ECE3411 – Fall 2017

Lec5a.

Bus and Communication Interfaces
Task Based Programming Revisited
Real Time Operating Systems

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

Copied from Lecture 6a, ECE3411 – Fall 2015, by
Marten van Dijk and Syed Kamran Haider

UCONN

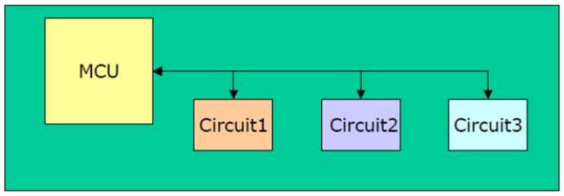
Bus and Communication Interfaces

- 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

2

Serial synchronous interfaces

- Local serial interconnection of microcontrollers and peripheral circuits/functions
- Required features:
  - Low complexity
  - Low to medium data rate
  - Small physical footprint/few pins
  - Short distances
  - Low cost

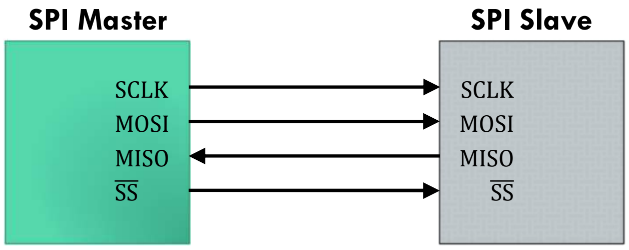


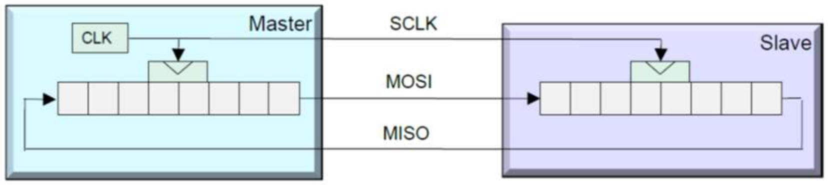
- Most MCUs have built-in peripheral units for communicating with external circuits, e.g. ATmegaAVR (SPI and TWI (I2C))
- Great abundance of different types of peripheral circuits that implements synchronous serial interfaces (Flash, EEPROM, AD, DA, RTC, Display drivers, sensors etc.)

3

SPI: Serial Peripheral Interface

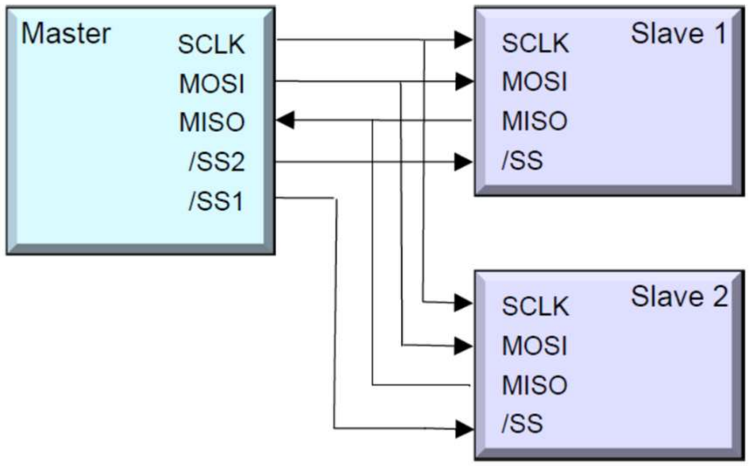
- Synchronous Data Transfer
- Master/Slave configuration
- 4-Line Bus
- Full Duplex operation





4

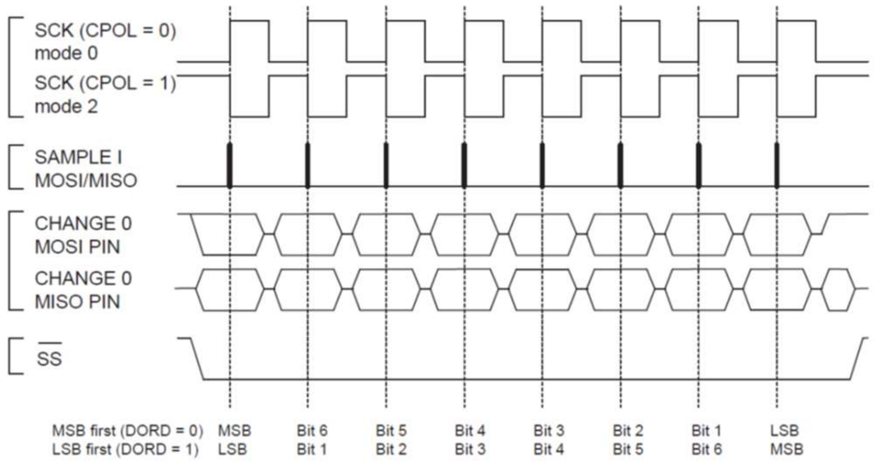
SPI Master with Multiple Slaves



5

SPI Frame Transfer

Figure 18-3. SPI Transfer Format with CPHA = 0


SCK (CPOL = 0) mode 0
SCK (CPOL = 1) mode 2
SAMPLE I MOSI/MISO
CHANGE 0 MOSI PIN
CHANGE 0 MISO PIN
$\overline{\text{SS}}$
MSB first (DORD = 0) MSB Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 LSB
LSB first (DORD = 1) LSB Bit 1 Bit 2 Bit 3 Bit 4 Bit 5 Bit 6 MSB

6

MicroWire (μWire)

- Essentially a subset of SPI
- SPI mode 0 → (CPOL, CPHA) = (0, 0)
- Often found in half duplex “three-wire mode”
- Common bi-directional serial data line → only three wires needed (SIO, SCLK, CS)
- Used in e.g. RTCs (real-time clocks) and serial EEPROMs

7

Task Based Programming

8

Example: How are the tasks scheduled?

```
while (1)
{
        if (task1_timer == 0)       // if task1_timer is not already equal to 0,
                                    // it is being decremented every 1 millisecond
                                    // during a timer ISR
        {
                task1_timer = t1;
                task1();            // task1 takes m1 milliseconds
        }

        if (task2_timer == 0)       // if task2_timer is not already equal to 0,
                                    // it is being decremented every 1 millisecond
                                    // during a timer ISR
        {
                task2_timer = t2;
                task2();            // task2 takes m2 milliseconds
        }
}
```

9

Example Cont'd

- Suppose t1=5, m1=1, t2=10, and m2=15
- What is the frequency f1 in Hz at which task1() is called?
- What is the frequency f2 in Hz at which task2() is called?

- Answer:
  - Since both task1_timer and task2_timer are decremented to 0 during the execution of task2(), task1() and task2() alternate.
  - Therefore, f1=f2 = 1 every 16 ms which is equal to 1000/16 Hz.

10

Example Cont'd

- Suppose t1=20, m1=1, t2=10, and m2=15
- What is the frequency f1 in Hz at which task1() is called?
- What is the average frequency f2 in Hz at which task2() is called?

- Answer:
  - Since task2_timer is decremented to 0 during the execution of task2(), task2() is called as often as possible.
  - When it is task1()'s turn to be executed, it takes more than one and less than two executions of task2_timer to get task1() decremented to 0.
  - Therefore, the execution pattern converges to a repetition of task2() (takes 15 ms), task2() (takes 15 ms), task1() (takes 1 ms) giving
    - a frequency f_1=1000/31 Hz and
    - an average frequency f_2=2 * 1000/31.

11

Example Cont'd

- Suppose t1=20, m1=1, t2=25, and m2=15
- What is the frequency f1 in Hz at which task1() is called?
- What is the frequency f2 in Hz at which task2() is called?

- Answer:
  - During the time that task2() is executed (which takes 15 ms), task1_timer (which initial value is 20) is decremented to a value v<=5.
  - The MCU will be idle for v ms after which task2_timer is decremented to 25-15-v and task1_timer just turned into 0.
  - So, after v ms task1() is executed taking 1ms during which task1_timer reduces to 19 and task2_timer reduces by 1 to 9-v.
  - The MCU will be idle for another 9-v ms after which task1_timer is equal to 10+v and task2_timer just turned into 0.
  - Now task2() is executed (which takes 15 ms) after which task1_timer is equal to 0 and task2_timer is equal to 10.
  - The same argument is now repeated for v=0 showing that the execution pattern converges to a repetition of task2() (takes 15 ms), task1() (takes 1 ms), idle time (takes 9 ms) giving
    - a frequency f_1=f_2=1000/25 Hz.

12

Example Cont'd

- Suppose t1=4, m1=1, t2=8, and m2=4.
- Assume initially task1_timer = 0 and task2_timer = t2
- What is the average frequency f1 in Hz at which task1() is called?
- What is the average frequency f2 in Hz at which task2() is called?

- Answer:
  - Task 1 executes during the intervals [12n,12n+1], [12n+5,12n+6], for integers n>=0.
  - Task 2 executes during intervals [12n+8,12n+12] for integers n>=0.
  - This gives frequencies f_1=1000*2/12 Hz and f_2=1000/12 Hz.

13

Real Time OS

- What follows is extracted or copied from MIT 16.07 (Perry)

- What is an Operating System (OS)?
- Basic operating system design concepts
- What is a Real Time OS (RTOS)?
- Realtime Kernel Design Strategies

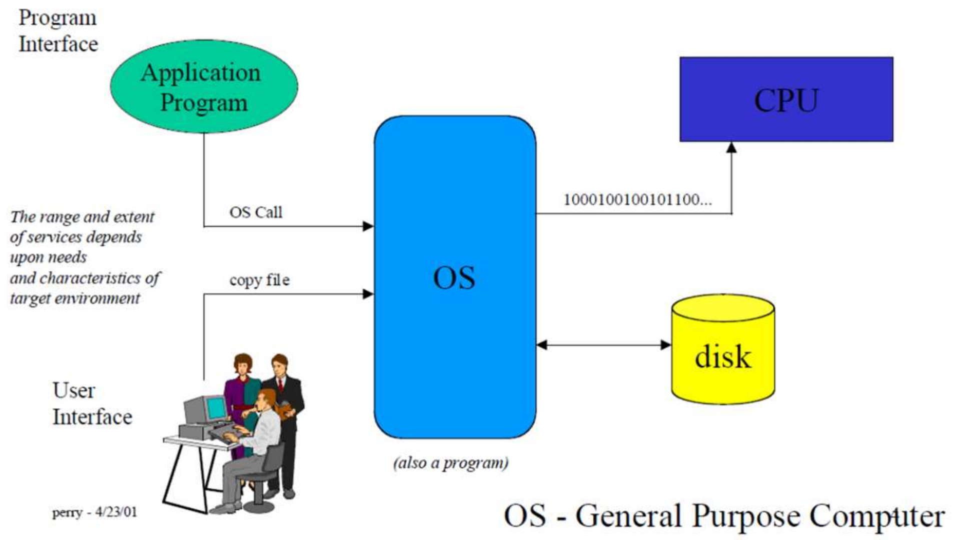
14

What is an operating system?

An organized collection of software extensions of hardware that serve as...

- control routines for operating a computer (for example, to gain access to computer resources (like file I/O))
- an environment for execution of programs

OS Services



What does an OS do?

- Manages computer system resources (processor, memory, I/O, etc.)
  - Keeps track of status and "owner" of each resource
  - Decides who gets resource
  - Decides how long the resource can be in use
- In systems that support *concurrent execution* of programs, it
  - Resolves conflicts for resources
  - Optimizes performance given multiple users

17

Types of operating systems

- Simplest = small kernel on embedded processor
- Most complex = full featured commercial OS
  - Multi-user security
  - Graphics support
  - Networking support
  - Peripherals communication
  - Concurrent execution of programs

18

OS Hierarchy

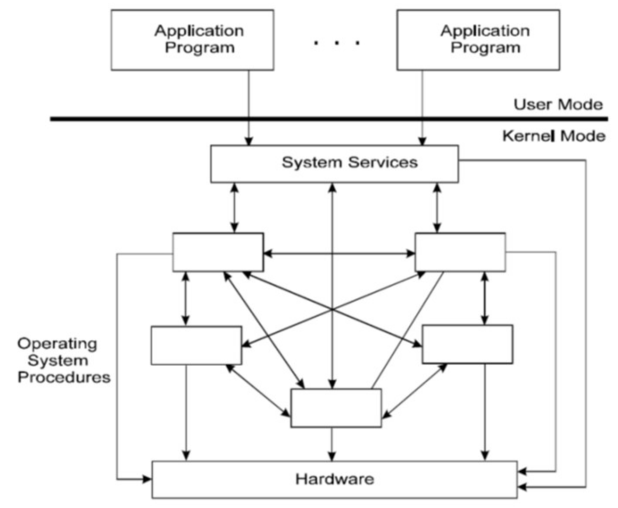
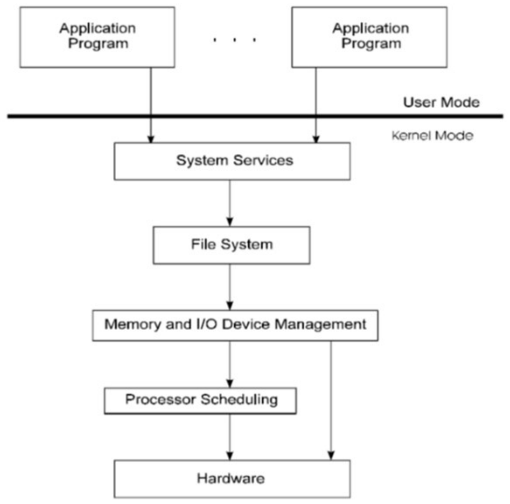


Figure 2.1: Monolithic Operating System

19

OS Hierarchy



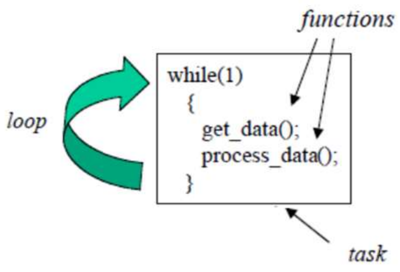
Taken from http://www.cloudbus.org/~raj/

Figure 2.2: Layered Operating System

20

Tasks & Functions

- A task is a process that repeats itself
  - Loop forever
  - Essential building block of real time software systems
- A function is a procedure that is called. Once called, it runs and then exits possibly returning a value.



```
while(1)
  {
    get_data();
    process_data();
  }
```

21

RTOS

- Often RTOS = OS Kernel
- An embedded system is designed for a single purpose so the user shell and file/disk access features are unnecessary
- RTOS gives you control over your resources
  - No background processes that “just happen”
  - Bounded number of tasks
- RTOS gives you control over timing by allowing:
  - Manipulation of task priorities
  - Choice of scheduling options

22

Components OS Kernel

- Task Scheduler: To determine which task will run next in a multitasking system
- Task Dispatcher: To perform necessary bookkeeping to start a task
- Intertask Communication: To support communication between one process (i.e. task) and another

23

Realtime Kernel Design Strategies

- Polled Loop Systems
- Interrupt Driven Systems
- Multi-Tasking
- Foreground/Background Systems

24

Polled Loops

- Simplest RT kernel
- A single and repetitive instruction tests a flag that indicates whether or not an event has occurred
  - Examples: Non-blocking LCD instructions, Non-blocking "get string" over the UART channel
- No intertask communication or scheduling needed. Only single tasks exist
- Excellent for handling high-speed data channels, especially when
  - Events occur at widely spaced intervals and
  - Processor is dedicated to handling the data channel

25

Polled Loops

- Pros:
  - Simple to write and debug
  - Response time easy to determine (as compared to our task-based programming example with two rather than a single task)
- Cons:
  - Can fail due to burst of events
  - Generally not sufficient to handle complex systems
  - Waste of CPU time, especially when event being polled occurs infrequently

26

Using Polled Loops

- Often used inside other real time schemes to, e.g.,
  - Poll a suite of sensors for data
  - Check for user inputs (keyboard, keypad, UART data)
- Opposite of interrupt driven systems

What is an Interrupt (recap)?

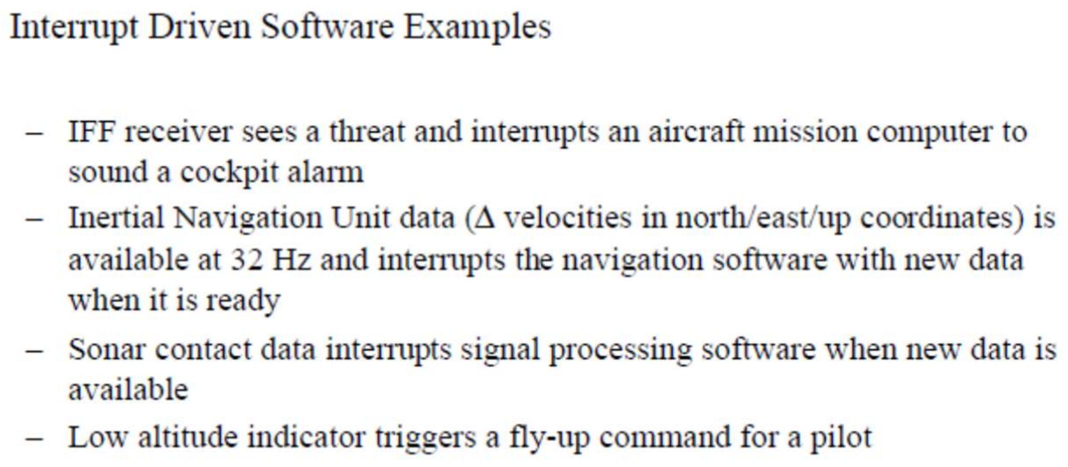
- A HW signal that initiates and event
- Upon receipt of an interrupt, the processor
  - Completes the instruction being executed
  - Saves the program counter (so as to return to the same execution point)
  - Loads the program counter with the location of the interrupt handler code (ISR)
  - Executes 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

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)
  - Disabed (processor doesn't check or ignores all of them)
  - Masked (processor only sees some interrupts)

29

Examples interrupt-driven system



Interrupt Driven Software Examples

– IFF receiver sees a threat and interrupts an aircraft mission computer to sound a cockpit alarm
– Inertial Navigation Unit data (Δ velocities in north/east/up coordinates) is available at 32 Hz and interrupts the navigation software with new data when it is ready
– Sonar contact data interrupts signal processing software when new data is available
– Low altitude indicator triggers a fly-up command for a pilot

30

ECE3411 – Fall 2017
Lab 5a.

SPI: Serial Peripheral Interface

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

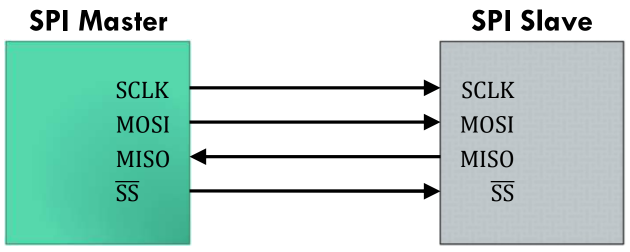
Copied from Lab 6c, ECE3411 – Fall 2015, by
Marten van Dijk and Syed Kamran Haider
With the help of:
www.wikipedia.org
ATmega328P Datasheet

UCONN

SPI: Serial Peripheral Interface

The SPI bus specifies four logic signals:

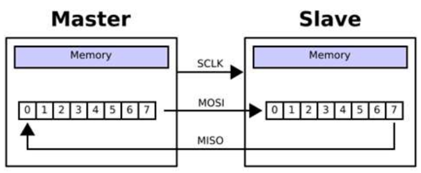
- SCLK : Serial Clock (output from master).
- MOSI : Master Output, Slave Input (output from master).
- MISO : Master Input, Slave Output (output from slave).
- SS : Slave Select (active low, output from master).

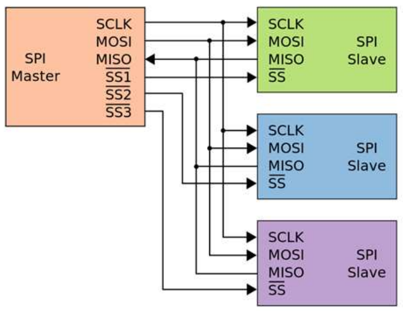


2

SPI Master & Slaves

- The SPI bus can operate with a single Master device and with one or more Slave devices.
- In case of multiple slaves, the master selects the slave device with a logic 0 on the select (SS) line.
- During each SPI clock cycle, the master sends a bit on the MOSI line and the slave reads it, while the slave sends a bit on the MISO line and the master reads it.
- This sequence is maintained even when only one-directional data transfer is intended.





3

SPI Data Modes

- There are four combinations of SCK phase and polarity with respect to serial data, which are determined by control bits CPHA and CPOL.

Table 18-2. SPI Modes

SPI Mode	Conditions	Leading Edge	Trailing eDge
0	CPOL=0, CPHA=0	Sample (Rising)	Setup (Falling)
1	CPOL=0, CPHA=1	Setup (Rising)	Sample (Falling)
2	CPOL=1, CPHA=0	Sample (Falling)	Setup (Rising)
3	CPOL=1, CPHA=1	Setup (Falling)	Sample (Rising)

4

SPI Frame Transfer with CPHA=0

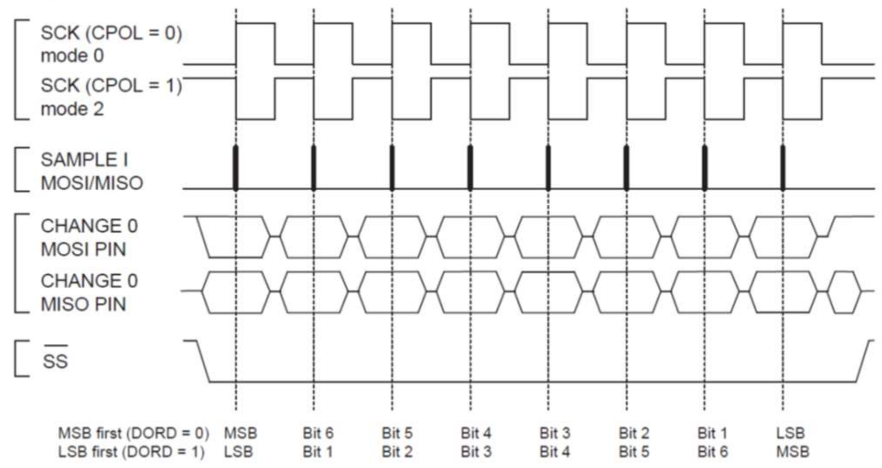


Figure 18-3. SPI Transfer Format with CPHA = 0
SCK (CPOL = 0)
mode 0
SCK (CPOL = 1)
mode 2
SAMPLE I
MOSI/MISO
CHANGE 0
MOSI PIN
CHANGE 0
MISO PIN
SS
MSB first (DORD = 0) MSB Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 LSB
LSB first (DORD = 1) LSB Bit 1 Bit 2 Bit 3 Bit 4 Bit 5 Bit 6 MSB

5

SPI Frame Transfer with CPHA=1

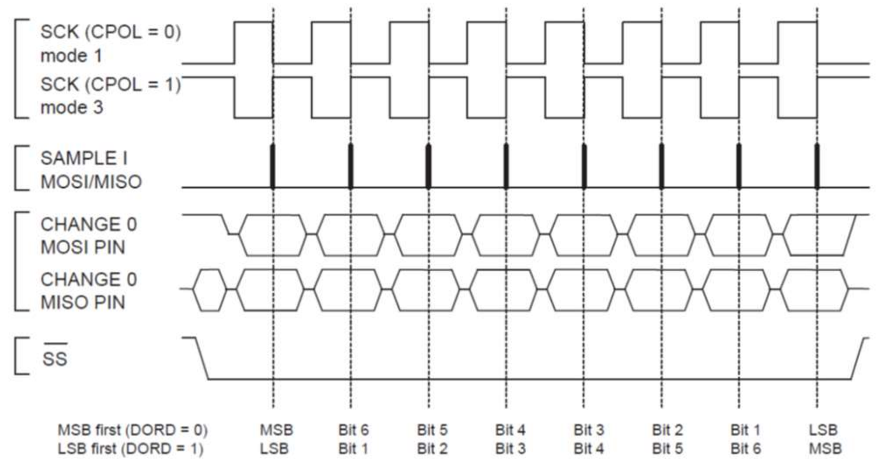


Figure 18-4. SPI Transfer Format with CPHA = 1
SCK (CPOL = 0)
mode 1
SCK (CPOL = 1)
mode 3
SAMPLE I
MOSI/MISO
CHANGE 0
MOSI PIN
CHANGE 0
MISO PIN
SS
MSB first (DORD = 0) MSB Bit 6 Bit 5 Bit 4 Bit 3 Bit 2 Bit 1 LSB
LSB first (DORD = 1) LSB Bit 1 Bit 2 Bit 3 Bit 4 Bit 5 Bit 6 MSB

6

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.

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.

Important Notes

- On Xplained Mini, ATmega328P is programmed by ATmega32U4
- Programming in ISP mode and/or enabling/disabling fuses uses SPI bus.
- If you program in ISP mode, you'll need to restart Atmel Studio every time you program the ATmega328P.
- Therefore, use debugWire interface to program the ATmega328P for this lab.
- REMEMBER: debugWire interface requires DWEN fuse to be enabled, which is done over SPI bus between ATmega32U4 and ATmega328P.
  Therefore, if you plan to connect MOSI and MISO pins together to perform loopback testing of SPI, do so only after entering into debugWire interface and programming the ATmega328P at least once. This will enable the DWEN fuse before the SPI pins MOSI and MISO are shorted together.

Task1: SPI Loopback Testing

Write a simple program to test SPI in loopback mode. In particular:

- Configure SPI in Master mode
- Read a potentiometer's voltage through ADC every 100ms (only upper 8 bits).
- Transmit the byte containing voltage reading over SPI.
- Loopback the transmitted byte by connecting MOSI and MISO pins together according to the instructions given on previous slide.
- Print on LCD the byte value received over SPI.

Task2: SPI Master Slave Communication

Extend Task1 so that you can exchange voltage readings between yours and your friend's board over SPI bus.

- Configure your board as SPI Master and ask your friend to configure his as SPI Slave.
- Make proper wire connections of SS, SCK, MOSI and MISO pins between the two boards.
- The slave MCU should read its ADC value and write it to SPDR register every 50ms.
- Transmit Master's voltage value every 100ms. This will also result in receiving the last voltage value written in SPDR register in the slave MCU.
- For both Master and Slave, print both transmitted and received values on LCD.

Homework: Use SPI interrupts on both Master and Slave sides for non-blocking SPI communication.

ECE3411 – Fall 2017
Lec5b.

Real Time Operating Systems Cont'd

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

Copied from Lecture 6a and Lecture 7b, ECE3411 – Fall 2015,
by Marten van Dijk and Syed Kamran Haider

Slides on RTOS are extracted or copied from MIT 16.07 (Smith) and "Embedded Software Architecture", Cook & Freudenberg, 2008.

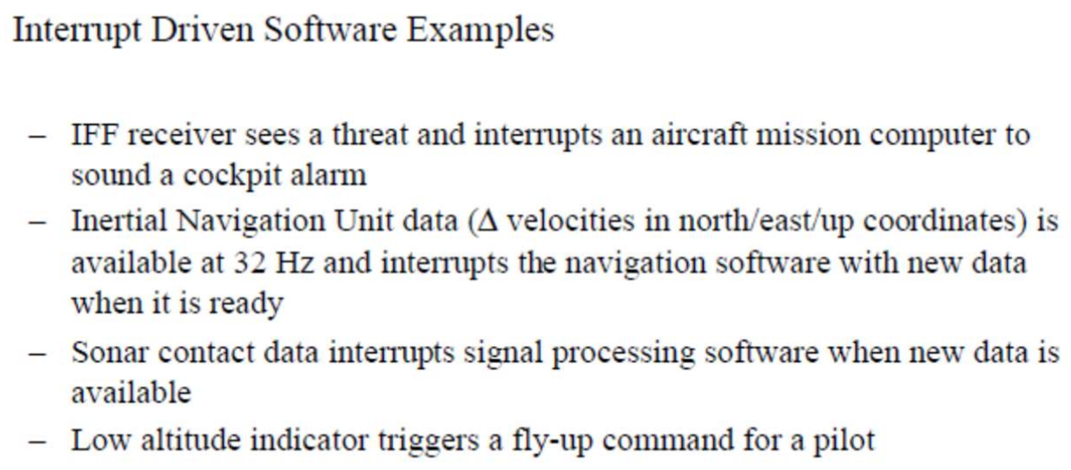
Realtime Kernel Design Strategies

- Polled Loop Systems
- Interrupt Driven Systems
- Multi-Tasking
- Foreground/Background Systems

Polled Loops

- Simplest RT kernel
- A single and repetitive instruction tests a flag that indicates whether or not an event has occurred
  - Examples: Non-blocking LCD instructions, Non-blocking "get string" over the UART channel
- No intertask communication or scheduling needed. Only single tasks exist
- Excellent for handling high-speed data channels, especially when
  - Events occur at widely spaced intervals and
  - Processor is dedicated to handling the data channel

Examples interrupt-driven system



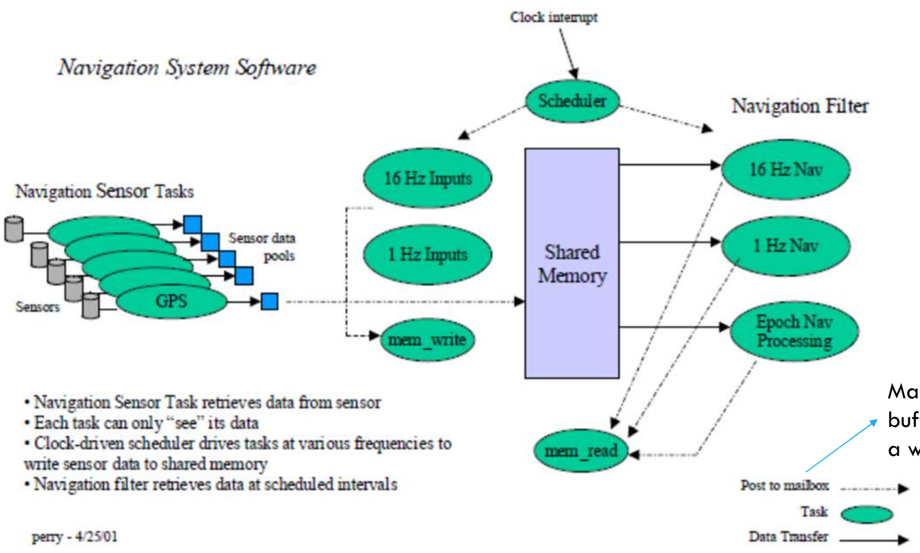
Interrupt Driven Software Examples

- IFF receiver sees a threat and interrupts an aircraft mission computer to sound a cockpit alarm
- Inertial Navigation Unit data (Δ velocities in north/east/up coordinates) is available at 32 Hz and interrupts the navigation software with new data when it is ready
- Sonar contact data interrupts signal processing software when new data is available
- Low altitude indicator triggers a fly-up command for a pilot

Multitasking

- Separate tasks that share one processor (or processors)
- Each task executes within its own context
  - Owns processor
  - Sees its own variables
  - May be interrupted
- Tasks may interact to execute as a whole program

Example



- Navigation Sensor Task retrieves data from sensor
- Each task can only "see" its data
- Clock-driven scheduler drives tasks at various frequencies to write sensor data to shared memory
- Navigation filter retrieves data at scheduled intervals

perry - 4/25/01

Mailbox is a circular buffer with a read and a write pointer

Context Switching

- When the CPU switches from one task to running another, its is said to have *switched contexts*
- Save the minimum needed to restore the interrupted process
  - Contents of registers
  - Contents of the program counter
  - Contents of coprocessor registers (if applicable)
  - Memory page registers
  - Memory-mapped I/O
  - Special variables
- During context switching, interrupts are often disabled
- Real time systems require minimal times for context switches

7

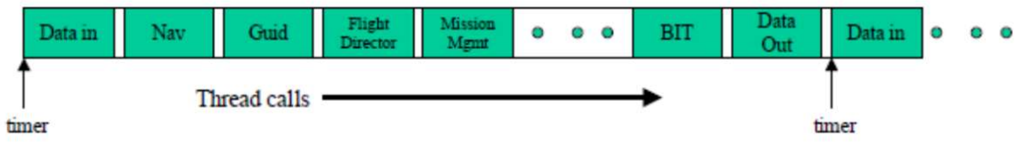
Multitasking

- How do many tasks share the same CPU?
  - Cyclic executive systems
  - Round robin systems
  - Pre-emptive priority systems

8

Cyclic Executive Systems

- Calls to statically ordered threads

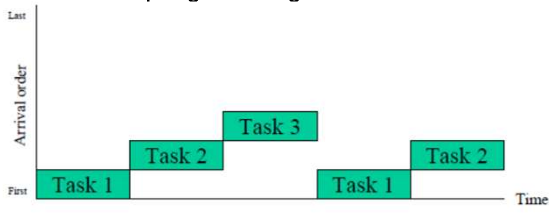


- Pros
  - Easy to implement (used extensively in complex safety critical systems)
- Cons
  - Not efficient in overall usage of CPU processing
  - Does not provide optimal response time

9

Round Robin Systems

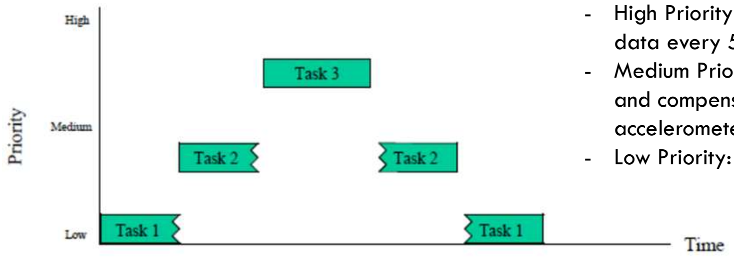
- Several processes execute sequentially to completion
- Often in conjunction with a cyclic executive
- Each task is assigned a fixed time slice
- Fixed rate clock initiates an interrupt at a rate corresponding to the time slice
  - Task executes until it completes or its execution time expires
  - Context saved if task does not complete
- Just like our task-based programming without fixed times slices per task



10

Pre-emptive Priority Systems

- Higher priority task can preempt a lower priority task if it interrupts the lower-priority task
- Priorities assigned to each interrupt are based upon the urgency of the task associated with the interrupt
- Priorities can be fixed or dynamic
  - Round Robin Systems - Preemptive Scheduling of 3 Tasks



Example: Aircraft Navigation System
- High Priority: Task that checks accelerometer data every 5ms
- Medium Priority: Task that collects gyro data and compensates this data and the accelerometer data every 40ms
- Low Priority: Display update, Built-in-Test (BIT)

11

Problems Multitasking

- High priority tasks hog resources and starve low priority tasks
- Low priority tasks share a resource with high priority tasks and block high priority tasks

- How does a RTOS deal with some of these issues?
  - Rate Monotonic Systems (higher execution frequency = higher priority)
  - Priority Inheritance

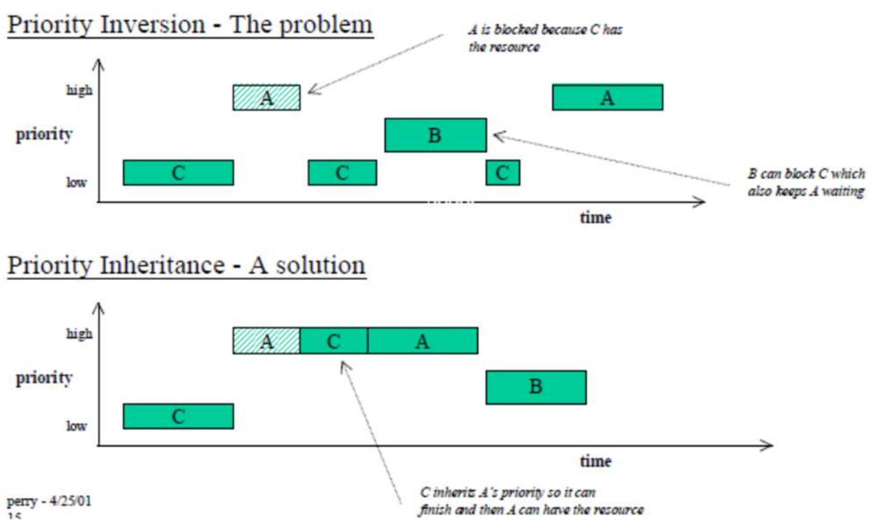
12

Priority Inversion / Priority Inheritance

- Task A and Task C share a resource
- Task A is high priority
- Task C is low priority
- Task A is blocked when Task C runs (effectively assigning A to C's priority, hence priority inversion)
- Task A will be blocked for longer, if Task B of medium priority comes along to keep Task C from finishing
- A good RTOS would sense this condition and temporarily promote Task C to the high priority of Task A (Priority Inheritance)

13

Priority Inversion / Priority Inheritance


Priority Inversion - The problem
A is blocked because C has the resource
high
priority
low
A
B
C
C
C
A
B can block C which also keeps A waiting
time
Priority Inheritance - A solution
high
priority
low
A
C
A
B
C
time
C inherits A's priority so it can finish and then A can have the resource
perry - 4/25/01

14

Foreground/Background Systems

- Most common hybrid solution for embedded applications
- Involve interrupt driven (foreground) AND noninterruptive driven (background) processes
- All realtime solutions are just a special case of foreground/background systems
  - Polled loops = background only system
  - Interrupt-only systems = foreground only system
- Anything not time-critical should be in background
  - Background is process with lowest priority

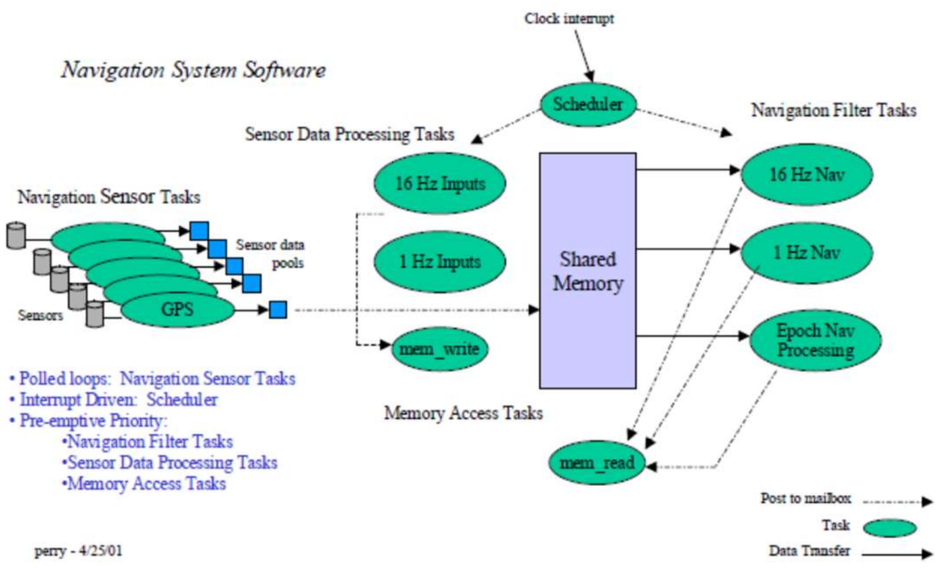
15

Foreground/Background Systems

- Gives hybrid systems = combining what we have seen so far
  - Polled loops
  - Interrupt-driven systems
  - Multi-tasking
    - Pre-emptive priority or
    - Round robin or
    - Cyclic executive

16

Back to the multitasking example



17

Multitasking Pros & Cons

- Pros
  - Segments the problem into small, manageable piece (modular computer system design principle)
  - Makes more modular software (can reuse portions more easily)
  - Allows software designer to prioritize certain tasks over others
- Cons
  - Depending upon implementation, timing may not be deterministic (jitter caused by variations in timing of incoming data)
  - Context switching adds overhead

18

Full Featured RTOS

- Expand foreground/background solution
  - Add network interfaces
  - Add device drivers
  - Add complex debugging tools
- Most common choice for complex systems
- Many commercial operating systems available

Choosing a RTOS approach

- How do you know which one is right for your application?
- Look at what is driving your system (arrival pattern of data)
  - Irregular (known but varying sequence of intervals between events)
  - Bursty (arbitrary sequence with bound on number of events)
  - Bounded (minimum interarrival interval)
  - Bounded with average rate (unpredictable event times, but cluster around mean)
  - Unbounded (statistical prediction only)
- What is the critical I/O?
- Are there absolute hard deadlines?

Choosing a RTOS approach

How do you know which one is right for your application? Let's look at some real life choices.

- Reusable Launch Vehicle for satellites. Thrust Vector Control SW requires new attitude data every 40 msec or rocket becomes unstable.
  - *We chose cyclic executive.*
- Navigation and Control System for submarine. Interface to multiple sensors at multiple data rates. Information from the Inertial Reference Unit is most critical, but exact timing of input data is not essential.
  - *We chose preemptive priority scheme running on a commercial RTOS. Important tasks given highest priority.*

21

Choosing a RTOS approach

How do you know which one is right for your application? Let's look at some real life choices.

- Avionics System requires new data from flight control surfaces, navigation equipment, and radar system every 50 msec.
  - *Cyclic executive. Each task runs to completion. Tasks run in series. Last tasks may not finish before 50msec interrupt occurs.*
- Microcontroller running to switch radar antennae and check for incoming signal. If the signal is there, power up the signal processing chip.
  - *We chose polled loop.*

22

ECE3411 – Fall 2017
Lab 5b.

DAC: Digital to Analogue Conversion

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

Copied from Lab 6c, ECE3411 – Fall 2015, by
Marten van Dijk and Syed Kamran Haider
With the help of:
www.wikipedia.org
ATmega328P Datasheet

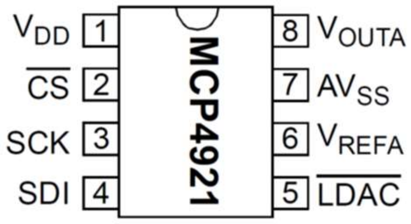
UCONN

DAC: Digital to Analog Converter

We use an external DAC for this lab: MCP4921

- 12 bit resolution.
- SPI interface.

1	V_{DD}	Positive Power Supply Input (2.7V to 5.5V)
2	CS	Chip Select Input. (SPI Slave Select)
3	SCK	SPI Serial Clock Input
4	SDI	SPI Serial Data Input (MOSI)
5	LDAC	Synchronization input used to transfer DAC settings from serial latches to the output latches.
6	V_{REFA}	DAC_A Voltage Input (AV_{SS} to V_{DD})
7	AV_{SS}	Analog ground
8	V_{OUTA}	DAC_A Output

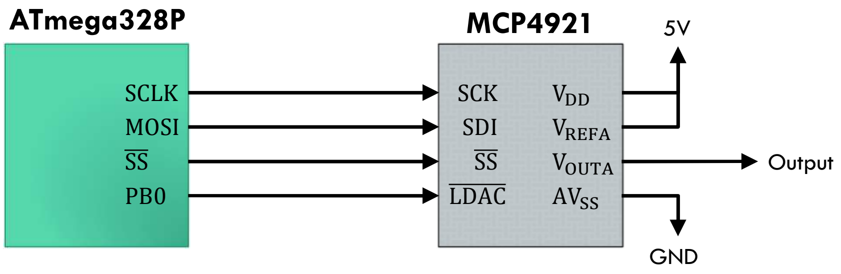


2

DAC SPI Interface

MCP4921 acts as SPI Slave and only receives data → MISO is not connected.

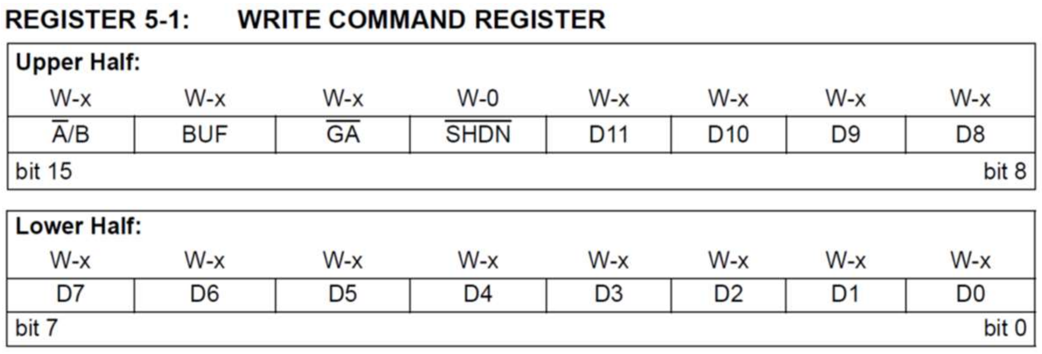
- Connect the ATmega328P with MCP4921 as shown in the figure below.
- Notice that LDAC pin also needs to be connected to a GPIO pin on ATmega328P.



3

DAC SPI Frame Format

- MCP4921 receives a 16-bit word from the MCU in two 8-bit SPI transactions.
- The format of the 16-bit frame containing 4 command and 12 data bits is shown below.



REGISTER 5-1: WRITE COMMAND REGISTER

Upper Half:

W-x	W-x	W-0	W-0	W-x	W-x	W-x	W-x
$\overline{A}$/B	BUF	$\overline{GA}$	$\overline{SHDN}$	D11	D10	D9	D8
bit 15							bit 8

Lower Half:

W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x
D7	D6	D5	D4	D3	D2	D1	D0
bit 7							bit 0

4

DAC Command Bits

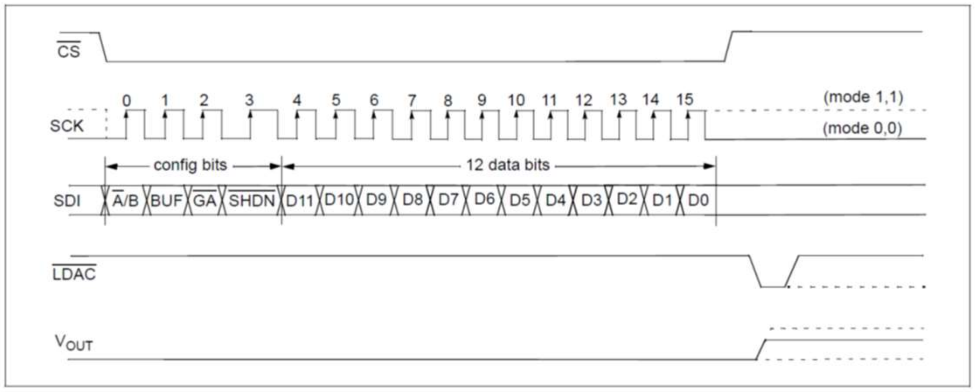
- The upper 4 bits of the 16 bit word are DAC command bits.
- The description of the 16 bit frame bits is as follows:

bit 15	**A/B:** DAC$_A$ or DAC$_B$ Select bit
	1 = Write to DAC$_B$
	0 = Write to DAC$_A$
bit 14	**BUF:** V$_{REF}$ Input Buffer Control bit
	1 = Buffered
	0 = Unbuffered
bit 13	**$\overline{\text{GA}}$:** Output Gain Select bit
	1 = 1x ($V_{OUT} = V_{REF} * D/4096$)
	0 = 2x ($V_{OUT} = 2 * V_{REF} * D/4096$)
bit 12	**$\overline{\text{SHDN}}$:** Output Power Down Control bit
	1 = Output Power Down Control bit
	0 = Output buffer disabled, Output is high impedance
bit 11-0	**D11:D0:** DAC Data bits
	12 bit number “D” which sets the output value. Contains a value between 0 and 4095.

5

DAC SPI Interface Timing

- The figure below shows the timing of one SPI transaction (command + data) between the MCU and DAC.
- You need to implement the same timing through SPI interface on ATmega328P.



6

Task: Controlling LED Glow

Write a simple program to control the glow of a LED using DAC.

In particular:

- Configure the SPI in Master mode.
- Read a potentiometer's voltage through ADC every 100ms (full 10 bit resolution).
- Normalize the 10-bit ADC reading to a 12-bit digital value for DAC.
- Transmit the 4-bit command and 12-bit data value to DAC over SPI.
- Don't forget to generate a LOW pulse at LDAC pin after transmission.
- Print the ADC's and DAC's readings on LCD.

Homework: Use DAC to generate a 100Hz sine wave with a peak-to-peak amplitude of 5V.

ECE3411 – Fall 2017

Lec5c.

I^2C
RedBot & DC Motor
Servo Motor Control

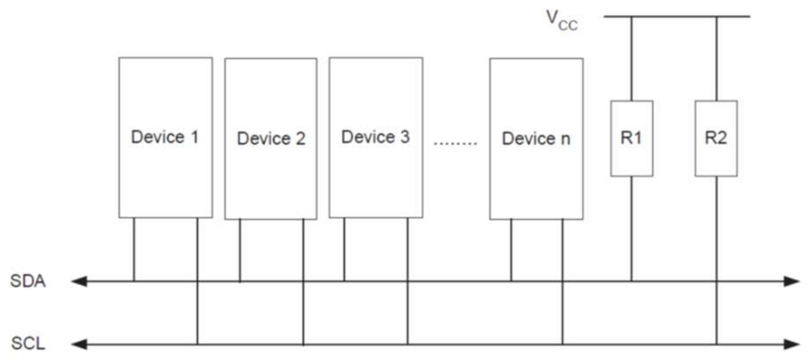
Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: marten.van_dijk@uconn.edu

Slides on I2C and Servo Motor Control copied from Lecture 7b, ECE3411 – Fall 2015, by Marten van Dijk and Syed Kamran Haider
Slides on RedBot adapted from Wikipedia and "RedBot Project" offered by Sung Yuel Park in Spring 2016

UCONN

I^2C: 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.

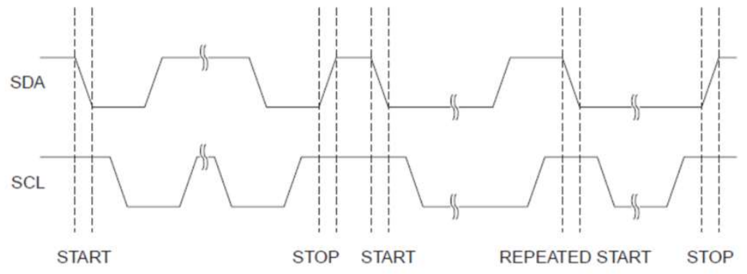


2

I^2C START and STOP Conditions

- START and STOP conditions are signaled by changing the level of the SDA line when the SCL line is high.
- When a new START condition is issued between a START and STOP condition, this is referred to as a REPEATED START condition

Figure 21-3. START, REPEATED START and STOP conditions

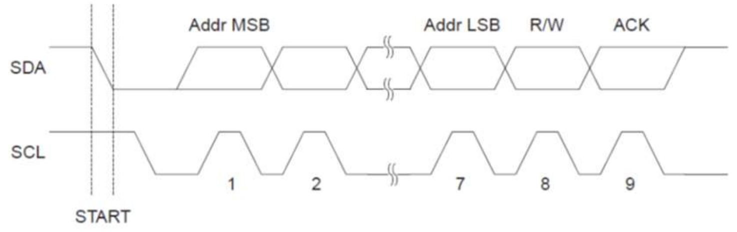


3

I^2C Address Packet Format

- All address packets transmitted on the TWI bus are 9 bits long:
  - 7 address bits, one READ/WRITE control bit and an acknowledge bit.
- When a Slave recognizes that it is being addressed, it should acknowledge by pulling SDA low in the ninth SCL (ACK) cycle.
- The Master can then transmit a STOP condition (by pulling SDA high), or a REPEATED START condition to initiate a new transmission.

Figure 21-4. Address Packet Format

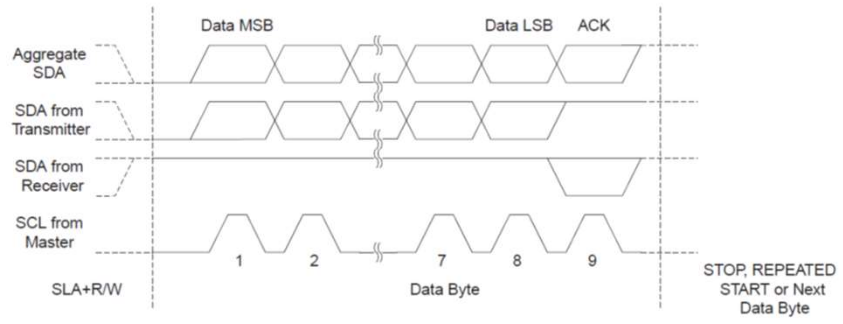


4

I^2C Data Packet Format

- All data packets transmitted on the TWI bus are 9 bits long:
  - One data byte and one acknowledge bit.
- An Acknowledge (ACK) is signaled by the Receiver pulling the SDA line low during the ninth SCL cycle. If the Receiver leaves the SDA line high, a NACK is signaled.

Figure 21-5. Data Packet Format

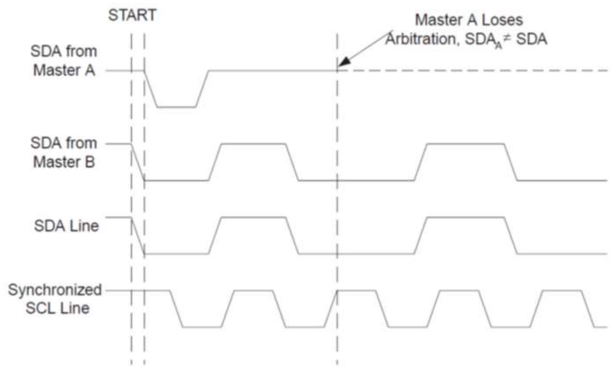


5

I^2C Bus Arbitration

- Arbitration is carried out by all masters (any device can become a master) 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



6

RedBot Project

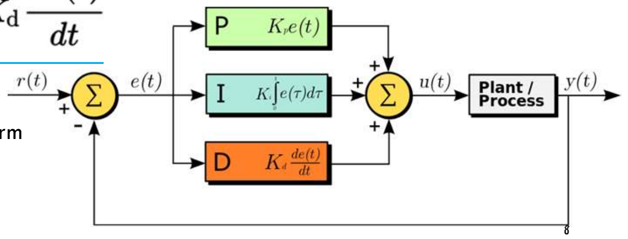
Titles	AVR Functions	Additional HW	SW Scenario
Line Follower	ADC, PWM, GPIO, UART	IR Sensor, H-bridge driver	Based on IR sensor input, RedBot needs to move along a line (black electrical tape)

7

PID Control

- A **proportional–integral–derivative controller** (PID controller) is a control loop feedback mechanism (controller) commonly used in industrial control systems.
- It continuously calculates an error value e(t) as the difference between a desired set-point and a measured process variable and applies a correction based on proportional, integral, and derivative terms.
  - e(t) = |(max possible "blackness" measured by the line sensor) – (currently measured "blackness" by the line sensor) |
  - u(t) = is the correcting rotation speed of the vehicle (measured as the duty cycle)

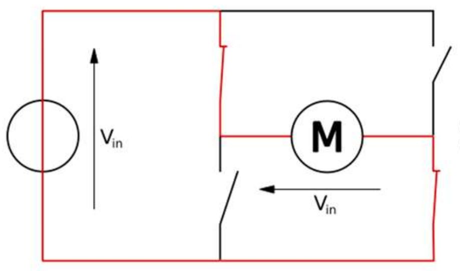
$$u(t) = \underbrace{K_{\mathrm{p}} e(t)}_{\text{Proportional Term}} + \underbrace{K_{\mathrm{i}} \int_0^t e(\tau)\, d\tau}_{\text{Integral Term}} + \underbrace{K_{\mathrm{d}} \frac{de(t)}{dt}}_{\text{Derivative Term}}$$



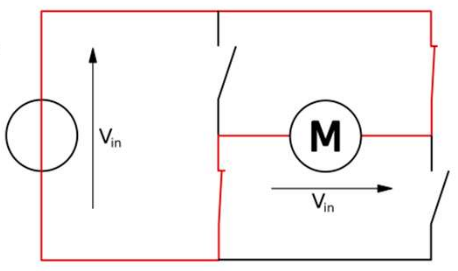
8

H Bridges

- An H bridge enables a voltage to be applied across a load in either direction. It is widely used in robotics to allow DC motors to run forwards or backwards.



Go forward



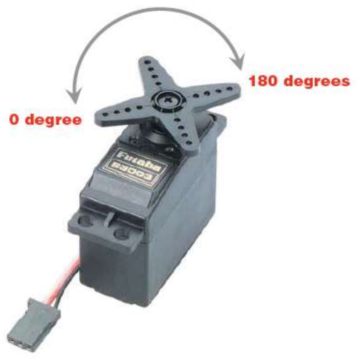
Go backward

In RedBot, the control signals of an H bridge will be derived from two GPIO signals and a PWM signal. The two GPIO signals control how this motor is connected in this H bridge, and the duty cycle of the PWM signal controls how often it is connected. So, GPIOs control the direction, and PWM signal controls the speed.

9

Servo Motor

- A Servo is a small device that has an output shaft that can be positioned to specific angular positions based on input PWM signal.
- The servo motor has a potentiometer that is connected to the output shaft and allows the control circuitry to monitor the current angle of the servo motor.
- A normal servo is used to control an angular motion of between 0 and 180 degrees.



Ref: http://lizarum.com/assignments/physical_computing/2008/servo.html

10

Servo Motor Applications

- Servos are typically used to control elevators, rudders and ailerons.

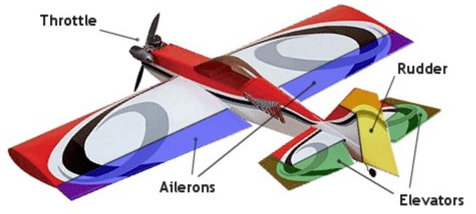
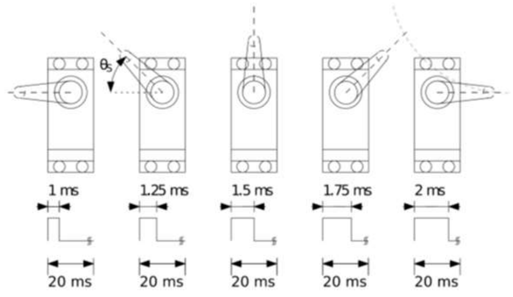


Image Refs: http://www.greatplanes.com/discontinued/gpma1414.html
http://www.rc-airplane-world.com/rc-airplane-controls.html

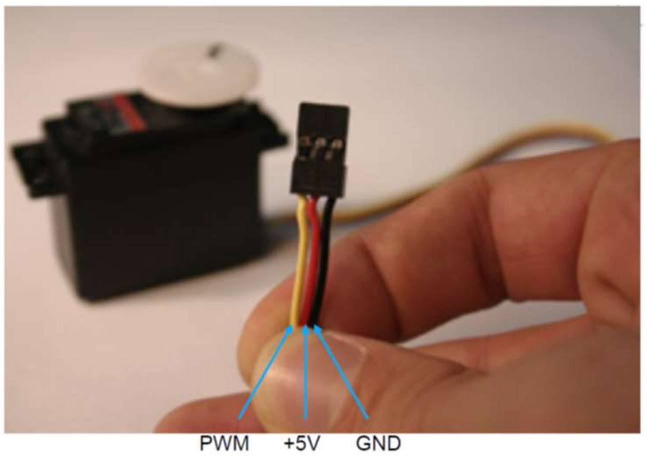
Controlling the servo

- The servo is controlled using a 50 Hz PWM signal (i.e. signal period = 20 ms)
- The angle of the servo is determined by the pulse width (i.e. the duty cycle)
  - 1.5ms corresponds to the center position.
- By varying the pulse width, we can control the angle
- The pulse width must never be outside the range 0.9 to 2.1ms



Connecting the servo

- Typically the servo connectors have 3 wires which should be connected as follows:
  - Red → VCC (+5V)
  - Black → GND (0V)
  - Yellow → PWM signal



ECE3411 – Fall 2017
Lab5c

I^{2}C: Inter Integrated Circuit

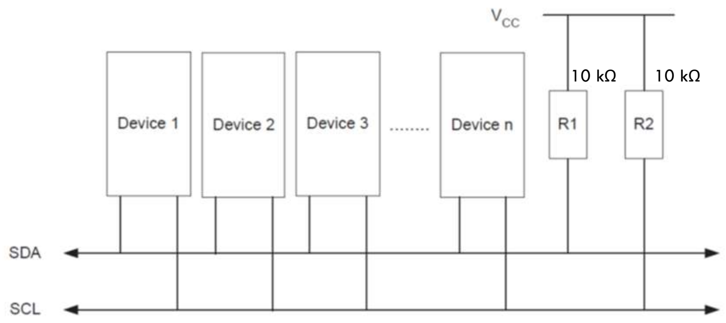
Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: marten.van_dijk@uconn.edu

UCONN

With the help of:
ATmega328P Datasheet

I^{2}C: Inter Integrated Circuit

- 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).
- A pull-up resistor (typically 10 kΩ) is needed for each of the TWI bus lines.
- All devices connected to the bus have individual addresses.



2

I^2C Terminologies

- I^2C (TWI) protocol allows several devices (up to 128) to be connected.
- Each device is identified by a configurable 7-bit address.
- Each device can communicate with any other device
  - The transmitter address the receiver by its 7-bit address.

Table 21-1. TWI Terminology

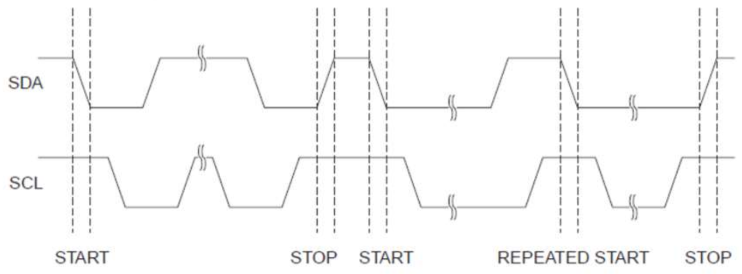
Term	Description
Master	The device that initiates and terminates a transmission. The Master also generates the SCL clock.
Slave	The device addressed by a Master.
Transmitter	The device placing data on the bus.
Receiver	The device reading data from the bus.

3

I^2C START and STOP Conditions

- START and STOP conditions are signaled by changing the level of the SDA line when the SCL line is high.
- When a new START condition is issued between a START and STOP condition, this is referred to as a REPEATED START condition

Figure 21-3. START, REPEATED START and STOP conditions

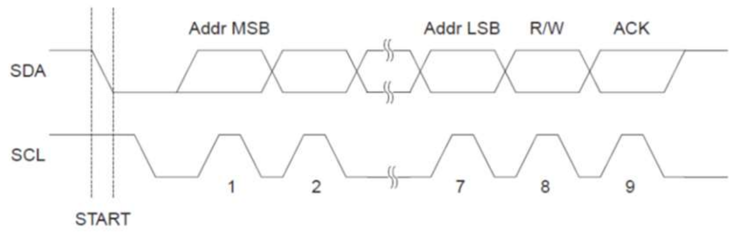


4

I^{2}C Address Packet Format

- All address packets transmitted on the TWI bus are 9 bits long:
  - 7 address bits, one READ/WRITE control bit and an acknowledge bit.
- When a Slave recognizes that it is being addressed, it should acknowledge by pulling SDA low in the ninth SCL (ACK) cycle.
- The Master can then transmit a STOP condition, or a REPEATED START condition to initiate a new transmission.

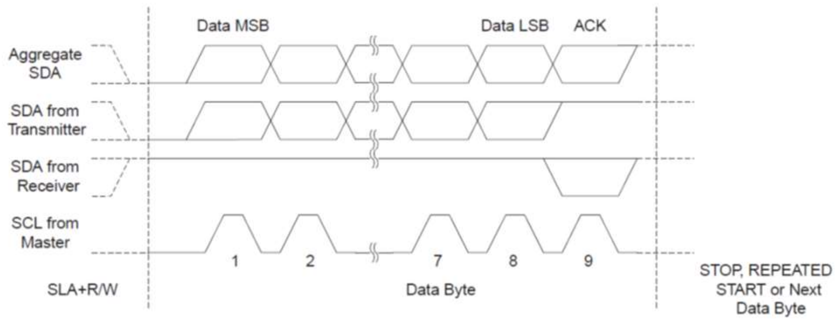
Figure 21-4. Address Packet Format



I^{2}C Data Packet Format

- All data packets transmitted on the TWI bus are 9 bits long:
  - One data byte and one acknowledge bit.
- An Acknowledge (ACK) is signaled by the Receiver pulling the SDA line low during the ninth SCL cycle. If the Receiver leaves the SDA line high, a NACK is signaled.

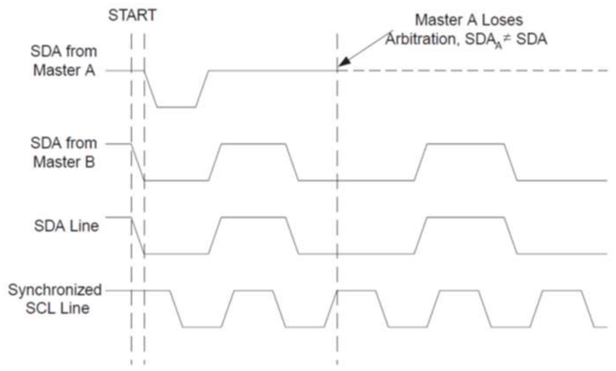
Figure 21-5. Data Packet Format



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.

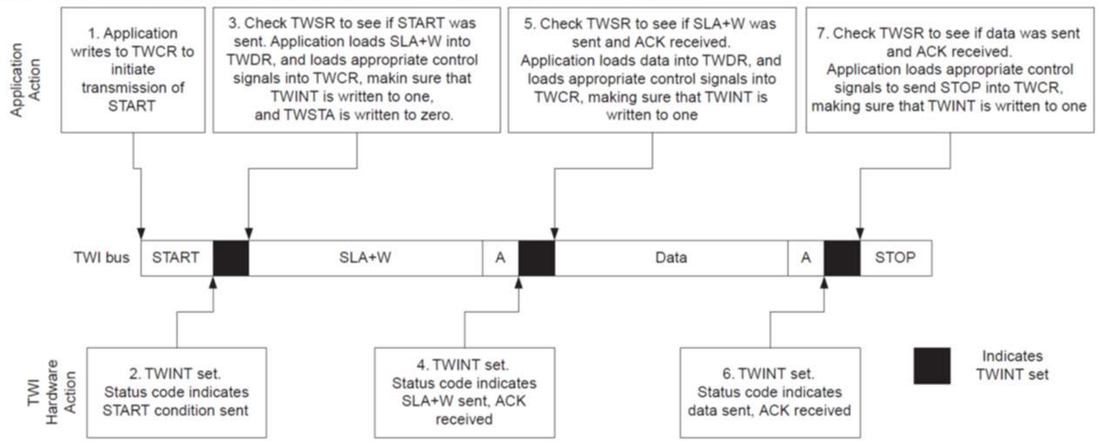
Figure 21-8. Arbitration Between Two Masters



7

A typical I^2C Transmission

Figure 21-10. Interfacing the Application to the TWI in a Typical Transmission


Application Action
1. Application writes to TWCR to initiate transmission of START
3. Check TWSR to see if START was sent. Application loads SLA+W into TWDR, and loads appropriate control signals into TWCR, makin sure that TWINT is written to one, and TWSTA is written to zero.
5. Check TWSR to see if SLA+W was sent and ACK received. Application loads data into TWDR, and loads appropriate control signals into TWCR, making sure that TWINT is written to one
7. Check TWSR to see if data was sent and ACK received. Application loads appropriate control signals to send STOP into TWCR, making sure that TWINT is written to one
TWI bus
START
SLA+W
A
Data
A
STOP
TWI Hardware Action
2. TWINT set. Status code indicates START condition sent
4. TWINT set. Status code indicates SLA+W sent, ACK received
6. TWINT set. Status code indicates data sent, ACK received
Indicates TWINT set

8

A typical I^{2}C Transmission Summary

- When the TWI has finished an operation and expects application response, the TWINT Flag is set. The SCL line is pulled low until TWINT is cleared.
- When the TWINT Flag is set, the user must update all TWI Registers with the value relevant for the next TWI bus cycle. As an example, TWDR must be loaded with the value to be transmitted in the next bus cycle.
- After all TWI Register updates and other pending application software tasks have been completed, TWCR is written. When writing TWCR, the TWINT bit should be set.
- Writing a one to TWINT clears the flag. The TWI will then commence executing whatever operation was specified by the TWCR setting.

9

I^{2}C Transmission Example

```
uint8_t TWI_Master_Transmit(uint8_t Address, uint8_t Data)
{
        TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN);        // Send START condition
        while (!(TWCR & (1<<TWINT)));                  // Wait for TWINT Flag set.
        if ((TWSR & 0xF8) != START)                    // Check value of TWI Status Register.
                ERROR();
        TWDR = (Address << 1) | (WRITE);               // Load SLA_W (Slave Address & Write) into TWDR Register.
        TWCR = (1<<TWINT) | (1<<TWEN);                 // Clear TWINT bit in TWCR to start transmission of address.
        while (!(TWCR & (1<<TWINT)));                  // Wait for TWINT Flag set.
        if ((TWSR & 0xF8) != MT_SLA_ACK)               // Check value of TWI Status Register.
                ERROR();
        TWDR = Data;                                   // Load DATA into TWDR Register.
        TWCR = (1<<TWINT) | (1<<TWEN);                 // Clear TWINT bit in TWCR to start transmission of data.
        while (!(TWCR & (1<<TWINT)));                  // Wait for TWINT Flag set.
        if ((TWSR & 0xF8) != MT_DATA_ACK)              // Check value of TWI Status Register.
                ERROR();
        TWCR = (1<<TWINT)|(1<<TWEN)| (1<<TWSTO);       // Transmit STOP condition.
}
```

Note: The code above assumes that several definitions have been made, for example by using include-files.

10

I^2C Reception Example

```
void TWI_Slave_Initialize(uint8_t Address)
{
        TWAR = (Address << 1)|(1);              // Load Slave Address into TWAR Register.
        TWCR = (1<<TWEA)|(1<<TWEN);             // Enable TWI & Acknowledgements.
}
```

```
uint8_t TWI_Slave_Receive(void)
{
        TWCR = (1<<TWEA)|(1<<TWEN);                   // Enable TWI & Acknowledgements.
        while (!(TWCR & (1<<TWINT)));                 // Wait for TWINT Flag set (once this slave is addressed)
        if ((TWSR & 0xF8) != 0x60)                    // Check value of TWI Status Register.
                ERROR();
        TWCR = (1<<TWINT) | (1<<TWEN);                // Clear TWINT bit start reception of data.
        while (!(TWCR & (1<<TWINT)));                 // Wait for TWINT Flag set.
        if ((TWSR & 0xF8) != 0x80)                    // Check if Data has been received & ACK has been returned
                ERROR();
        TWCR = (1<<TWINT) | (1<<TWEN);                // Clear TWINT bit.
        return TWDR;                                  // Read TWDR Register.
}
```

Note: The code above assumes that several definitions have been made, for example by using include-files.

11

Task1: I^2C Master Slave Communication

Write a program to send ADC voltage readings to your friend's board over I^2C bus.

- Configure your board as I^2C Master (f_{SCL} = 200kHz) and ask your friend to configure his as I^2C Slave.
- Make proper wire connections of SCK and SDA pins between the two boards.
  Don't forget to put a 10 kΩ pullup resister on each line.
- In Master MCU, read a potentiometer's voltage through ADC every 100ms (only upper 8 bits).
- Transmit Master's voltage value every 100ms.
- For Master, print the transmitted reading on UART.
- For Slave, print the received reading on UART.

Homework: Use I^2C interrupts on both Master and Slave sides for non-blocking I^2C implementation.

12

Department of Electrical and Computing Engineering

UNIVERSITY OF CONNECTICUT

ECE 3411 Microprocessor Application Lab: Fall 2017

Problem Set P5

There are 6 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/18)	2 (x/20)	3 (x/16)	4 (x/16)	5 (x/20)	6 (x/10)	Total (xx/100)

Name:

Student ID:

1. [18 points]: The following code tries to implement a 1kHz PWM signal whose duty cycle is varied in way that it results in 10Hz sawtooth waveform. The MCU clock frequency is 16MHz. List all the bugs that you can identify in this code, and mention how would you fix them.

```
uint16_t step = 80;
uint16_t time_period = 16000;
uint16_t duty_cycle = 0;

void main(void)
{
    /* Configuring Timer 1 for PWM generation */
    OCR1A = time_period-1;
    OCR1B = duty_cycle;
    TCCR1A |= (1<<WGM11) | (1<<WGM10);    // turn on Fast PWM mode
    TCCR1B |= (1<<WGM13) | (1<<WGM12);    // turn on Fast PWM mode
    TIMSK1 |= (1<<OCIE1B);                // Enable Interrupt
    TCCR1B |= (1<<CS10);                  // Set pre-scaler @ 1

    while(1);    // Nothing to do
}

ISR(TIMER0_COMPA_vect)
{
    duty_cycle += step;
    duty_cycle = duty_cycle % (time_period-1);
    OCR1B = duty_cycle;
}
// ---------------------------------------------------------------
```

2. [20 points]: Answer the following short questions:

a. Why is 128 the maximum number of devices that can be connected together in one I2C network?

b. In task 1 of Lab 5c, you were required to establish communication between two MCUs via I2C. What wire connections were needed?

c. How many wires does the SPI protocol use?

d. Building on to part(c), what is each wire used for?

Initials:

3. [16 points]: Encircle the correct answer for the following:

a. SPI and I^2C protocols are:

(a) Asynchronous
(b) Synchronous
(c) None of the above

b. SPI and I^2C protocols:

(a) SPI is half duplex whereas I^2C is full duplex.
(b) SPI is full duplex whereas I^2C is half duplex.
(c) Both SPI and I^2C are full duplex.
(d) Both SPI and I^2C are half duplex.

c. Suppose a SPI Slave wants to send 2 byes of data to a SPI Master. Select the correct statement from the following:

(a) Slave initiates SPI communication by itself and sends 2 bytes of data.
(b) Slave asks the Master to initiate SPI communication and then sends 2 bytes of data to Master.
(c) Master initiates SPI communication by itself and sends 2 bytes of data to Slave.
(d) None of the above

d. Which one is correct for transferring a data byte from an I^2C Master to Slave:

(a) Master sends Address, Slave ACKs, Master sends Data, Slave ACKs.
(b) Slave sends Address, Master ACKs, Master sends Data, Slave ACKs.
(c) Slave requests Address, Slave ACKs, Master sends Data, Master ACKs.
(d) None of the above

Initials:

4. [16 points]:

a. What is context switching? List at least three kinds of data which needs to be saved in a context switch.

b. What is the mechanism (as explained in lecture) behind controlling the angle of a servo motor using one PWM signal control?

Initials:

5. [20 points]: *Earliest Deadline First* (EDF) algorithm schedules the task whose deadline is closest in time.
Complete the following chart according to **non-preemptive** version of *Earliest Deadline First* (EDF) scheduling algorithm. The task specifications are given in the following table.

Task	Required CPU Time (ms)	Task Period (ms)
Task A	1	6
Task B	2	5
Task C	4	10

The upward arrows (↑) in the chart show that the task is ready to be scheduled. Hence each next arrow also represents the deadline of the task needed to be executed in the previous period.

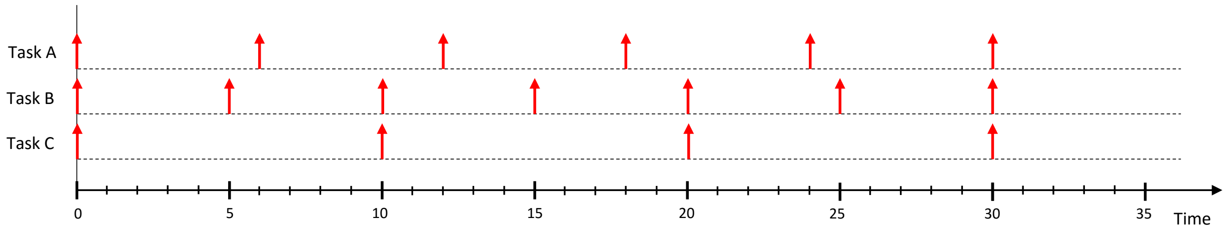


Figure 1: EDF Scheduling Chart.

Initials:

6. [10 points]: Can you shortly describe what you have learned and feel confident about using in the future?

End of Problem Set

Initials:

Department of Electrical and Computing Engineering

UNIVERSITY OF CONNECTICUT

ECE 3411 Microprocessor Application Lab: Fall 2017

Problem Set A5

There are 4 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/20)	2 (x/30)	3 (x/30)	4 (x/20)	Total (xx/100)

Name:

Student ID:

1. [20 points]: Below is a program layout with comments explaining what happens during program execution. Also the meaning of all register initializations is given (there is no need to look into the ATmega328P data sheet).

You should pay attention to the main body which initializes Timer1, and polls its value before an ADC measurement in sleep mode starts and after the execution of an "ADC task" finishes. The difference of the two values is converted to micro seconds and added to a variable busy. The goal for busy is to measure the time during which the MCU is doing "useful" work. The code that is related to busy is highlighted with vertical bars.

After the program layout below, the first subproblem asks you what is truly measured by busy in the program and the second subproblem asks you to explain the code which converts the difference to micro seconds.

```
... we assume a clock frequency of 20MHz    ...
... inclusion of packages                   ...
... declaration of global variables         ...

// ------------------------------------------------- //

ISR (TIMER0_COMPA_vect)
{
    /* Update task timer */
    if (taskADC_timer >0 )  {--taskADC_timer;}
}

// ------------------------------------------------- //

ISR (ADC_vect)
{
    /* Read a 10-bit conversion */
    AinLow = (int)ADCL;
    Ain = (int)ADCH*256;
    Ain = Ain + AinLow;
}

// ------------------------------------------------- //

void taskADC(void)
{
    /* Reset task timer */
    taskADC_timer = 400;

    //Convert Ain into a voltage
    voltage = ((1.0*Ain)/1024.0)*5.0;

    ... Some more computation: sometimes taking more and sometimes taking less time ...
    ... However, no matter how long taskADC() takes, its execution is always <= 200 ms ...
}
```

Initials:

```
int main(void)
{
    ... initialization variables ...

|    // set up timer 1 for 3.2 micro second counter increments
|    TCCR1B = 3;           //set prescalar to divide by 64

    //set up timer 0 such that ISR(TIMER0_COMPA_vect) is called every 1 milli second
    OCR0A  = 77;          //Set the compare reg to 78 time ticks
    TIMSK0 = (1<<OCIE0A); //Turn on timer 0 cmp match ISR
    TCCR0B = 4;           //Set prescalar to divide by 256
    TCCR0A = (1<<WGM01);  //Turn on clear-on-match

    // initialize the ADC
    ADMUX  = 6;                            // Select ADC Channel 6
    ADCSRA = (1<<ADEN) | (1<<ADIE) + 7 ; // Enable AD converter, enable its interrupt,
                                         // set prescalar (notice that the ADSC bit is
                                         // not set, so no ADC conversion is started)
    SMCR   = (1<<SM0) ;                    // Choose ADC sleep mode

    sleep_enable();
    sei();

    while (1)
    {
        if (taskADC_timer == 0)
        {
|           // Measure timer 1
|           T1poll_before = TCNT1;

            //Perform an ADC measurement in sleep mode, and execute taskADC
            sleep_cpu();
            taskADC();

|           //Measure timer 1 again and update busy with the amount of micro seconds that
|           //have passed: every TCNT1 to TCNT1+1 increment takes 3.2 micro seconds.
|           T1poll_after = TCNT1;
|
|           if (T1poll_after > T1poll_before) {
|               busy += (T1poll_after - T1poll_before)*3.2;
|           } else {
|               busy += ( (T1poll_after - T1poll_before) + 65536 ) * 3.2;
|           }
        } /* end of if (taskADC_timer == 0) */
    } /* end of while(1) */
} /* end of main() */
```

Initials:

The data sheet (section 9.4) writes for the ADC Noise Reduction Mode that ”... the SLEEP instruction makes the MCU enter ADC Noise Reduction mode, stopping the CPU but allowing the ADC, the external interrupts, 2-wire Serial Interface address match, Timer/Counter2 and the Watchdog to continue operating (if enabled) ...”. This means that all other hardware modules stop working, in particular, the other timers/counters stop incrementing.

A. (16 points) Answer with ”never”, ”sometimes”, or ”always”, whether the execution times (measured in micro seconds) of the following procedures are added into `busy` variable. Explain your answers.

(a) `ISR(TIMER0_COMPA_vect)`

(b) `ISR(ADC_vect)`

(c) `sleep_cpu()`

(d) `taskADC()`

Initials:

B. (4 points) The program assumes that `taskADC()` always takes $\leq$ 200 ms. Use this assumption to explain why the code

```
if (T1poll_after > T1poll_before) {
    busy += ( T1poll_after - T1poll_before ) * 3.2;
} else {
    busy += ( (T1poll_after - T1poll_before) + 65536 ) * 3.2;
}
```

correctly adds to `busy` the time in micro seconds that passed between the polling of `T1poll_before` and the polling of `T1poll_after`.

Initials:

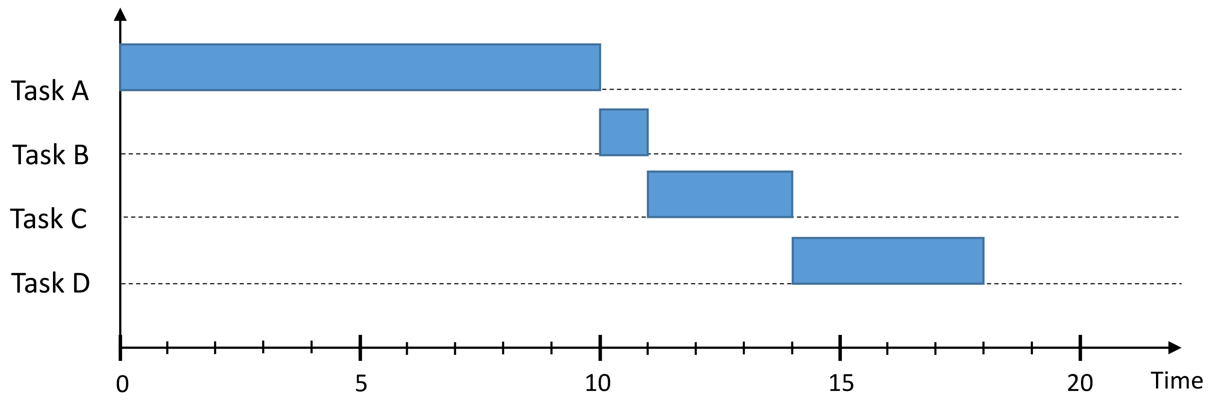
2. [30 points]: Table 1 shows the characteristics of four tasks that need to be scheduled on the MCU.

"Ready Time" indicates when the corresponding task is ready to execute. In this example, all the tasks are ready and want to execute as soon as the system starts, i.e. at time 0. "Required CPU Time" indicates how many time units are needed for the task to finish.

Table 1: Task Specifications

Task	Ready Time	Required CPU Time
A	0	10
B	0	1
C	0	3
D	0	4

The following figure shows an example of First Come First Serve scheduling of these tasks assuming the order A, B, C, and then D.



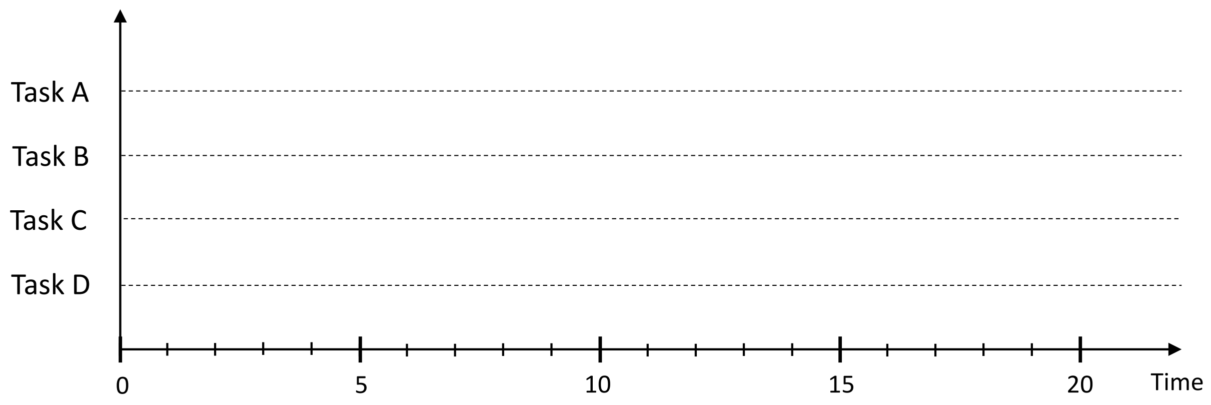
The following table shows the completion times of the tasks and their corresponding wait times (i.e. while the task is suspended and waiting for the CPU) under First Come First Serve scheduling scheme.

Table 2: Analysis under First Come First Serve Scheduling

Task	Ready Time	Required CPU Time	Completion Time	Wait Time
A	0	10	10	0
B	0	1	11	10
C	0	3	14	11
D	0	4	18	14
Average			13.25	8.75

Initials:

A. Round Robin Scheduling: (15 points)
Plot how the tasks A, B, C, and D will be scheduled on the CPU under **Round Robin Scheduling** with a **time slice of 1 time unit**. I.e. assuming an order of A, B, C, and D; the tasks take turns for the CPU and each task gets the CPU for 1 time unit in each turn until the task finishes. Assume that no time is wasted in context switching between the tasks.



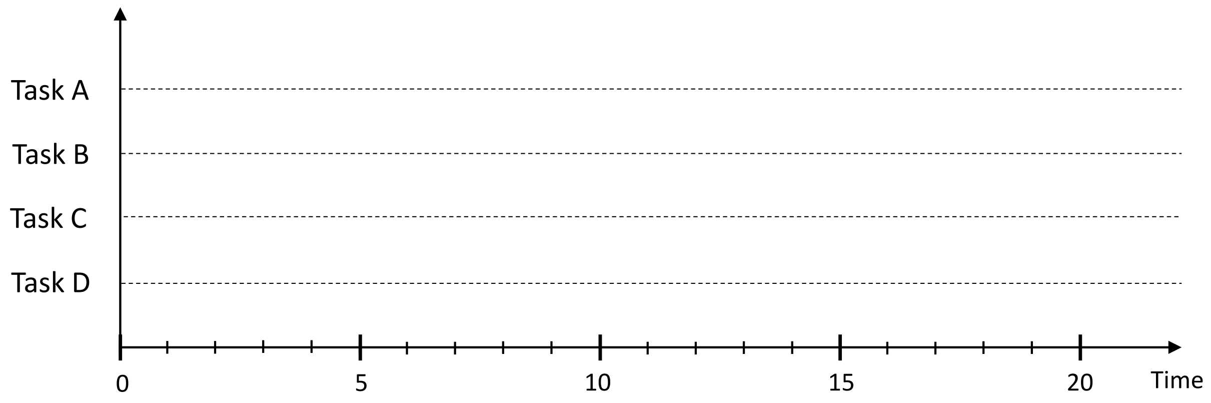
Use the plot above to complete the following table.

Table 3: Analysis under Round Robin Scheduling

Task	Ready Time	Required CPU Time	Completion Time	Wait Time
A	0	10		
B	0	1		
C	0	3		
D	0	4		
Average				

Initials:

B. Shortest Remaining Time First Scheduling: (15 points)
Plot how the tasks A, B, C, and D will be scheduled on the CPU under **Shortest Remaining Time First Scheduling**. I.e. whichever task needs the shortest amount of CPU time to finish gets the CPU first. Once this task is finished, another task that needs the smallest CPU time is executed. Assume that no time is wasted in scheduling a new task once a task finishes.



Use the plot above to complete the following table.

Table 4: Analysis under Shortest Remaining Time First Scheduling

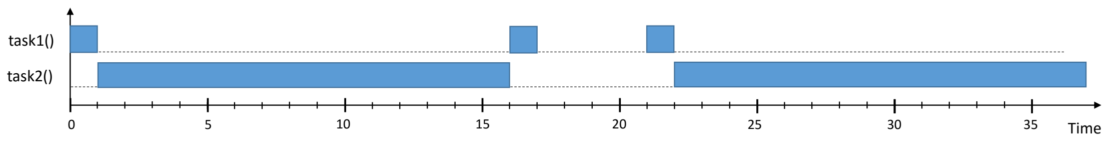
Task	Ready Time	Required CPU Time	Completion Time	Wait Time
A	0	10		
B	0	1		
C	0	3		
D	0	4		
Average				

Initials:

3. [30 points]: Consider the following while loop in the main code:

```
/* If not already equal to 0, task1_timer and task2_timer
   are decremented every 1 millisecond in a timer ISR */
task1_timer = 0;    // Initializing. task1 is ready to execute
task2_timer = 0;    // Initializing. task2 is ready to execute
while (1)
{
   if (task1_timer == 0)  // if task1 is ready to run.
   {
      task1_timer = t1;
      task1();    // task1 takes m1 milliseconds to execute.
   }
   if (task2_timer == 0)  // if task2 is ready to run.
   {
      task2_timer = t2;
      task2();    // task2 takes m2 milliseconds to execute.
   }
}
```

Suppose if `t1=5`, `m1=1` and `t2=20`, `m2=15` then the corresponding execution pattern is shown below:



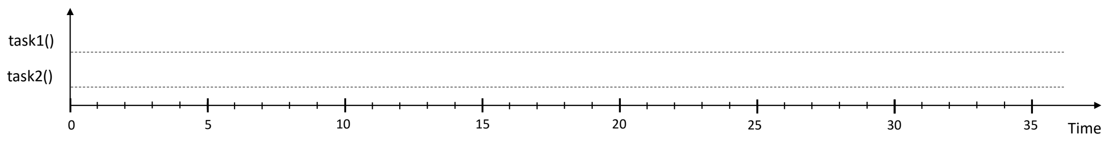
Here `task1()` executes for 1 ms, `task2()` executes for 15 ms, and the MCU is idle for 4 ms. The same pattern starts getting repeated over and over.

In this example pattern, `task2()` is executed once per 21 ms (a frequency of 1/(21 ms)) and `task1()` is executed on average two times per 21 ms (an average frequency of 2/(21 ms)).

Similar to the above mentioned example, answer the following questions.

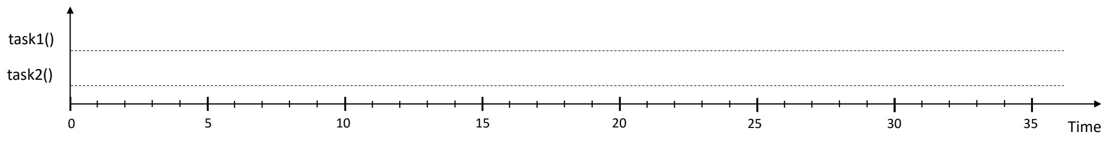
Initials:

A. Suppose `t1=5`, `m1=1`, `t2=10`, and `m2=15`.
Draw the execution pattern of the two tasks.



What is the frequency $f1$ in Hz at which `task1()` is called?
What is the frequency $f2$ in Hz at which `task2()` is called?

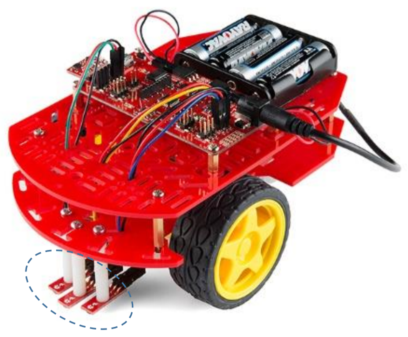
B. Suppose `t1=20`, `m1=1`, `t2=10`, and `m2=15`.
Draw the execution pattern of the two tasks.



What is the frequency $f1$ in Hz at which `task1()` is called?
What is the average frequency $f2$ in Hz at which `task2()` is called?

Initials:

4. [20 points]: The robot shown below is a RedBot that you will be using in the last lab. It is a line follower which can follow a black line. It has three line sensors in the front encircled by the dashed line. Write a function to control the movement of this RedBot with the help of the given functions.



The functions you can use are:

- void left_wheel_forward(void);
- void left_wheel_backward(void);
- void right_wheel_forward(void);
- void right_wheel_backward(void);

The inputs of your program are the values sampled by three line sensors. For each line sensor, suppose that it reads more than T when it is over the black line. And assume the black line is as thick as a single sensor. Fill the program below.

```
void redbot_control(int left_sensor, int middle_sensor, int right_sensor)
{









}
```

Initials:

End of Problem Set

Initials:

Department of Electrical and Computing Engineering

UNIVERSITY OF CONNECTICUT

ECE 3411 Microprocessor Application Lab: Fall 2017

Independent LAB5

There are 2 independent lab questions in LAB5.

You may not discuss independent labs in any way, shape, or form with anyone else and you are not allowed to lookup solutions from other sources.

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

Name:

Student ID:

1. [Pass/Fail points]: In this task, you need to implement *non-preemptive* version of **Shortest Remaining Time First** scheduling for a given set of periodic tasks. The task specifications are given in the following table.

Table 1: Task Specifications

Task	Required CPU Time (ms)	Task Period (ms)
`task_0()`	400	1000
`task_1()`	200	500
`task_2()`	100	600

You are provided with a C source file named `scheduling_SW.c` which provides a framework to run these tasks.

You need to implement the function called `SRTF_scheduler()` such that it schedules the tasks w.r.t. *shortest remaining time first* scheduling. Once you have implemented the scheduler, connect a UART terminal to your MCU and see the resulting order of the tasks.

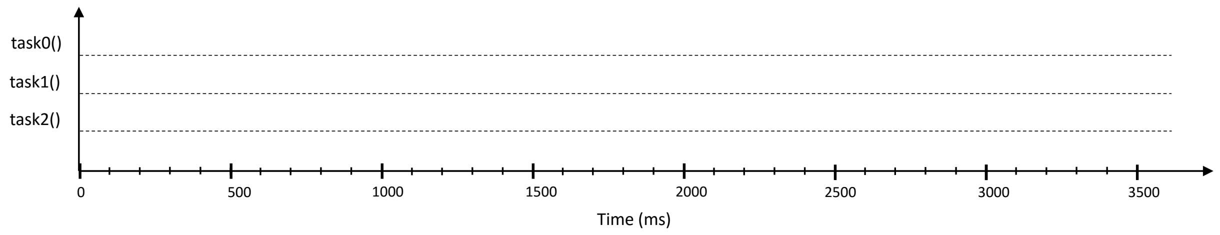
Write the resulting pattern of the tasks in the space given below. Just write the task numbers to show the order in which they are scheduled.

Table 2: Scheduler Output

No.	1	2	3	4	5	6	7
Task #							
No.	8	9	10	11	12	13	14
Task #							

Notice that if your scheduler is not working properly, at least one of the task would miss its deadline and a message will be printed on UART followed by suspension of the program.

It would be easier for you to program/verify your scheduler's behavior by plotting the intended behavior on a paper first. For this purpose, you may use the space provided below.



Initials:

2. [Pass/Fail points]: In this task, you need to implement **Non-blocking SPI**. Write a simple program to test SPI in loopback mode. In particular:

- Configure SPI in Master mode with SPI interrupt enabled.
- Configure Timer1, with compare match interrupt enabled, in CTC mode to overflow after every 100ms.
- In Timer1 ISR, initiate an ADC conversion to read a potentiometers voltage (only upper 8 bits) every 100ms.
- In ADC ISR, once the conversion is complete, initiate a SPI transmission to transmit the byte containing voltage reading over SPI.
- Loopback the transmitted byte by connecting MOSI and MISO pins together.
- Once SPI interrupt triggers, print on LCD the byte value received over SPI.

Implement this system by filling in the gaps in the code layout given below.
Notice that busy waiting on SPIF flag after initiating a SPI transmission is **not allowed**.

The following code snippet provides the necessary layout and definitions.

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <avr/pgmspace.h>
#include <inttypes.h>
#include <avr/interrupt.h>
#include <stdio.h>
#include <string.h>
#include "lcd_lib.h"

// SPI related definitions
#define DDR_SPI     DDRB
#define SPI_SS      2
#define SPI_MOSI    3
#define SPI_MISO    4
#define SPI_SCK     5

// Variables
volatile unsigned int Ain;
volatile uint8_t data_byte;

// LCD Strings
char lcd_buffer[17]; // LCD display buffer
const uint8_t LCD_Master[] PROGMEM = "Master: ";

//----------------------------------------------------------------------



// All initializations
```

Initials:

```
void initialize_all(void)
{
    /* Configure LCD, Timer1, ADC and SPI */
}

//------------------------------------------------------------------------

// Timer 1 Compare Match A ISR (TCNT1 = OCR1A)
ISR (TIMER1_COMPA_vect)
{
    /* Write your code here */
}
//------------------------------------------------------------------------

// ADC ISR
ISR(ADC_vect)
{
    /* Write your code here */
}
//------------------------------------------------------------------------

// SPI ISR
ISR(SPI_STC_vect)
{
    /* Write your code here */
}
//------------------------------------------------------------------------

/* Main Function */
int main(void)
{
    initialize_all();     // Initialize everything
    sei();                // Enable Global Interrupts
    while(1);             // Nothing to do.
}
//------------------------------------------------------------------------
```

Initials:

End of LAB5

Initials: