ECE3140 / CS3420
Embedded Systems

Lecture 4. Input/Output

Prof. José F. Martínez

Cornell University

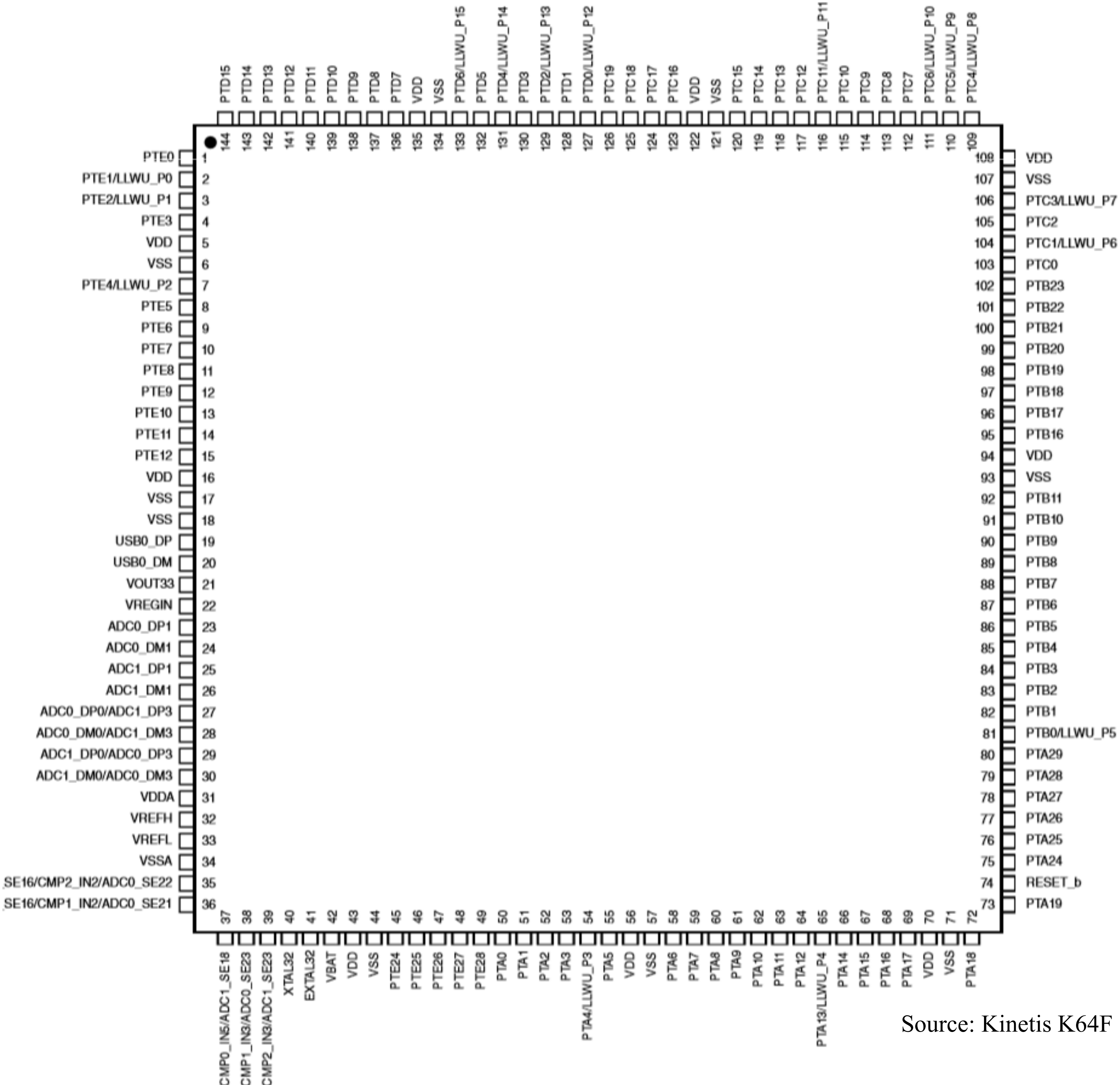
Interacting with the Physical World

- 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

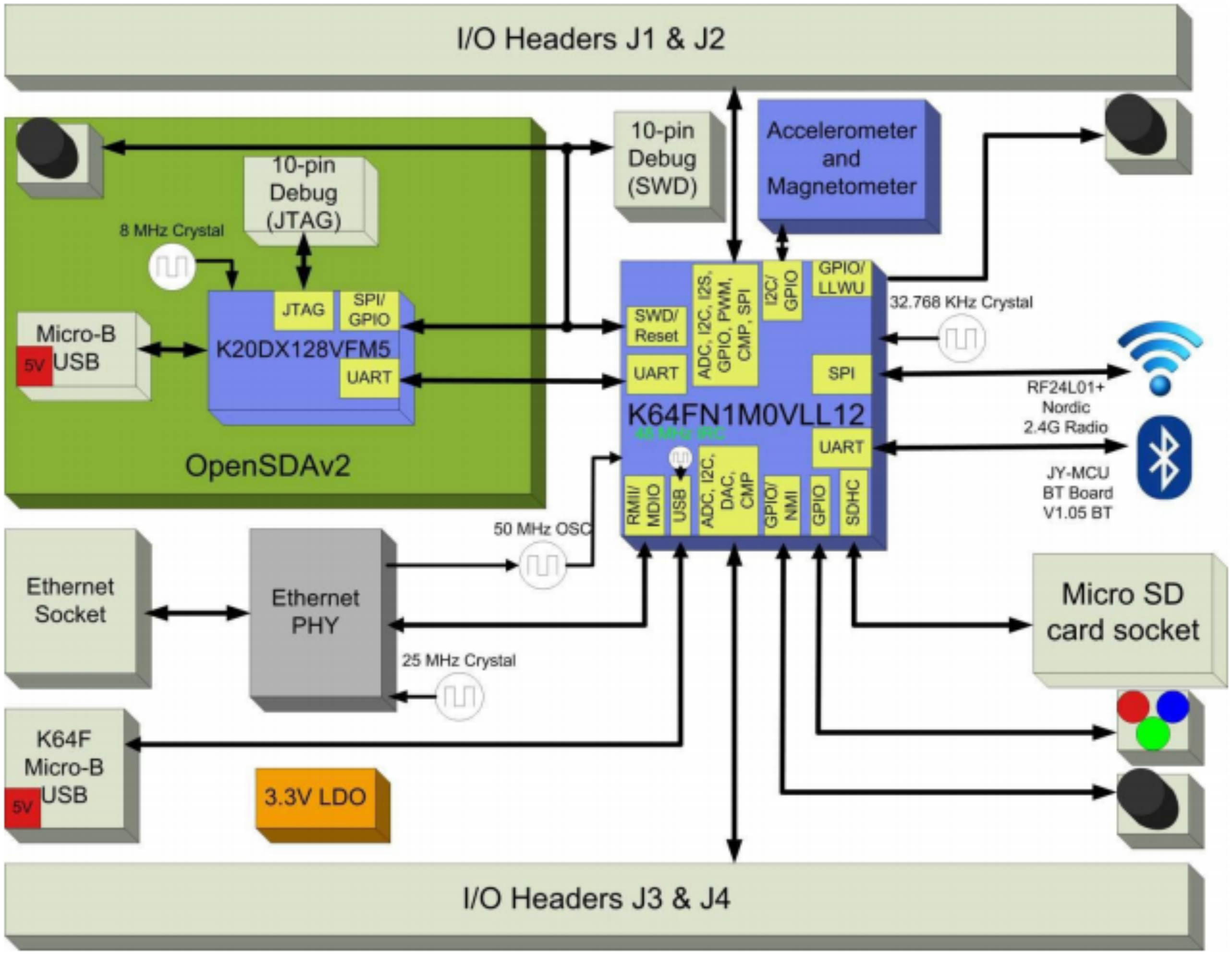
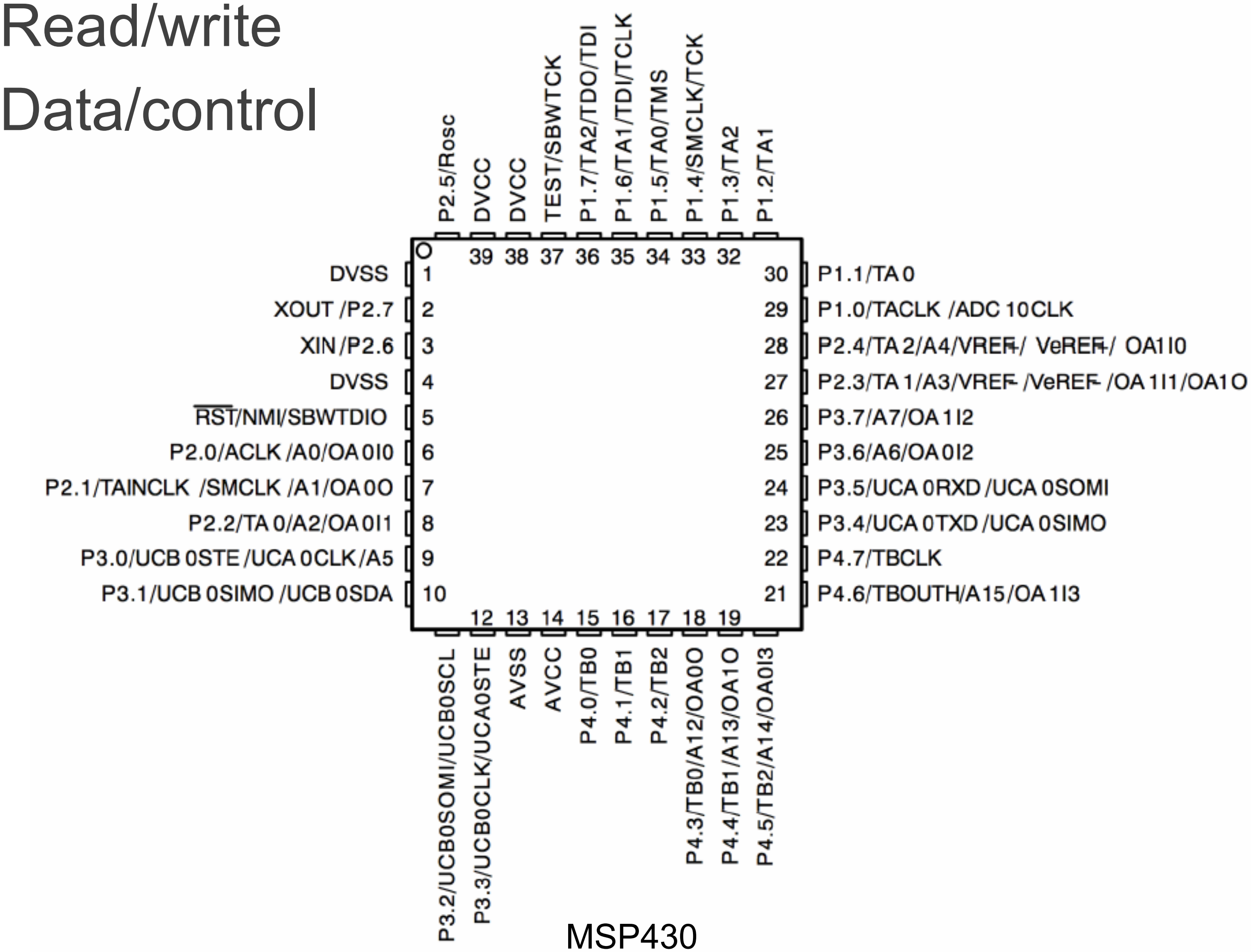


Figure 1. FRDM-K64F block diagram

Source: FRDM-K64F Freedom Module User’s Guide

I/O Ports

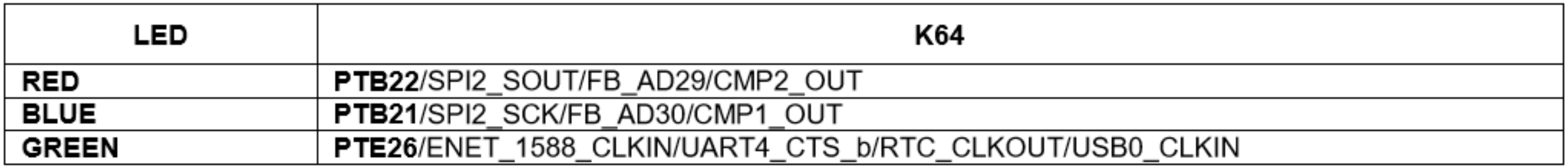
- I/O locations/groups are typically called ports
  - Read/write
  - Data/control



MSP430

LED Connection for FRDM-K64F

Table 6. LED signal connections



LED	K64
RED	**PTB22**/SPI2_SOUT/FB_AD29/CMP2_OUT
BLUE	**PTB21**/SPI2_SCK/FB_AD30/CMP1_OUT
GREEN	**PTE26**/ENET_1588_CLKIN/UART4_CTS_b/RTC_CLKOUT/USB0_CLKIN

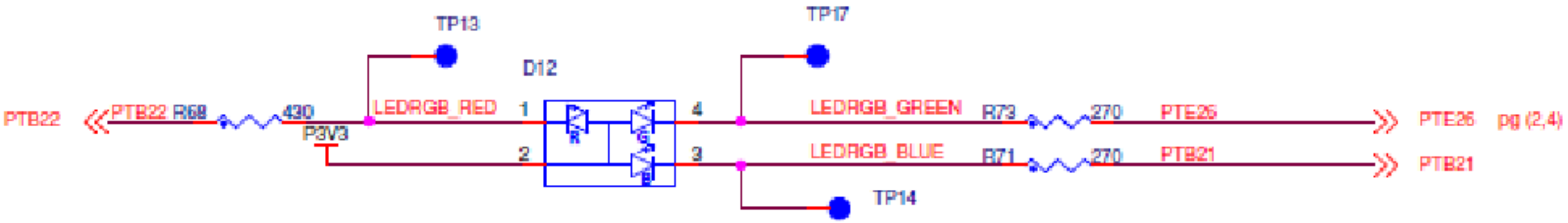
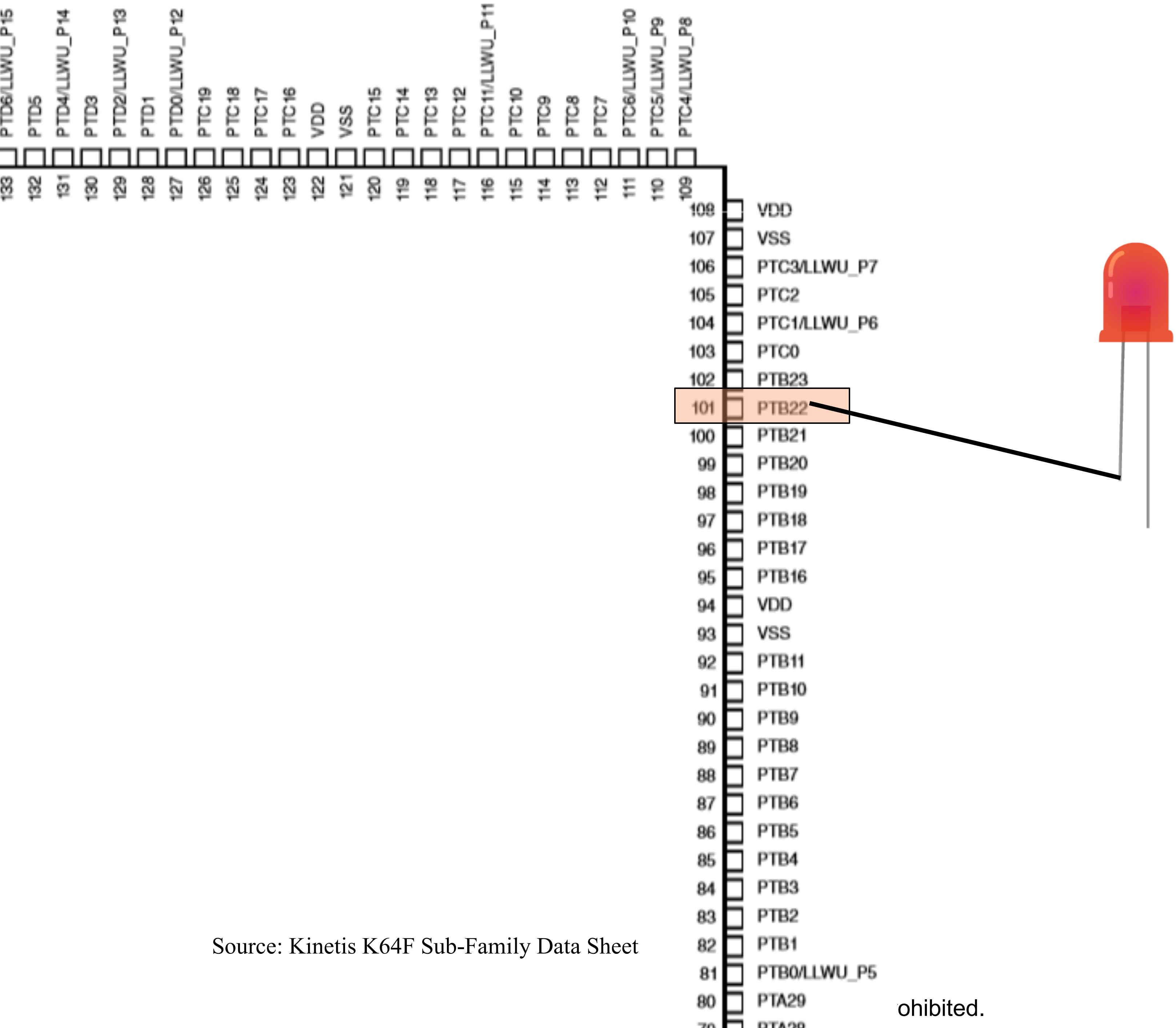


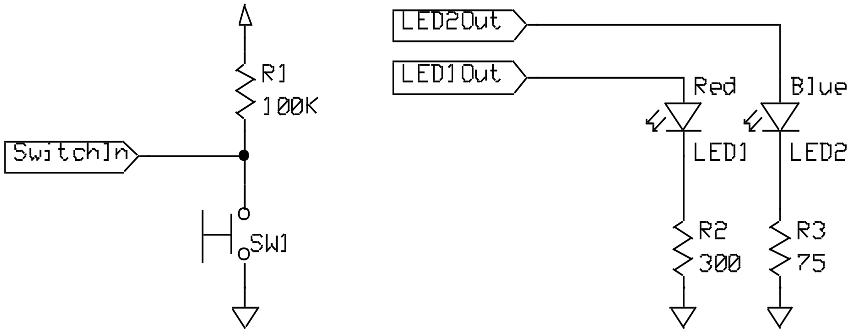
Figure 14. Tricolor LED

Source: FRDM-K64F Freedom Module User’s Guide



Source: Kinetis K64F Sub-Family Data Sheet

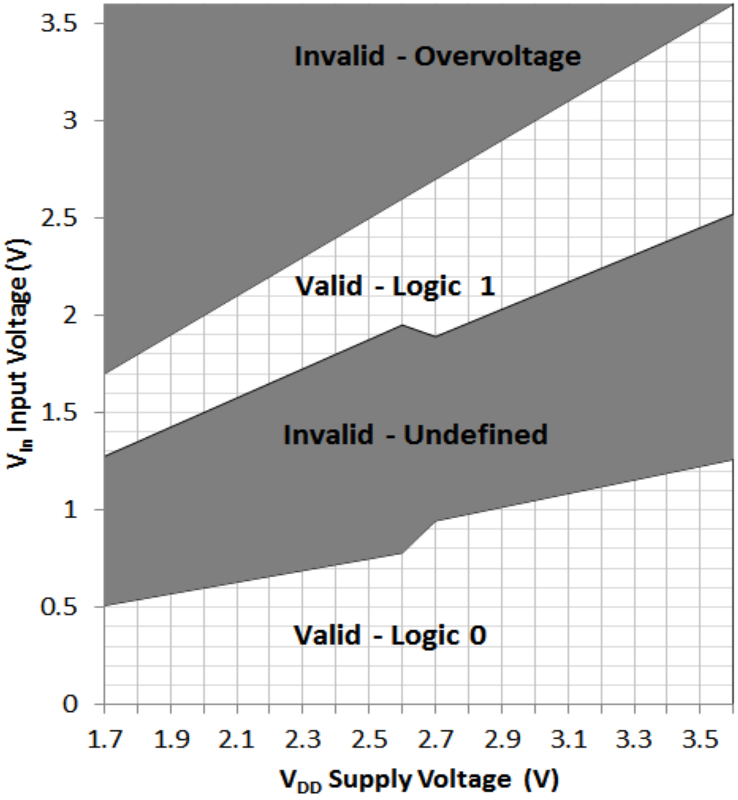
General-Purpose Input and Output



- 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

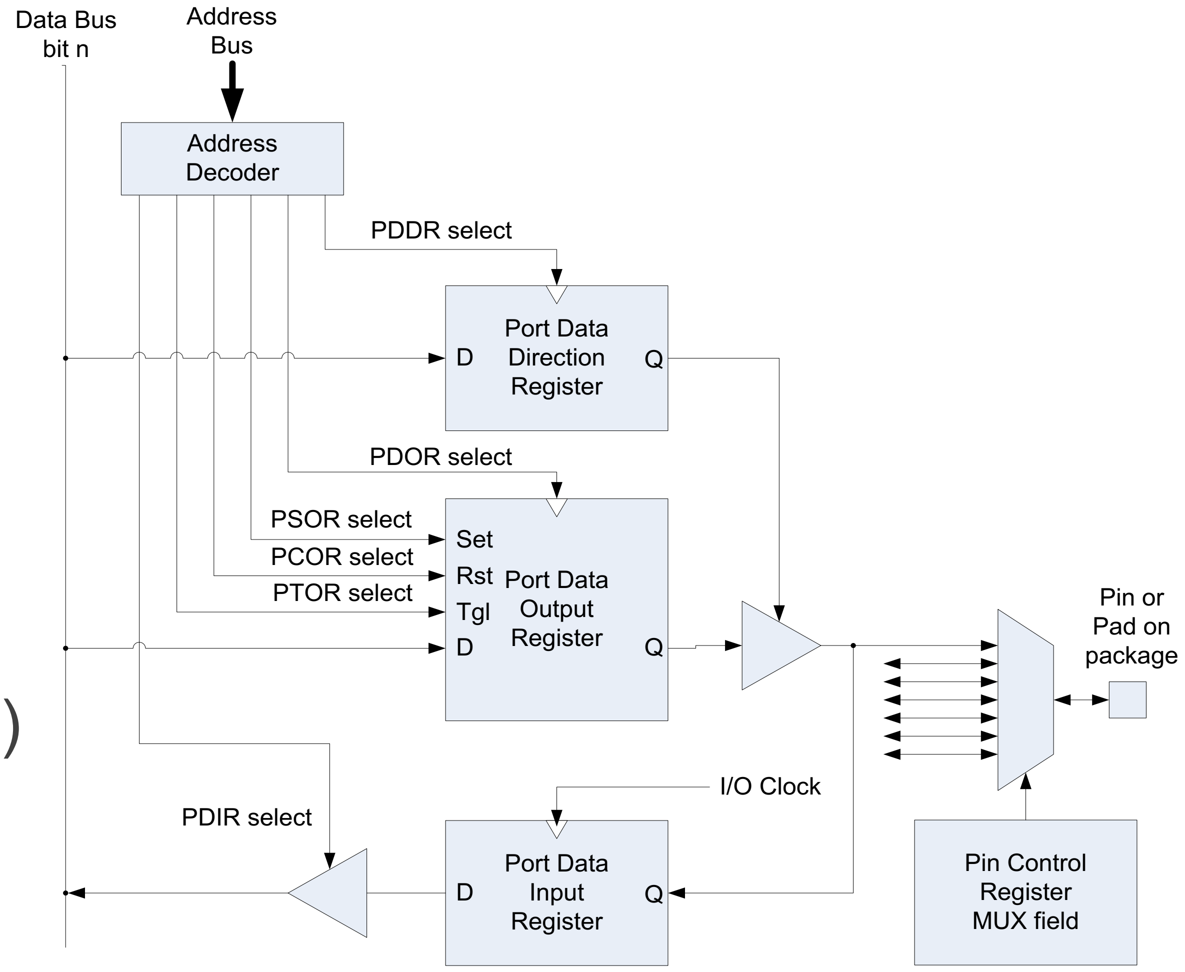
What’s a One? A Zero?

- 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

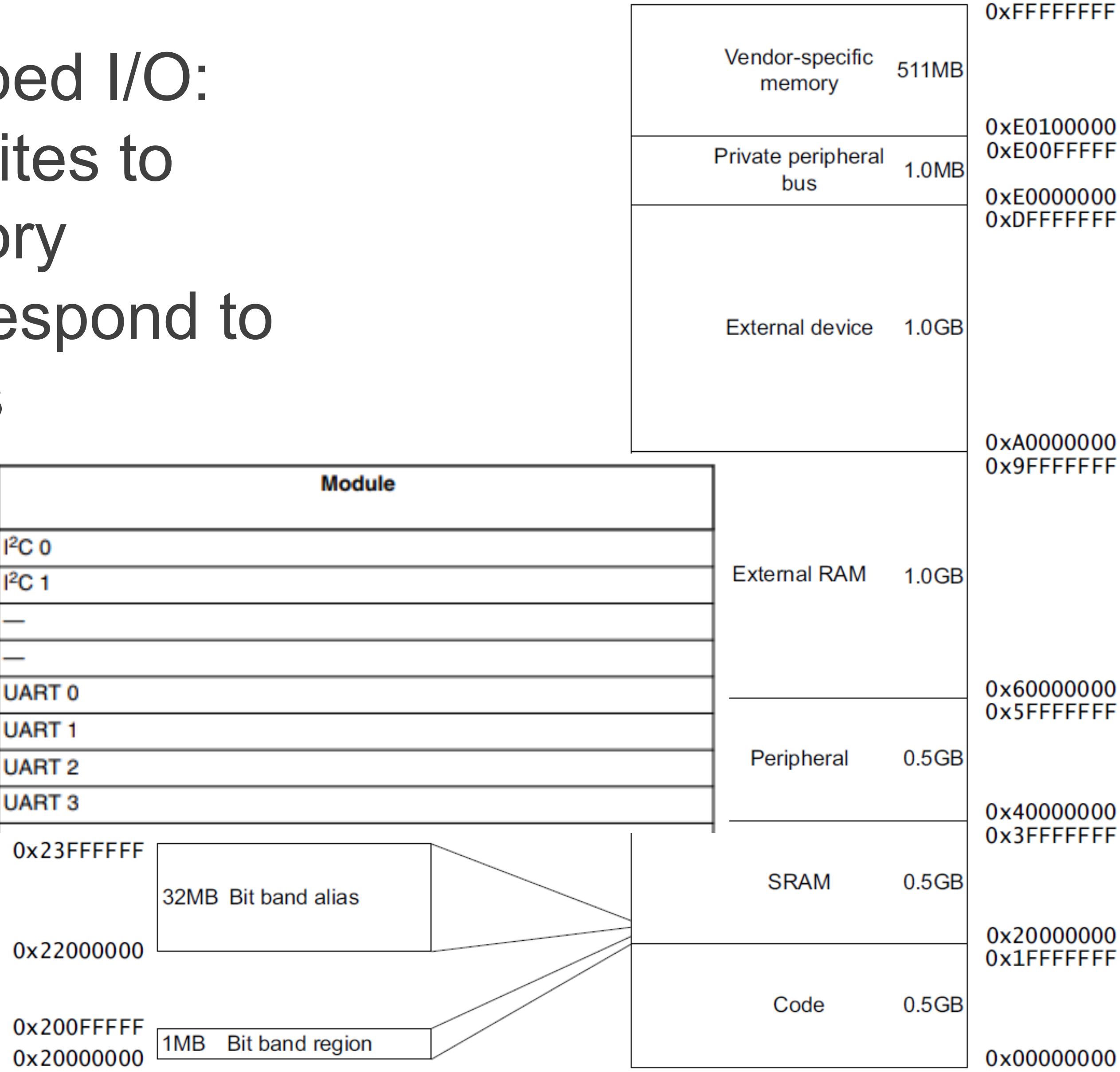
- Example: SNAP ISA
  - `add $15,$1,$2`
    - Register 15 is mapped for output operations
  - `add $1,$15,$2`
    - Register 15 is also mapped for input operations

- 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

System 32-bit base address	Slot number	Module
0x4006_6000	102	I²C 0
0x4006_7000	103	I²C 1
0x4006_8000	104	—
0x4006_9000	105	—
0x4006_A000	106	UART 0
0x4006_B000	107	UART 1
0x4006_C000	108	UART 2
0x4006_D000	109	UART 3



Source: ARM Cortex-M4

Privilege Levels

- Which software should manages the IO?

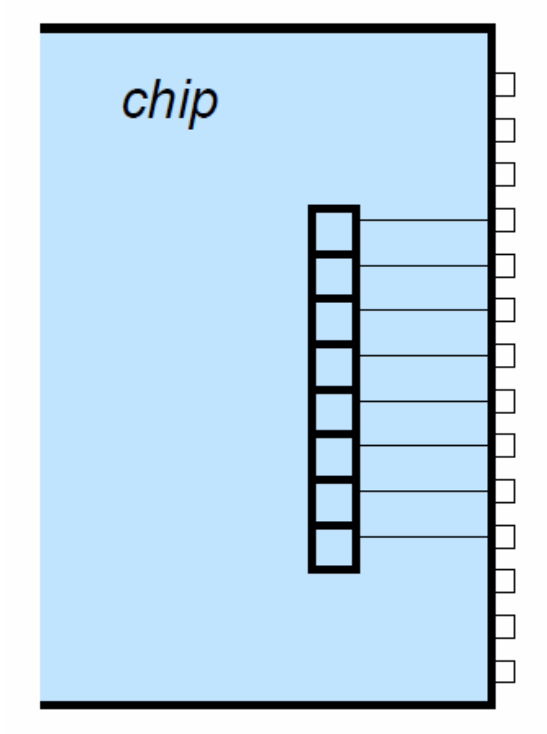
- Processors often support multiple privilege levels
  - Supervisor / “Privileged” in Cortex-M4
    - Access to all resources
  - User / “Unprivileged” in Cortex-M4
    - Limited access to certain instructions and memory/peripheral

- 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*

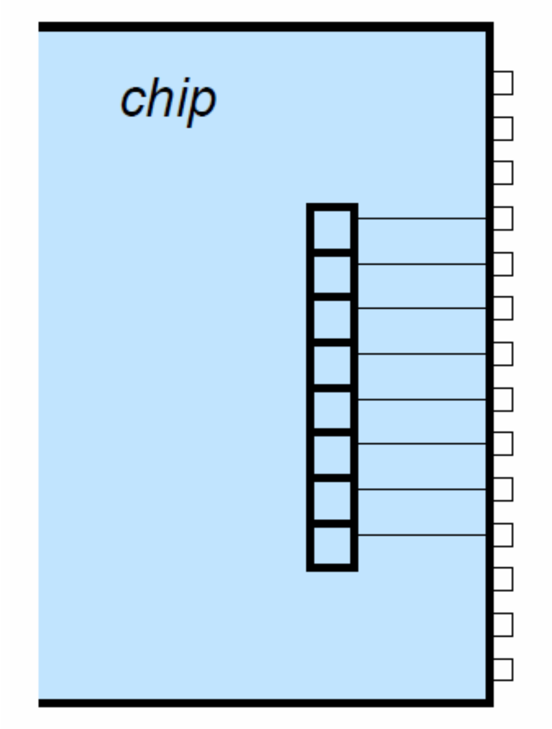
- How does software know **if** the value is valid?
  - Use a valid bit
    - 9-bit input, with 8-bits of data
    - Toggle 9th bit to indicate new data
  - Use encoded data
    - One-hot encoding
    - 01 = *false*, 10 = *true*, 00 = *no data*

- How does software know **when** a new input is ready?

Option 1: Polling

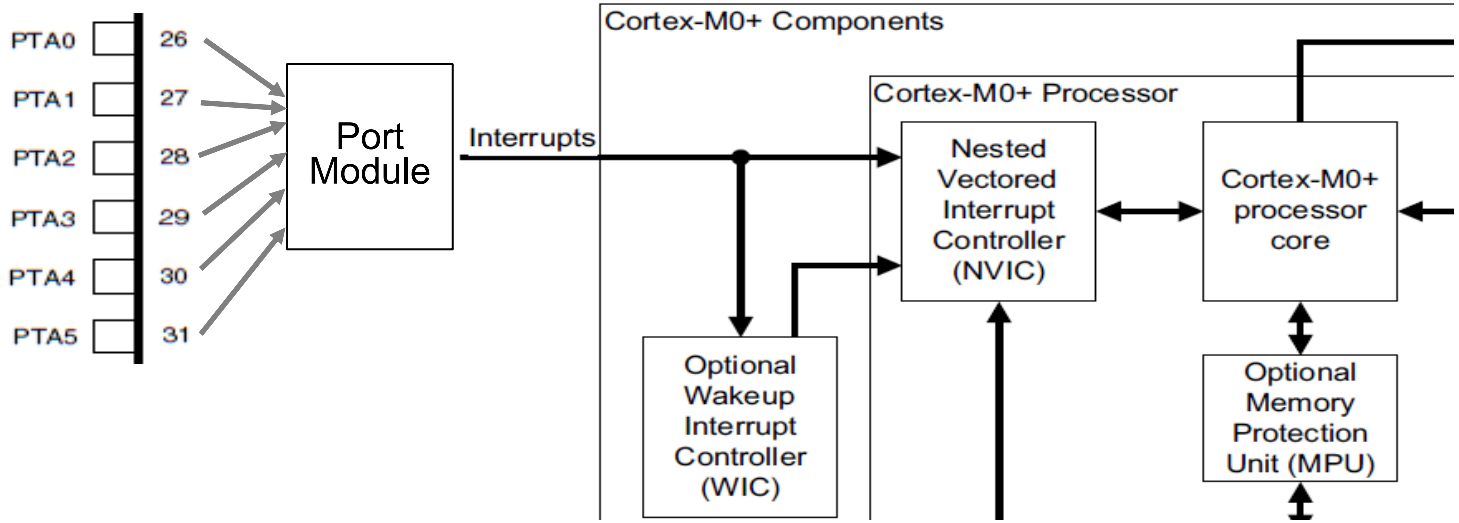
- Use software to check
    - Keep reading the value in a loop

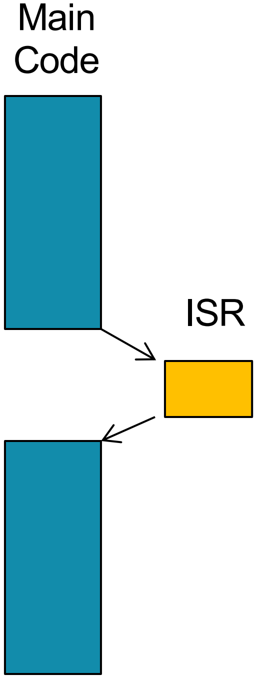
```
while (1) {
    // read a GPIO port
    // check the value
}
```



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


PTA0
PTA1
PTA2
PTA3
PTA4
PTA5
26
27
28
29
30
31
Port Module
Interrupts
Cortex-M0+ Components
Cortex-M0+ Processor
Nested Vectored Interrupt Controller (NVIC)
Cortex-M0+ processor core
Optional Wakeup Interrupt Controller (WIC)
Optional Memory Protection Unit (MPU)



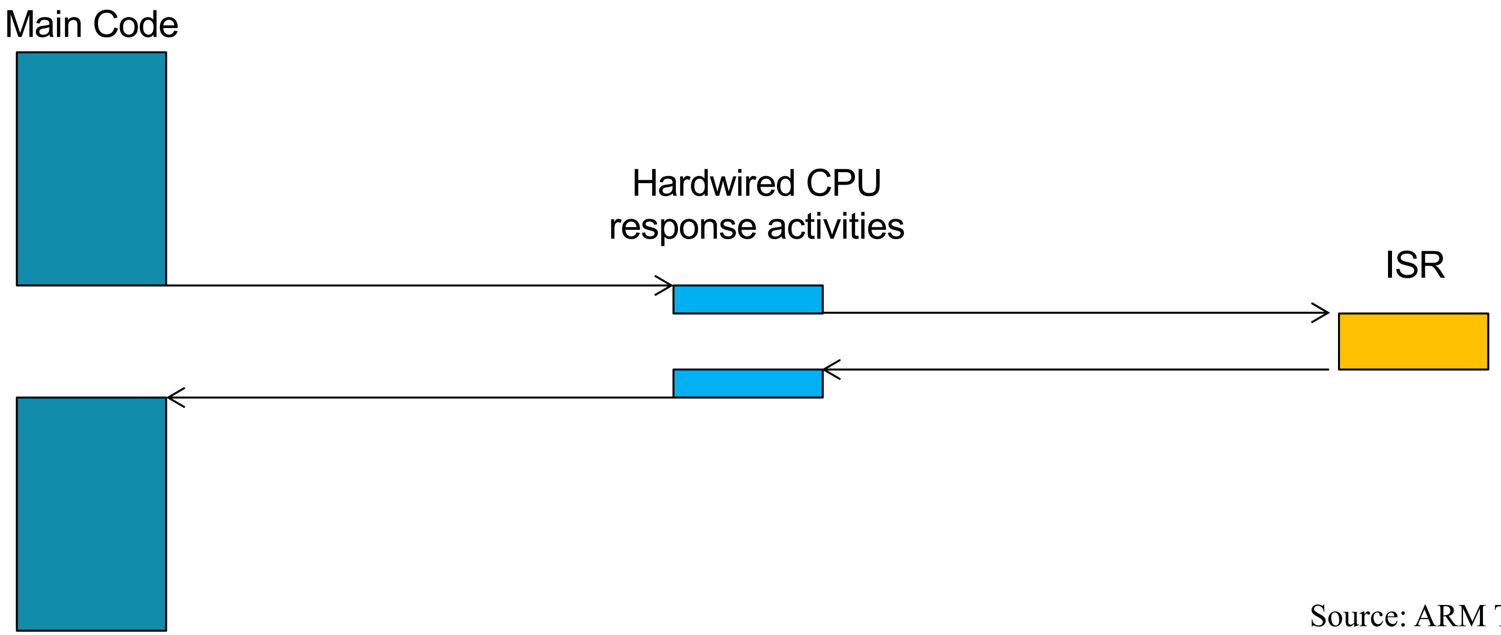
Polling vs. Interrupt

- 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

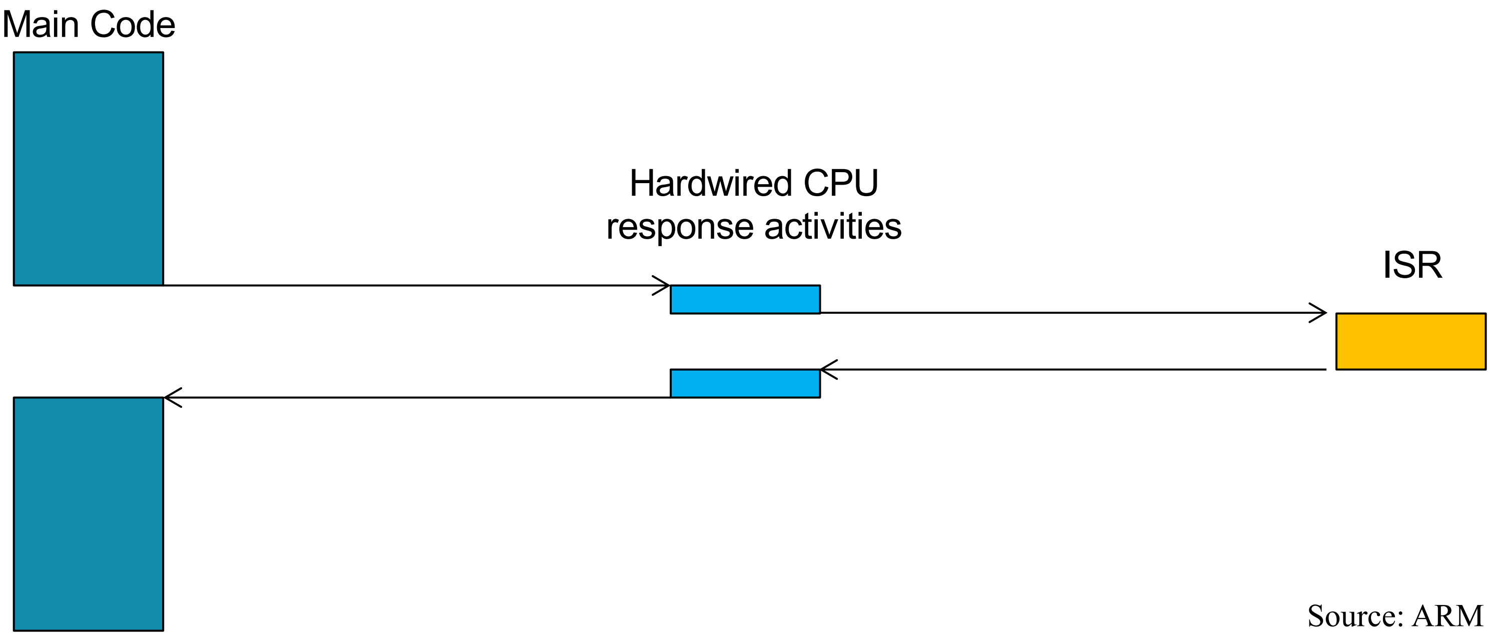
- Interrupts (asynchronous)
  - I/O device interrupt, reset, etc.

- 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

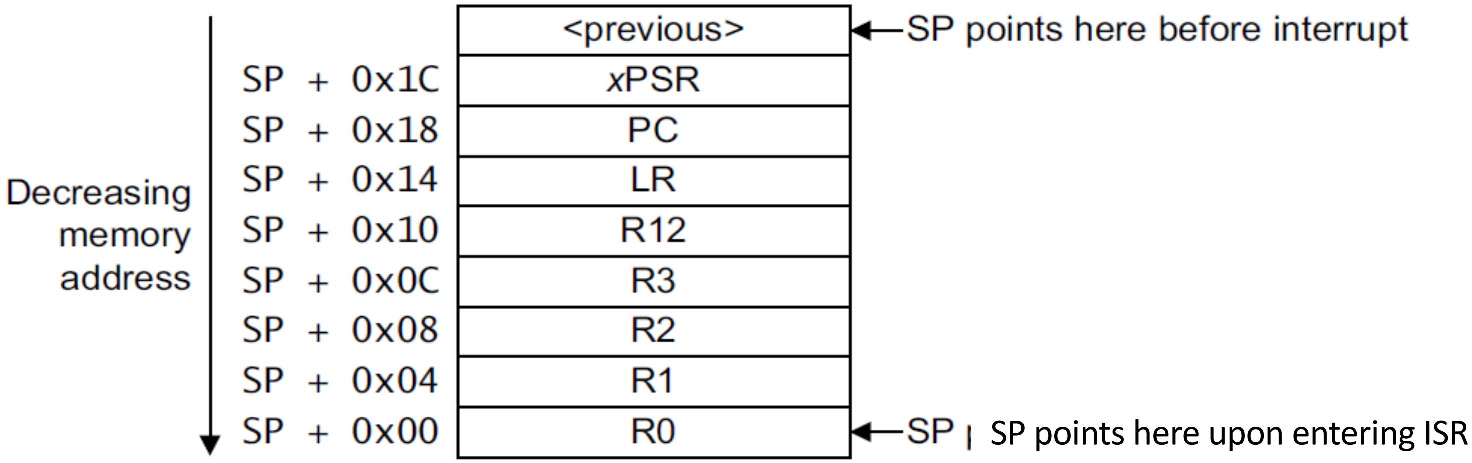
CPU's Hardwired Exception Processing

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



Decreasing memory address

	<previous>
SP + 0x1C	*x*PSR
SP + 0x18	PC
SP + 0x14	LR
SP + 0x10	R12
SP + 0x0C	R3
SP + 0x08	R2
SP + 0x04	R1
SP + 0x00	R0

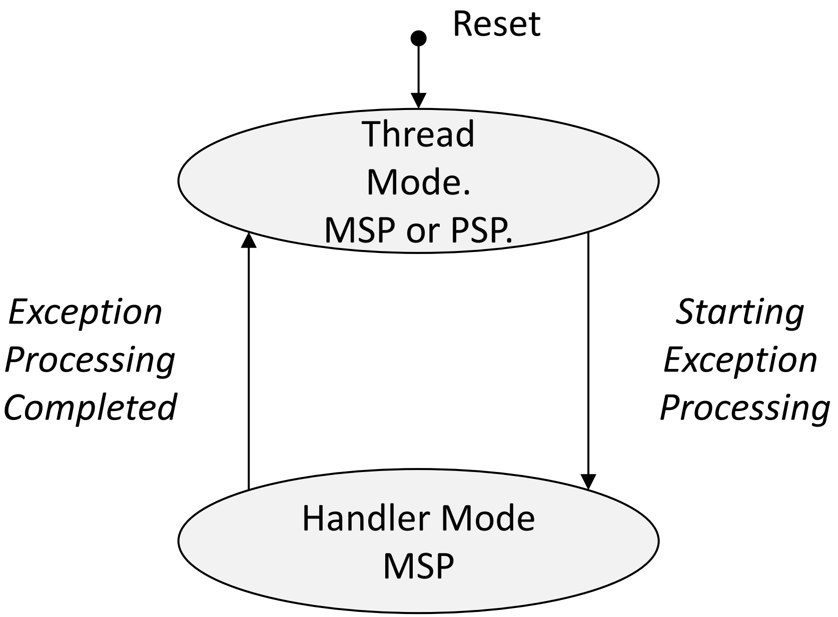
SP points here before interrupt

SP SP points here upon entering ISR

- Two SPs: Main (MSP), process (PSP)
- Which is active depends on operating mode, CONTROL register bit 1
- Stack grows toward smaller addresses

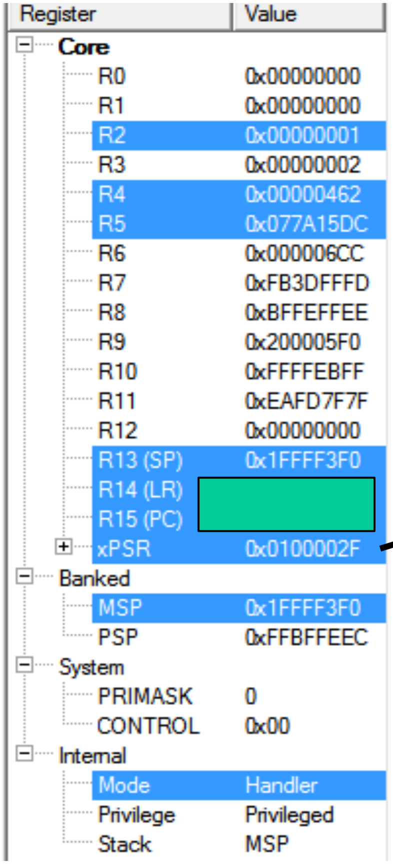
3. Switch to Handler/Privileged Mode

- Thread mode
  - Privileged or unprivileged
  - MSP or PSP
- Handler mode
  - Privileged
  - Always uses Main SP



Source: ARM Tutorial Slides

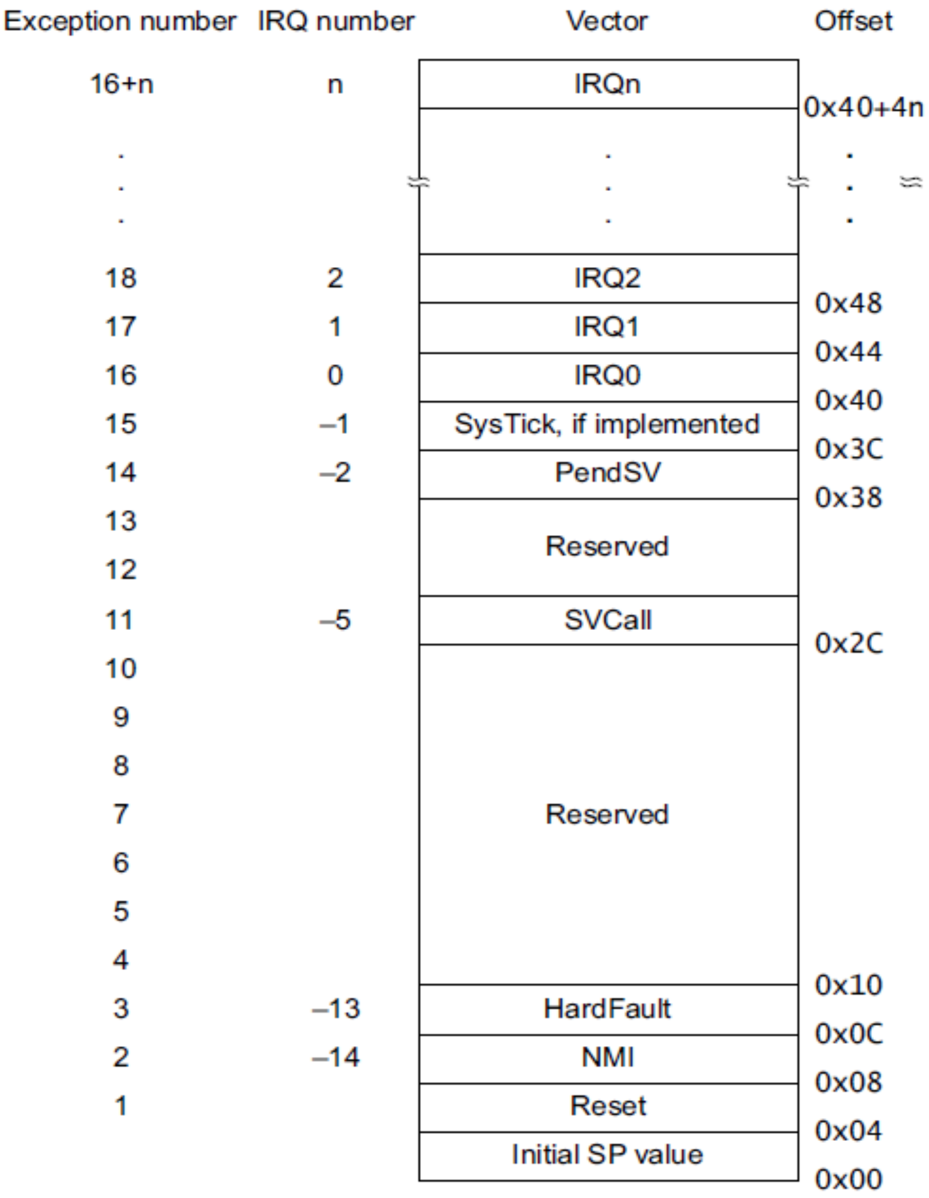
Update IPSR with Exception Number



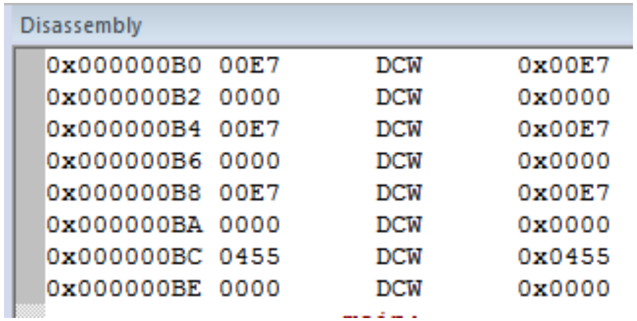
Register	Value
Core	
R0	0x00000000
R1	0x00000000
R2	0x00000001
R3	0x00000002
R4	0x00000462
R5	0x077A15DC
R6	0x000006CC
R7	0xFB3DFFFD
R8	0xBFFEFFEE
R9	0x200005F0
R10	0xFFFFEBFF
R11	0xEAFD7F7F
R12	0x00000000
R13 (SP)	0x1FFFF3F0
R14 (LR)	
R15 (PC)	
xPSR	0x0100002F
Banked	
MSP	0x1FFFF3F0
PSP	0xFFBFFEEC
System	
PRIMASK	0
CONTROL	0x00
Internal	
Mode	Handler
Privilege	Privileged
Stack	MSP

PORTD_IRQ is Exception number 0x2F
(interrupt number + 0x10)

Vector Table Example



Exception number	IRQ number	Vector	Offset
16+n	n	IRQn	0x40+4n
.		.	.
.		.	.
.		.	.
18	2	IRQ2	0x48
17	1	IRQ1	0x44
16	0	IRQ0	0x40
15	–1	SysTick, if implemented	0x3C
14	–2	PendSV	0x38
13		Reserved	
12			
11	–5	SVCall	0x2C
10		Reserved	
9			
8			
7			
6			
5			
4			0x10
3	–13	HardFault	0x0C
2	–14	NMI	0x08
1		Reset	0x04
		Initial SP value	0x00



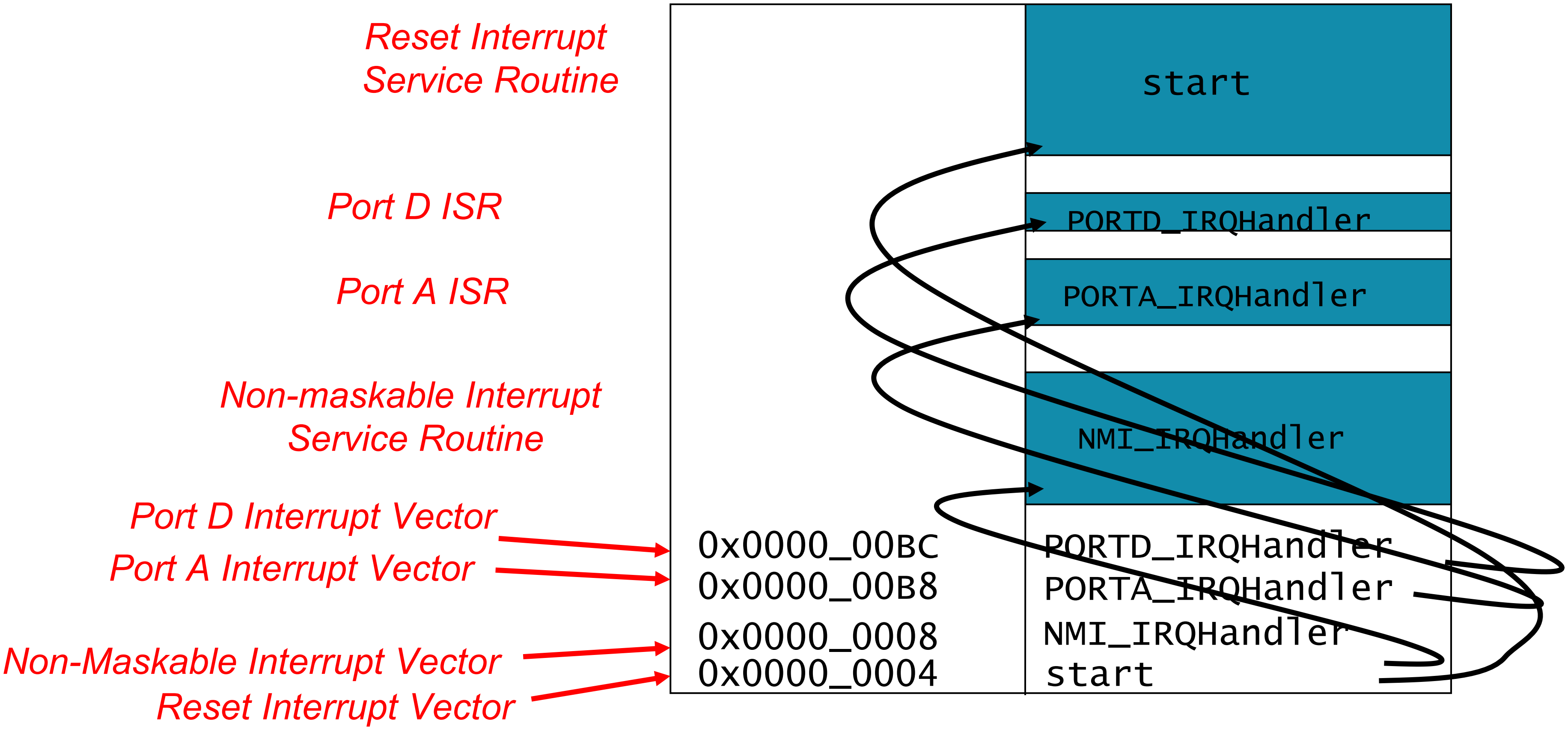
Disassembly

```
0x000000B0 00E7      DCW      0x00E7
0x000000B2 0000      DCW      0x0000
0x000000B4 00E7      DCW      0x00E7
0x000000B6 0000      DCW      0x0000
0x000000B8 00E7      DCW      0x00E7
0x000000BA 0000      DCW      0x0000
0x000000BC 0455      DCW      0x0455
0x000000BE 0000      DCW      0x0000
```

- 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



Source: ARM Tutorial Slides

5. Load LR With EXC_RETURN Code

EXC_RETURN	Return Mode	Return Stack	Description
0xFFFF_FFF1	0 (Handler)	0 (MSP)	Return to exception handler
0xFFFF_FFF9	1 (Thread)	0 (MSP)	Return to thread with MSP
0xFFFF_FFFD	1 (Thread)	1 (PSP)	Return to thread with PSP

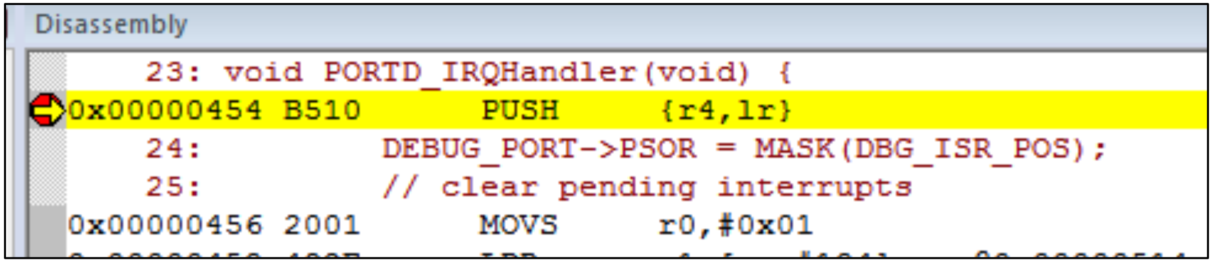
- EXC_RETURN value generated by CPU to provide information on how to return
  - Which SP to restore registers from? MSP (0) or PSP (1)
    - Previous value of SPSEL
  - Which mode to return to? Handler (0) or Thread (1)
    - Another exception handler may have been running when this exception was requested

Source: ARM Tutorial Slides

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



```
    23: void PORTD_IRQHandler(void) {
0x00000454 B510      PUSH     {r4,lr}
    24:         DEBUG_PORT->PSOR = MASK(DBG_ISR_POS);
    25:         // clear pending interrupts
0x00000456 2001      MOVS     r0,#0x01
```

Source: ARM Tutorial Slides

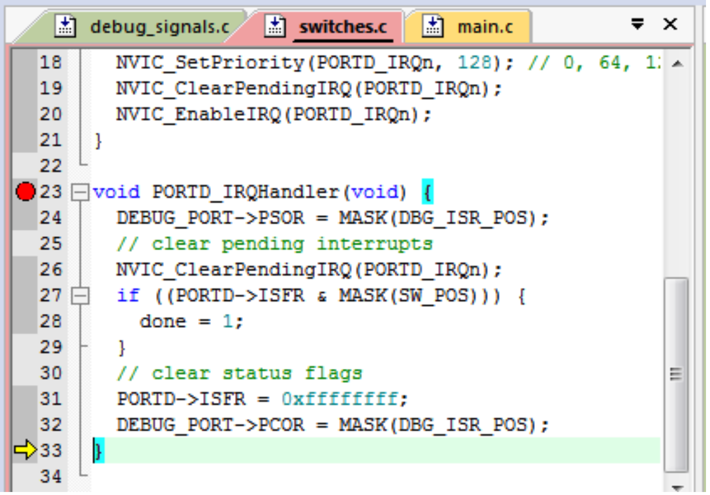
Exiting an Exception Handler

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

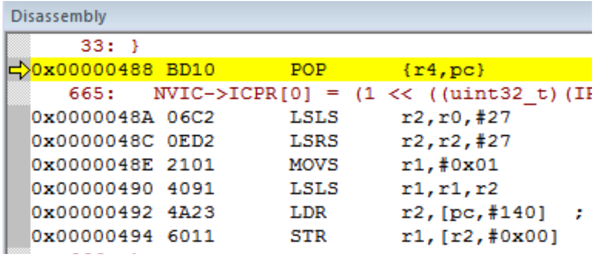
1. Execute Instruction for Exception Return

- 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



```
  NVIC_SetPriority(PORTD_IRQn, 128); // 0, 64, 1
  NVIC_ClearPendingIRQ(PORTD_IRQn);
  NVIC_EnableIRQ(PORTD_IRQn);
}

void PORTD_IRQHandler(void) {
  DEBUG_PORT->PSOR = MASK(DBG_ISR_POS);
  // clear pending interrupts
  NVIC_ClearPendingIRQ(PORTD_IRQn);
  if ((PORTD->ISFR & MASK(SW_POS))) {
    done = 1;
  }
  // clear status flags
  PORTD->ISFR = 0xffffffff;
  DEBUG_PORT->PCOR = MASK(DBG_ISR_POS);
}
```



```
    33: }
0x00000488 BD10      POP      {r4,pc}
   665:    NVIC->ICPR[0] = (1 << ((uint32_t)(I
0x0000048A 06C2      LSLS     r2,r0,#27
0x0000048C 0ED2      LSRS     r2,r2,#27
0x0000048E 2101      MOVS     r1,#0x01
0x00000490 4091      LSLS     r1,r1,r2
0x00000492 4A23      LDR      r2,[pc,#140]  ;
0x00000494 6011      STR      r1,[r2,#0x00]
```

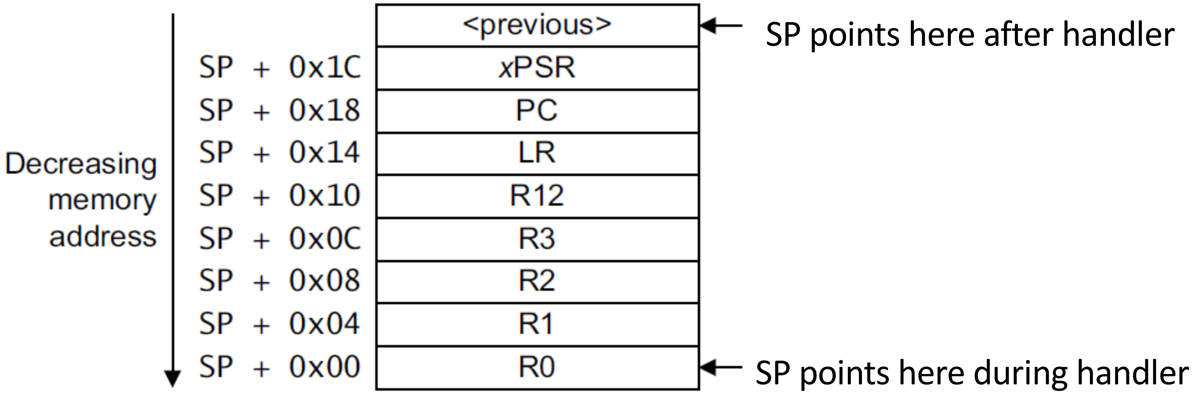
Source: ARM Tutorial Slides

2. Select Stack, Restore Context

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

EXC_RETURN	Return Stack	Description
0xFFFF_FFF1	0 (MSP)	Return to exception handler with MSP
0xFFFF_FFF9	0 (MSP)	Return to thread with MSP
0xFFFF_FFFD	1 (PSP)	Return to thread with PSP

- Pop the registers from that stack



<previous>

SP + 0x1C xPSR

SP + 0x18 PC

SP + 0x14 LR

SP + 0x10 R12

SP + 0x0C R3

SP + 0x08 R2

SP + 0x04 R1

SP + 0x00 R0

Decreasing memory address

SP points here after handler

SP points here during handler

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

```
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

```
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