COE718: Embedded Systems Design

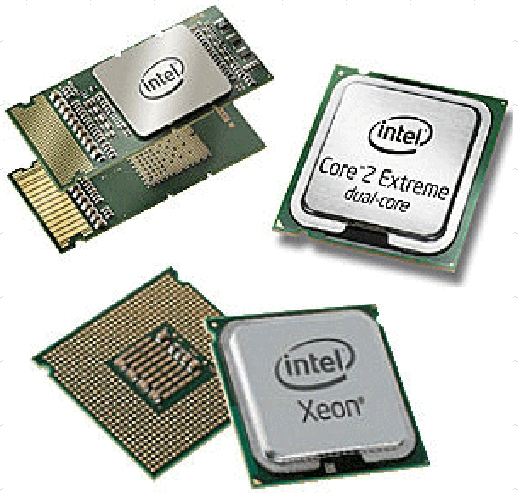
Lecture 11:

Embedded Systems - IO, Memory, Interfacing etc

D.W. Lewis, “Fundamentals of Embedded Software”, Chapter 8 and 11

Anita Tino

Microcontrollers versus Microprocessors



Microcontrollers versus Microprocessors

- uP
  - IC which has a CPU inside (cache and support controllers)
  - Don't have RAM, ROM/Flash or other peripherals embedded on the same die
    - External and flexible
  - Referred to as a "computer" on a chip – now considered SoCs
  - Meant for general purpose applications

Microcontrollers versus Microprocessors

- uC
  - Designed to perform specific tasks
  - i.e. Takes input, processes it, and outputs
    - Same can be argued for a uP
  - Interacts with external world via sensors, and other IOs
  - Consists of RAM, ROM/Flash, IO ports and a "processor" on a single chip

Microcontrollers versus Microprocessors

uP

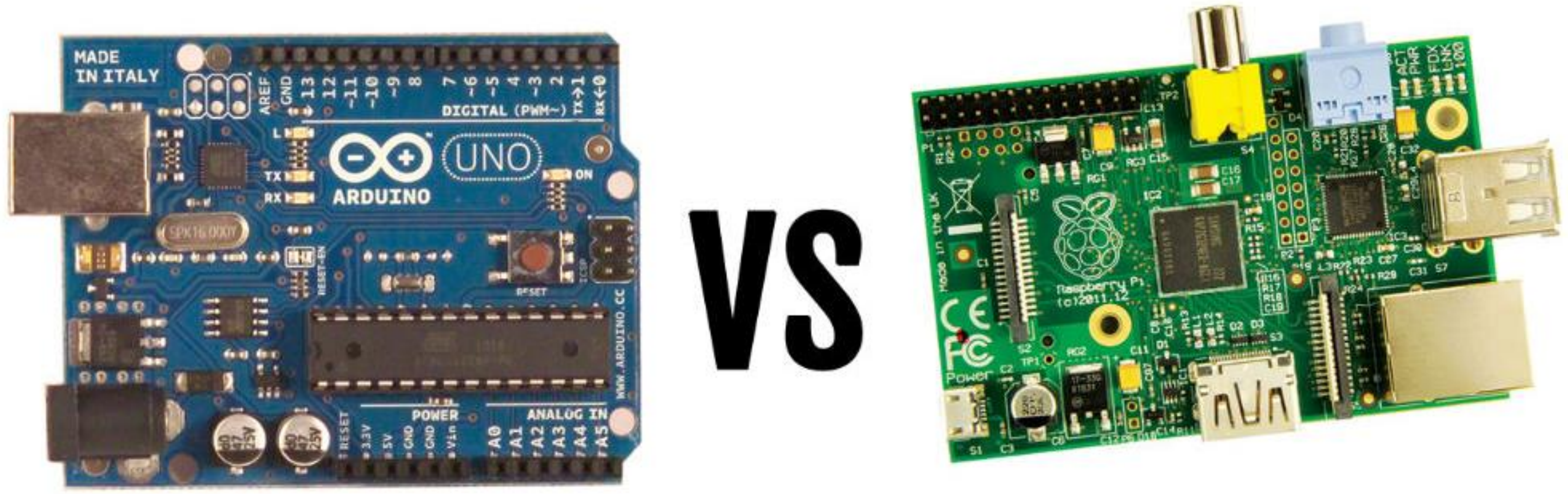
- Executes unspecific tasks, therefore requires high amount of resources such as RAM, ROM, HDD, IOs as well
- Clock speed – 600MHz to 2-3GHz
- Expensive
- Good at computationally intensive tasks

uC

- Specific tasks, therefore has the hardware it needs on board
- Clock speed – 15-60MHz, 100-200MHz
- Cheaper
- Good at processing specific applications, esp requiring IOs

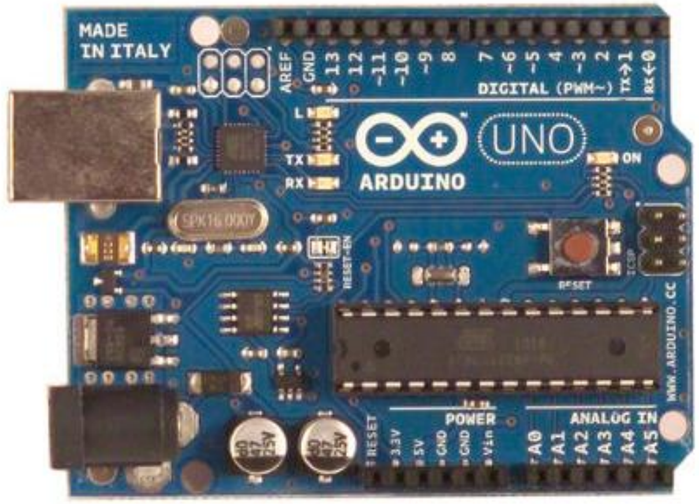
Microcontrollers versus Microprocessors

- Sometimes these lines can get a bit blurry....

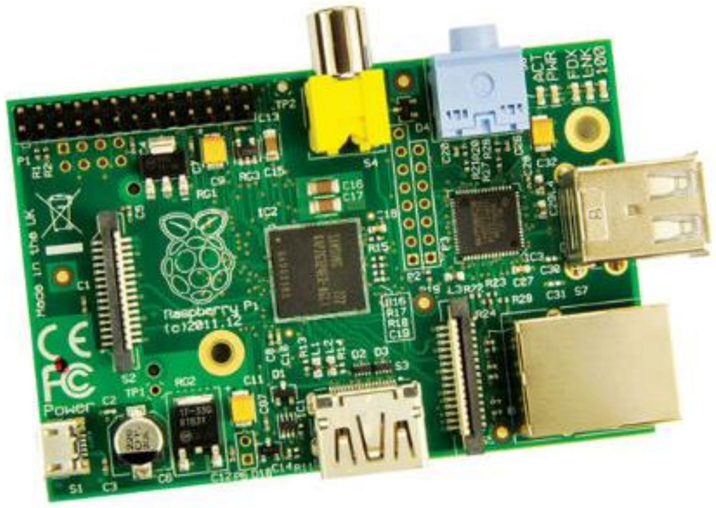


Microcontrollers versus Microprocessors

- Sometimes these lines can get a bit blurry....



VS

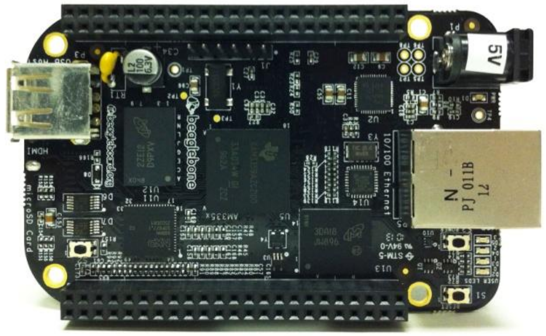


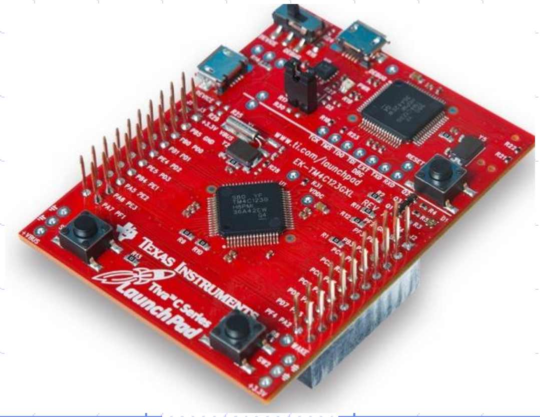
16MHz – ATmega328P
1KB EEPROM, 2KB SRAM, 32KB Flash
6 Analog IO, 14 Digital IO, 6 PWM
SPI, UART, I2C compatible etc

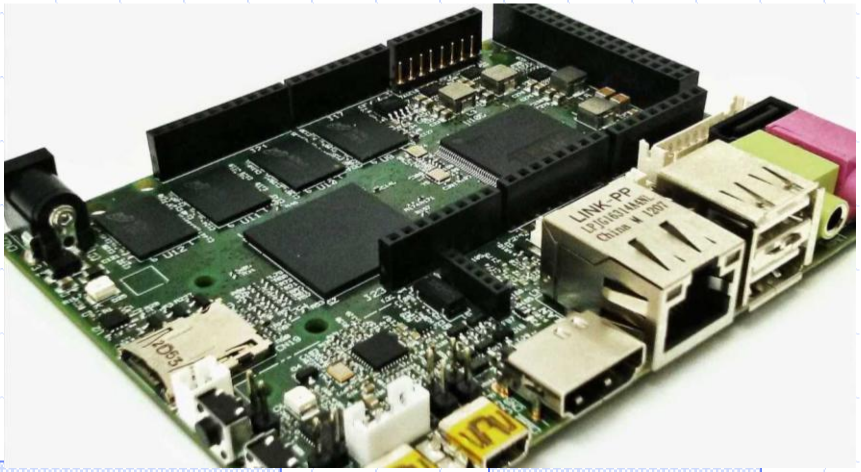
700MHz ARM11XX (Based on BCM2835 SoC)
512MB RAM, Store – uSD
HDMI, USB, Camera slots, Ethernet
8 digital GPIO, No Analog, 1 PWM
SPI, UART, I2C compatible

Microcontrollers versus Microprocessors

- Other solutions depending on your application requirements







Embedded Systems & IO

- IO devices
  - Let’s name a couple…
- Assumption– IO devices are much slower when compared to the speed of a CPU
  - Human interaction
  - Sensor data
  - Communication protocols

Embedded Systems & IO

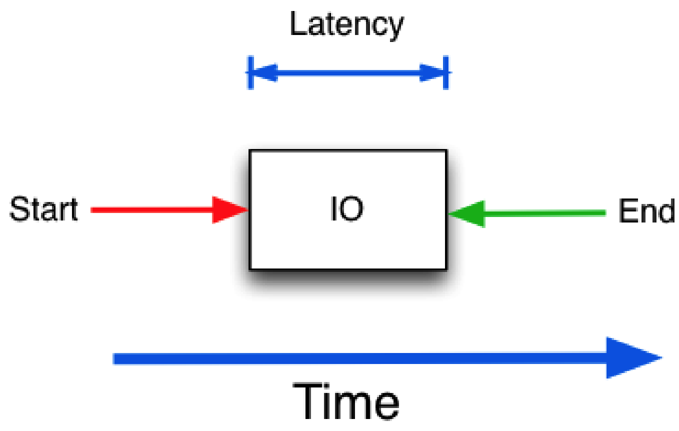
- IO and uP are separate devices
  - Physically independent
- IO devices process events/stimuli (inputs and outputs) and have data available for processing
  - Not under the control of the CPU
- Therefore, IO and uP need to be synchronized somehow
  - See if data is ready, read data etc

Embedded Systems & IO

- Things that we must take into consideration:
- **Transfer Rate** – Measure of the no. of bytes/sec transferred between the uP and device
- **Latency** – measure of the delay from the instant the device is ready until the time the 1st data byte is transferred
  - Aka response time

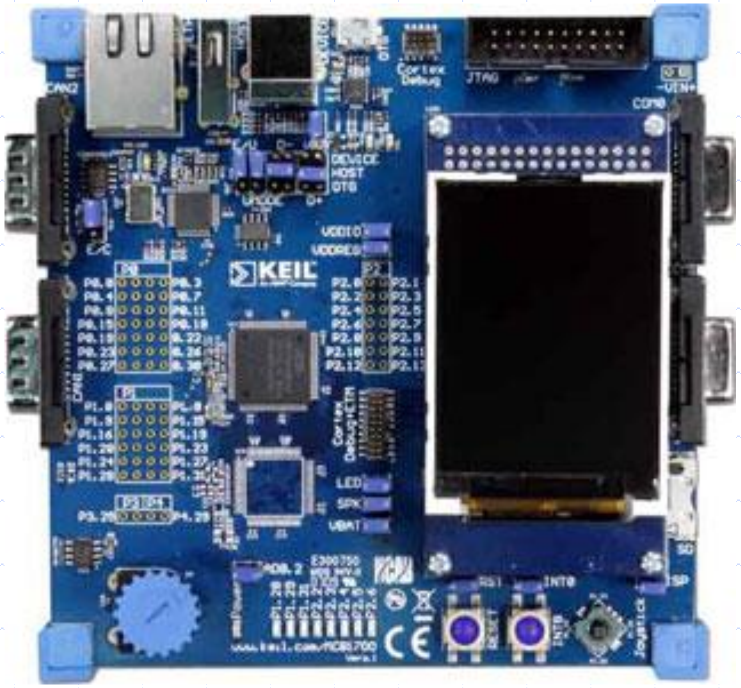
Embedded Systems & IO

- Why is latency important?
  - **Latency** – measure of the delay from the instant the device is ready until the time the 1st data byte is transferred



Embedded Systems & IO

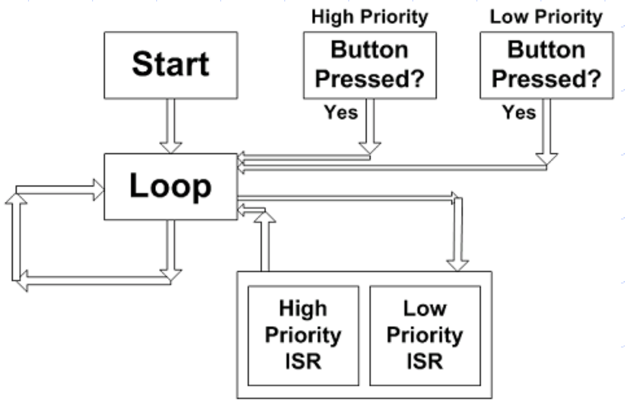
- IOs we've used in the lab?



- If so, how did we retrieve the data?

uP ←→ IO & Data Transfers

1. Polling
2. Interrupt Driven IO
   - Can be timers or input stimuli
3. ??



Polling

Advantages:

- Easiest way to implement data transfers with IO
- Provide good transfer rates

Polling

Advantages:

- Easiest way to implement data transfers with IO
- Provide good transfer rates

What could go wrong?

```
int main(){
…
    while(1){
        x =get_joystick();
        …
        function calls()()
        …
    }
}
```

Polling

Disadvantages:

- Unpredictable response times
  - Can’t afford this in real-time
  - Chance that uP may not even check the device for new data when it arrives bc it’s busy doing something else

Interrupt Driven IO

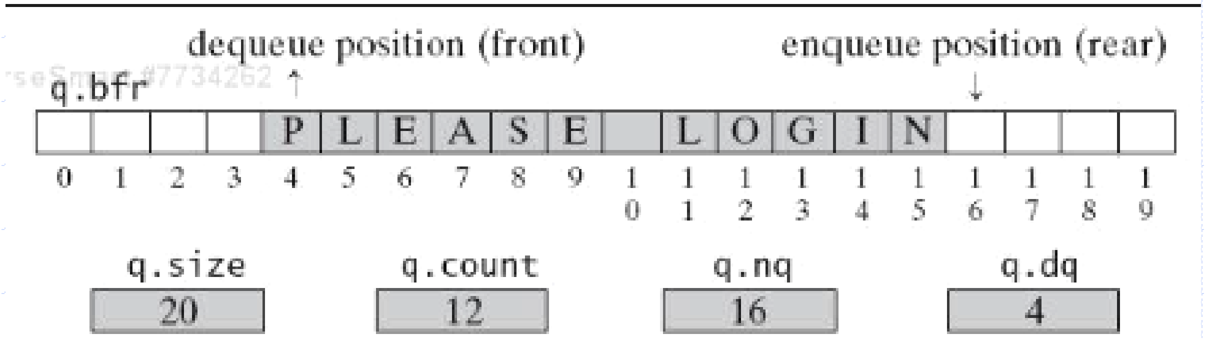
- IO data is transferred and processed in the ISR
- IO data is received in the background
  - Has dedicated hardware that obtains the IO data when ready and places it in a register/buffer
    - Eebot in COE/ELE538
  - ISR reads this data and interprets it (set flags etc)

Interrupt Driven IO

- Data which comes in more frequently uses buffers (FIFOS)
- IO controller manages the placement of data in the buffer
  - uP/ISR manages reading buffered data and processing (in order)
  - Buffers will have control registers that dictate the status to coordinate between the ISR and device

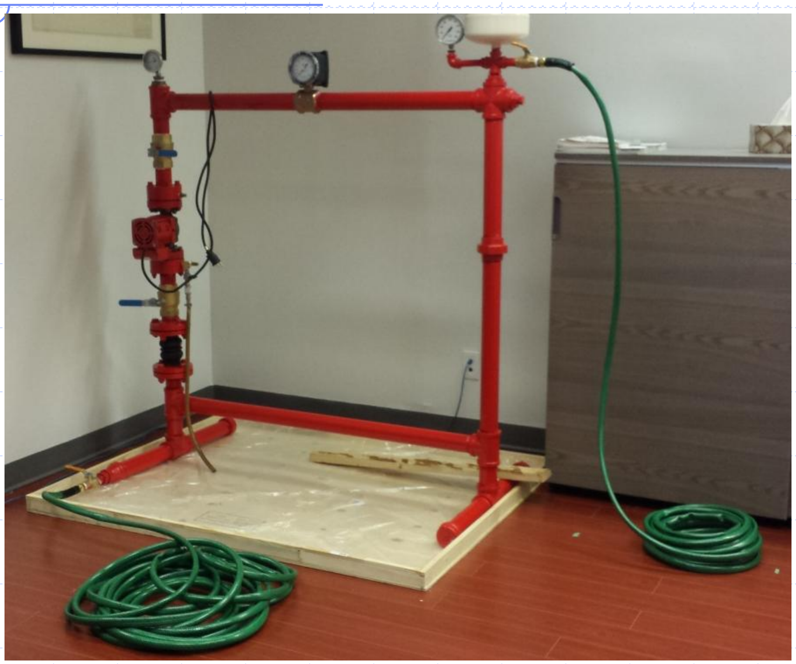
Interrupt Driven IO

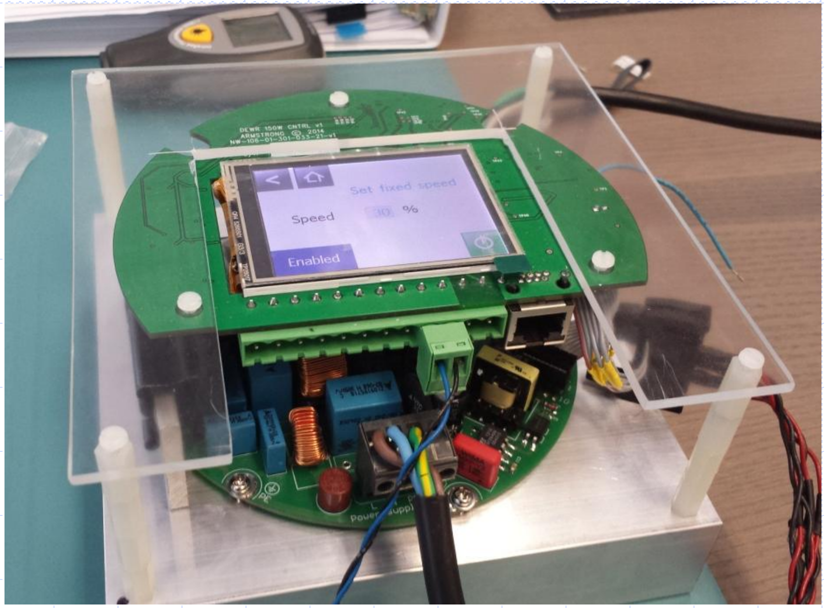
- Keyboard example



-Use APIs, but must call them correctly for functioning

Interrupt Driven IO





Consider the following…

- You have an embedded system that deals with graphics (video stream)
- Or a system that takes pictures with a connected camera which you must store
- Or a general situation where an IO device must transfer and/or store blocks and blocks of data to system memory

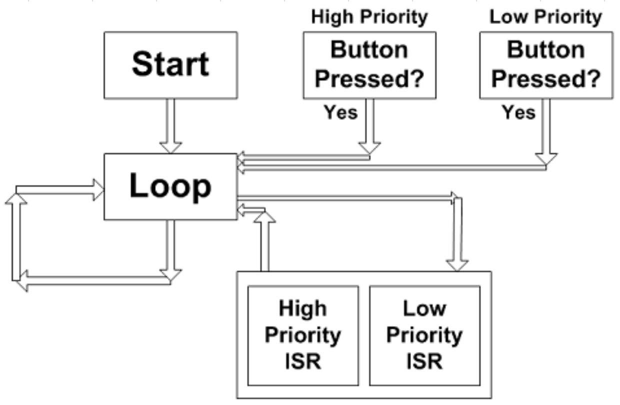
- Will polling and/or IO driven ISRs work?

Consider the following...

- The uP would have to (constantly):
  - Stop what it's doing
  - Transfer data from the IO bus to a buffer/reg
  - Read from the buffer/reg
  - Generate an address to store the data
  - And send the data to memory
- This would likely stall and/or utilize the processor majority of the time while data transfers

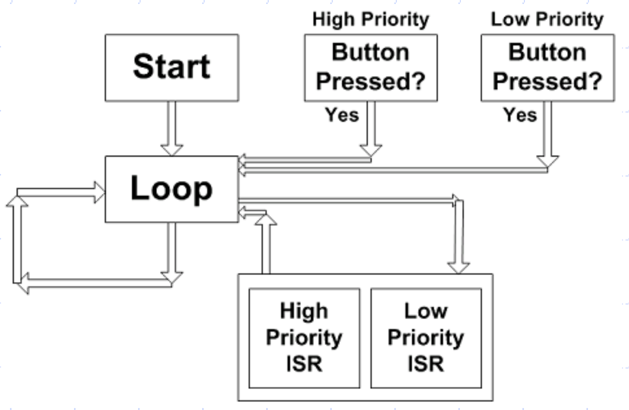
uP ←→ IO & Data Transfers

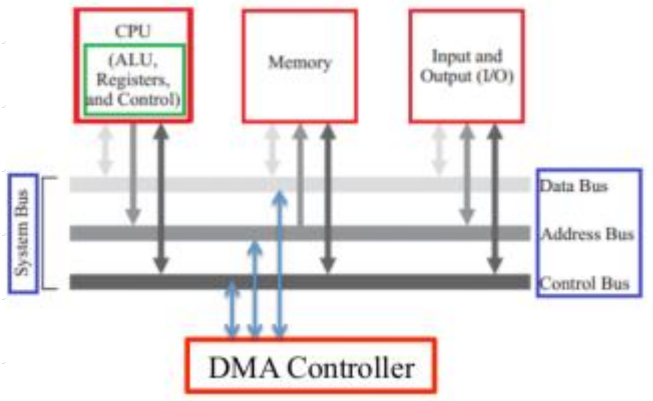
1. Polling
2. Interrupt Driven IO
   - Can be timers or input stimuli
3. ?? -- what do we do for large data blocks?



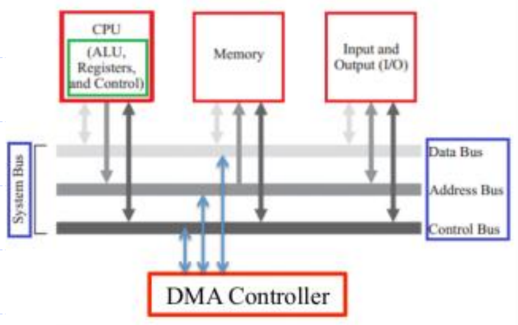
uP ←→ IO & Data Transfers

1. Polling
2. Interrupt Driven IO
   - Can be timers or input stimuli
3. Direct Memory Access (DMA)



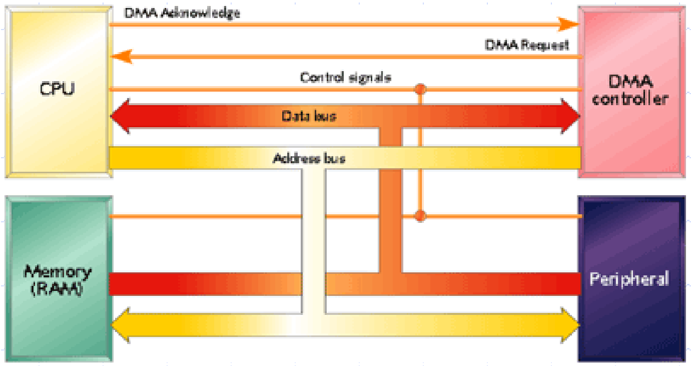


DMA



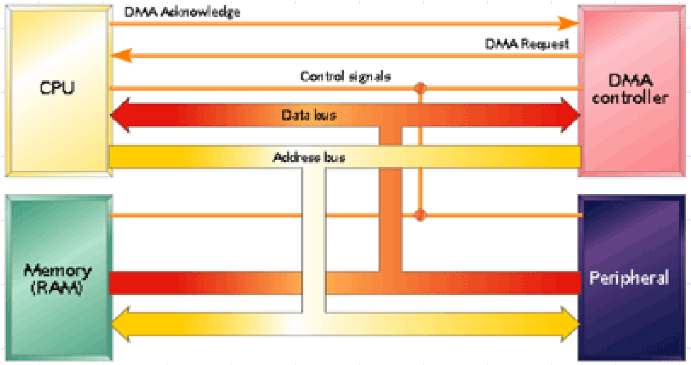
- Avoids using the uP to transfer data
- DMA engine transfers the IO device data directly to memory without CPU intervention
- CPU can go along doing what it needs to do until the DMA is done
  - processor will more than likely need to process this data at some time

DMA



- DMA must therefore generate all the control signal to memory that are necessary to allow the transfers to complete
- Issue – need a bus arbiter to decide who gets the bus at what time
  - Since CPU is also exe, should DMA or CPU get bus? Are there multiple IOs which also need the bus?

DMA



- DMA is initialized through software to specify the source and destinations of the transfer
  - Also number of bytes to transfer, number of transfers etc
- Once initialized, DMA takes over
- Once DMA has completed, generates an interrupt for the CPU to take over

DMA MODES AND TECHNIQUES

Estimating IO Performance

- To design a system that is robust, we must consider various requirements and limitations of our system and the IO device
  - See if our system can keep up (support the requirements) based on the proposed design

Estimating IO Performance

- Consider a system which must read data blocks coming from the RS232 port
- Consider the ARM Cortex-M CPU, running at 50MHz
- Specifications of RS232:
  - Block size = 1B
  - Max rate = 115,200 bits/sec => ~10KB/sec
- What method should we use? Polling, Interrupts, or DMA?

Estimating IO Performance

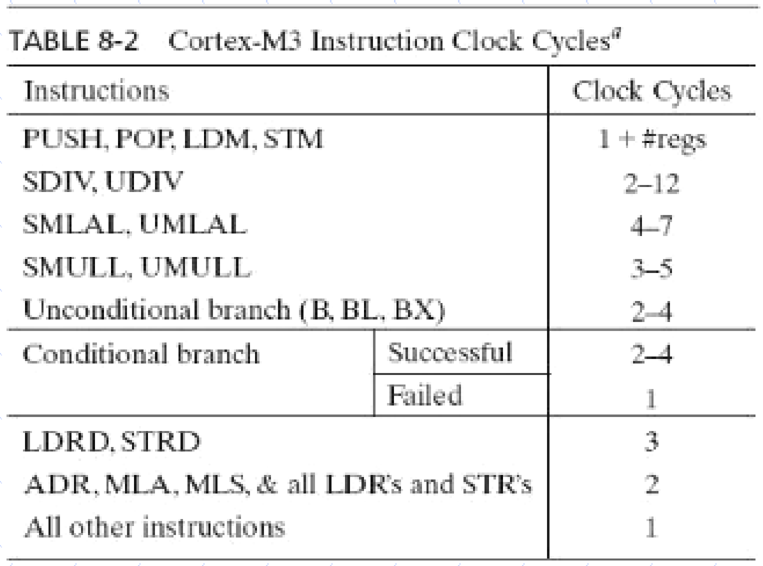


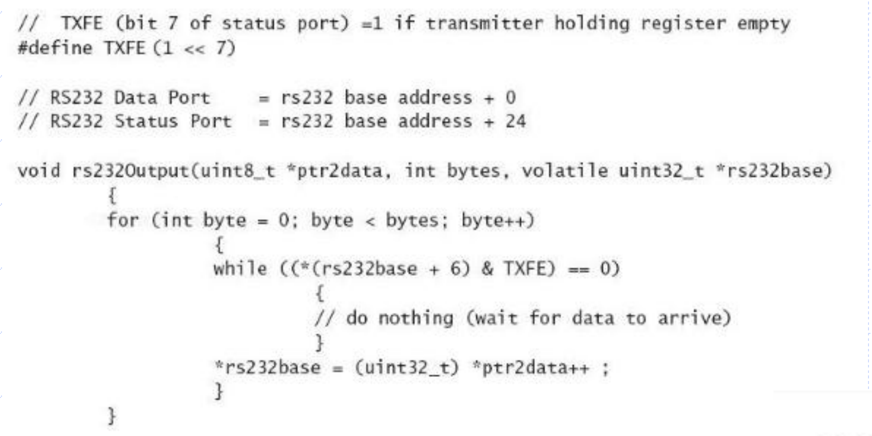
TABLE 8-2 Cortex-M3 Instruction Clock Cycles[a]

Instructions		Clock Cycles
PUSH, POP, LDM, STM		1 + #regs
SDIV, UDIV		2–12
SMLAL, UMLAL		4–7
SMULL, UMULL		3–5
Unconditional branch (B, BL, BX)		2–4
Conditional branch	Successful	2–4
	Failed	1
LDRD, STRD		3
ADR, MLA, MLS, & all LDR's and STR's		2
All other instructions		1

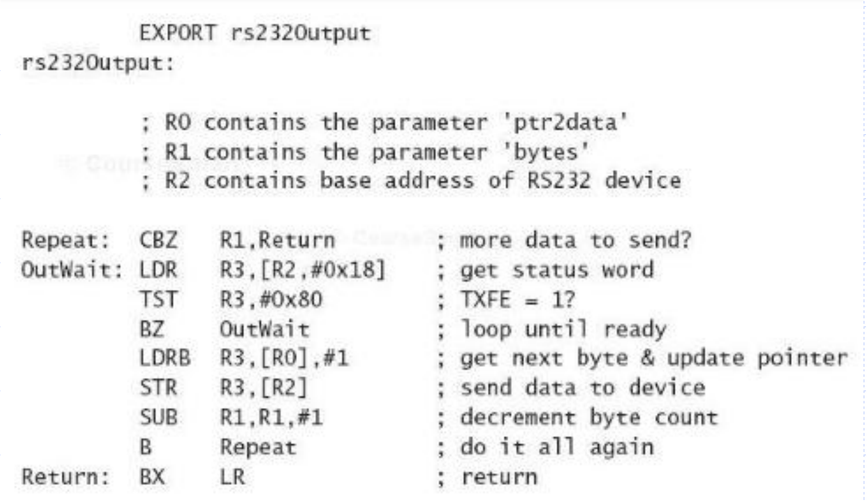
-Assumes CPU and memory run at the same speed

Estimating IO Performance

- Consider **polling**:



```
//  TXFE (bit 7 of status port) =1 if transmitter holding register empty
#define TXFE (1 << 7)

// RS232 Data Port    = rs232 base address + 0
// RS232 Status Port  = rs232 base address + 24

void rs232Output(uint8_t *ptr2data, int bytes, volatile uint32_t *rs232base)
        {
        for (int byte = 0; byte < bytes; byte++)
                  {
                  while ((*(rs232base + 6) & TXFE) == 0)
                           {
                           // do nothing (wait for data to arrive)
                           }
                  *rs232base = (uint32_t) *ptr2data++ ;
                  }
        }
```

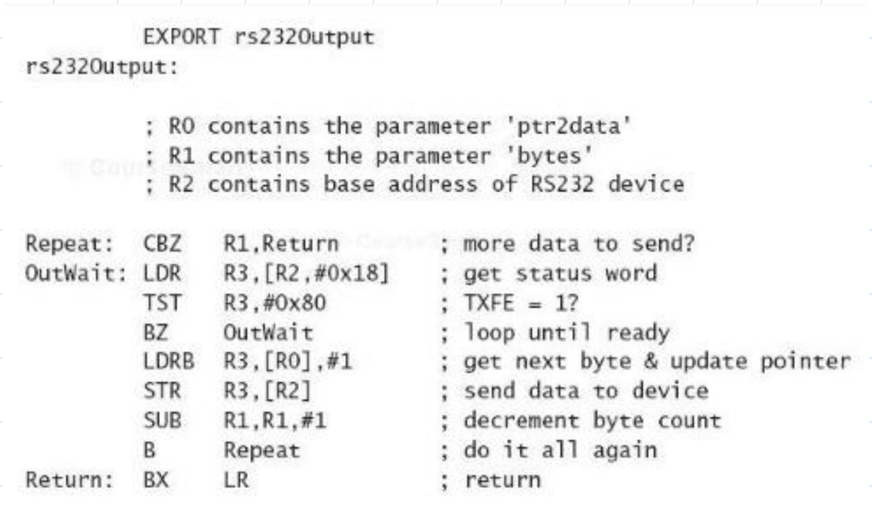


```
          EXPORT rs232Output
rs232Output:

          ; R0 contains the parameter 'ptr2data'
          ; R1 contains the parameter 'bytes'
          ; R2 contains base address of RS232 device

Repeat:   CBZ    R1,Return        ; more data to send?
OutWait:  LDR    R3,[R2,#0x18]    ; get status word
          TST    R3,#0x80         ; TXFE = 1?
          BZ     OutWait          ; loop until ready
          LDRB   R3,[R0],#1       ; get next byte & update pointer
          STR    R3,[R2]          ; send data to device
          SUB    R1,R1,#1         ; decrement byte count
          B      Repeat           ; do it all again
Return:   BX     LR               ; return
```

- Must repeatedly test to see if device is ready
- Unaware of response time
- But once ready, we have a deterministic delay

Estimating IO Performance

- Consider polling:



```
        EXPORT rs232Output
rs232Output:

        ; R0 contains the parameter 'ptr2data'
        ; R1 contains the parameter 'bytes'
        ; R2 contains base address of RS232 device

Repeat:  CBZ     R1,Return        ; more data to send?
OutWait: LDR     R3,[R2,#0x18]    ; get status word
         TST     R3,#0x80         ; TXFE = 1?
         BZ      OutWait          ; loop until ready
         LDRB    R3,[R0],#1       ; get next byte & update pointer
         STR     R3,[R2]          ; send data to device
         SUB     R1,R1,#1         ; decrement byte count
         B       Repeat           ; do it all again
Return:  BX      LR               ; return
```

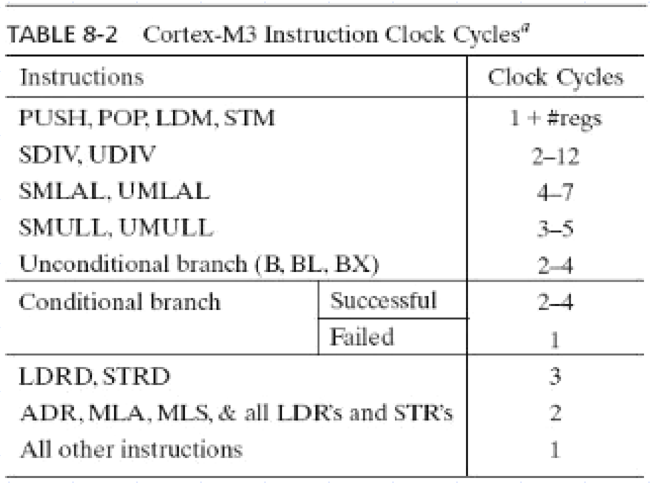


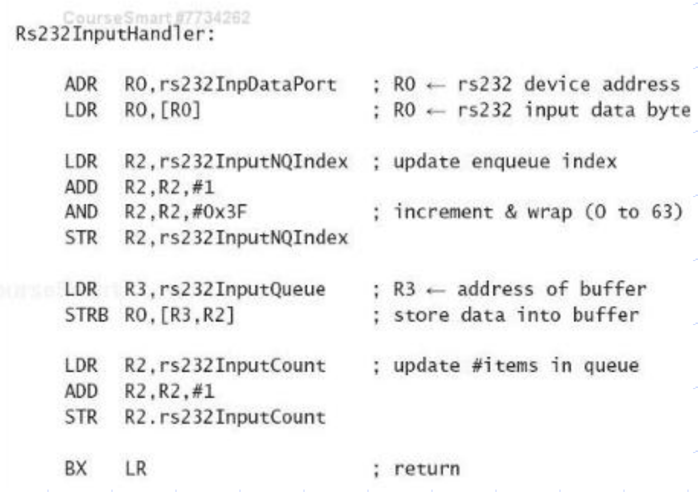
TABLE 8-2 Cortex-M3 Instruction Clock Cycles[a]

Instructions		Clock Cycles
PUSH, POP, LDM, STM		1 + #regs
SDIV, UDIV		2–12
SMLAL, UMLAL		4–7
SMULL, UMULL		3–5
Unconditional branch (B, BL, BX)		2–4
Conditional branch	Successful	2–4
	Failed	1
LDRD, STRD		3
ADR, MLA, MLS, & all LDR's and STR's		2
All other instructions		1

Calculations.....
What is the transfer rate (BW) we can achieve?
Can it support RS232?

Estimating IO Performance

- Consider **Interrupts**:


CourseSmart #7734262

```
Rs232InputHandler:

    ADR   R0,rs232InpDataPort    ; R0 ← rs232 device address
    LDR   R0,[R0]                ; R0 ← rs232 input data byte

    LDR   R2,rs232InputNQIndex   ; update enqueue index
    ADD   R2,R2,#1
    AND   R2,R2,#0x3F            ; increment & wrap (0 to 63)
    STR   R2,rs232InputNQIndex

    LDR   R3,rs232InputQueue     ; R3 ← address of buffer
    STRB  R0,[R3,R2]             ; store data into buffer

    LDR   R2,rs232InputCount     ; update #items in queue
    ADD   R2,R2,#1
    STR   R2.rs232InputCount

    BX    LR                     ; return
```

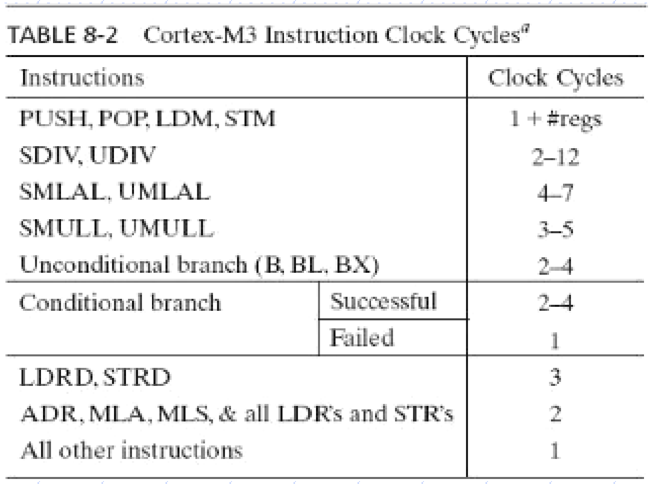
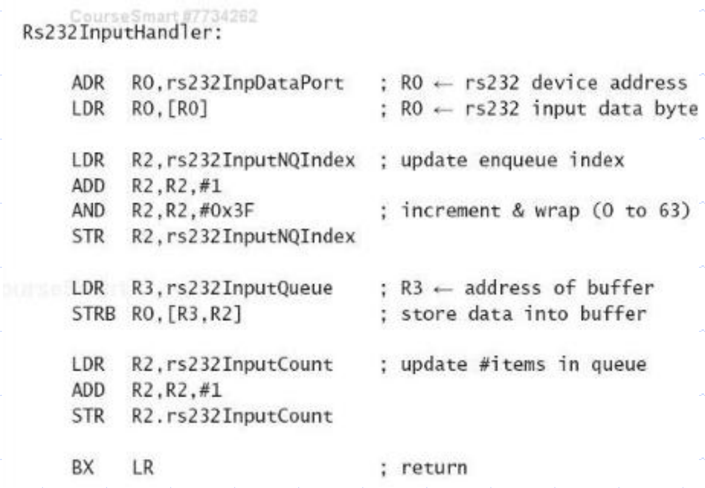


TABLE 8-2 Cortex-M3 Instruction Clock Cycles[a]

Instructions		Clock Cycles
PUSH, POP, LDM, STM		1 + #regs
SDIV, UDIV		2–12
SMLAL, UMLAL		4–7
SMULL, UMULL		3–5
Unconditional branch (B, BL, BX)		2–4
Conditional branch	Successful	2–4
	Failed	1
LDRD, STRD		3
ADR, MLA, MLS, & all LDR's and STR's		2
All other instructions		1

- Assume tail chaining -> 6 c.c

Estimating IO Performance

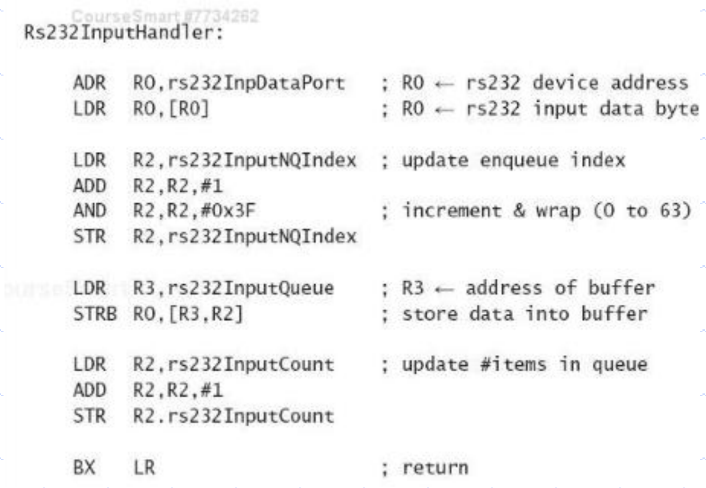


CourseSmart #7734262

```
Rs232InputHandler:

    ADR   R0,rs232InpDataPort    ; R0 ← rs232 device address
    LDR   R0,[R0]                ; R0 ← rs232 input data byte

    LDR   R2,rs232InputNQIndex   ; update enqueue index
    ADD   R2,R2,#1
    AND   R2,R2,#0x3F            ; increment & wrap (0 to 63)
    STR   R2,rs232InputNQIndex

    LDR   R3,rs232InputQueue     ; R3 ← address of buffer
    STRB  R0,[R3,R2]             ; store data into buffer

    LDR   R2,rs232InputCount     ; update #items in queue
    ADD   R2,R2,#1
    STR   R2.rs232InputCount

    BX    LR                     ; return
```

- Gave us less than polling
  - Overhead for interrupt handling costs!
- How can we improve this cost?

Estimating IO Performance



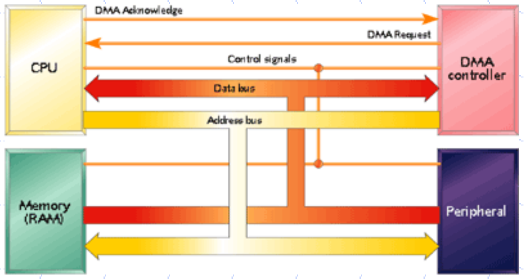
CourseSmart #7734262

```
Rs232InputHandler:

    ADR  R0,rs232InpDataPort    ; R0 ← rs232 device address
    LDR  R0,[R0]                ; R0 ← rs232 input data byte

    LDR  R2,rs232InputNQIndex   ; update enqueue index
    ADD  R2,R2,#1
    AND  R2,R2,#0x3F            ; increment & wrap (0 to 63)
    STR  R2,rs232InputNQIndex

    LDR  R3,rs232InputQueue     ; R3 ← address of buffer
    STRB R0,[R3,R2]             ; store data into buffer

    LDR  R2,rs232InputCount     ; update #items in queue
    ADD  R2,R2,#1
    STR  R2.rs232InputCount

    BX   LR                     ; return
```

- Gave us less than polling
  - Overhead for interrupt handling costs!
- How can we improve this cost?
  - Only transferring 1B at a time; why not more?

Estimating IO Performance

DMA Method



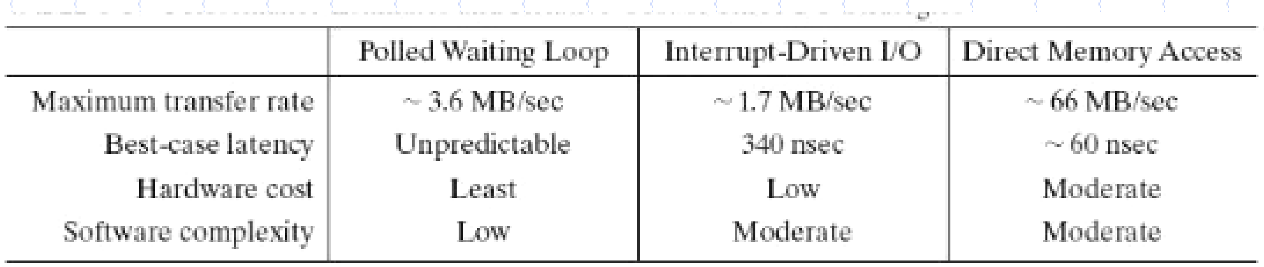
- Cost of additional hardware
- Remember that memory, IO and DMA are external to the CPU, so can not consider c.c time directly as in other cases
  - We consider a multiple of CPU's c.c time

Estimating IO Performance

DMA Method

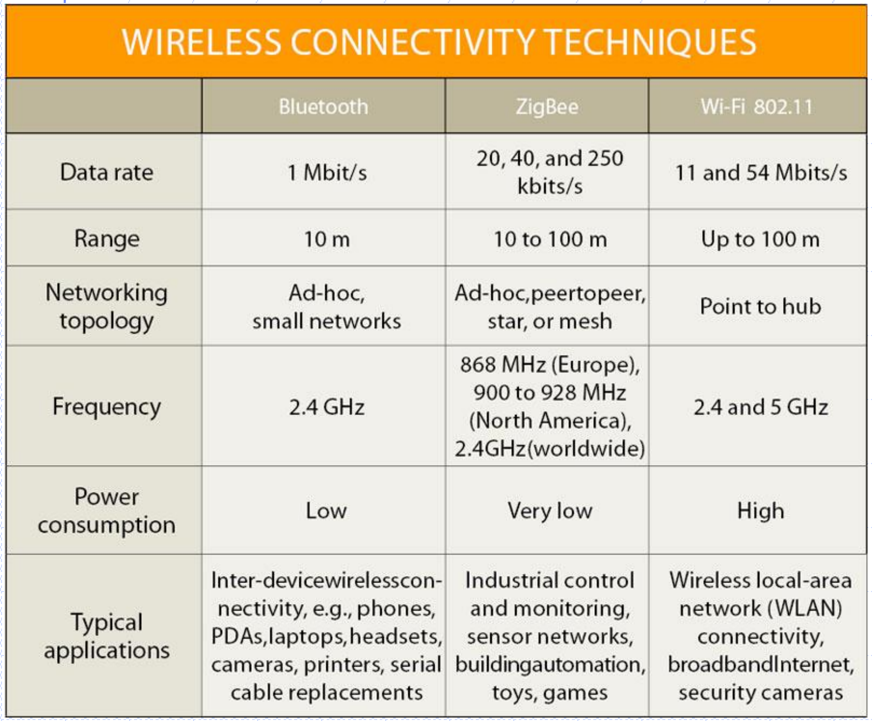
- Assume 32b are transferred / c.c
- Assume we require additional 2-3c.c in comparison to the CPU (i.e. Lower clock rate of IO, memory and DMA)
  - Assume: access/grant and release of bus with additional 2 c.c + 2c.c
  - Conclusion: working at 60ns/c.c with DMA
- Transfer rate?

Estimating IO Performance

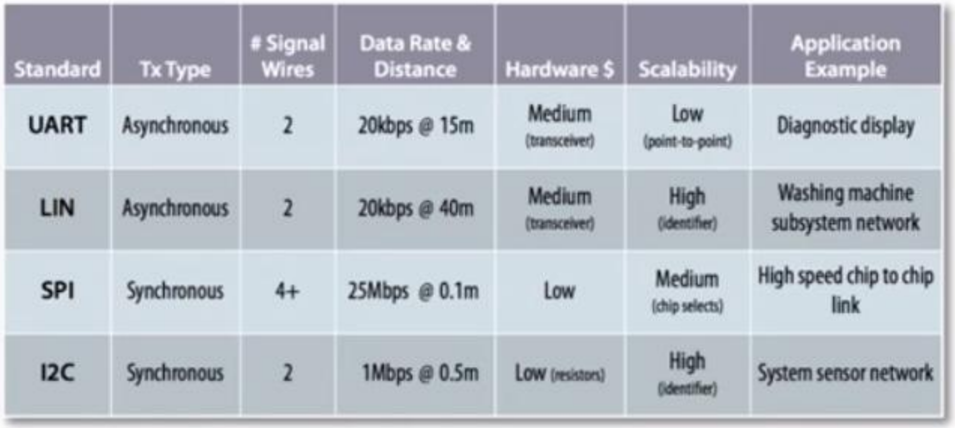


	Polled Waiting Loop	Interrupt-Driven I/O	Direct Memory Access
Maximum transfer rate	~ 3.6 MB/sec	~ 1.7 MB/sec	~ 66 MB/sec
Best-case latency	Unpredictable	340 nsec	~ 60 nsec
Hardware cost	Least	Low	Moderate
Software complexity	Low	Moderate	Moderate

Other Communication Protocols



WIRELESS CONNECTIVITY TECHNIQUES			
	Bluetooth	ZigBee	Wi-Fi 802.11
Data rate	1 Mbit/s	20, 40, and 250 kbits/s	11 and 54 Mbits/s
Range	10 m	10 to 100 m	Up to 100 m
Networking topology	Ad-hoc, small networks	Ad-hoc,peertopeer, star, or mesh	Point to hub
Frequency	2.4 GHz	868 MHz (Europe), 900 to 928 MHz (North America), 2.4GHz(worldwide)	2.4 and 5 GHz
Power consumption	Low	Very low	High
Typical applications	Inter-devicewirelesscon-nectivity, e.g., phones, PDAs,laptops,headsets, cameras, printers, serial cable replacements	Industrial control and monitoring, sensor networks, buildingautomation, toys, games	Wireless local-area network (WLAN) connectivity, broadbandInternet, security cameras



Standard	Tx Type	# Signal Wires	Data Rate & Distance	Hardware $	Scalability	Application Example
UART	Asynchronous	2	20kbps @ 15m	Medium (transceiver)	Low (point-to-point)	Diagnostic display
LIN	Asynchronous	2	20kbps @ 40m	Medium (transceiver)	High (identifier)	Washing machine subsystem network
SPI	Synchronous	4+	25Mbps @ 0.1m	Low	Medium (chip selects)	High speed chip to chip link
I2C	Synchronous	2	1Mbps @ 0.5m	Low (resistors)	High (identifier)	System sensor network

-GSM/GPRS (i.e. 3GSM = 600Kb/s -> 1.4Mb/s
-Ethernet (10Mb/sec)
-USB2.0 => 480 Mb/s (60MB/s), USB3.0 => 640MB/s (5Gb/s)

Memory Requirements in ES

- Memory is not usually in abundance in embedded systems... Why?
- How do you measure memory needs of your system?
- If you anticipate incorrectly, what should you consider?

Memory Requirements in ES

- Take a look at the final project....
  - Cortex-M3 running @ 100MHz
  - 64KB on-chip RAM
  - 512KB on-chip Flash

- Code = ?
- Avg picture size (.c) = ?
- => Take a look at .axf file
  - 498KB

Memory Management in SW

- Important when dealing with interrupts, and/or multiple tasks accessing the same variables and data

const

static

extern

volatile

uint_t

Memory Management in SW

const

- Qualifier which means that this variable cannot be modified throughout the code
- Has static storage duration
- Example – images in final project

```
const unsigned char myimage[]
```

Memory Management in SW

extern

- Extends visibility of variables and C functions
  - Mostly for variables, extern is implied for functions declared/included in .h
- Allows us to define a variable in 1 file, and set a value(s) to it in another file
  - Must be re-declared in the file of use as a global variable (only 1st time used)

Memory Management in SW

volatile

- Compiler cannot apply optimizations to the variable, or reorder access to the variable

- Example – Bit banding

Memory Management in SW

static

- Permanent variables within their function scope
  - Unlike global variables, not known outside their function or file
- Static maintains its value between function calls
  - Tells compiler to make variable limited in scope while allowing it to persist throughout the life of the function

Memory Management in SW

static

- Stored to global memory vs stack
- Variable is only initialized the 1st the time the function is called, can not be changed until the next function call

- Example 1
- Example 2

Memory Management in SW

register

- Variable will be used frequently
- You as the programmer are trying to give a hint to the compiler (for optimization) to suggest this variable is important
  - Want the CPU to hold the variable in a register vs keep str/ld back to register

Memory Management in SW

uint_t, int_t

- Allows the programmer to specify the width of the integer type
  - i.e. Number of bits needed in the variable
  - Use <stdinth>
  - Ex: uint_8 var = 0xFF;
  - Ex: uint_16 var2 = 0xDEAF
- Especially useful when designing hw/sw systems that send data back & forth

Image Processing

- In general and in embedded systems

OpenCV

- Processes real-time, graphical applications
- C++, C, Python and Java support
- Open source computer vision and machine learning software library

https://www.youtube.com/watch?v=o3LSOq6OC4I
https://www.youtube.com/watch?v=bcswZLwhTUI