ECE3411 – Fall 2017
Lec6a.

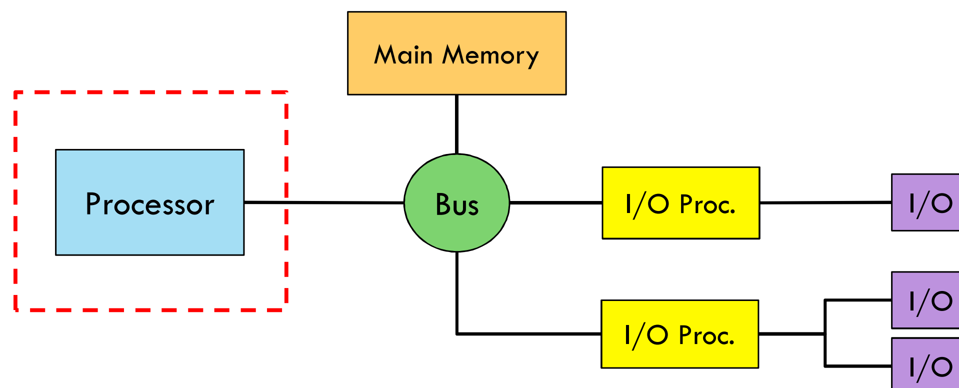
Advanced topics in Embedded Systems Design

Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: marten.van_dijk@uconn.edu

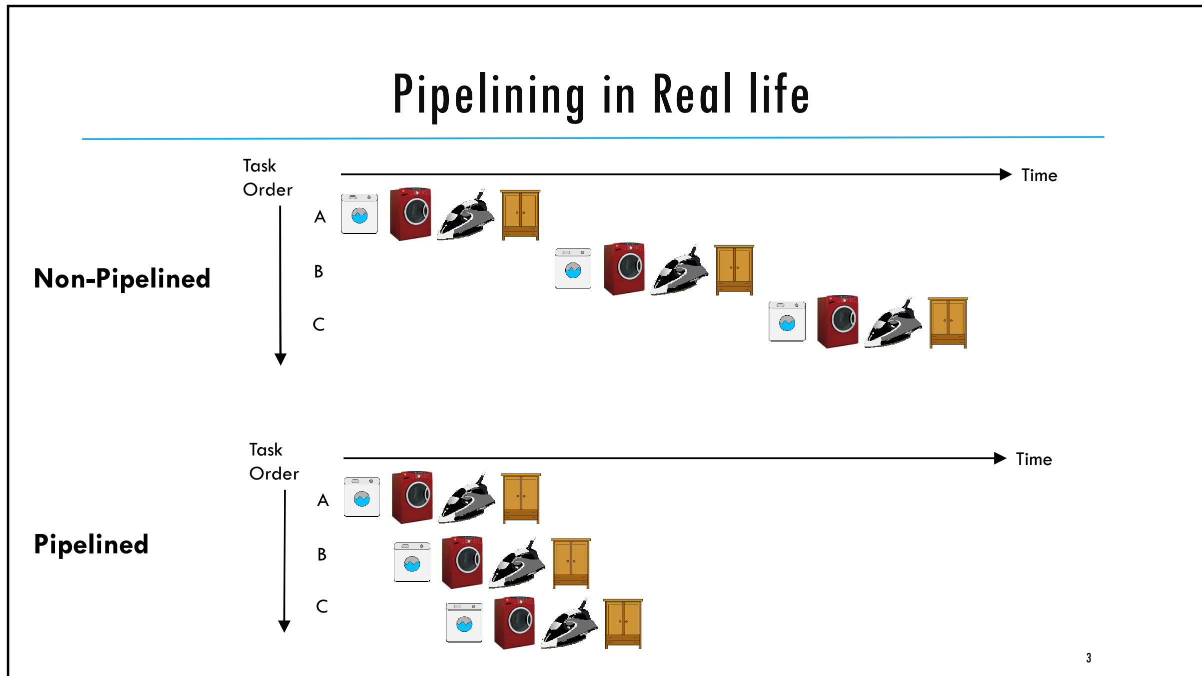
UCONN

Slides are copied from Lecture 7a, ECE3411 – Fall 2015
by Marten van Dijk and Syed Kamran Haider.

Basic Computer Components



2

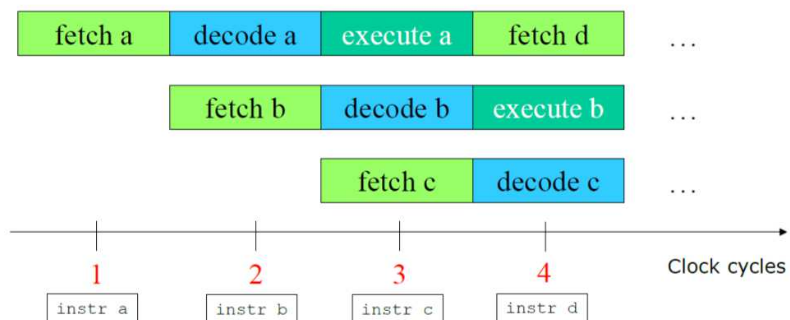


Instruction Pipelining

- Method for increasing the instruction throughput of the processor [IPS]
- Instruction execution may be partitioned into a fixed number of sequential steps, e.g. (ARM7):
  - Fetch (instruction from memory)
  - Decode (opcode and operands)
  - Execute…
- Stages may be executed in parallel, e.g. the CPU works with several instructions at the same time
  → Increased Throughput
- Some instructions and code sequences may reduce the performance gain (pipeline stalls) because of dependencies
  → Developing techniques to avoid stalls is very "hot" in current microprocessor research

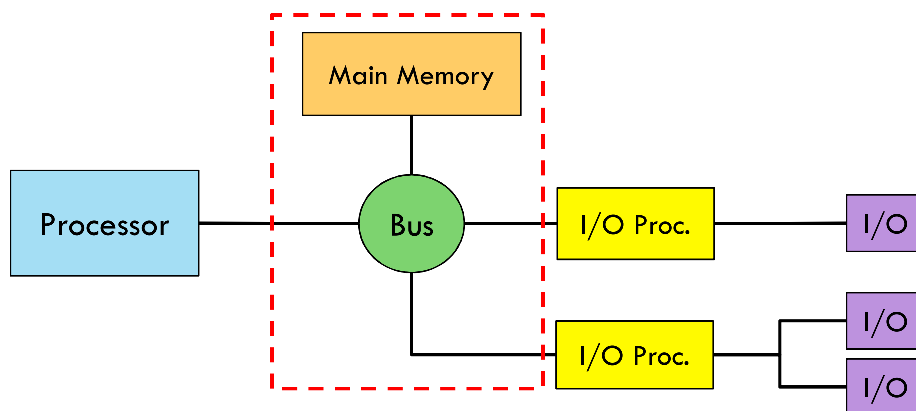
4

3-Stage Pipeline



5

Basic Computer Components



6

Main Memory

- Stores instructions and data for the processor
- Connected to the processor via the memory bus

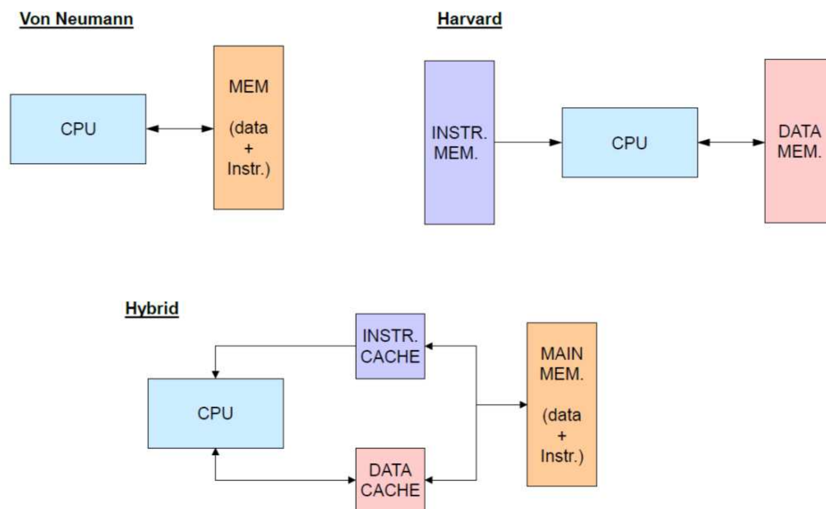
Two main types:

- Volatile, RAM (random byte wise read and write)
  - SRAM
  - DRAM
- Non-volatile, ROM (read (and sometimes write…))
  - OTPROM
  - EPROM
  - EEPROM
  - FLASH
  - FRAM

Memory Architectures

- Von Neumann machine/architecture
  - Memory viewed as a long continuous column of memory cells
  - No principal difference between data and instructions
  - No difference between different data types
  - Common physical storage for instructions and data
- Harvard architecture
  - Principal difference between data and instructions
  - Physically separate memories and busses for data and instructions
  - May have different word length for data and instructions
- Hybrid architecture
  - Harvard architecture between CPU and cache memory
  - In case of cache-miss: Von Neumann between CPU and main memory

Memory Architectures



9

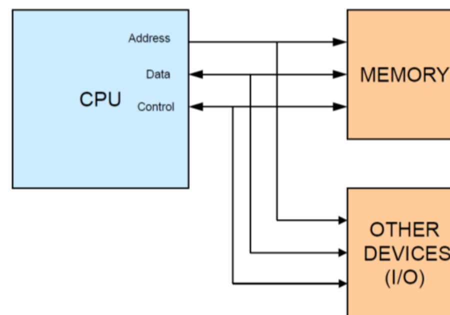
Bus and Communication Interfaces

- Parallel Bus Systems
  - Processor Buses – AVR etc.
  - Industrial Buses
    - VMEbus
    - CompactPCI
    - PC/104
    - ...
- Serial Local Buses
  - SPI
  - MicroWire
  - I2C
  - 1-Wire
- Serial Lines (1 to 1, 1 to N)
  - UART
  - RS-232C
  - RS-422
  - USB
- Networks (N to M)
  - CAN
  - RS-485
  - LAN/Ethernet
- Wireless Communication
  - IR/IrDA
  - ISM
  - WiFi
  - Bluetooth
  - Zigbee

10

Parallel Bus Interfaces

- A bus is defined as a group of signal lines that shares a common function and that connects the processor to the memory and I/O devices in the system
- The “three bus” system is the most common parallel bus architecture:
  - Address
  - Data
  - Control



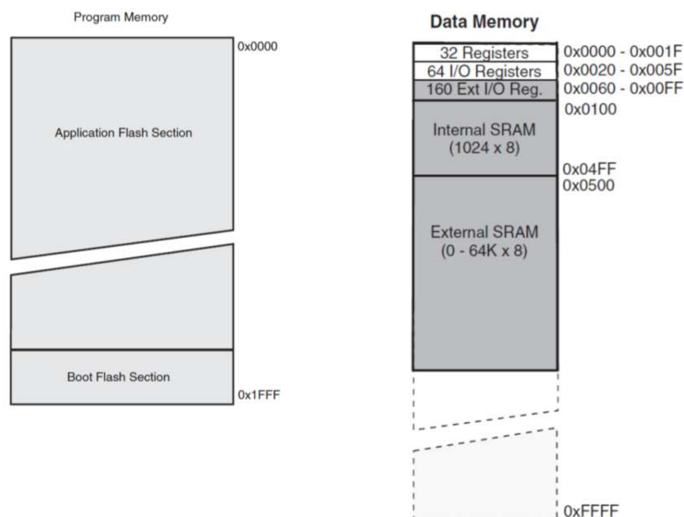
11

Address Space

- The range of memory locations addressable by a processor.
- Typically reflected by the width of the address bus
  - 16 bit → 2^{16} = 64 KB
  - 32 bit → 2^{32} = 4 GB
- Linear (flat) address space
  - One contiguous block of bytes (words)
  - Logical address = physical address
- Paged, Segmented
  - Organized in pages and/or segments:offsets (logical addresses)
  - Conversion between logical & physical addresses needed

12

Address space of AVR ATmega162

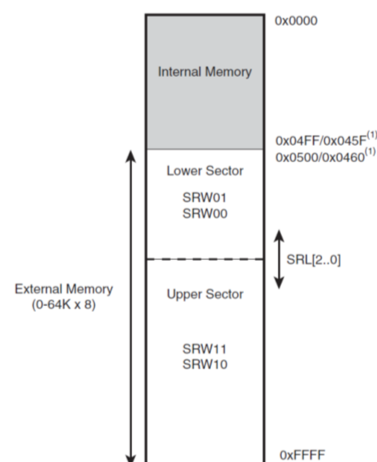


13

Interfacing External SRAM to ATmega162

The External Memory interface consists of:

- AD7:0 → Multiplexed low-order address bus and data bus
- A15:8 → High-order address bus (configurable number of bits)
- ALE → Address latch enable
- RD → Read strobe
- WR → Write strobe



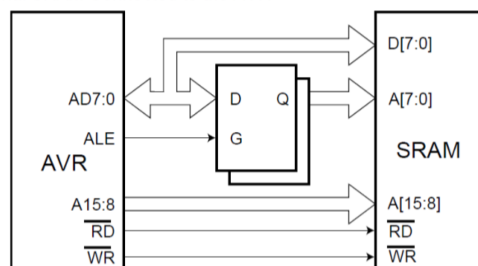
14

Interfacing External SRAM to ATmega162

The External Memory interface consists of:

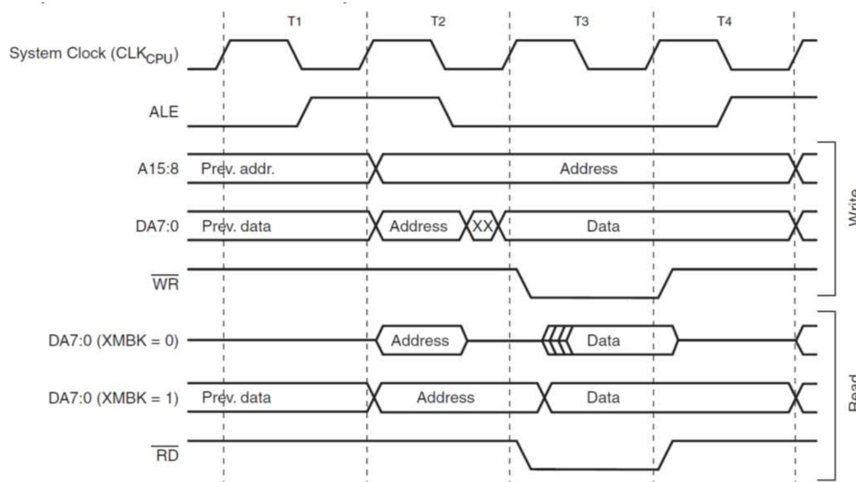
- AD7:0 → Multiplexed low-order address bus and data bus
- A15:8 → High-order address bus (configurable number of bits)
- ALE → Address latch enable
- RD → Read strobe
- WR → Write strobe

External SRAM Connected to the AVR



15

Processor/Bus cycle ATmega162 (without wait states)



16

Wait states

Problem: The processor is normally much faster than the peripheral devices connected to the bus interface

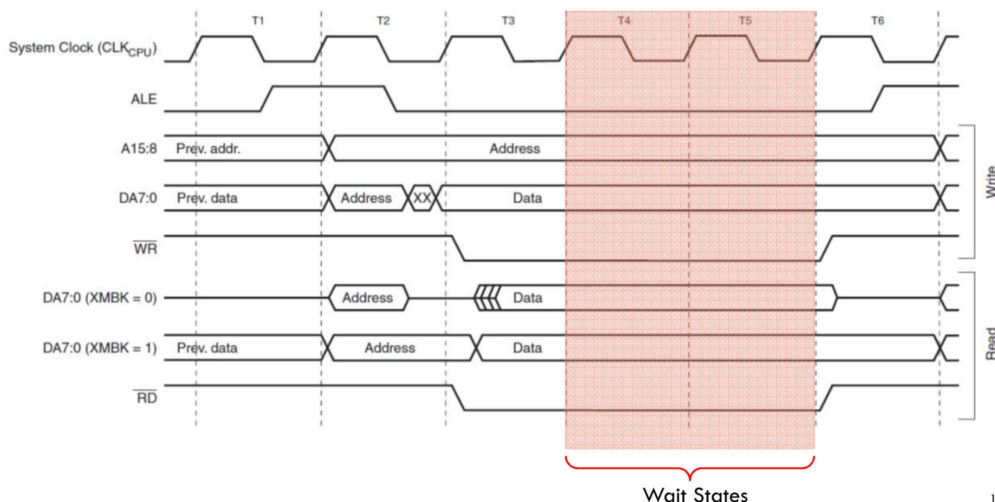
- Peripheral devices may not be able to respond to processor requests within the next clock state
- Data on the bus may be invalid when they are read by the processor

Solution: *Wait states*

- Synchronous processors: Injects one or more extra clock cycles into the bus cycle.
- Asynchronous processors: Delayed assertion of the DTACK signal.

17

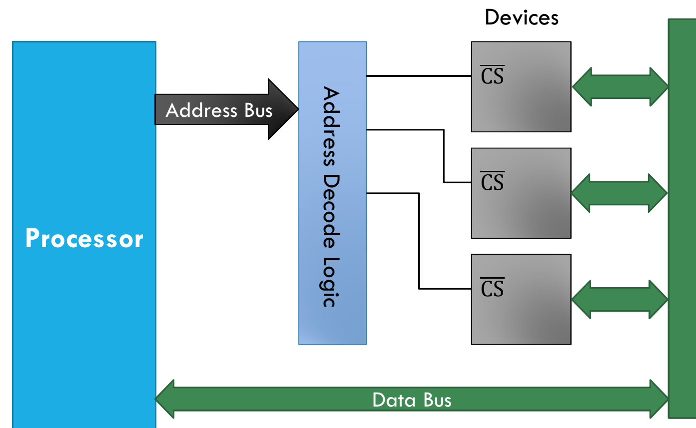
Processor/Bus cycles ATmega162 – 2 wait states



18

Address Decoding

- Address space is divided among several devices.
- Address Decoding Logic is configured according to the address space mapping.
- Address Decoding logic enables device(s) based on the address requested by the processor.



19

Example: Memory Map & Address Decoding

Design an AVR-based computer that includes four devices in its address space:

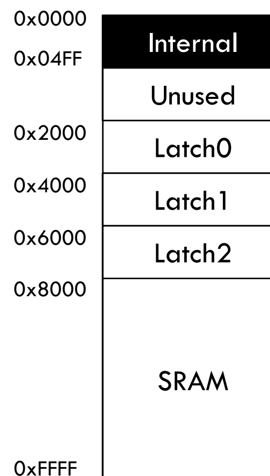
- One SRAM 32K, for data storage
- Digital outputs for driving 24 LEDs using three 8-bit latches

Make a memory map and address decoder for the system (use partial decoding)

20

Selected Memory Map

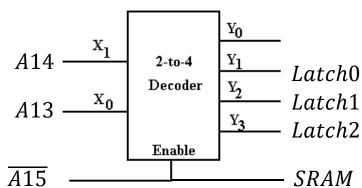
Device	Address Range	A15 … A0
Internal/Unused	0x0000 – 0x1FFF	0000 0000 0000 0000 0001 1111 1111 1111
Latch0	0x2000 – 0x3FFF	0010 0000 0000 0000 0011 1111 1111 1111
Latch1	0x4000 – 0x5FFF	0100 0000 0000 0000 0101 1111 1111 1111
Latch2	0x6000 – 0x7FFF	0110 0000 0000 0000 0111 1111 1111 1111
SRAM	0x8000 – 0xFFFF	1000 0000 0000 0000 1111 1111 1111 1111

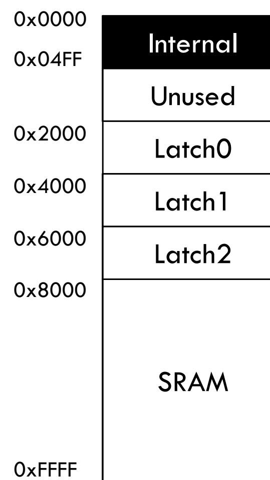


21

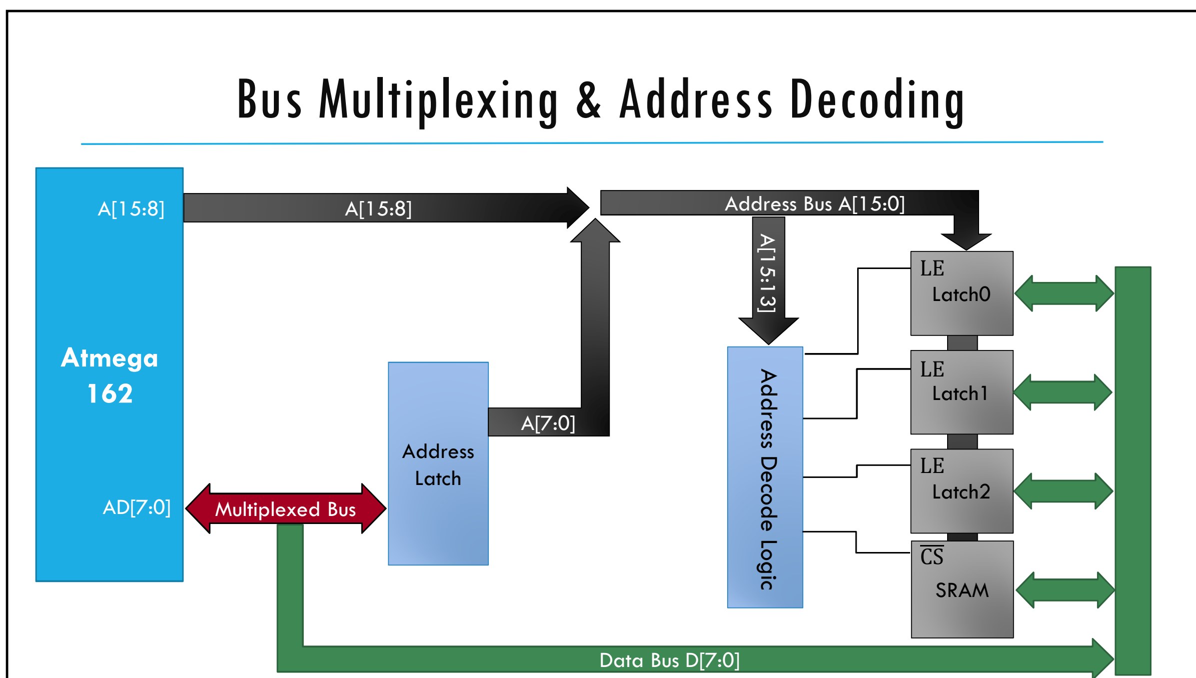
Address Decoding of selected Memory Map

Device	Address Range	A15 … A0
Internal/Unused	0x0000 – 0x1FFF	000x xxxx xxxx xxxx
Latch0	0x2000 – 0x3FFF	001x xxxx xxxx xxxx
Latch1	0x4000 – 0x5FFF	010x xxxx xxxx xxxx
Latch2	0x6000 – 0x7FFF	011x xxxx xxxx xxxx
SRAM	0x8000 – 0xFFFF	1xxx xxxx xxxx xxxx

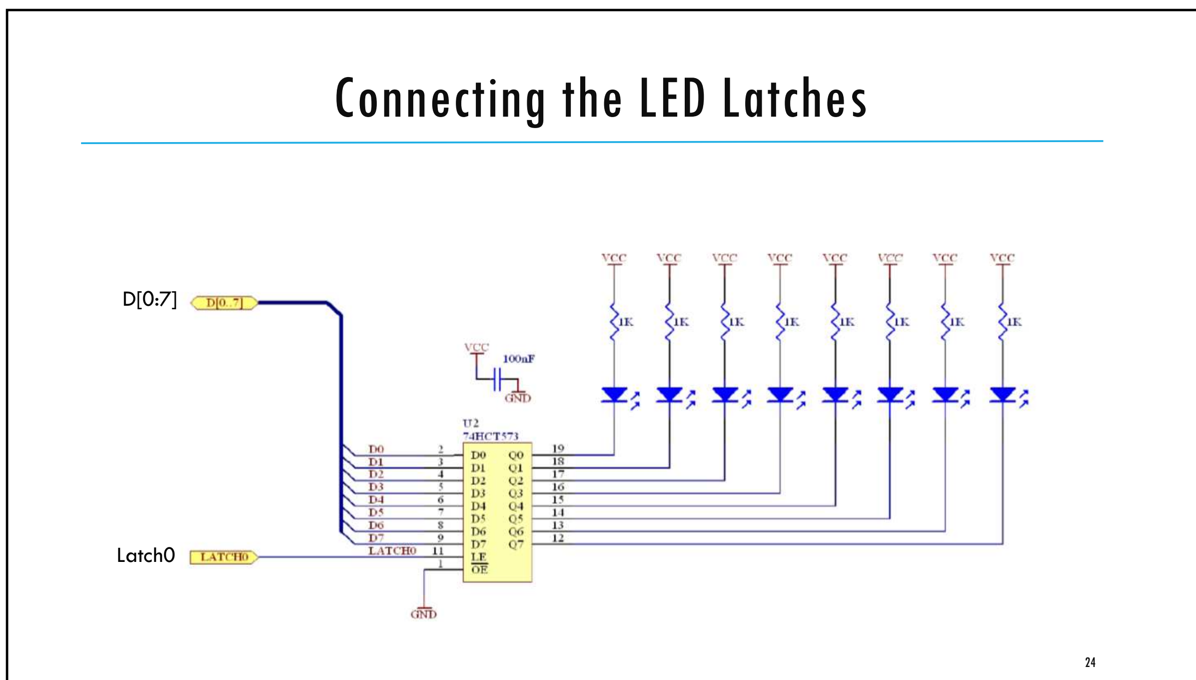




22



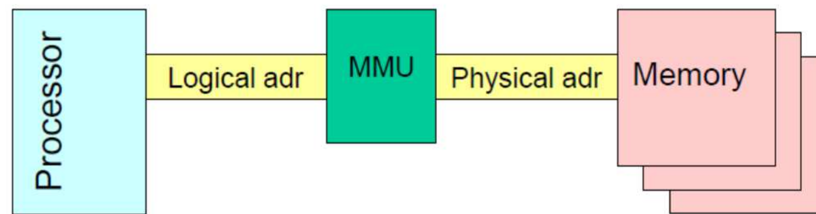
Bus Multiplexing & Address Decoding
A[15:8]
A[15:8]
Atmega
162
AD[7:0]
Multiplexed Bus
Address
Latch
A[7:0]
Address Bus A[15:0]
A[15:13]
Address Decode Logic
LE
Latch0
LE
Latch1
LE
Latch2
CS
SRAM
Data Bus D[7:0]



Connecting the LED Latches
D[0:7]
Latch0
VCC
1K
100nF
GND
U2
74HCT573
LATCH0
24

Memory Management

- Translation between logical and physical memory space.
- Memory Management Unit (MMU) handles the address translation (and other jobs).



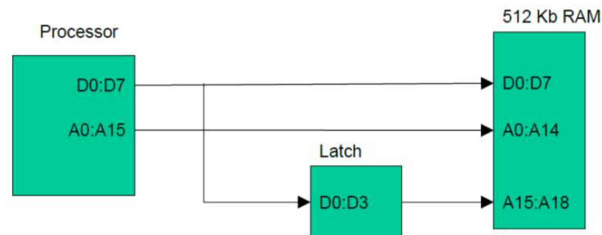
Memory Management

- Physical memory > logical memory
  - Banked memory → Dividing the physical memory into N partitions (banks) where the size of each partition is equal to (or lesser than) the processor's logical address space
- Physical memory < Logical memory
  - Virtual memory → Exploits the entire logical memory space of the processor. On-demand loading of data blocks to the physical memory from secondary storage (paging).
- Protection of memory regions
  - Monitor the address bus and intercept in case of unauthorized access to critical memory regions (OS, I/O space, interrupt tables etc.) (MPU)
- Isolation of tasks/threads
  - Prevent unauthorized access between the memory spaces of threads in a multitasking system

Example: Banked memory

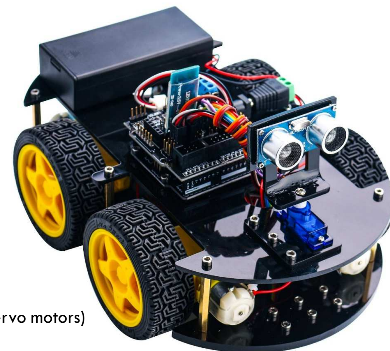
In some cases it will be necessary to expand the physical memory beyond the logical memory space of the processor, e.g. 512Kb memory on a 16 bit address buss.

- **Solution:** Map a data latch into the processor's address room and use it to keep the 4 MSB of the 19 bit physical address
  →16 memory banks of 32 Kb = 512 Kb available to the application
  → Needs software control



Spring 2018: Advanced MCU Applications Lab

- What?
  - Advanced course on Microcontrollers' Applications.
- Instructor
  - Marten van Dijk
- When?
  - Next semester: Spring 2018
- Who can join?
  - Everyone who has taken ECE3411
- What will be taught?
  - *Parallel Bus Interfaces* (for external SRAM and other devices)
  - *Graphic OLED Display*
  - Controller Area Network (CAN Protocol)
  - *Wireless Protocols* (E.g. Bluetooth)
  - *Analog Sensors Interfacing* (E.g. Ultrasonic Sensors) & *Motor Control* (DC motors, Servo motors)
  - *Real-time Operating Systems*
  - Case studies
  - Final Project: Collision Avoidance Robot

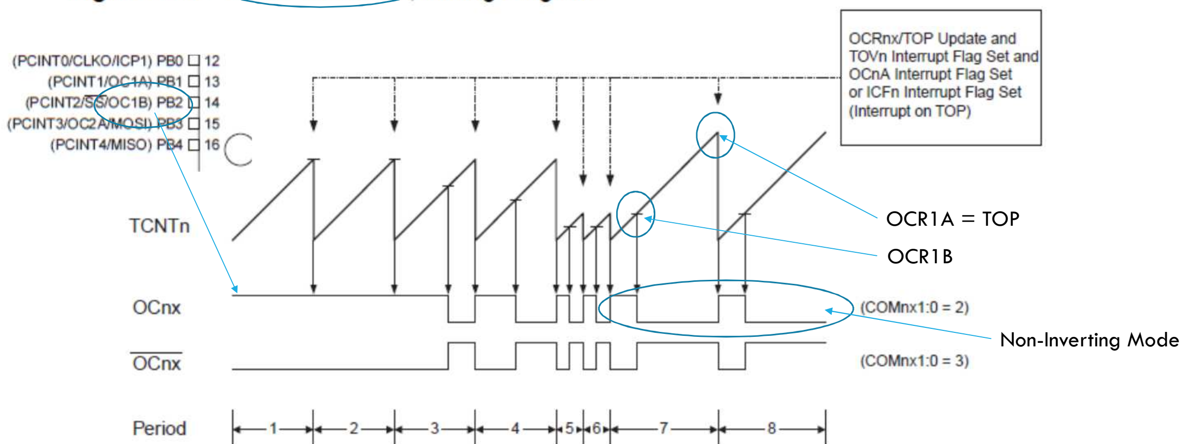


ECE3411 – Fall 2017
Lect6b

Review Session

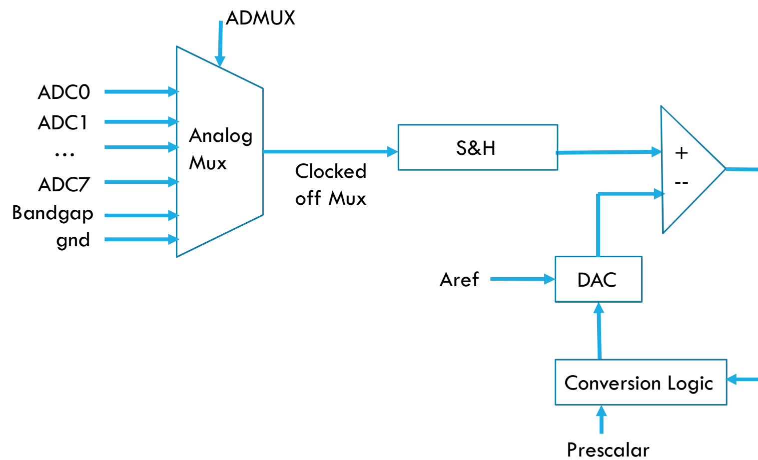
Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: marten.van_dijk@uconn.edu

Pulse Width Modulation using Timer 1



2

ADC



3

Example code ADC, no interrupt

```
void main(void)
{
        DDRC &= 0x00;    // PC1 = ADC1 is set as input

        uart_init();
        stdout = stdin = stderr = &uart_str;

        // ADLAR set to 1 → left adjusted result in ADCH
        // MUX3:0 set to 0001 → input voltage at ADC1
        ADMUX = (1<<MUX0) | (1<<ADLAR);

        // ADEN set to 1 → enables the ADC circuitry
        // ADPS2:0 set to 111 → prescalar set to 128 (104us per conversion)
        ADCSRA = (1<<ADEN) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0);

        // Start A to D conversion
        ADCSRA |= (1<<ADSC);
        fprintf(stdout,"\n\rStarting ADC demo...\n\r");
```

Takes more than 1ms, hence conversion will finish which takes 104us

4

Example code ADC, no interrupt

```
        while (1)
        {
                // Read from ADCH to get the 8 MSBs of the 10 bit conversion
                Ain = ADCH;

                // Typecast the volatile integer into floating type data, divide by maximum 8-bit value, and
                // multiply by 5V for normalization
                Voltage = (float)Ain/256.00 * 5.00;

                //ADSC is cleared to 0 when a conversion completes.  Set ADSC to 1 to begin a conversion.
                ADCSRA |= (1<<ADSC);

                // Write Voltage to string format and print (3 char string + "." + 2 decimal places)
                dtostrf(Voltage, 3, 2, VoltageBuffer);
                fprintf(stdout,"%s\n\r",VoltageBuffer);
        }

        return 0;
}
```

Takes more than 1ms, hence conversion will finish which takes 104us

5

ADC Noise Reduction

9.11.1 SMCR – Sleep Mode Control Register

The Sleep Mode Control Register contains control bits for power management.

Bit	7	6	5	4	3	2	1	0	
0x33 (0x53)	–	–	–	–	SM2	SM1	SM0	SE	SMCR
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Table 9-2. Sleep Mode Select

SM2	SM1	SM0	Sleep Mode
0	0	0	Idle
0	0	1	ADC Noise Reduction
0	1	0	Power-down
0	1	1	Power-save
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Standby[(1)]
1	1	1	External Standby[(1)]

6

Watchdog Timer

```
#include <avr/wdt.h>

#include <avr/eeprom.h>
#define eeprom_true 0  //Suppose you want to store a flag at position 0
#define eeprom_data 1  //Suppose you want to store data at position 1

ISR (WDT_vect)
{
        eeprom_write_dword((uint32_t*)eeprom_data,mode);        //Write our current mode to EEPROM
        eeprom_write_byte((uint8_t*)eeprom_true, 'T');           //Set write flag TRUE
}

void Initialize(void)
{
        … all other initialization …
        WDTCSR |= (1<<WDCE) | (1<<WDE);  // Set Watchdog Condition Edit for four cycles
        WDTCSR = (1<<WDIE) | (1<<WDE) | (1<<WDP3);   // Set WDT Int and Reset; Prescalar at 4.0s.
}
```

7

Watchdog Timer

```
int main(void)
{
        // WDOG Interrupt and Reset Disable, this only matters if reset occurs.
        wdt_reset();                                  // Reset Watchdog timer
        MCUSR  &= ~(1<<WDRF);                         // Shut off Watchdog Reset Flag
        WDTCSR |= (1<<WDCE) | (1<<WDE);      // Set Watchdog Change Enable and WD Enable
        WDTCSR  = 0x00;                       // Disable Watchdog

        Initialize();
        // Read TimeOut from EEPROM
        if (eeprom_read_byte((uint8_t*)eeprom_true) == 'T')
        {
                mode = eeprom_read_dword((uint32_t*)eeprom_data);
        }
        else
        {
                mode = 0; // Begin in normal mode
        }
        while (1) { ….. }
}
```

8

What is an Interrupt (recap)?

- A HW signal that initiates and event
- Upon receipt of an interrupt, the processor
  - Completes the instruction being executed
  - Saves the program counter (so as to return to the same execution point)
  - Loads the program counter with the location of the interrupt handler code (ISR)
  - Executes the interrupt handler (ISR)
- In practice, real time systems can handle several interrupts in priority fashion
  - Interrupts can be enabled/disabled (By setting appropriate registers.)
  - Highest priority interrupts serviced first (Which ones have the highest priority in Atmega328P?)
- Processor must check for interrupts very frequently: If any have arrived, it stops immediately and runs the associated ISR
  - Processor repeats: do one operation; check interrupts; if interrupts then suspend task and run ISR

9

ISR

- ISR is a program run in response to an interrupt
  - Disables all interrupts
  - Clears the interrupt flag that got it called
  - Runs code to service the event
  - Re-enables interrupts
  - Exits so the processor can go back to its running task
- Should be as fast as possible, because nothing else can happen when an interrupt is being serviced (when interrupts happen very frequently, tasks are being stalled and progress very slowly, in the worst case one instruction per ISR)
- Interrupts can be
  - Prioritized (service some interrupts before others)
  - Disabled (processor doesn't check or ignores all of them)
  - Masked (processor only sees some interrupts)

10

Scheduling Policies

Static Scheduling Schemes

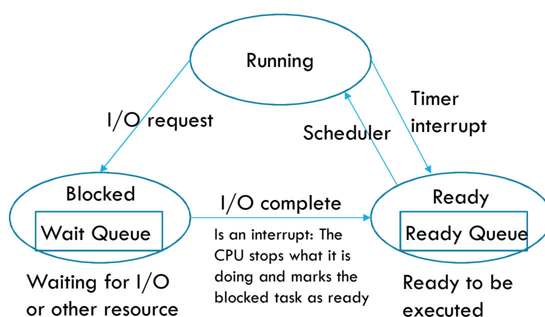
- Round-robin scheduling
- Rate-monotonic scheduling
- Deadline-monotonic scheduling
- Shortest Remaining Time First

Dynamic Scheduling Schemes

- Earliest deadline first scheduling
- Least slack time scheduling

Task State Diagram

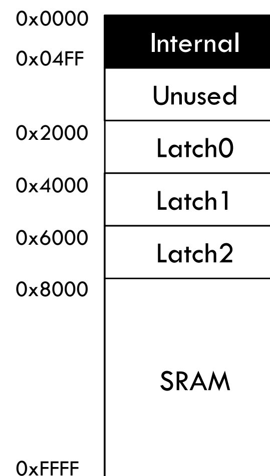
- A task/process goes through several states during its life in a multitasking system
- Tasks are moved from one state to another in response to the stimuli marked on the arrows

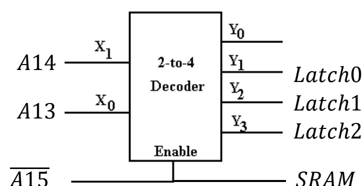


- Any tasks that are ready to run sit on the *ready queue*. This queue may be prioritized so the most important task runs next.
- When the scheduler decides the current task has had enough time on the CPU, either because it finished or its time slice is up, the *Running* task is moved to the ready queue. Then the first task on the ready queue is selected for Running.
- If the Running task needs I/O or needs a resource that is currently unavailable, it is put on the *blocked queue*. When its resource becomes available, it goes back to Ready.

Address Decoding of selected Memory Map

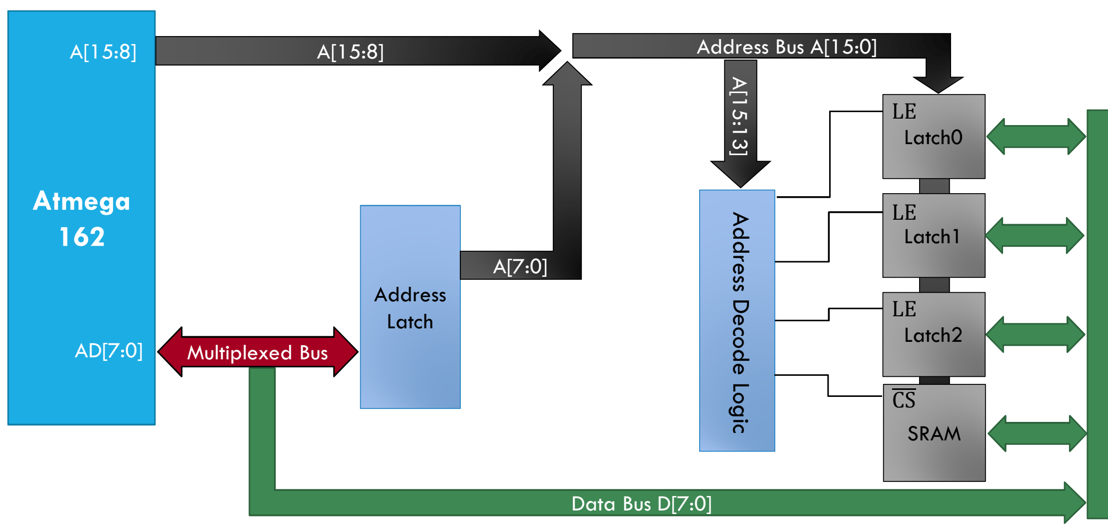
Device	Address Range	A15 … A0
Internal/Unused	0x0000 – 0x1FFF	000x xxxx xxxx xxxx
Latch0	0x2000 – 0x3FFF	001x xxxx xxxx xxxx
Latch1	0x4000 – 0x5FFF	010x xxxx xxxx xxxx
Latch2	0x6000 – 0x7FFF	011x xxxx xxxx xxxx
SRAM	0x8000 – 0xFFFF	1xxx xxxx xxxx xxxx





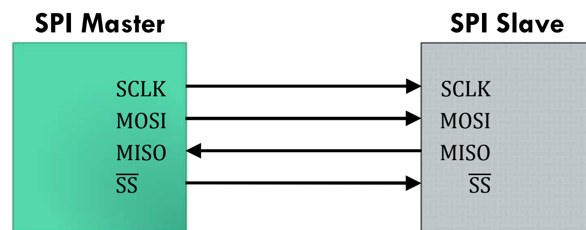
13

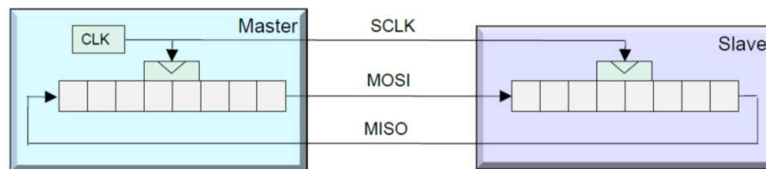
Bus Multiplexing & Address Decoding



SPI: Serial Peripheral Interface

- Synchronous Data Transfer
- Master/Slave configuration
- 4-Line Bus
- Full Duplex operation





15

SPI Master Example

```
void SPI_MasterInit(void)
{
        /* Set SS, MOSI and SCK output, all others input */
        DDR_SPI = (1<<DD_SS) | (1<<DD_MOSI) | (1<<DD_SCK);
        /* Enable SPI, Master, set clock rate fck/128 */
        SPCR = (1<<SPE) | (1<<MSTR) | (1<<SPR1) | (1<<SPR0);
}
```

```
uint8_t SPI_Master_Transceiver(uint8_t cData)
{
        PORTB &= ~(1<<SPI_SS);          // Pull Slave_Select low
        SPDR = cData;                   // Start transmission
        while( !(SPSR & (1<<SPIF)) );   // Wait for transmission complete
        PORTB |= (1<<SPI_SS);           // Pull Slave Select High
        return SPDR;                    // Return received data
}
```

Note:
DDR_SPI in the examples must be replaced by the actual Data Direction Register controlling the SPI pins.
DD_SS, DD_MOSI, DD_MISO and DD_SCK must be replaced by the actual data direction bits for these pins.
E.g. if MOSI is placed on pin PB3, replace DD_MOSI with DDB3 and DDR_SPI with DDRB.
SPI_SS should be replaced with actual bit position of SS pin in the port corresponding to SPI pins.

16

SPI Slave Example

```
void SPI_SlaveInit(void)
{
        /* Set MISO output, all others input */
        DDR_SPI = (1<<DD_MISO);
        /* Enable SPI */
        SPCR = (1<<SPE);
}
```

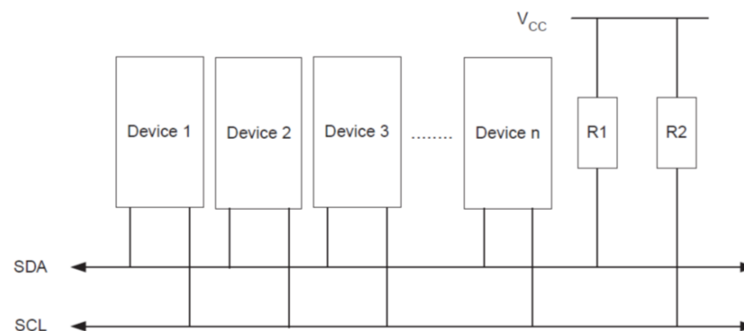
```
uint8_t SPI_SlaveReceive(void)
{
        /* Wait for reception complete */
        while(!(SPSR & (1<<SPIF)));
        /* Return Data Register */
        return SPDR;
}
```

Note:
DDR_SPI in the examples must be replaced by the actual Data Direction Register controlling the SPI pins.
DD_MOSI, DD_MISO and DD_SCK must be replaced by the actual data direction bits for these pins.
E.g. if MOSI is placed on pin PB5, replace DD_MOSI with DDB5 and DDR_SPI with DDRB.

17

I²C: Inter Integrated Circuit bus

- Also known as Two Wire Interface (TWI)
- Allows up to 128 different devices to be connected using only two bi-directional bus lines, one for clock (SCL) and one for data (SDA).
- All devices connected to the bus have individual addresses.



18

I^2C Bus Arbitration

- Arbitration is carried out by all masters continuously monitoring the SDA line after outputting data.
- If the value read from the SDA line does not match the value the Master had output, it has lost the arbitration.

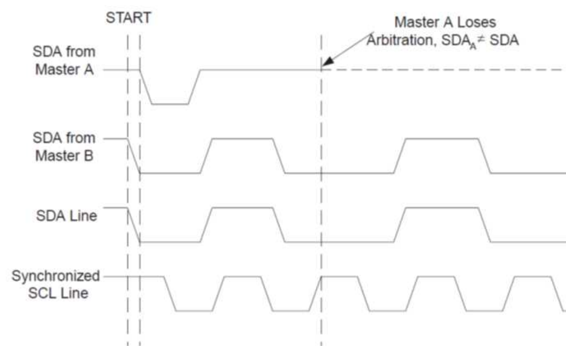


Figure 21-8. Arbitration Between Two Masters

19

Department of Electrical and Computing Engineering

UNIVERSITY OF CONNECTICUT

ECE 3411 Microprocessor Application Lab: Fall 2017

Problem Set A6

There are 5 questions in this problem set. Answer each question according to the instructions given in at least 3 sentences on own words.

If you find a question ambiguous, be sure to write down any assumptions you make.
Be neat and legible. If we can't understand your answer, we can't give you credit!

Any form of communication with other students is considered cheating and will merit an F as final grade in the course.

SUBMIT YOUR ANSWERS IN A HARDCOPY FORMAT.

Do not write in the box below

1 (x/6)	2 (x/30)	3 (x/30)	4 (x/24)	5 (x/10)	Total (xx/100)

Name:

Student ID:

1. [6 points]: Answer the following questions:

a. What is the purpose of so called *wait states* in external parallel bus interface of a processor?

b. The ARM Cortex-M features a Harvard architecture. What does that mean and what performance advantages are associated with this architecture?

Initials:

2. [30 points]: You need to implement a simple controller to monitor the altitude of an RC airplane in flight. The altitude is measured by an on-board altitude sensor that supports a standard SPI slave interface (refer to Figure 1).

The sensor provides an 8-bit value between 0 and 255 which linearly corresponds to an altitude of 0 to 1020 feet respectively.

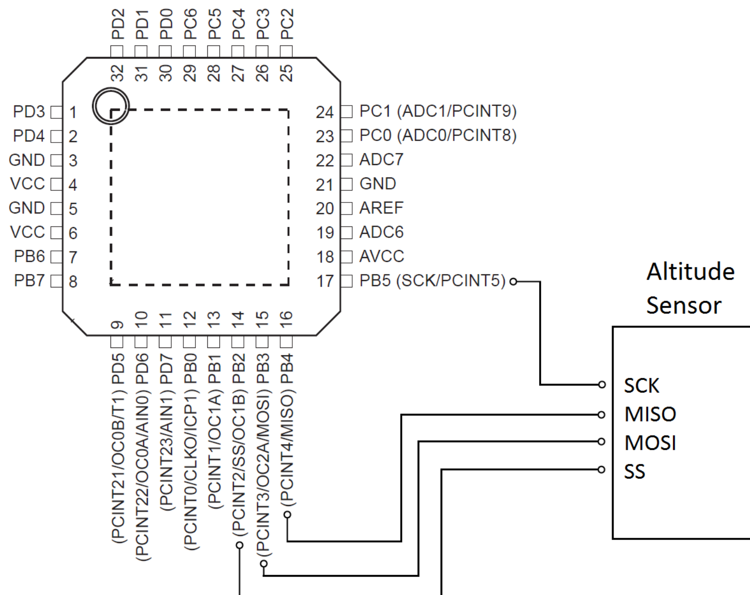


Figure 1: Hardware configuration of the controller.

The MCU needs to read the sensor data over SPI strictly every 10 milliseconds. This data is stored in a global variable called `altitude`, and used in the following two tasks which are already implemented in a library:

- `void Altitude_Monitor(void)`
  - This task monitors safety critical events related to the altitude.
  - It needs to execute every 10 milliseconds, immediately once the new SPI data is received.
  - It takes negligible time to execute on the MCU.
  - It is a high priority task and it must not be interrupted by any other task.
- `void Altitude_Display(void)`
  - This task displays the current altitude on LCD screen.
  - It needs to execute every 1 second.
  - It takes tens of milliseconds to execute on the MCU.
  - It is a low priority task and it must be interruptable by other high priority tasks.

Initials:

A. System level design choices: Answer the following questions:

a. Why or why not the specifications/properties of task `Altitude_Monitor()` will be violated if:

- This task is called inside the main function.
- This task is called inside an ISR.

Explain your answer.

b. Why or why not the specifications/properties of task `Altitude_Display()` will be violated if:

- This task is called inside the main function.
- This task is called inside an ISR.

Explain your answer.

Initials:

B. Displaying the altitude: Given that the CPU clock frequency is 16.384 MHz (i.e. 16384000 Hz), use *task based programming* approach to call `Altitude_Display()` task every 1 second using **only** Timer2.

```
/* Assume all necessary header files are included */
/* Declare & initialize your variables here */






int main(void)
{
    /* Assume any initialization for Altitude_Display() is already done */
    /* Perform Timer2 initializations here */












    sei();                    // Enable Global Interrupts
    while(1)
    {
        /* Your code here */






    }
} /* End of main() */

/* Timer 2 ISR: write the ISR code */
ISR(TIMER2_COMPA_vect)
{




} /* End of Timer2 ISR */
```

Initials:

C. Reading and monitoring the altitude: Given that the CPU clock frequency is 16.384 MHz (i.e. 16384000 Hz), extend the code of Part B such that:

- The MCU reads the sensor data over SPI every 10 milliseconds.
- The task `Altitude_Monitor()` is called every 10 milliseconds such that, while it is executing, it is not interruptable by any other task.

Notice that you may use **only** Timer2.

```
/* Assume all necessary header files are included */
/* Declare your variables here */
volatile uint8_t altitude;    // Store the received sensor data in this variable.




int main(void)
{
    // Set SS, MOSI and SCK output, MISO input
    DDRB |= (1<<DDB2)|(1<<DDB3)|(1<<DDB5);

    /* Perform all Timer2 related initializations here */
    /* You may simply write ``Copied Over from Part B'' or
        write new code here to match new requirements */




    /* Perform SPI Initializations here, enable interrupt */



    sei();                    // Enable Global Interrupts
    while(1)
    {
        /* You may simply write ``Copied Over from Part B'' or
             write new code here to match new requirements */




    }
} /* End of main() */
```

Initials:

```
/* Timer 2 ISR: write the ISR code */
ISR(TIMER2_COMPA_vect)
{
    /* Manage Software counter */









    /* Start SPI transmission */







} /* End of Timer2 ISR */

/* SPI ISR: Write the ISR code */
ISR(SPI_STC_vect)
{
    /* Process the sensor data received over SPI */

















} /* End of SPI ISR */
```

Initials:

3. [30 points]: You need to implement a simple controller to steer an RC airplane in flight. The airplane can be turned towards left/right or it can fly straight based on the positions of two ailerons, one on each wing, as shown in Figure 2.

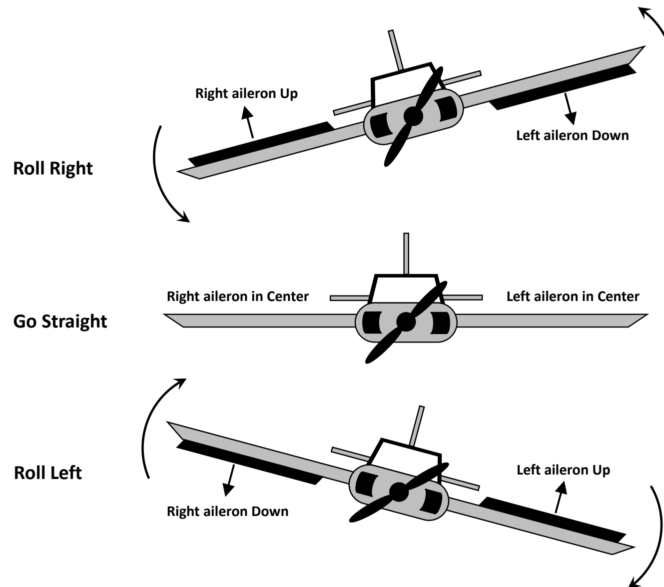


Figure 2: Aileron positions and airplane's direction of motion.

Each aileron is controlled by a servo motor that requires a PWM signal of 62.5Hz. The position of the aileron is controlled by the duty cycle of the PWM signal as specified in the following table.

Aileron Position	PWM Frequency (Hz)	PWM Period (ms)	PWM Duty Cycle (ms)
Aileron Down	62.5 Hz	$16ms$	$1.25ms$
Aileron Centered	62.5 Hz	$16ms$	$1.50ms$
Aileron Up	62.5 Hz	$16ms$	$1.75ms$

Two push switches SW0 and SW1, connected to external interrupts INT0 and INT1 respectively, are used to control the direction of the airplane motion according to a finite state machine (FSM) shown in Figure 3. Upon startup, the system is in `STRAIGHT` state. When SW0 or SW1 is pushed once, the system goes to `RIGHT` or `LEFT` state respectively, and the airplane turns right or left by manipulating the ailerons according to the Figure 2.

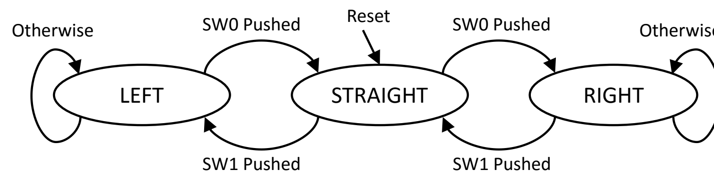


Figure 3: FSM of airplane steering controller.

Initials:

The hardware configuration of ATmega328P based controller is shown in the figure below.

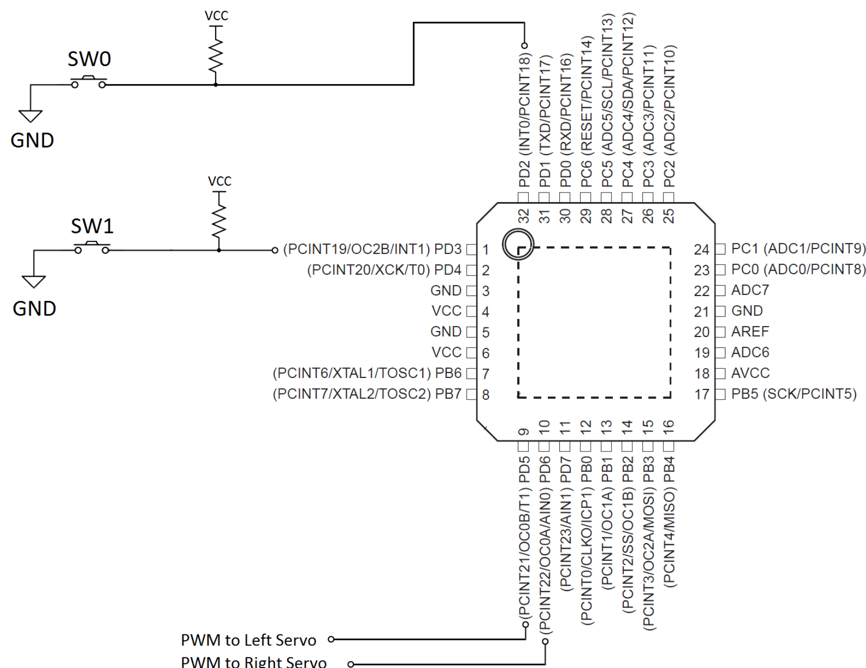


Figure 4: Hardware configuration of the controller.

The following code snippet provides the necessary layout and definitions.

```
#define F_CPU 16384000UL        // CPU runs on 16.384 MHz
#include <avr/io.h>
#include <inttypes.h>
#include <avr/interrupt.h>

// Definitions for State Machine
#define STRAIGHT     0
#define RIGHT        1
#define LEFT         2
volatile uint8_t Current_State = STRAIGHT;

// Definitions for PWM
volatile uint8_t left_duty_cycle;
volatile uint8_t right_duty_cycle;

/* Main Function */
int main(void)
{
    initialize_PWMs();        // Configure PWM related Timer & Signals
    initialize_Switches();    // Configure External Interrupts
    sei();                    // Enable Global Interrupts
    while(1);                 // Nothing to do.
}
```

Initials:

A. Initializing Timer0 and PWMs:

Given that the CPU clock frequency is 16.384 MHz (i.e. 16384000 Hz), you need to generate two 62.5Hz PWM signals for the right and left servo on PD6 and PD5 respectively (as shown in Figure 4), using **only** Timer0.

For this purpose, both channel A and B of Timer0 need to generate a PWM signal each for each wing. Therefore, in order to synchronize the duty cycle updates for both, we require you to use **only one** Timer0 ISR to reload the duty cycle values stored in variables `left_duty_cycle` and `right_duty_cycle` into Timer0 register(s). Which ISR should it be?

(a) `TIMER0_COMPA_vect`

(b) `TIMER0_COMPB_vect`

(c) `TIMER0_OVF_vect`

Complete the function `initialize_PWMs()` by properly initializing Timer0 for this purpose, and by also configuring the PWM pins as necessary. Notice that as soon as this function is executed, the PWMs signals will start driving the servos. Therefore you need to configure the initial duty cycle of both PWMs to be $1.50ms$ for centered position.

```
initialize_PWMs()
{
    /* Configure Timer0 & pins for generating two PWMs here */
```

```
} */ End of initialize_PWMs() */
```

Initials:

B. Configuring External Interrupts:

The switches SW0 and SW1 are connected to INT0 and INT1 as shown in Figure 4. Assume that these switches are hardware debounced (i.e. no software debouncing is needed).

Complete the function `initialize_Switches()` by configuring the external interrupts (INT0 and INT1) properly. While configuring, keep in mind what logic value will be passed to the interrupt pin if the corresponding switch is pushed (refer to Figure 4).

```
initialize_Switches()
{
    /* Configure INT0 and INT1 here */













} */ End of initialize_Switches() */
```

Initials:

C. Implementing the controller FSM:
Assuming that SW0 and SW1 are hardware debounced (i.e. no software debouncing is needed), implement the controller FSM from Figure 3 inside INT0 and INT1 ISRs. In each of the states, modify the duty cycle variables `left_duty_cycle` and `right_duty_cycle` accordingly. We assume that these duty cycle values are used to program the PWM duty cycles in the ISR indicated in Part A which we do not ask you to program.
Hint: It would be useful to split the FSM from Figure 3 into two FSMs, one for each input switch.

```
// External Interrupt INT0 ISR
ISR(INT0_vect)
{
    /* Complete the FSM here */
    switch (Current_State)
    {
        case LEFT:









        break;

        case STRAIGHT:









        break;

        case RIGHT:









        break;
    }
} /* End of ISR(INT0_vect) */
```

Initials:

```
// External Interrupt INT1 ISR
ISR(INT1_vect)
{
    /* Complete the FSM here */
    switch (Current_State)
    {
        case LEFT:








        break;

        case STRAIGHT:








        break;

        case RIGHT:








        break;
    }

} /* End of ISR(INT1_vect) */
```

Initials:

4. [24 points]: You need to design an AVR-based system that includes four external devices in its address space:

- Two SRAMs of size 16 kilobytes each (i.e. 2^{14} unique addresses) for data storage, having the following control signals:
  - $\overline{WE}$: Write Enable (Active Low).
  - $\overline{OE}$: Output Enable (Active Low).
  - $\overline{CE}$: Chip Enable (Active Low).
- Two 8-bit latches to drive digital outputs for 16 LEDs.
  - LE: Latch Enable (Active High).
  - $\overline{OE}$: Output Enable (Active Low).

The system should be based on a MCU of the type Atmel AVR ATmega162 (shown in Figure 5).

ATmega162

Left pin function	Pin	Pin	Right pin function
(OC0/T0) PB0	1	40	VCC
(OC1/T1) PB1	2	39	PA0 (AD0/PCINT0)
(RXD1/AIN0) PB2	3	38	PA1 (AD1/PCINT1)
(TXD1/AIN1) PB3	4	37	PA2 (AD2/PCINT2)
($\overline{SS}$/OC3B) PB4	5	36	PA3 (AD3/PCINT3)
(MOSI) PB5	6	35	PA4 (AD4/PCINT4)
(MISO) PB6	7	34	PA5 (AD5/PCINT5)
(SCK) PB7	8	33	PA6 (AD6/PCINT6)
$\overline{RESET}$	9	32	PA7 (AD7/PCINT7)
(RXD0) PD0	10	31	PE0 (ICP1/INT2)
(TXD0) PD1	11	30	PE1 (ALE)
(INT0/XCK1) PD2	12	29	PE2 (OC1B)
(INT1/ICP3) PD3	13	28	PC7 (A15/TDI/PCINT15)
(TOSC1/XCK0/OC3A) PD4	14	27	PC6 (A14/TDO/PCINT14)
(OC1A/TOSC2) PD5	15	26	PC5 (A13/TMSI/PCINT13)
($\overline{WR}$) PD6	16	25	PC4 (A12/TCK/PCINT12)
($\overline{RD}$) PD7	17	24	PC3 (A11/PCINT11)
XTAL2	18	23	PC2 (A10/PCINT10)
XTAL1	19	22	PC1 (A9/PCINT9)
GND	20	21	PC0 (A8/PCINT8)

Figure 5: ATmega162 Pin Configuration.

By completing the subsections A and B, design an interface between the ATmega162 and the external devices needed for this system.

Initials:

A. Address Mapping & Decoding: (10 points)

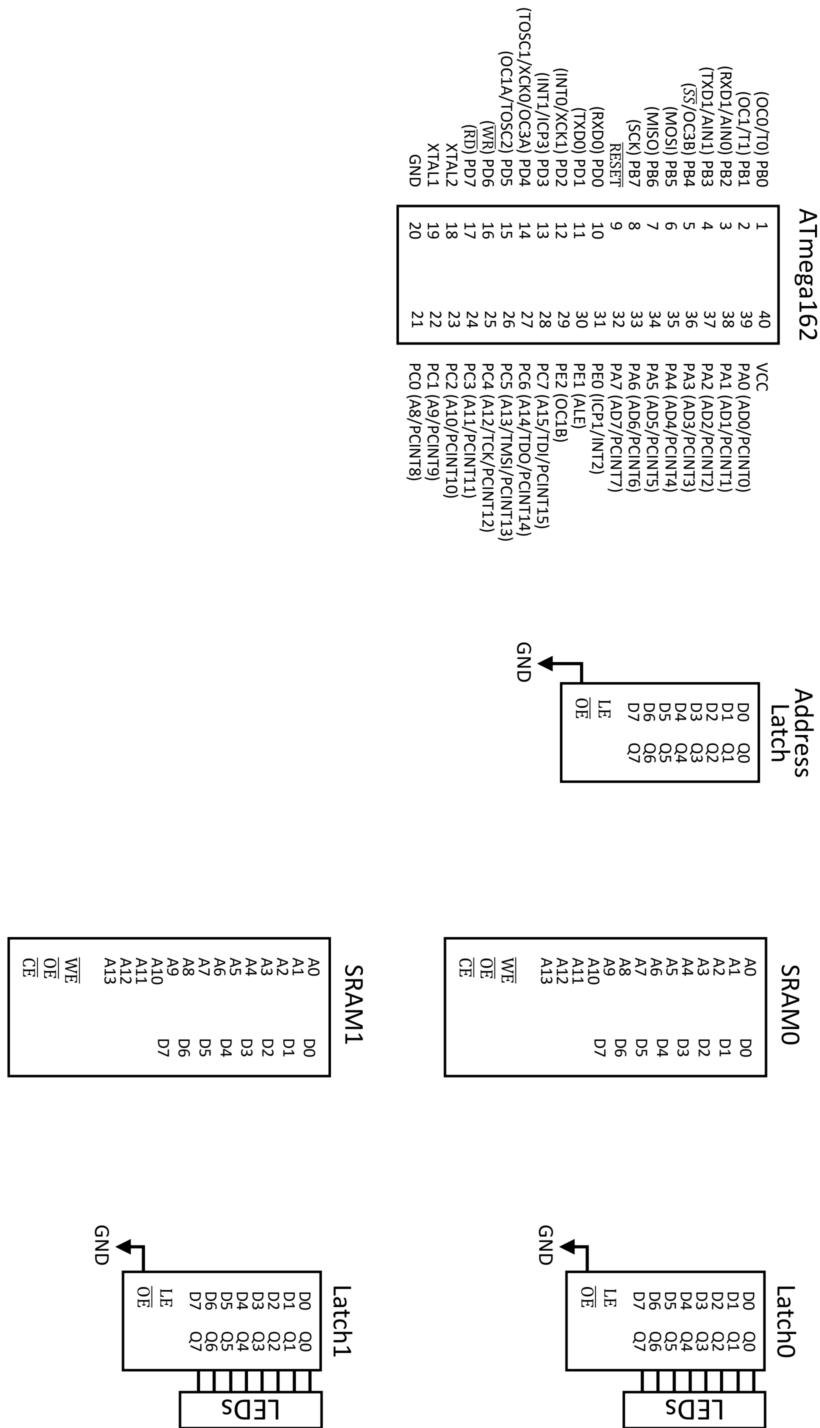
Explain how you will organize the address space of the system and its associated decoding logic (remember that the lower 1280 addresses of ATmega162 are reserved).

Show your calculations/methodology and design the address decoding logic.

Initials:

B. Overall System Interconnects: (14 points)

Draw a detailed schematic of the system which shows the interconnection of components/chips (details may be limited to main signal lines/paths). Specify and draw chips, circuits and signals that you may find necessary to include.



Initials:

5. [10 points]: Can you shortly describe what you have learned and feel confident about using in the future?

End of Problem Set

Initials:

ECE3411 – Fall 2017
Independent LAB6.

RedBot

Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: marten.van_dijk@uconn.edu

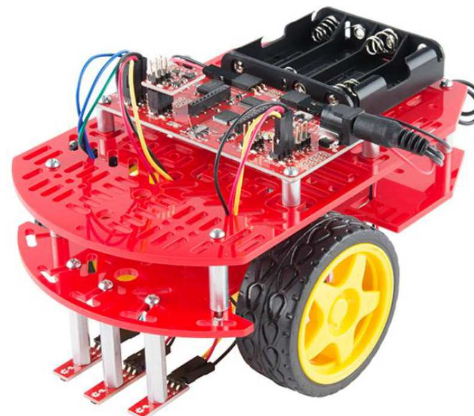
UCONN

Slides of I2C are copied from Lab 7b, ECE3411 – Fall 2015, by Marten van Dijk and Syed Kamran Haider
Some of these slides are extracted or copied from "RedBot Project" offered at Sung Yeul Park in Spring 2016

RedBot

- Using Sparkfun's RedBot Line Follower kit, you will implement a small robot that follows a line of electrical tape.
- Infrared Sensors are used to sample the desired path in reference to the robot's trajectory.
- Movement is actuated by two PWM controlled H-bridge modules.

- Description: https://www.sparkfun.com/products/13166
- Get started: https://learn.sparkfun.com/tutorials/getting-started-with-the-redbot
- Schematic: https://cdn.sparkfun.com/datasheets/Robotics/RedBot_Mainboard_v14.pdf



Warning: Please do not write anything to EEPROM, since this seems to prevent further programming of the MCU.

2

Using Atmel Studio to Program Arduino Board

- Since the board on RedBot is an Arduino Board, we are not able to directly use Atmel Studio to program it. We need to setup an external programmer in Atmel Studio for programming this board.
- Please follow the following steps to setup the external programmer.

1. Download Avrdude from http://mirror.rackdc.com/savannah//avrdude/avrdude-5.11-Patch7610-win32.zip
2. Unzip the downloaded file, rename the directory to ***avrdude***, and copy it into your **C** drive
3. Connect your board to your computer, open Device Manager, check the **COM** port.
4. Open Atmel Studio, go to Tools -> External Tools

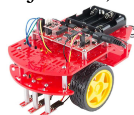
3

Setup External Programmer

5. Fill the dialog box like this:
6. The Arguments field in the dialog box is

-C "C:\avrdude\avrdude.conf" -p atmega328p -c arduino -P COM9 -b 115200 -U flash:w:"$(ProjectDir)Debug\$(TargetName).hex":i

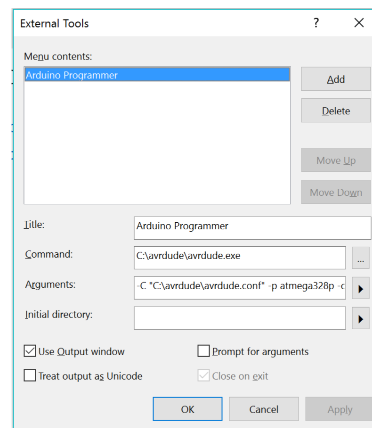
for most of



The Arguments field in the dialog box is

-C "C:\avrdude\avrdude.conf" -p atmega328p -c arduino -P COM9 -b 57600 -U flash:w:"$(ProjectDir)Debug\$(TargetName).hex":i

for most of



Note: Update your COM number accordingly. If your programmer does not work after setup, change to the other argument. It must be one of these two.

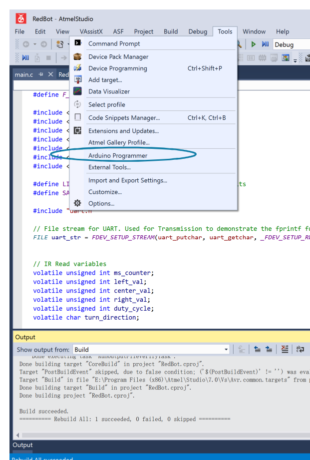
4

Use External Programmer

Since the MCU on this board is also ATMega 328P, you just need to create the project and program as usual.

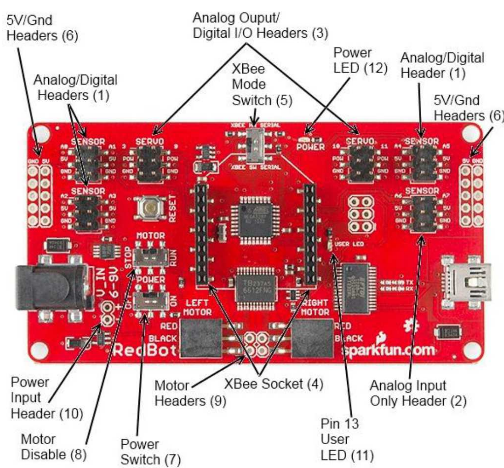
Then first build the project, and click Tools-> Arduino Programmer to program your board.

Notice: This Arduino Programmer can only be used to program the board (not to build the solution), so you should always rebuild your solution before you program it.



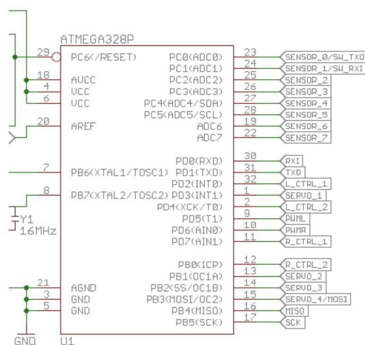
5

RedBot Mainboard



6

ATMega 328P Pin Assignment

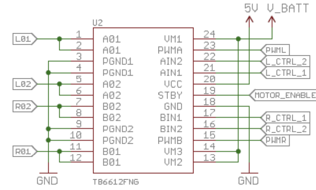


Pin #	Pin Name	Port Name	Ext Circuit
19, 22~29	ADC	PC0~5	ADC input
30, 31	USART	PD0, PO1	XBEE
32, 2	L_CTRL_1/2	PD2, PD4	Left Motor
1, 13~15	SERVO_1/2/3/4	PD3, PB1, PB2, PB3	
9~10	PWML/R	PD5, PD6	
11~12	R_CTRL_1/2	PD7, PB0	Right Motor
29	RESET	PC6	
7,8	CLK	PB6, PB7	
18, 4, 6	AVCC, VCC		
21, 3, 5	AGND, GND		
20	AREF		
16, 17	MISO, SCK	PB4, PB5	6PIN ISP

Line Sensor

- QRE1113: Miniature Reflective Object Sensors
- The sensor works by detecting reflected light coming from its own infrared LED.
- By measuring the amount of reflected infrared light, it can detect transitions from light to dark (lines) or even objects directly in front of it.

Motor Control Mechanism



- TB6612FNG: Dual DC motor driver IC
- Four modes: CW, CCW, Short brake, and stop
- Speed control: PWM duty ratio
- Input1, 2: determine/control Mode
- STBY: motor enable pin

Input				Output		
IN1	IN2	PWM	STBY	OUT1	OUT2	Mode
H	H	H/L	H	L	L	Short brake
L	H	H	H	L	H	CCW
		L	H	L	L	Short brake
H	L	H	H	H	L	CW
		L	H	L	L	Short brake
L	L	H	H	OFF (High impedance)		Stop
H/L	H/L	H/L	L	OFF (High impedance)		Standby

9

Task 1a: Reading Data from Sensors

- You are required to first initialize ADC and sample three sensors in a round robin fashion.
- Print ADC reads on your screen over UART.
- Test your sensors over a white surface and a black electrical tape, and figure out a proper threshold to distinguish "on tape" and "off tape" states.

10

Task 1b: Controlling Motors

- Generate the correct command for your motors.
  - Test clockwise and counter-clockwise rotation.
  - Test stop command.
- Generate a PWM signal to control the speed of your motors.

Task 1c: Integration

- Use the data from three sensors to adjust the speed and direction of two motors.
- Test your simple line follower.

Task 2: PID Control

- Improve your line follower by implementing a PID controller in order to make your RedBot move more smoothly.
- Hint: In order to generate an error value in the PID controller, you can first use your thresholds to convert three raw ADC reads to a three-bit value. Then convert this three-bit value into an error value which should be a signed value.
  - E.g. when the left sensor is on the tape and the other two are not on the tape, you first convert it to 0b100, and then convert it to error value -2.
  - E.g. when the left sensor and middle sensor are on the tape, convert it to error -1.
  - This requires you to use fuzzy thresholds, e.g., you can take an average of the sensor values for a white surface and a black tape.
- Hint: The integral of errors can be calculated as a summation of all the errors.
- Hint: The derivative of errors can be calculated as current error – last error.
- Tip: The constant for the I term and D term do not need to be large in comparison with the constant for the P term.

13

Task 3: Counting the laps

- There is one spot on the track where all the three sensors will sample black tapes.
  - Use this as an indicator of having completed one lap.
  - Whenever the RedBot passes this area, you need to toggle the on-board LED D13, which is connected with PB5.
  - Hint: Don't forget to implement a debounced button to prevent toggling this LED more than once within one lap.

- **You need to demonstrate your RedBot on Wednesday Dec. 6th during lab hours.**
- **Email your commented code and submit a hard copy by noon Dec. 5th**
- **No revision or 24-hr extension token can be used !!!**
- **Based on your code we will ask a couple of questions.**

14