ECE3140 / CS3420
Embedded Systems

Lecture 4. Input/Output

Prof. José F. Martínez
Interacting with the Physical World

- How does a processor interact with the physical world?
  - . . . what happens to make the LED blink?

- We need a way for the processor to communicate with other devices

- Lots of mechanisms possible
Outline

- General I/O concepts
  - I/O ports and physical connections
  - General-Purpose Input Output (GPIO)
  - Software accesses to I/O ports
  - Polling vs. interrupts

- Interrupt/exception handling
  - Exception handling in ARM

- Handling multiple input sources

- References (ARM specific)
  - Chapter 2 and Chapter 4 (pp.99-117)
    - Embedded Systems Fundamentals with ARM Cortex-M based Microcontrollers
  - K64 Sub-Family Reference Manual
  - FRDM-K64F Freedom Module User’s Guide
Source: Kinetis K64F Sub-Family Data Sheet
K64F Peripherals

Figure 1. FRDM-K64F block diagram

Source: FRDM-K64F Freedom Module User’s Guide
**I/O Ports**

- I/O locations/groups are typically called **ports**
  - Read/write
  - Data/control
LED Connection for FRDM-K64F

Table 6. LED signal connections

<table>
<thead>
<tr>
<th>LED</th>
<th>K64</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED</td>
<td>PTB22/SPI2_SOUT/FB_AD29/CMP2_OUT</td>
</tr>
<tr>
<td>BLUE</td>
<td>PTB21/SPI2_SCK/FB_AD30/CMP1_OUT</td>
</tr>
<tr>
<td>GREEN</td>
<td>PTE26/ENET_1588_CLKIN/UART4_CTS_b/RTC_CLKOUT/USB0_CLKIN</td>
</tr>
</tbody>
</table>

Figure 14. Tricolor LED

Source: FRDM-K64F Freedom Module User’s Guide
Source: Kinetis K64F Sub-Family Data Sheet
General-Purpose Input and Output

- **GPIO = General-purpose input and output (digital)**
  - Input: program can determine if input signal is a 1 or a 0
  - Output: program can set output to 1 or 0

- Can use this to interface with external devices

- Example
  - Input: switch
  - Output: LEDs
What’s a One? A Zero?

- Signal’s value is determined by voltage
- Input threshold voltages depend on supply voltage $V_{DD}$
- Exceeding $V_{DD}$ or GND may damage chip
GPIO Port Bit Circuitry in MCU

- **Control**
  - Direction
  - MUX

- **Data**
  - Output (different ways to access it)
  - Input
Accessing I/O Ports

- How to access an I/O port from software?
  - Special instructions
  - Special registers
  - Special memory locations
Option 1: I/O Instructions

- Special instruction in the ISA for input/output

- Example: Z80 ISA
  - `out (243), A`
    - Output value stored in register A to port 243
  - `in A,254`
    - Read the value in port 254 and store it in register A

- Ports are special operands
Option 2: Special Registers

- Special register in the ISA for input/output

- Example: SNAP ISA
  - `add $15,$1,$2`
    - Register 15 is mapped for output operations
  - `add $1,$15,$2`
    - Register 15 is also mapped for input operations

- I/O operation determined by writing specific values to $15.
Option 3: Memory Mapped I/O

- Memory mapped I/O: Reads and writes to specific memory locations correspond to I/O operations

<table>
<thead>
<tr>
<th>System 32-bit base address</th>
<th>Slot number</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4006_6000</td>
<td>102</td>
<td>I²C 0</td>
</tr>
<tr>
<td>0x4006_7000</td>
<td>103</td>
<td>I²C 1</td>
</tr>
<tr>
<td>0x4006_8000</td>
<td>104</td>
<td>—</td>
</tr>
<tr>
<td>0x4006_9000</td>
<td>105</td>
<td>—</td>
</tr>
<tr>
<td>0x4006_A000</td>
<td>106</td>
<td>UART 0</td>
</tr>
<tr>
<td>0x4006_B000</td>
<td>107</td>
<td>UART 1</td>
</tr>
<tr>
<td>0x4006_C000</td>
<td>108</td>
<td>UART 2</td>
</tr>
<tr>
<td>0x4006_D000</td>
<td>109</td>
<td>UART 3</td>
</tr>
</tbody>
</table>

Source: ARM Cortex-M4
Privilege Levels

- Which software should manages the IO?

- Processors often support multiple privilege levels
  - Supervisor / “Privileged” in Cortex-M4
    - Access to all resources
  - User / “Unprivileged” in Cortex-M4
    - Limited access to certain instructions and memory/peripheral

- Which mode should a processor starts running in?
Handling Outputs

- On-chip registers connected to I/O pins

- Implementing output instructions:
  - Write register for output values
  - Change in state appears on the pins
    - ... after a small delay
Handling Input

- Need communication disciplines

- How does software know if the value is valid?
  - Use a valid bit
    - 9-bit input, with 8-bits of data
    - Toggle 9th bit to indicate new data
  - Use encoded data
    - One-hot encoding
    - 01 = false, 10 = true, 00 = no data

- How does software know when a new input is ready?
Option 1: Polling

- Use software to check
  - Keep reading the value in a loop

```c
while (1) {
    // read a GPIO port
    // check the value
}
```
Option 2: Interrupts

- A peripheral device notifies a processor that there is a new input
  - Run Interrupt Service Routine (ISR)
  - Return to the original (interrupted) program
Polling vs. Interrupt

- **Polling**
  - Simple
  - Slow - need to explicitly check to see if switch is pressed
  - Wasteful of CPU time - the faster a response we need, the more often we need to check
  - Scales badly - difficult to build system with many activities which can respond quickly. Response time depends on all other processing.

- **Interrupt**
  - Efficient - code runs only when necessary
  - Fast - hardware mechanism
  - Scales well
  - More complex to implement
  - Requires additional hardware
Interrupt Handling Sequence

- Main code is running
- Interrupt trigger occurs
- Processor does some hard-wired processing
- Processor executes ISR, including return-from-interrupt instruction at the end
- Processor resumes the main code

Source: ARM Tutorial Slides
Interrupt Handling vs. Function Calls

- How is calling an exception/interrupt handler different from a subroutine call? How to divide work between ‘caller’ (interrupt SW) and ‘callee’ (ISR)?

Input parameters:

Caller-saved vs. callee-saved registers:

Privileged mode:
Interrupt Handling

- **Enter an interrupt handler**
  - HW saves PC [and possibly more registers]
  - [HW puts an interrupt/exception number in a register]
  - HW switches to a privileged ‘interrupt handler’ mode
  - HW jumps to the PC specified in the interrupt vector table

- **Interrupt Service Routine (ISR)**
  - [ISR saves/restores additional registers that will be used]
  - [ISR may disable interrupts while it’s running]
  - ISR finds out the reason for an interrupt and processes it
  - ISR runs ‘return-from-interrupt’ instruction

- **Exit an interrupt handler**
  - HW restored HW-saved registers
  - HW switches the privilege mode back
  - HW jumps to the saved PC
Interrupts, Exceptions, and Traps

- Broadly, exceptions may refer to all types of events that interrupt a normal program execution

- Interrupts (asynchronous)
  - I/O device interrupt, reset, etc.

- Exceptions (synchronous)
  - Arithmetic overflow, FP anomaly, page fault, misaligned memory access, memory protection violation, illegal instruction, etc.

- System calls / Traps (synchronous)
  - SVCall
Exception Processing Sequence

- Main code is running
- Interrupt trigger occurs
- Processor does some hard-wired processing
- Processor executes ISR, including return-from-interrupt instruction at the end
- Processor resumes the main code

Source: ARM Tutorial Slides
CPU’s Hardwired Exception Processing

1. Finish current instruction (except for lengthy instructions)
2. Push context (8 32-bit words) onto current stack (MSP or PSP)
   - xPSR, Return address, LR (R14), R12, R3, R2, R1, R0
3. Switch to handler/privileged mode, use MSP
4. Load PC with address of exception handler
5. Load LR with EXC_RETURN code
6. Load IPSR with exception number
7. Start executing code of exception handler

Usually 16 cycles from exception request to execution of first instruction in handler

Source: ARM Tutorial Slides
2. Push Context onto Current Stack

- Two SPs: Main (MSP), process (PSP)
- Which is active depends on operating mode, CONTROL register bit 1
- Stack grows toward smaller addresses
3. Switch to Handler/Privileged Mode

- **Thread mode**
  - Privileged or unprivileged
  - MSP or PSP

- **Handler mode**
  - Privileged
  - Always uses Main SP

Source: ARM Tutorial Slides
Update IPSR with Exception Number

PORTD_IRQ is Exception number 0x2F
(interrupt number + 0x10)
### Vector Table Example

<table>
<thead>
<tr>
<th>Exception number</th>
<th>IRQ number</th>
<th>Vector</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16+n</td>
<td>IRQn</td>
<td>$0x40+4n$</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>IRQ2</td>
<td>$0x48$</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>IRQ1</td>
<td>$0x44$</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>IRQ0</td>
<td>$0x40$</td>
</tr>
<tr>
<td>15</td>
<td>-1</td>
<td>SysTick, if implemented</td>
<td>$0x3C$</td>
</tr>
<tr>
<td>14</td>
<td>-2</td>
<td>PendSV</td>
<td>$0x38$</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>SVCall</td>
<td>$0x2C$</td>
</tr>
<tr>
<td>11</td>
<td>-5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- PORTD ISR is IRQ #31 (0x1F), so vector to handler begins at $0x40+4*0x1F = 0xBC$
- Why is the vector odd? 0x0000_0455
- LSB of address indicates that handler uses Thumb code

Source: ARM Tutorial Slides
4. Load PC With Address Of Exception Handler

- **Reset Interrupt Service Routine**
  - Port D ISR
  - Port A ISR
- **Non-maskable Interrupt Service Routine**
  - Port D Interrupt Vector
  - Port A Interrupt Vector
- **Non-Maskable Interrupt Vector**
  - Reset Interrupt Vector

<table>
<thead>
<tr>
<th>Vector Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000_0004</td>
<td>PORTD_IRQHandler</td>
</tr>
<tr>
<td>0x0000_00B8</td>
<td>PORTA_IRQHandler</td>
</tr>
<tr>
<td>0x0000_00BC</td>
<td>NMI_IRQHandler</td>
</tr>
<tr>
<td>0x0000_0008</td>
<td>PORTD_IRQHandler</td>
</tr>
<tr>
<td>0x0000_0008</td>
<td>PORTA_IRQHandler</td>
</tr>
<tr>
<td>0x0000_0004</td>
<td>NMI_IRQHandler</td>
</tr>
<tr>
<td>0x0000_0004</td>
<td>start</td>
</tr>
</tbody>
</table>

Source: ARM Tutorial Slides
5. Load LR With EXC_RETURN Code

<table>
<thead>
<tr>
<th>EXC_RETURN</th>
<th>Return Mode</th>
<th>Return Stack</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF_FFF1</td>
<td>0 (Handler)</td>
<td>0 (MSP)</td>
<td>Return to exception handler</td>
</tr>
<tr>
<td>0xFFFF_FFF9</td>
<td>1 (Thread)</td>
<td>0 (MSP)</td>
<td>Return to thread with MSP</td>
</tr>
<tr>
<td>0xFFFF_FFFD</td>
<td>1 (Thread)</td>
<td>1 (PSP)</td>
<td>Return to thread with PSP</td>
</tr>
</tbody>
</table>

- **EXC_RETURN** value generated by CPU to provide information on how to return
  - Which SP to restore registers from? MSP (0) or PSP (1)
    - Previous value of SPSEL
  - Which mode to return to? Handler (0) or Thread (1)
    - Another exception handler may have been running when this exception was requested
6. Start Executing Exception Handler

- Exception handler starts running, unless preempted by a higher-priority exception.

- Exception handler may save additional registers on stack:
  - For example, handler may call a subroutine and save LR and R4 in the following example.

Source: ARM Tutorial Slides
Exiting an Exception Handler

1. Execute instruction triggering exception return processing
2. Select return stack, restore context from that stack
3. Resume execution of code at restored address

Source: ARM Tutorial Slides
1. Execute Instruction for Exception Return

- No “return from interrupt” instruction
- Use regular instruction instead
  - BX LR - Branch to address in LR by loading PC with LR contents
  - POP {..., PC} - Pop address from stack into PC
- … with a special value EXC_RETURN loaded into the PC to trigger exception handling processing
  - BX LR used if EXC_RETURN is still in LR
  - If EXC_RETURN has been saved on stack, then use POP

Source: ARM Tutorial Slides
2. Select Stack, Restore Context

- Check EXC_RETURN to determine from which SP to pop the context

<table>
<thead>
<tr>
<th>EXC_RETURN</th>
<th>Return Stack</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF_FFF1</td>
<td>0 (MSP)</td>
<td>Return to exception handler with MSP</td>
</tr>
<tr>
<td>0xFFFF_FFF9</td>
<td>0 (MSP)</td>
<td>Return to thread with MSP</td>
</tr>
<tr>
<td>0xFFFF_FFFD</td>
<td>1 (PSP)</td>
<td>Return to thread with PSP</td>
</tr>
</tbody>
</table>

- Pop the registers from that stack

Source: ARM Tutorial Slides
Outline

- Sharing data between ISR and other threads
  - Volatile variables
  - Non atomic updates

- Disabling interrupts

- Handling multiple input sources
Example: Digital Clock

```c
int Minutes; // Updated every minute via timer ISR

// main program
  int hour, min;

...

while (1) {
    hour = Minutes/60; // i1
    min = Minutes%60; // i2
    DisplayTime(hour, min); // Displays hh:mm
}

// ISR for timer interrupts (every minute)
  ...
  Minutes++;```

Problem: Variables Kept in Registers

- Compilers assume that variables in memory do not change spontaneously, and optimize based on that belief
  - *Don’t reload a variable from memory if current function hasn’t changed it*
  - Read variable from memory into register (faster access)
  - Write back to memory at end of the procedure, or before a procedure call, or when compiler runs out of free registers

- This optimization can fail
  - Example: reading from input port, polling for key press
    - while (SW_0) ; will read from SW_0 once and reuse that value
    - Will generate an infinite loop triggered by SW_0 being true

- Variables for which it fails
  - Memory-mapped peripheral register – register changes on its own
  - Global variables modified by an ISR – ISR changes the variable
  - Global variables in a multithreaded application – another thread or ISR changes the variable
The Volatile Directive

- Need to tell compiler which variables may change outside of its control
  - Use volatile keyword to force compiler to reload these vars from memory for each use

```c
volatile unsigned int num_ints;

volatile int * var; // or
int volatile * var;
```

- Now each C source read of a variable (e.g. status register) will result in an assembly language LDR instruction
Problem: Non-Atomic Updates

volatile int Minutes; // Updated every minute via timer ISR
int hour, min;
...
hour = Minutes/60; // i1
min = Minutes%60; // i2
DisplayTime(hour,min); // Displays hh:mm

Q: Assume Minutes=119 before i1. What are possible outcomes of this program?

A: 1:59
B: 2:00
C: 1:00
D: A or B
E: A or B or C
Disabling Interrupts

Two major types of interrupts:

- **Non-maskable**
  - Can’t disable them
  - Example: reset

- **Maskable**
  - User-controlled
  - Can selectively activate them
Core Exception Mask Register (ARM)

- Similar to “Global interrupt disable” bit in other MCUs

- PRIMASK - Exception mask register (CPU core)
  - Bit 0: PM Flag
    - Set to 1 to prevent activation of all exceptions with configurable priority
    - Clear to 0 to allow activation of all exception
  - Access using CPS, MSR and MRS instructions
  - Use to prevent data race conditions with code needing atomicity

- CMSIS-CORE API
  - void __enable_irq() - clears PM flag
  - void __disable_irq() - sets PM flag
  - uint32_t __get_PRIMASK() - returns value of PRIMASK
  - void __set_PRIMASK(uint32_t x) - sets PRIMASK to x
Multiple IO Devices

- Which device raised an interrupt?
- Which interrupt service routine to run?

Common approaches
- (Polling)
- Interrupt Vector Table (IVT) + Multiple IRQ signals
- Interrupt + Polling
- Daisy chain
Prioritization (ARM)

- Exceptions are prioritized to order the response simultaneous requests (smaller number = higher priority)

- Priorities of some exceptions are fixed
  - Reset: -3, highest priority
  - NMI: -2
  - Hard Fault: -1

- Priorities of other (peripheral) exceptions are adjustable
  - Value is stored in the interrupt priority register (IPR0-7)
    - 0x00
    - 0x40
    - 0x80
    - 0xC0

Source: ARM Tutorial Slides
Special Cases of Prioritization (ARM)

- Simultaneous exception requests?
  - Lowest exception type number is serviced first

- New exception requested while a handler is executing?
  - New priority higher than current priority?
    - New exception handler preempts current exception handler
  - New priority lower than or equal to current priority?
    - New exception held in pending state
    - Current handler continues and completes execution
    - Previous priority level restored
    - New exception handled if priority level allows
Daisy Chain

- Wiring scheme where multiple devices are wired together in sequence or in a ring