Topics

1) How does hardware notify software of an event?
2) What architectural design is used for bare metal?
3) How can we get accurate timing?
4) How can we use the watchdog?

Interrupt Service Routine

- IRQ:
  - Hardware generates IRQs for certain events
    - timers
    - serial data arrived
    - board losing power, etc.

- Developer writes an Interrupt Service Routine (ISR) to handle a given interrupt.
  - Hardware executes the correct ISR for an interrupt.
  - "main" thread suspended while in ISR.
  - Foreground Task...
  - Background Task...
Minimize Time in ISR

- Do minimal work to service an IRQ
  - Postpone "real" work for background thread:
    - Set flag in ISR; process flag in main().
    - Don't do blocking IO (UART) in ISR. [ex: printf]
- Why?
  - ISR often runs with interrupts disabled, so..
    - Worst-case latency can critically impede real-time performance.
    - EX: sampling audio channel regularly

ISR Programming Issues

- Global variables written in ISR should be volatile
  - Prevents compiler optimization; value may change unexpectedly.
  
  volatile static _Bool uartDataWaiting = false;
  - it's effectively multithreaded.
- Avoid using much stack in ISR
  - ISR's stack usage can be on top of normal usage.
  - In multi-threaded RTOS...
- Complications
  - Can have nested interrupts
  - Can prioritize interrupts

Common Interrupts

- Some common interrupts
  - Timer
  - UART Rx [and many UART interrupts]
    - Demo: Show bm_uartRx.c
  - GPIO changes
  - Network packet Rx
  - Analog-to-digital data sample ready
Modular Approach

- Use modules with clear interfaces
  - .h files: expose an interface.
  - .c files: implements functionality.
- In our bare metal examples, makefiles automatically compile all .c files for you.

- Names
  - Control buzzer in buzzer.h/.c
  - Give functions names such as:
    int Buzzer_init();
    int Buzzer_on(int durationInSec);
  - All global variables, and functions not shared in .h..

Common Bare Metal Architecture

Main App Module

```c
main()
init()
while (1) {
    // do background
    // work here
    if (Serial_keyWaiting())
        Serial_handleKey();
}
```

Serial Module

```c
serialRxISR()
- record RX'd char
- set key RX flag
- clear hardware
Serial_handleKey()
- process key, taking required actions.
- clear key RX flag.
```

Timer Module

```c
timerISR()
- track current "time":
  time_ms += 100;
- set timer expired flag
- call a few functions to do some very light work.
- nothing may block or take much time.
- clear hardware
```

Common Pattern

- ISR Common pattern
  1.
  2.
  - clears flag,
  - calls function to do some work
    (OK to take a “long” time to complete)
- Modularize
  - Encapsulate flag and all functions to set/check flag into a module.
  - Advanced:
    Instead of Serial module processing user inputs and do customized work in ISR, give module function pointers for callbacks from the ISR (or elsewhere) for triggering custom work from Serial module.

Demo: Fake Typer

- Demo: fakeTyper
  - Multiple files, clear interfaces
  - Trace main()
- Function
  - FakeTyper_init(void) -- from main()
  - FakeTyper_setMessage(char *message) -- from serial
  - FakeTyper_notifyOnTimerIsr(void) -- from timer
  - FakeTyper_doBackgroundWork(void) -- from main()
- Multiple Interrupts
  - If using multiple interrupts (like timer and serial) only call IntAINTCInit() once:
    It resets the Arm Interrupt Controller, which will turn off any enabled interrupts
Timers

- **Hardware Resources**
  - Processor has 8 configurable 32-bit hardware timers
- **Can generate an interrupt when timer overflows**
  - Process:
    1. Counts up to 0xFFFF FFFF + 1
    2. Generates interrupt
    3. Load TLDR on reset...

- Change TLDR’s value to control timer period
  - Larger numbers = shorter duration.

---

**Timer Diagram**

- **ISR triggers at 0xFFFF FFFF + 1 (overflow)**
- **Timer counts up at 25 MHz or 32.786 kHz**
- **ISR triggers**
- **On overflow, counter reloads to value from TLDR register**

---

**Clock Divider**

- **Clock Divider**
  - Clock has 2 possible sources: 32.768 kHz, 25Mhz
  - Can prescale (divide down) the clock.
  -...
  - Effectively uses bit N of an extra counter to drive the clock into the timer.

- **Computing Timeout**
  
  \[ \text{Frequency (Hz)} = \frac{1}{\text{Period}} \]
  
  \[ \text{Period} = \frac{xFFFF FFFF - TLDR + 1}{\text{Timer Clock Frequency}} \times 2^N \]

  - Frequency (Hz) = 1 / period (in s)
Simplified Example

- Imagine an 8-bit timer with a 32Hz clock (not MHz, Hz!)
- With no divider (N=0), what is the maximum duration of the timer?

\[
Period = \frac{xFF - x\text{00} + 1}{32} * 2^0 \\
= \frac{256}{32} \\
= 2^8 = 2^3 = 8 \text{ s}
\]

TLDR

\[+1\]

\[\text{Timer Clock Frequency}\]

\[\text{Period} = \frac{\text{xFFFF FFFF} - \text{TLDR} + 1}{2^N} \]

- Imagine an 8-bit timer with a 32Hz clock (not MHz, Hz!)
- What divider (N) and TLDR is needed to get a 200s duration (period)?
  - Since 32 = 2^5, so start with N = 5.
  - We want 200s timeout:
    \[
    \text{200} = 0xFF - \text{TLDR} + 1 \\
    \text{TLDR} = 0xFF - 200 + 1 \\
    = 55 + 1 \\
    = 56
    \]

\[
\text{Period} = \frac{256 - 56 + 1}{32} * 2^5 = 200 \text{ s}
\]

Max on 32bit timer

- With our 32 bit timer and a 25MHz clock with N=0, what is the maximum timer period?

\[
\text{Period} = \frac{\text{xFFFF FFFF} - 0 + 1}{25 \text{ MHz}} * 2^0 
\]

2s on 32bit timer

- With our 32 bit timer and a 25MHz clock, how can we get a 2s period?

\[
2s = \frac{\text{xFFFF FFFF} - \text{TLDR} + 1}{25 \text{ MHz}} \\
2s * 25 \text{ MHz} = \text{xFFFF FFFF} - \text{TLDR} + 1 \\
\text{TLDR} = \text{xFFFF FFFF} - 2 * 25 M + 1
\]

...Skipping some math...

\[
= \text{xFD05 0F80}
\]

- Note: Simply compute number of timer clock cycles, and take 2's complement.
With our 32 bit timer and a 25MHz clock and N = 2, how long a period will the previous TLDR give?

- Previous slide computed TLDR for 2s period.
- Prescale N=2 means $2^2$ slowdown
- So N=2 makes previous TLDR.

**Demo and Practical Timing**

- Demo:
  - Show bm_timer.c

- Practical Timing
  - If you want to double the period, double TLDR's distance to 0xFFFF FFFF
  - If your timing is a little off:
    Use stopwatch to time 10-100 timeouts and then scale TLDR

**Watchdog Timeout**

- Watchdog Timer
  - 32 bit counter triggers..
  - Runs on 32,768khz clock
  - Hitting watchdog resets it to WDT_WLDR's value
  - Hit by writing a changing value to WDT_WTGR (WatchdogTimerTriggerSet)

- Demo
  - bm_fakeTyper :: timers.c
  - show computation of timeout
  - show hit function.
Reset Source

- Processor knows why it last rebooted
  - Cold
  - Watchdog
  - External reset (reset button)
  - Software reset (instruction)

- Reset Register (PRM_RSTST)
  See TRM p1336 [show docs]
  - indicates reset source
  - must clear bits (write all 1's) to have next reboot show only the correct source.
  - Not in .h files:
    #define PRM_DEV 0x44E00F00 // base
    #define PRM_RSTST_OFFSET 0x08 // reg offset

Summary

- Hardware generates IRQ, serviced by ISR
  - ISR does minimal work on “foreground” task; processing/blocking IO done in main(): “background”
- Use good modular C design: encapsulation
- Hardware timers give precise timing.
  - Set reload register (TLDR) to control period:
    \[
    \text{Period} = \frac{\text{FFFF FFFF} - \text{TLDR} + 1}{\text{Timer Clock Frequency}} \times 2^N
    \]
- Watchdog timer reboots on an overflow.
- Use reset registers to track reset source.