Week 7. SWIs & Memory Mapped I/O

18-342: Fundamentals of Embedded Systems

Rajeev Gandhi
INI & ECE
Carnegie Mellon University

Carnegie Mellon

Electrical & Computer ENGINEERING

Overview of the week

- How to install a SWI handler?
- Re-entrant SWI handlers
- Interfacing hardware and I/O devices to processors
  - Memory-mapped I/O vs. Port-mapped I/O
- Differences between registers and memory-mapped registers
- Introduction to Timers
  - Timers on the gumstix board
  - Memory-mapped registers of the timers
- Watchdog timers

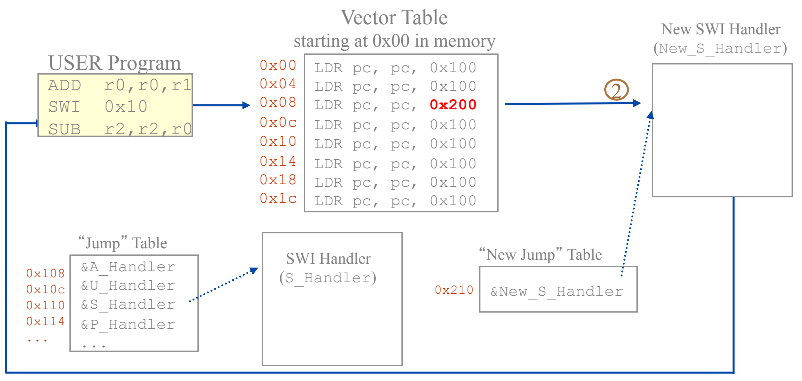
- Announcement:
  - No lecture on Thursday
  - Possible lab 2 recitation by the TAs on Friday

Adding your custom SWIs

- Suppose you are given an OS with some of the standard syscalls already defined
- You want to add a few more custom syscalls that you want to be able to provide your application developers
- How do we add our custom SWIs without losing the SWI handling already provided?
  - Modify the code for the `C_SWI_Handler` if you have the source code
  - Wire in your SWI Handler before the system SWI handler

Installing your SWI handler

- Two ways to install your own SWI handler
- Method 1
  - Change the address of `S_Handler` (stored in `softvec)` to point to the new SWI handler
- Method 2 (done in lecture)
  - Define a new location for `softvec`
  - Store the address of the new SWI handler in `softvec`
  - Change the instruction stored in vector table location 0x08 to point to the new `softvec`



```
ADD   r0,r0,r1
SWI   0x10
SUB   r2,r2,r0
```

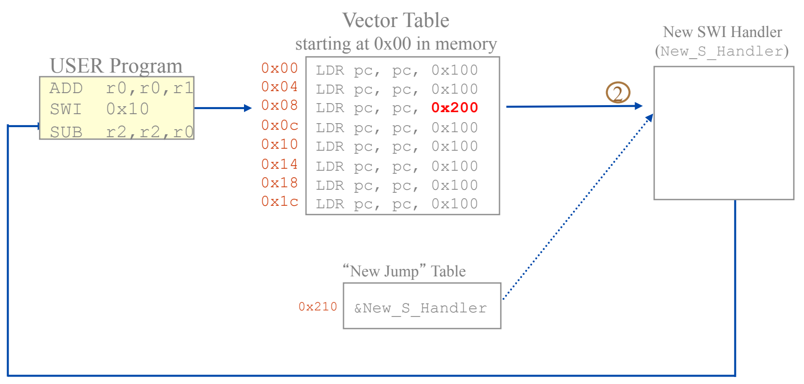
```
0x00  LDR pc, pc, 0x100
0x04  LDR pc, pc, 0x100
0x08  LDR pc, pc, 0x200
0x0c  LDR pc, pc, 0x100
0x10  LDR pc, pc, 0x100
0x14  LDR pc, pc, 0x100
0x18  LDR pc, pc, 0x100
0x1c  LDR pc, pc, 0x100
```

```
0x108  &A_Handler
0x10c  &U_Handler
0x110  &S_Handler
0x114  &P_Handler
...    ...
```

```
0x210  &New_S_Handler
```

Installing the SWI handler

- Load `pc` method
  - Compute the offset to be used in the `LDR` instruction (at 0x08)
    - Take the address of the word containing the address of the new SWI handler
    - Subtract the address of the SWI exception vector in the vector table
    - Subtract 8 bytes
    - Check that the result can be represented in 12 bits
  - Logically OR this with 0xe59FF000 (the opcode for `LDR pc, [pc, #0]`)
  - Store this new `LDR` instruction in the SWI exception vector

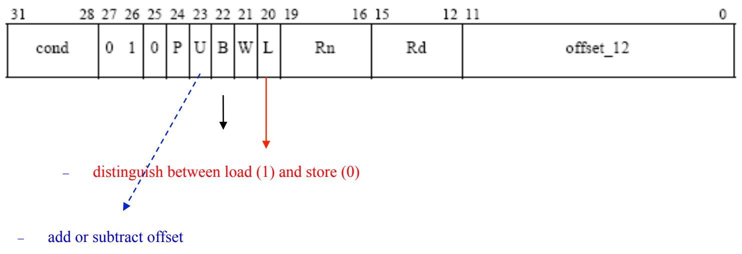


`LDR` Instruction

- `LDR rd, [rn, #offset]`   `rd` ← `mem[rn+offset]`

  – Decimal numbers prefixed by #

- `rd, rn` can be any register (r0 – r15)
- Binary encoding of `LDR/STR` instruction (with immediate offset addressing mode)



Loading the Vector Table

```
unsigned Install_Handler(unsigned location, unsigned int *vector)
/*Updates the contents of 'vector' stored at 0x08 to contain LDR pc, [pc, #offset] instruction to cause
    long branch to              'location' */
/*Function returns the original contents of 'vector' */
{
    unsigned offset;
   unsigned vec, oldvec;
    offset = ((unsigned) location - (unsigned) vector - 0x8)
    if(offset & 0xFFFFF000)  /* check if the offset can be represented using 12 bits */
    {     printf("Installation of handler failed");
          exit(0); }
    vec =(offset|0xe59FF000);/* vec now contains LDR pc, [pc, #offset] */
    oldvec = *vector;       /* sa                        )8 */
    *vector = vec; /* replace the contents of 0x08 with the new LDR instruction */
    return(oldvec);                       ction at 0x08 for chaining */
}
```

location = 0x210

vector = 0x08

*vector = 0xe59ff100

vec = 0xe59ff200

*vector = 0xe59ff200

oldvec = 0xe59ff100

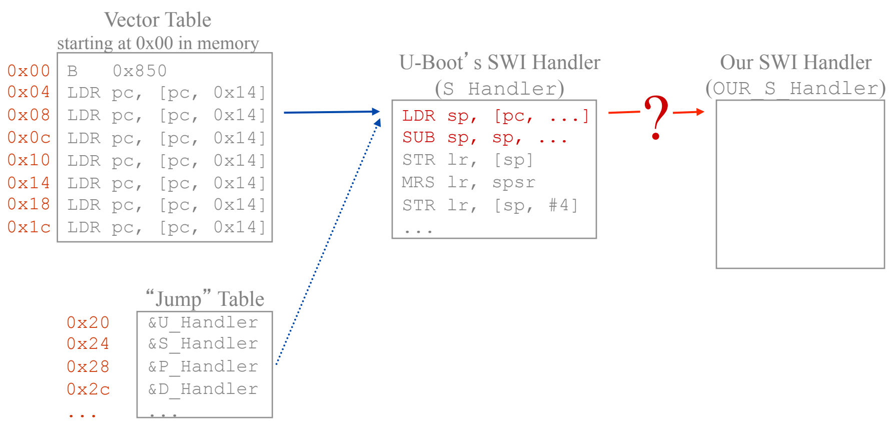
The following code calls function

```
unsigned *vector= (unsigned *) 0x08;
unsigned *location= (unsigned *) 0x210; /*The address of the new softvec */

Install_Handler((unsigned) location, vector);




*location= (unsigned) New_S_Handler;//Store the address of your New_S_Handler in swiaddr
```

Source : *Jumpstart Programming Techniques & ADS Developer's Guide*

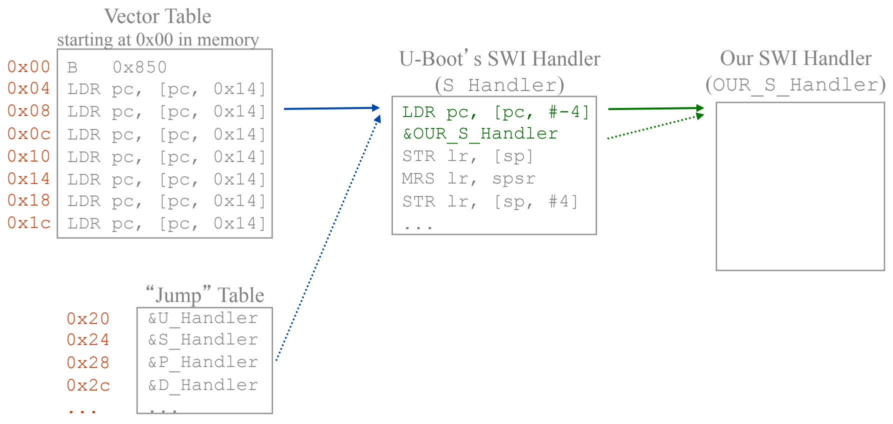
Installing the SWI Handler – Our Way

- **Problem**: Can't write to either the Vector Table or the "Jump" Table (both in ROM)
- **Solution**: Hijack the first few instructions of U-Boot's SWI Handler
  - Redirect the processor to our own handler
  - Only eight bytes guaranteed to be available (arbitrary lab restriction)



Installing the SWI Handler – Our Way

- Which instructions to use?
  - Analogous situation to the SWI vector in the Vector Table
  - One instruction must transfer control to Our SWI Handler
  - Eight bytes available → four bytes for instruction, four bytes for address



Reentrant Exception Handlers

- Reentrant programming – used to describe code which can have concurrent invocations which should ideally not interfere with each other

- What happens if you want to call another SWI from your SWI handler?
  - When an SWI occurs, the ARM
    - Sets appropriate CPSR mode bits to 10011 (`svc` mode)
    - Copies CPSR into `spsr_svc`
    - Stores return address (`pc` – 4) in `lr_svc`
    - Sets `pc` to vector address 0x08

- Need to save certain registers (that do not need to be saved otherwise)

- Do you see any issues here?

Program Status Register Instructions

- Two instructions to directly control a Program Status Register
- MRS
  - Transfers the contents of either the `cpsr` or the `spsr` into a register
- MSR
  - Transfers the contents of a register into the `cpsr` or the `spsr`

```
PRE    cpsr = nzcvqIFt_SVC

   MRS  r1, cpsr
   BIC   r1, r1, #0x080
   MSR  cpsr, r1

POST   cpsr = nzcvqiFt_SVC
```

A Reentrant `SWI` Handler for Lab2

```
import C_SWI_Handler
export S_Handler
S_Handler
    SUB      sp, sp, #4            ; leave room on stack for SPSR
    STMFD    sp!, {r0-r12, lr} ; store user's gp registers
    MRS      r2, spsr              ; get SPSR into gp registers
    STR      r2, [sp, #14*4]       ; store SPSR above gp registers
    MOV      r1, sp                ; pointer to parameters on stack
    LDR      r0, [lr, #-4]         ; extract the SWI number
    BIC      r0,r0,#0xff000000 ; get SWI # by bit-masking
    BL       C_SWI_Handler         ; goto handler (see prev lecture)
    LDR      r2, [sp, #14*4]       ; restore SPSR (NOT “sp!”)
    MSR      spsr, r2              ; restore SPSR from r2
    LDMFD    sp!, {r0-r12, lr} ; unstack user's registers
    ADD      sp, sp, #4          ; remove space used to store SPSR
    MOVS     pc, lr                ; return from handler
```

Source : *Jumpstart Programming Techniques*

Interfacing Peripheral Devices to the Processor

- So far we have looked at the ARM instruction set, programmer' s model
- Up next: How do we interface peripheral devices to the processor?
- We will look at
  - How do we set up (configure) peripheral devices?
  - How do we check the status of the devices?
  - How do we communicate with peripheral devices?

Software Addressing of I/O Devices

- Two ways of addressing I/O devices from the CPU
  - Memory-mapped I/O
    - Devices are mapped in memory address space, e.g., the 7-segment LED
    - Standard load and store instruction can manipulate devices
  - Port-mapped I/O
    - Devices are **not** kept in memory address space
    - **Special processor instructions** request data from devices
      - Example

```
IN  REG, PORT
OUT REG, PORT
```

- Which one is better?
  - Memory-mapped I/O uses the same load/store paradigm, but costs some of the address space
  - Full address space is available for port-mapped I/O, but requires extra instructions and control signals from the CPU

Example

- Device manufacturer will typically specify the registers that will be used to set up and control the device
- The (board) hardware designers will specify the address of these registers on your system
- You will write code to set up the devices, use the devices

Example

- Example: Suppose your hardware board has a 7-segment LED display (not present on gumstix)
  - Assume that the device manufacturer specifies that there is a register that can be written to display a character on the LED
  - The device manufacturer will also provide a table that determines the contents of the register for each character to be displayed)
  - The hardware (board) designer will specify the address where this register is mapped (assume that you are given that the device is mapped at 0x20200000
- If you wanted to display a character "P" on the LED, the code you will write will look like

```
LDR   R0,=0x20200000
MOV   R1,#0x0C
STRB  R1,[R0]
```

```
// LED character map
#define LEDcharP       0x0c
#define LEDcharH       0x09
#define LEDcharA       0x08
…
```

Writing Code to Access the Devices

- Portability issues – hard-coding the address may pose problems in moving to a new board where the address of the register is different

```
LDR   R0,=0x20200000
MOV   R1,#0x0C
STRB  R1,[R0]
```

- **Should** use "EQU" assembler directive

```
BASE  EQU 0x20200000
LDR   R0, =BASE
```

- *Can* also access devices using C programs
  - C pointers can be used to write to a specific memory location

```
 unsigned char *ptr;
 ptr  = (unsigned char *) 0x20200000;
*ptr = (unsigned char) 0x0C;
```

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

```
void test()
{
        int *ptr = (int *) 0x20200000;
        int val=0;
        do {
                val+=*ptr;
        } while(val <= 100);

}
```

Initial value: r1 contains 0x20200000
Assume r0 contains val

```
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
```

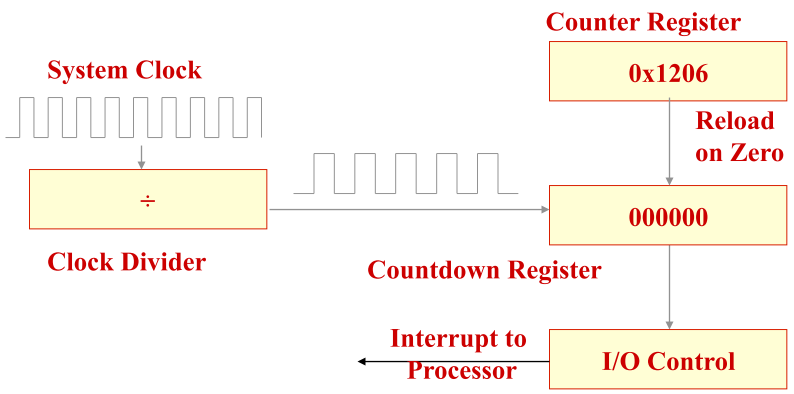
```
void test()
{
        volatile int *ptr=(int *) 0x2020…;
        int val=0;
        do {
                val+=*ptr;
        } while(val <= 100);

}
```

Initial value: r2 contains 0x2020…
Assume r0 contains val

```
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
```

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 0x40A0_0000 0x40A0_0004 0x40A0_0008 0x40A0_000C | OS Timer Match Register 0-3 (OSMR3, OSMR2, OSMR1, OSMR0) | 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
	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	0

Bits	Name	Description
<31:0>	OSMV	OS Timer Match Value. The value compared against the OS timer counter.

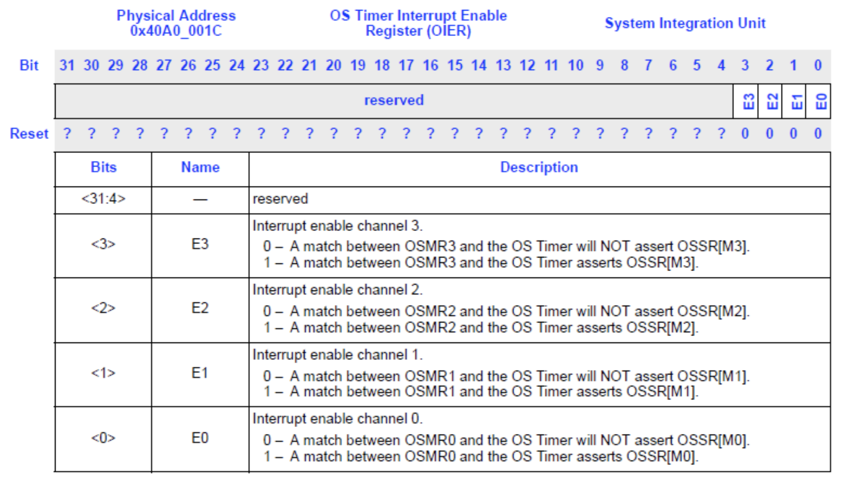
OSSR

Physical Address 0x40A0_0014 | OS Timer Status Register (OSSR) | 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																												M3	M2	M1	M0
Reset	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	0	0	0	0

Bits	Name	Description
<31:4>	—	reserved
<3>	M3	Match status channel 3. If OIER[3] is set then: 0 – OSMR[3] has NOT matched the OS timer counter since last being cleared. 1 – OSMR[3] has matched the OS timer counter.
<2>	M2	Match status channel 2. If OIER[2] is set then: 0 – OSMR[2] has NOT matched the OS timer counter since last being cleared. 1 – OSMR[2] has matched the OS timer counter.
<1>	M1	Match status channel 1. If OIER[1] is set then: 0 – OSMR[1] has NOT matched the OS timer counter since last being cleared. 1 – OSMR[1] has matched the OS timer counter.
<0>	M0	Match status channel 0. If OIER[0] is set then: 0 – OSMR[0] has NOT matched the OS timer counter since last being cleared. 1 – OSMR[0] has matched the OS timer counter.

OIER



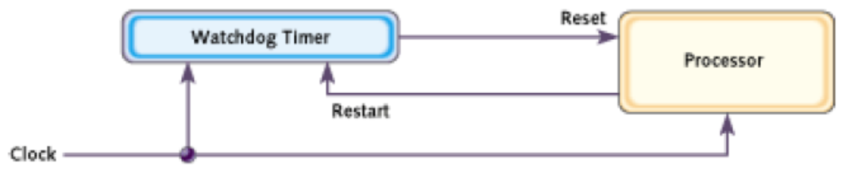
Physical Address 0x40A0_001C | OS Timer Interrupt Enable Register (OIER) | System Integration Unit

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	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	0	0	0	0

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].

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
  - What to do if some functions in the for loop *can* take longer than the maximum timer interval