
    uint32_t time;
    uint8_t mode;
    timer_t* next_timer;
} timer_t;

timer_t* current_timer;

void initTimer() {
    setupHardwareTimer();
    initLinkedList();
    current_timer = NULL;
}

error_t startTimerOneShot(timer_handler_t handler, uint32_t t) {
    // add handler to linked list and sort it by time
    // if this is first element, start hardware timer
}

error_t startTimerContinuous(timer_handler_t handler, uint32_t dt) {
    // add handler to linked list for (now+dt), set mode to continuous
    // if this is first element, start hardware timer
}

error_t stopTimer(timer_handler_t handler) {
    // find element for handler and remove it from list
}
__attribute__((__interrupt__)) void Timer1_IRQHandler() {
    timer_t * timer;
    MSS_TIM1_clear_irq();
    NVIC_ClearPendingIRQ(Timer1_IRQn);
    timer = current_timer;

    if (current_timer->mode == CONTINUOUS) {
        // add back into sorted linked list for (now+current_timer->time)
    }

    current_timer = current_timer->next_timer;

    if (current_timer != NULL) {
        // set hardware timer to current_timer->time
        MSS_TIM1_enable_irq();
    } else {
        MSS_TIM1_disable_irq();
    }

    (*timer->handler))(); // call the timer handler

    if (timer->mode != CONTINUOUS) {
        free(timer); // free the memory as timer is not needed anymore
    }
}