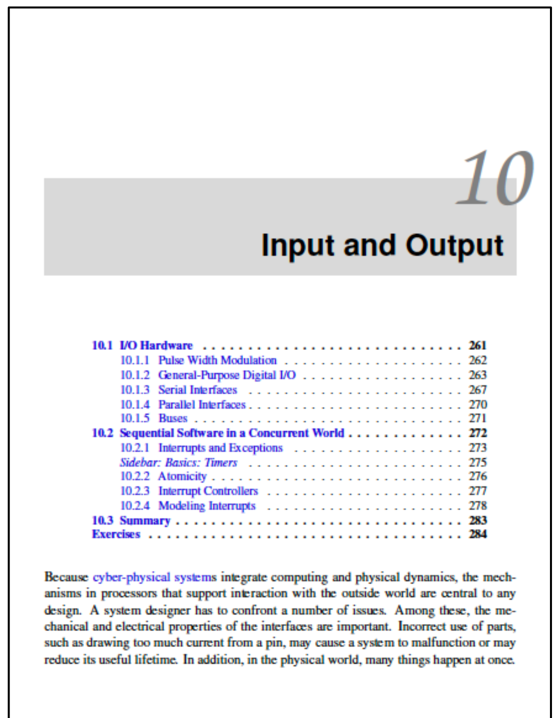
Lecture 7 : Input / Output

Slides were originally developed by Profs. Edward Lee and Sanjit Seshia, and subsequently updated by Profs. Gavin Buskes and Iman Shames.

Outline

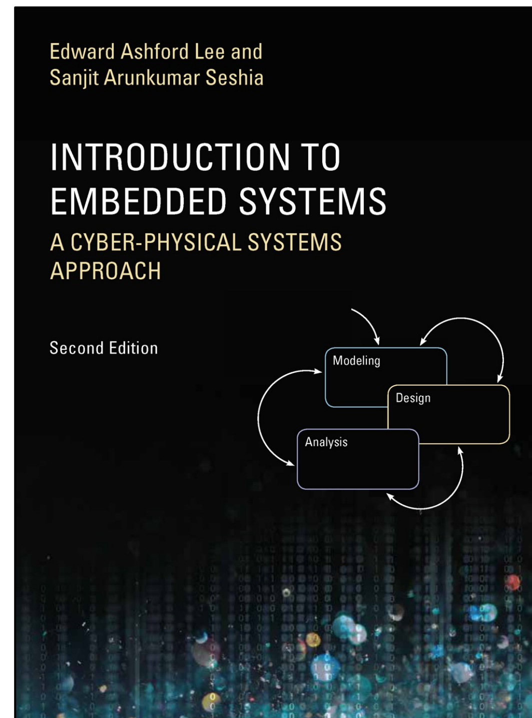
- Analog vs. digital, wired vs. wireless, serial vs. parallel
- Sampled or event triggered, bit rates
- Access control, security, authentication
- Physical connectors, electrical requirements (voltages and currents)



10

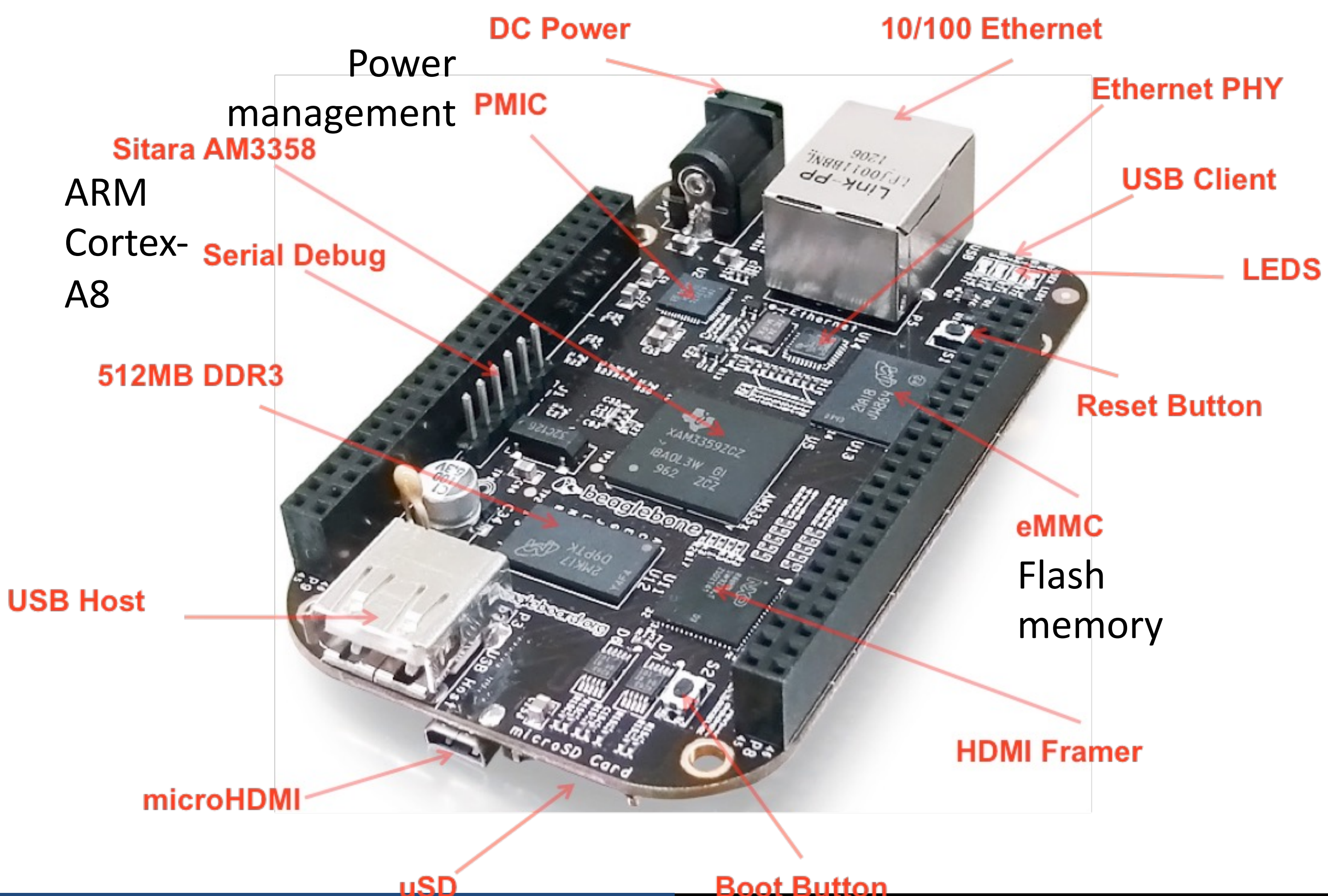
Input and Output

Because cyber-physical systems integrate computing and physical dynamics, the mechanisms in processors that support interaction with the outside world are central to any design. A system designer has to confront a number of issues. Among these, the mechanical and electrical properties of the interfaces are important. Incorrect use of parts, such as drawing too much current from a pin, may cause a system to malfunction or may reduce its useful lifetime. In addition, in the physical world, many things happen at once.

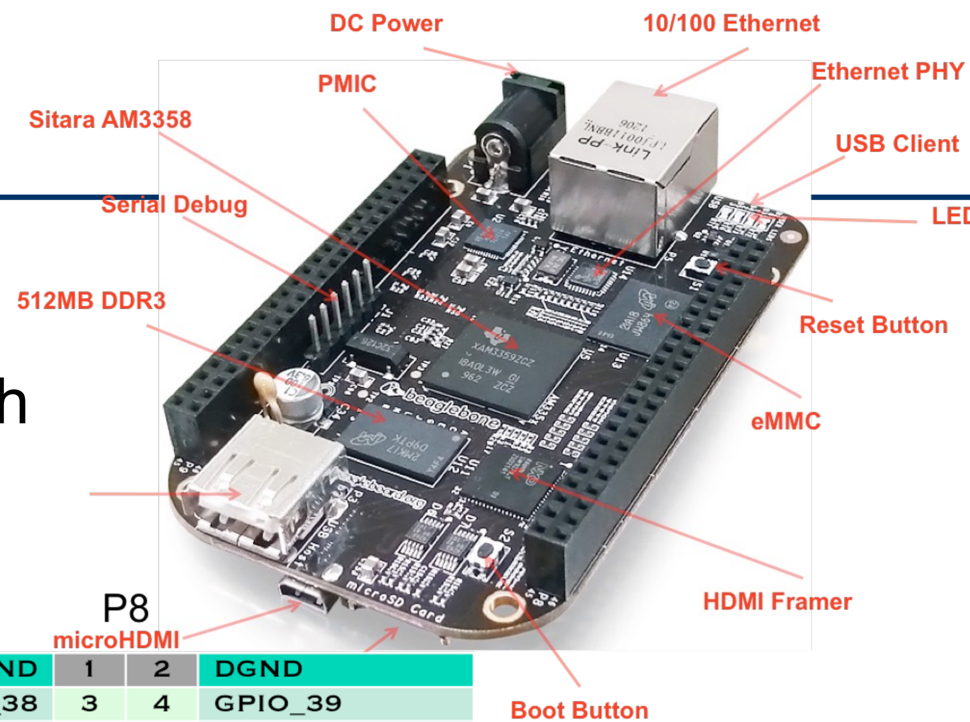


A Typical Microcomputer Board : Beaglebone Black from Texas Instruments

This board has analog and digital inputs and outputs. What are they? How do they work?



Beaglebone Black Header Configuration



One of eight configurations with SPI buses, analog I/O, etc.

P9

DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3
VDD_5V	5	6	VDD_5V
SYS_5V	7	8	SYS_5V
PWR_BUT	9	10	SYS_RESETN
GPIO_30	11	12	GPIO_60
GPIO_31	13	14	GPIO_40
GPIO_48	15	16	GPIO_51
SPI0_CS0	17	18	SPI0_D1
SPI1_CS1	19	20	SPI1_CS0
SPI0_D0	21	22	SPI0_SCLK
GPIO_49	23	24	GPIO_15
GPIO_117	25	26	GPIO_14
GPIO_125	27	28	SPI1_CS0
SPI1_D0	29	30	SPI1_D1
SPI1_SCLK	31	32	VDD_ADC
AIN4	33	34	GNDA_ADC
AIN6	35	36	AIN5
AIN2	37	38	AIN3
AIN0	39	40	AIN1
GPIO_20	41	42	SPI1_CS1
DGND	43	44	DGND
DGND	45	46	DGND

P8

DGND	1	2	DGND
GPIO_38	3	4	GPIO_39
GPIO_34	5	6	GPIO_35
GPIO_66	7	8	GPIO_67
GPIO_69	9	10	GPIO_68
GPIO_45	11	12	GPIO_44
GPIO_23	13	14	GPIO_26
GPIO_47	15	16	GPIO_46
GPIO_27	17	18	GPIO_65
GPIO_22	19	20	GPIO_63
GPIO_62	21	22	GPIO_37
GPIO_36	23	24	GPIO_33
GPIO_32	25	26	GPIO_61
GPIO_86	27	28	GPIO_88
GPIO_87	29	30	GPIO_89
GPIO_10	31	32	GPIO_11
GPIO_9	33	34	GPIO_81
GPIO_8	35	36	GPIO_80
GPIO_78	37	38	GPIO_79
GPIO_76	39	40	GPIO_77
GPIO_74	41	42	GPIO_75
GPIO_72	43	44	GPIO_73
GPIO_70	45	46	GPIO_71

Many GPIO pins can be reconfigured to be PWM drivers, timers, etc.

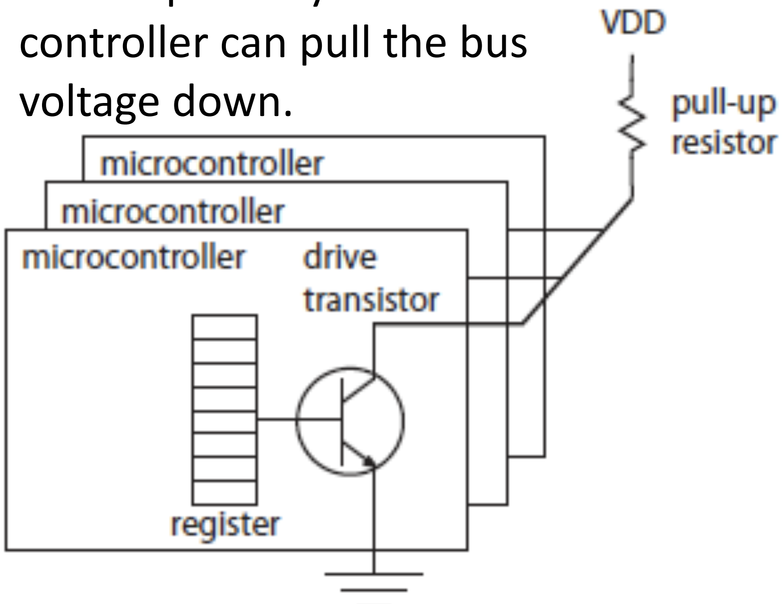
Simple Digital I/O: GPIO

- Open collector circuits are often used on GPIO (general-purpose I/O) pins of a microcontroller.

- The same pin can be used for input and output. And multiple users can connect to the same bus.

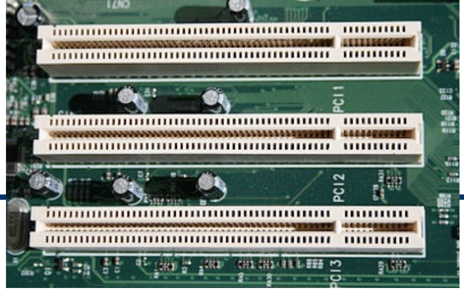
- Why is the current limited?

GPIO pins configured for bus output. Any one controller can pull the bus voltage down.



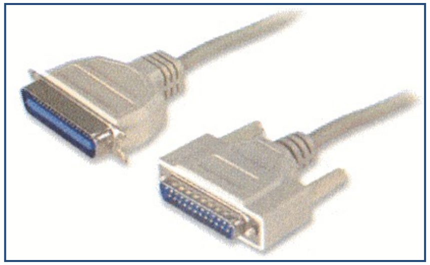
Wired Connections
Parallel vs. Serial Digital Interfaces

PCI



- Parallel (one wire per bit)
  - ATA: Advanced Technology Attachment
  - PCI: Peripheral Component Interface
  - SCSI: Small Computer System Interface
  - …
- Serial (one wire per direction)
  - RS-232
  - SPI: Serial Peripheral Interface bus
  - I^2C: Inter-Integrated Circuit
  - USB: Universal Serial Bus
  - SATA: Serial ATA
  - …
- Mixed (one or more “lanes”)
  - PCIe: PCI Express

SCSI

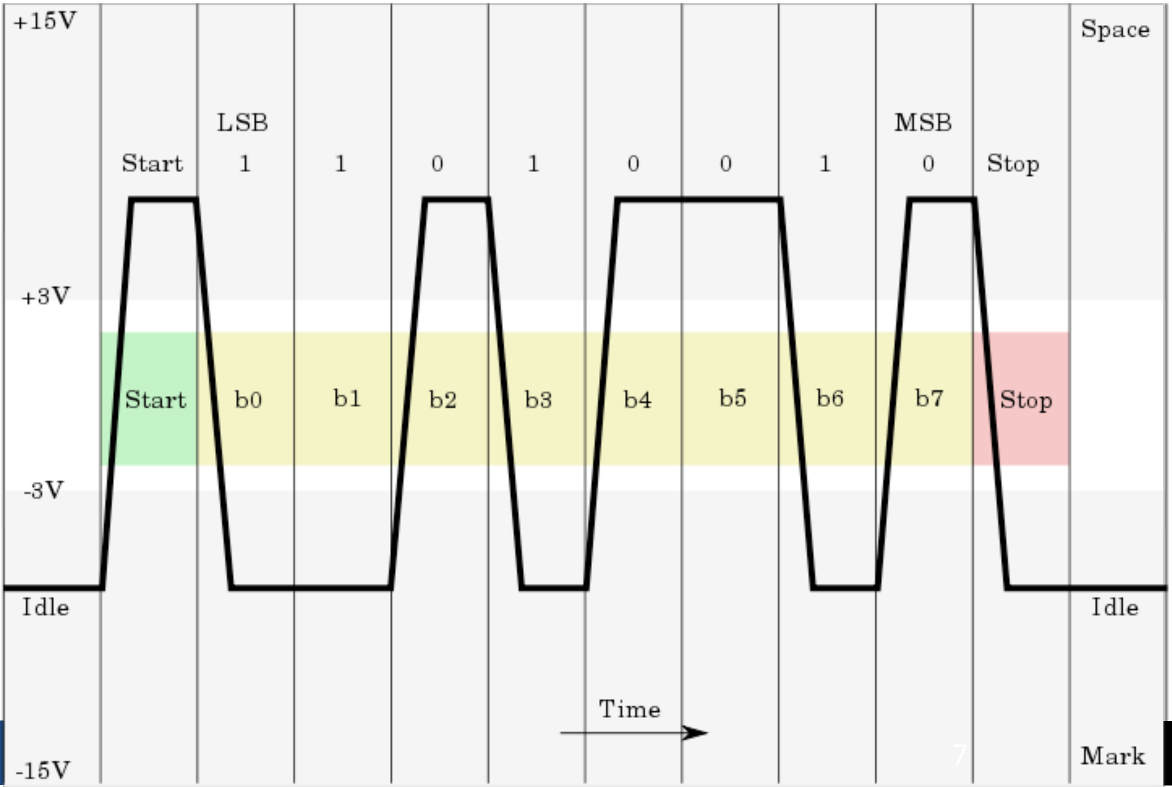


USB

RS-232

Serial Interfaces

- The old but persistent RS-232 standard supports asynchronous serial connections (no common clock).
- RS-232 relies on the clock in the transmitter being close enough in frequency to the clock on the receiver that upon detecting the start bit, it can just sample 8 more times and will see the remaining bits.



Many uses of RS-232 are being replaced by USB, which is electrically simpler but with a more complex protocol, or Bluetooth, which is wireless.

USB achieves higher speeds by beginning every packet with synchronization sequence of 8 bits. The receiver clock locks to this for the rest of the packet.

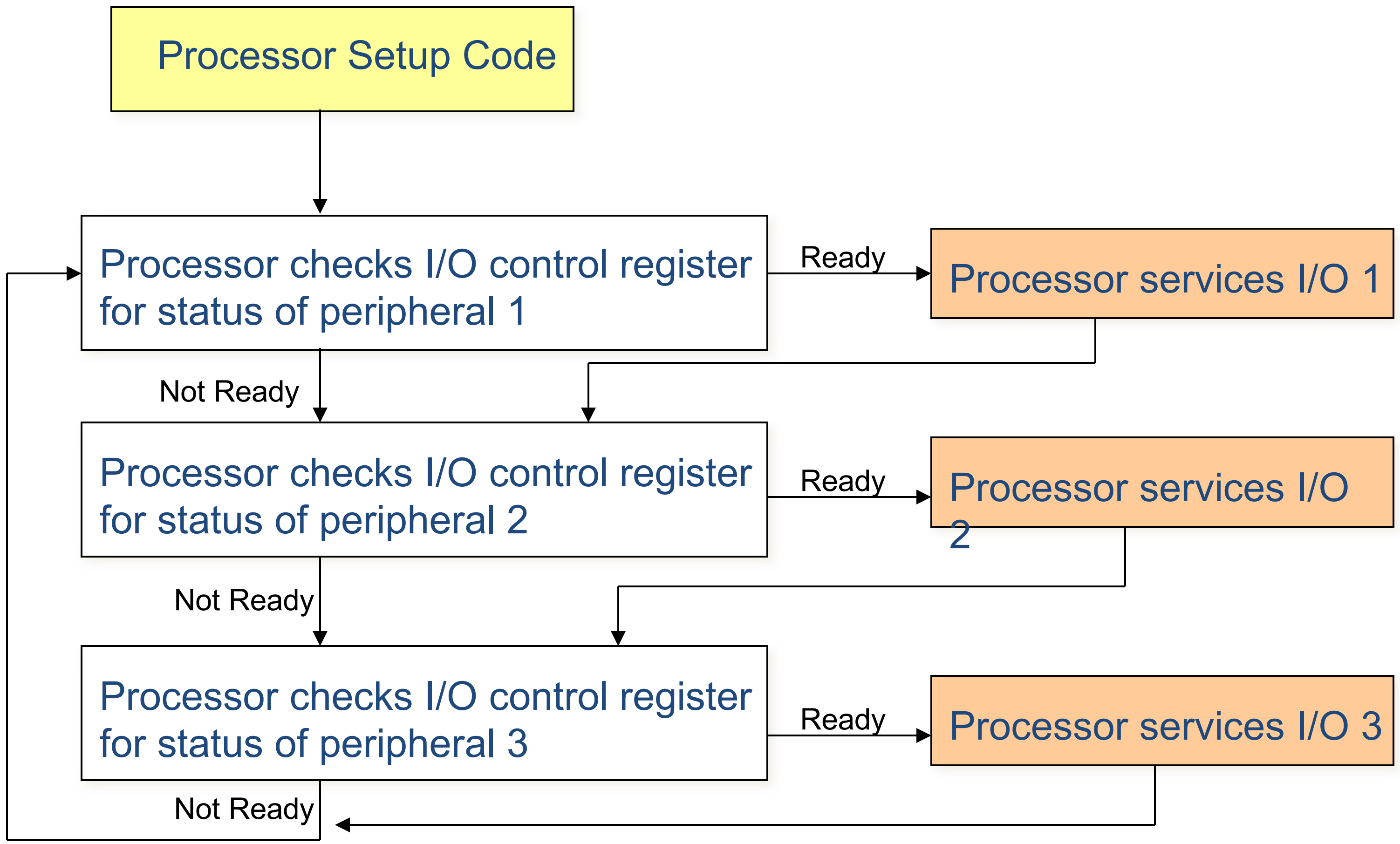
Uppercase ASCII "K" character (0x4b) with 1 start bit, 8 data bits, 1 stop bit. *Image license: Creative Commons ShareAlike 1.0 License*

Input/Output Mechanisms in Software

- Polling
  - Main loop uses each I/O device periodically.
  - If output is to be produced, produce it.
  - If input is ready, read it.

- Interrupts
  - External hardware alerts the processor that input is ready.
  - Processor suspends what it is doing.
  - Processor invokes an interrupt service routine (ISR).
  - ISR interacts with the application concurrently.

Polling


Processor Setup Code
Processor checks I/O control register for status of peripheral 1
Ready
Processor services I/O 1
Not Ready
Processor checks I/O control register for status of peripheral 2
Ready
Processor services I/O 2
Not Ready
Processor checks I/O control register for status of peripheral 3
Ready
Processor services I/O 3
Not Ready

Example Using a Serial Interface

In an Atmel AVR 8-bit microcontroller, to send a byte over a serial port, the following C code will do:

```
while(!(UCSR0A & 0x20));
UDR0 = x;
```

- x is a variable of type uint8.
- UCSR0A and UDR0 are variables defined in a header.
- They refer to memory-mapped registers in the UART (Universal Asynchronous Receiver-Transmitter)

Send a Sequence of Bytes

```
for(i = 0; i < 8; i++) {
   while(!(UCSR0A & 0x20));
   UDR0 = x[i];
}
```

How long will this take to execute?

- 57600 baud serial speed.
- 8/57600 =139 microseconds.
- If your processor operates at 18 MHz, 139 microseconds is equal to 2500 cycles.

Receiving via UART

Again, on an Atmel AVR:

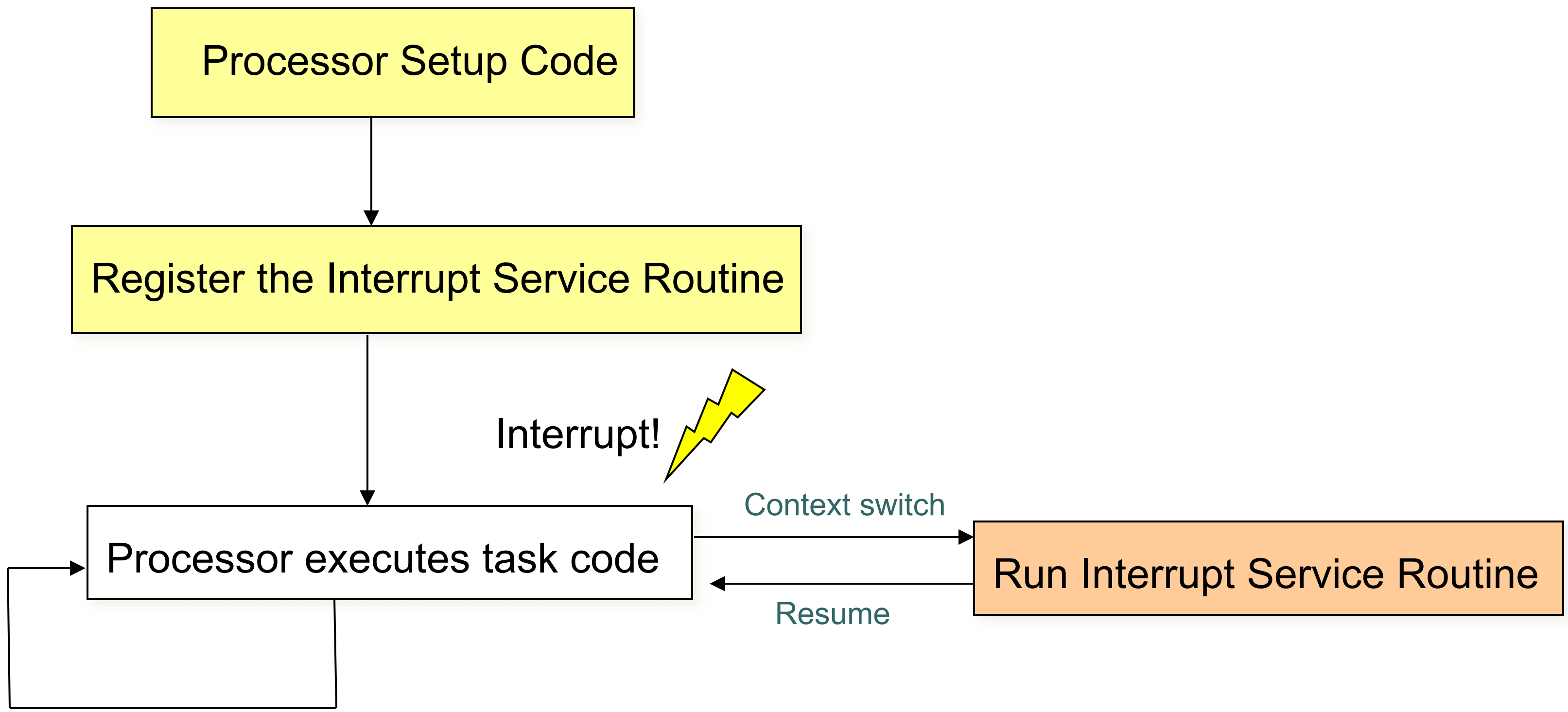
```
while(!(UCSR0A & 0x80));
return UDR0;
```

- Wait until the UART has received an incoming byte.
- *The programmer must ensure there will be one!*

Interrupts

- **Interrupt Service Routine**

  Short subroutine that handles the interrupt



Interrupts

The most typical and general program setup for the Reset and Interrupt Vector Addresses in ATmega168 is:

```
Address  Labels Code              Comments
0x0000          jmp    RESET        ; Reset Handler
0x0002          jmp    EXT_INT0     ; IRQ0 Handler
0x0004          jmp    EXT_INT1     ; IRQ1 Handler
0x0006          jmp    PCINT0       ; PCINT0 Handler
0x0008          jmp    PCINT1       ; PCINT1 Handler
0x000A          jmp    PCINT2       ; PCINT2 Handler
0x000C          jmp    WDT          ; Watchdog Timer Handler
0x000E          jmp    TIM2_COMPA   ; Timer2 Compare A Handler
0x0010          jmp    TIM2_COMPB   ; Timer2 Compare B Handler
0x0012          jmp    TIM2_OVF     ; Timer2 Overflow Handler
0x0014          jmp    TIM1_CAPT    ; Timer1 Capture Handler
```

Program memory addresses, not data memory addresses.

Source: ATmega168 Reference Manual

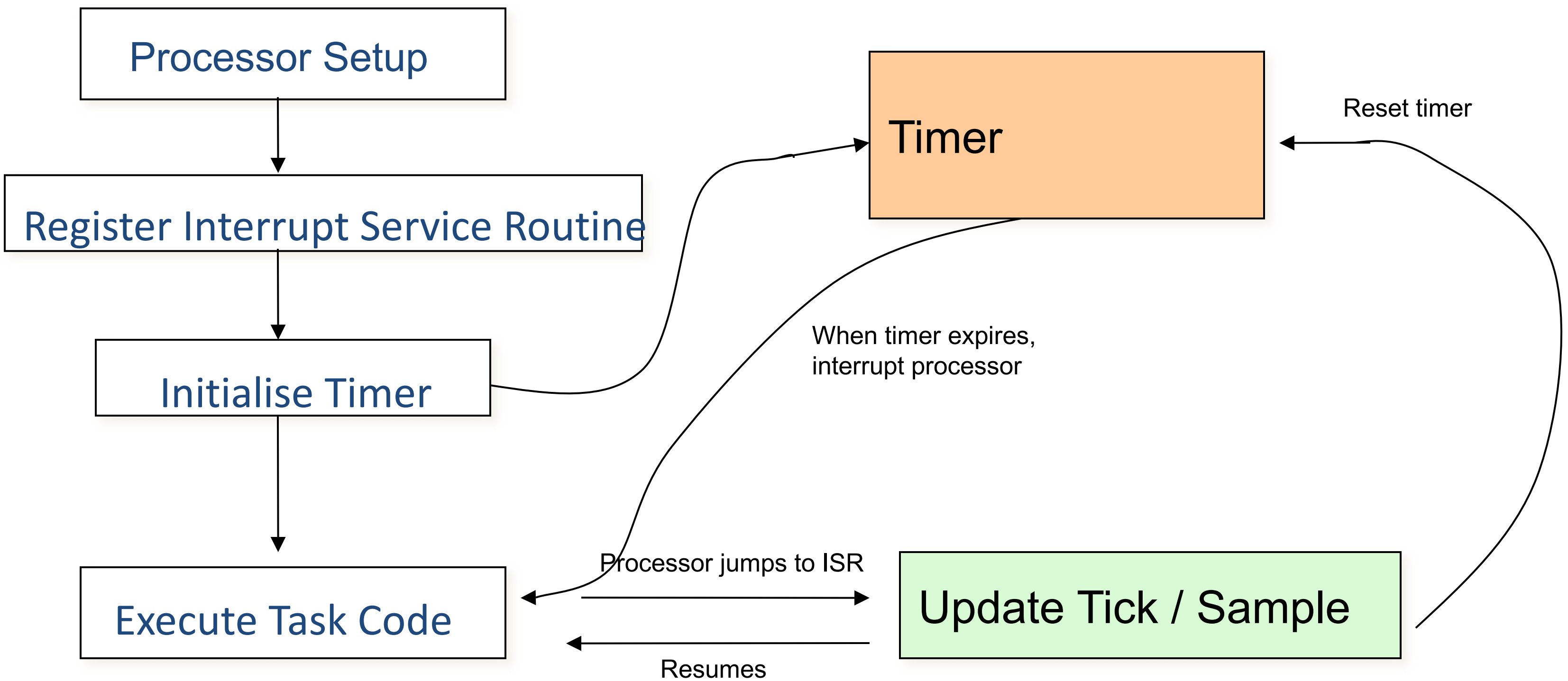
- Triggers:
  - A level change on an interrupt request pin
  - Writing to an interrupt pin configured as an output ("software interrupt") or executing special instruction
- Responses:
  - Disable interrupts.
  - Push the current program counter onto the stack.
  - Execute the instruction at a designated address in program memory.
- Design of interrupt service routine:
  - Save and restore any registers it uses.
  - Re-enable interrupts before returning from interrupt.

Interrupts are Evil

[I]n one or two respects modern machinery is basically more difficult to handle than the old machinery. Firstly, we have got the interrupts, occurring at unpredictable and irreproducible moments; compared with the old sequential machine that pretended to be a fully deterministic automaton, this has been a dramatic change, and many a systems programmer's grey hair bears witness to the fact that we should not talk lightly about the logical problems created by that feature.

(Dijkstra, "The humble programmer" 1972)

Timed Interrupt



Example : Set up a timer on an ARM Cortex M3 to trigger an interrupt every 1ms.

```
// Setup and enable SysTick with interrupt every 1ms
void initTimer(void) {
   SysTickPeriodSet(SysCtlClockGet() / 1000);
   SysTickEnable();
   SysTickIntEnable();
}

// Disable SysTick
void disableTimer(void) {
   SysTickIntDisable();
   SysTickDisable();
}
```

Number of cycles per sec.

Start SysTick counter

Enable SysTick timer interrupt

Source: Stellaris Peripheral Driver Library User's Guide

Example: Do something for 2 seconds then stop

```
volatile uint timer_count;
void ISR(void) {
  timer_count--;
}


int main(void) {
  // initialization code
  SysTickIntRegister(&ISR);
  ... // other init (prev slide)
  timer_count = 2000;
  while(timer_count != 0) {
    ... code to run for 2 seconds
  }
}
```

static variable: declared outside main() puts them in statically allocated memory (not on the stack)

volatile: C keyword to tell the compiler that this variable may change at any time, not (entirely) under the control of this program.

Interrupt service routine

Registering the ISR to be invoked on every SysTick interrupt

Concurrency

```
volatile uint timer_count;
void ISR(void) {
  timer_count--;
}


int main(void) {
  // initialization code
  SysTickIntRegister(&ISR);
  ... // other init
  timer_count = 2000;
  while(timer_count != 0) {
    ... code to run for 2 seconds
  }
}
```

concurrent code: logically runs at the same time. In this case, between any two machine instructions in main() an interrupt can occur and the upper code can execute.

What could go wrong?

Concurrency

```
volatile uint timer_count;
void ISR(void) {
  timer_count--;
}


int main(void) {
  // initialization code
  SysTickIntRegister(&ISR);
  ... // other init
  timer_count = 2000;
  while(timer_count != 0) {
    ... code to run for 2 seconds
  }
}
```

what if the interrupt occurs twice during the execution of this code?

What could go wrong?

Improved Example

```
volatile uint timer_count = 0;
void ISR(void) {
  if(timer_count != 0) {
    timer_count--;
  }
}
int main(void) {
  // initialization code
  SysTickIntRegister(&ISR);
  ... // other init
  timer_count = 2000;
  while(timer_count != 0) {
    ... code to run for 2 seconds
  }
}
```

Reasoning about concurrent code

```
volatile uint timer_count = 0;
void ISR(void) {
  if(timer_count != 0) {
    timer_count--;
  }
}
int main(void) {
  // initialization code
  SysTickIntRegister(&ISR);
  ... // other init
  timer_count = 2000;
  while(timer_count != 0) {
    ... code to run for 2 seconds
  }
}
```

can an interrupt occur here? If it can, what happens?

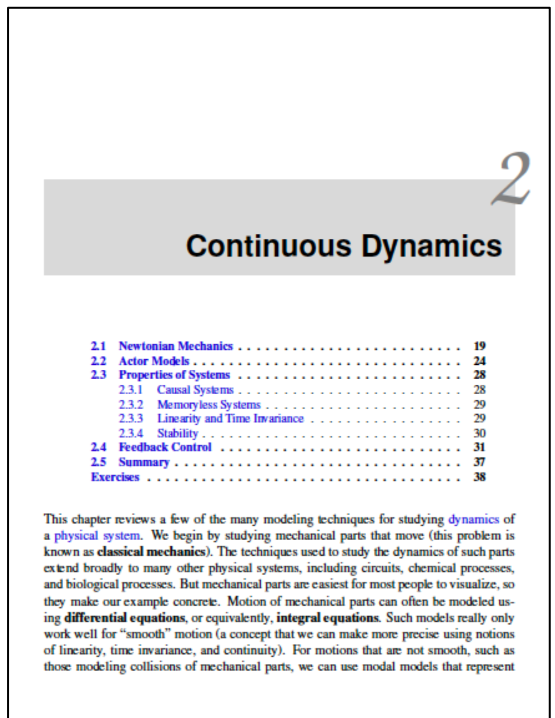
Issues to Watch For

- Interrupt service routine execution time
- Context switch time
- Nesting of higher priority interrupts
- Interactions between ISR and the application
- Interactions between ISRs
- …

- Interrupts introduce a great deal of nondeterminism into a computation. Very careful reasoning about the design is necessary.

Things to do …

- Download the textbook and read Chapter 2
- Complete the assignment and return by Friday August 19 mid-night!
- Read over Workshop 3 and do the pre-workshop work



2

Continuous Dynamics

This chapter reviews a few of the many modeling techniques for studying dynamics of a physical system. We begin by studying mechanical parts that move (this problem is known as **classical mechanics**). The techniques used to study the dynamics of such parts extend broadly to many other physical systems, including circuits, chemical processes, and biological processes. But mechanical parts are easiest for most people to visualize, so they make our example concrete. Motion of mechanical parts can often be modeled using **differential equations**, or equivalently, **integral equations**. Such models really only work well for "smooth" motion (a concept that we can make more precise using notions of linearity, time invariance, and continuity). For motions that are not smooth, such as those modeling collisions of mechanical parts, we can use modal models that represent