ARM CM-3 Interrupts
Prabal Dutta
27-Oct-2015

* *

Push-button / LED example

```
            mov    r0, #0x4  % PB
            mov    r1, #0x5  % LED
loop:       ldr    r2, [r0, #0]
            str    r2 [r1, #0]
            bl     do_work
            b      loop
```

When we implement something as a fixed-function device using an FSM, we usually sample all of the inputs and state of a system on every clock edge, in parallel. When we implement something on a processor, we actually time-multiplex the processor so we cannot simultaneously sample all inputs to a system. Rather, we hope that we can clock our processor fast enough that we can sample and respond to inputs in a fast enough manner, even though we have to take turns sampling the inputs one by one.

* *

Source: https://www.youtube.com/watch?v=jMnuQMYR3Ro

Exceptions: any event that can alter the normal CPU execution flow
  - Internal events that are abnormal (e.g. to the CPU core itself)
    - Invalid instruction
    - Illegal bus access

Interrupts: subset of exceptions
  - Hardward-driven signals
    - External signals
    - Peripheral flags

What happens when an interrupt or exception occurs?
- CPU saves current context (PC, registers, etc.)
  - Stack registers
- CPU grabs address of ISR from a vector table (each INT has an index table)
- CPU jumps to interrupt service handler
- ISR does operations, clear flags, etc.
- ISR exits
- CPU restores context and returns to main flow

* *

Sources of Interrupts on the M3
- Internal (defined by ARM)
  # No : Exception Type, Priority (Def. to 0 if Prog), Description
  # 0 : N/A, N/A, No exception running
  # 1 : Reset, -3 (Highest), Reset
  # 2 : NMI, -2, NMI (external Non-Maskable Interrupt)
  # 3 : Hard Fault, -1, All fault conditions if the core's fault hndl not enabled
  # 4 : MemManage Fault, Prog, Mem mgmt fault or MPU violation or illegal mem acc.
  # 5 : Bus Fault, Prog, Bus error (prefetch abort or data abort)
  # 6 : Usage Fault, Prog, Program error
  # 7 :
  # 8 :
  # 9 :
  # 10 :
  # 11 : SVCall, Prog, Supervisor call
  # 12 : Debug Monitor, Prog, Debug mon (brk pts, watchpoints, or ext. debug reqs)

  # 13 :
  # 14 : PendSV, Prog, Pendable request for system service
  # 15 : SYSTICK, Prog, System tick timer
- External (defined by chip manufacturers)
  # 16 : IRQ #0
  # 17 : IRQ #1
  # ...: ...
  # 255: IRQ #239

- Special registers in the CM-3
  - Following special registers in the M3
    - Program Status Registers (APSR, IPSR, EPSR)
    - Interrupt Mask Registers (PRIMASK, FAULTMASK, BASEPRI)
    - Control Register (CONTROL)
  - Such registers can only be accessed via MSR and MRS instructions
    - MRS <reg>, <special_reg> ; Read special register
    - MSR <special_reg>, <reg> ; Write special register
    - Examples
      MRS r0, APSR ; r0 <- APSR
      MRS r0, IPSR ; r0 <- IPSR
      MRS r0, EPSR ; r0 <- EPSR
      MSR APSR, r0 ; APSR <- r0
      MRS r0, PSR ; r0 <- APSR | IPSR | EPSR
      MSR PSR, r0 ; APSR <- r0 & 0xF8000000 ** double check these
                   ; IPSR <- r0 & 07000001FF
                   ; EPSR <- r0 & 0x0700FC00

- The exception/interrupt number is encoded in the bottom bits of the
  IPSR (Interrupt PSR).

- PRIMASK, FAULTMASK, BASEPRI registers are used to disable interrupts
  - PRIMASK and BASEPRI are useful for temporarily disabling interrupts in
    timing critical tasks
  - FAULTMASK temporarily disables fault handling when a task has crashed.
    Once the OS core starts cleaning up, it might not want to be interrupted
    by other faults caused by the crashed process. Therefore, FAULTMASK
    gives the OS kernel time to deal with fault conditions
  - PRIMASK: a 1-bit register. When set, it allows only the NMI and hard fault
    exceptions. All other interrutps and exceptions are masked. Default is 0,
    which means no masking.
  - FAULTMASK: a 1-bit register. When set, it allows only the NMI. All other
    faults are disabled (including hard faults)
  - BASEPRI: a register of up to 8 bits (depending on the bit width impl'd
    for priority level). It defines the masking priority level. When set,
    it disables all interrupts of the same or lower level (larger priority
    values). Higher priority interrupts can still be allowed. If this is
    set to 0, the masking function is disabled (this is the default).
  - How to access them?
    MRS r0, BASEPRI
    MRS r0, PRIMASK
    MRS r0, FAULTMASK
    MSR BASEPRI, r0
    MSR PRIMASK, r0
    MSR FAULTMASK, r0
  - Note: PRIMASK, FAULTMASK, and BASEPRI cannot be set in the user access level.

- The CONTROL register is used to define the privilege level and SP selection
  - Two bits
  - CONTROL[1] : Always 0 in handler mode. Either 0 or 1 in thread (base) mode.
                 Only writable when core is in thread mode and privileged.
                 In the user state or handler mode, writing to this bit is not
                 allowed. More details (see pg 32 in Yiu's M3 book).
  - CONTROL[0] : Only writable in privileged state. Once it enters the user

state, the only way to switch back to privileged is to trigger
an interrupt and change this bit in the interrupt handler.

- Accessing
  MRS r0, CONTROL ; Read control register into r0
  MSR CONTROL, r0 ; Write r0 into control register

- Operation Mode: CPU supports two modes and two privilege levels

	Supervisor Privileged	User Unprivileged	
When running exception handler	Handler Mode CONTROL[1]=0	Not allowed	
When not running exception handler	Thread Mode CONTROL[0]=0	Thread Mode CONTROL[0]=1	(e.g. main program)

  - On reset, CPU is in thread mode with privileged level access rights.
  - To switch to user level in thread mode, set CONTROL[0]=1
  - Note, once in user level, can't switch back (e.g. writing CONTROL will fault)
  - Only way to enter a privileged mode from user mode is to cause an exception
    - Fig 3.7 (Yiu) shows this happening
    - For a user level program to enter priv mode, it must:
      - cause interrupt (e.g. SVC call)
      - write to CONTROL[0]=0 in the ISR

- Interrupt Vector Table (IVT)
  - IVT starts by default after reset at Addr=0, but is relocatable
  - Each IVT entry is four bytes (one word), starting at Addr = 0
    - So, interrupt #n's ISR address is contained in memory location
      4*n (e.g. INT13 -> *((int32_t*)(0x34)) will give you ISR addr
  - NOTE: The LSB of each interrupt vector indicates whether the
    exception to be executed in ARM or Thumb state. Since CM-3 only
    supports Thumb, the LSB of all interrupt vectors should be 1!
  - The vectors in memory. Interrupt #n The vector table entries
    # No : Memory Add : Contains the ISR address for Exception Type
    ==
    # 0 : 0x00000000 : Starting value of the MSP
    # 1 : 0x00000004 : Reset
    # 2 : 0x00000008 : NMI, -2, NMI (external Non-Maskable Interrupt)
    # 3 : 0x0000000C : Hard Fault
    # 4 : 0x00000010 : MemManage Fault
    # 5 : 0x00000014 : Bus Fault, Prog, Bus error (prefetch abort or data abort)
    # 6 : 0x00000018 : Usage Fault, Prog, Program error

    # 11 : 0x0000002C : SVCall
    # 12 : 0x00000030 : Debug Monitor

    # 14 : 0x00000038 : PendSV, Prog, Pendable request for system service
    # 15 : 0x0000003C : SYSTICK
    # 16 : 0x00000040 : IRQ #0 - Note these are defined by chip mfg
    # 17 : 0x00000044 : IRQ #1
    # ...: ...
    # 255: 0x000003FF : IRQ #239

- Configuring the interrupts using the NVIC registes
  - Before you can use them (all but NMI and Reset), you have to configure interrupts
  - The NVIC configuration registers include the following:

    - ICTR : Interrupt Controller Type Register
    - ISER : Interrupt Set-Enable Registers
    - ICER : Interrupt Clear-Enable Registers
    - ISPR : Interrupt Set-Pending Registers
    - ICPR : Interrupt Clear-Pending Registers
    - IABR : Interrupt Active Bit Registers
    - IPR : Interrupt Priority Register
  - The NVIC registers are memory-mapped with the following addresses
    - ICTR : 0xE000E004 : RO : Interrupt Controller Type Register
    - ISER : 0xE000E100 : RW : NVIC_ISER0 -
             0xE000E11C : RW : NVIC_ISER7
    - ICER : 0xE000E180 : RW : NVIC_ICER0 -
             0xE000E19C : RW : NVIC_ICER7
    - ISPR : 0xE000E200 : RW : NVIC_ISPR0 -
             0xE000E21C : RW : NVIC_ISPR7
    - ICPR : 0xE000E280 : RW : NVIC_ICRP0 -
             0xE000E29C : RW : NVIC_ICRP0
    - IABR : 0xE000E300 : RO : NVIC_IABR0 -
             0xE000E31C : RO : NVIC_IABR7
    - IPR : 0xE000E400 : RW : NVIC_IRP0 -
             0xE000E4EC : RW : NVIC_IPR59

    - AIRC : 0xE000ED0C :   : Application Interrupt and Reset Control

  - To use them: set the bit that corresponds to the interrupt number,
    starting at the base address of the register.

    ```
    void enable_interrupts(int x) {
      uint32_t base = 0xE000E100;
      uint32_t idx  = x >> 5;
      uint32_t bit  = x & 0x1F;
      uint32_t mask = 1 << bit;
      *((uint32_t*)(base+4*idx)) = mask;
    }
    ```

  - Note: We *don't* need to read, modify, write these registers because the
    the only thing that matters is writing a 1 (writing a 0 does nothing).

- Interrupt Priorities
  - Higher priority <-> smaller number in priority level
  - Some exceptions have fixed priorities (i.e. reset, NMI, hard fault: neg #'s)
  - CM-3 supports
    - Three (3) fixed highest priority levels
    - Up to 256 programmable priority levels
      - Not all are implemented (minimum is three levels of priority)
      - If fewer are implemented, just use high order bits of Priority Level Reg.
      - Supports better code portability among processors (better than clipping MSB)

- Boot is an "exception" (see Fig 3.19)