ECE3411 – Fall 2015

Wrap Up

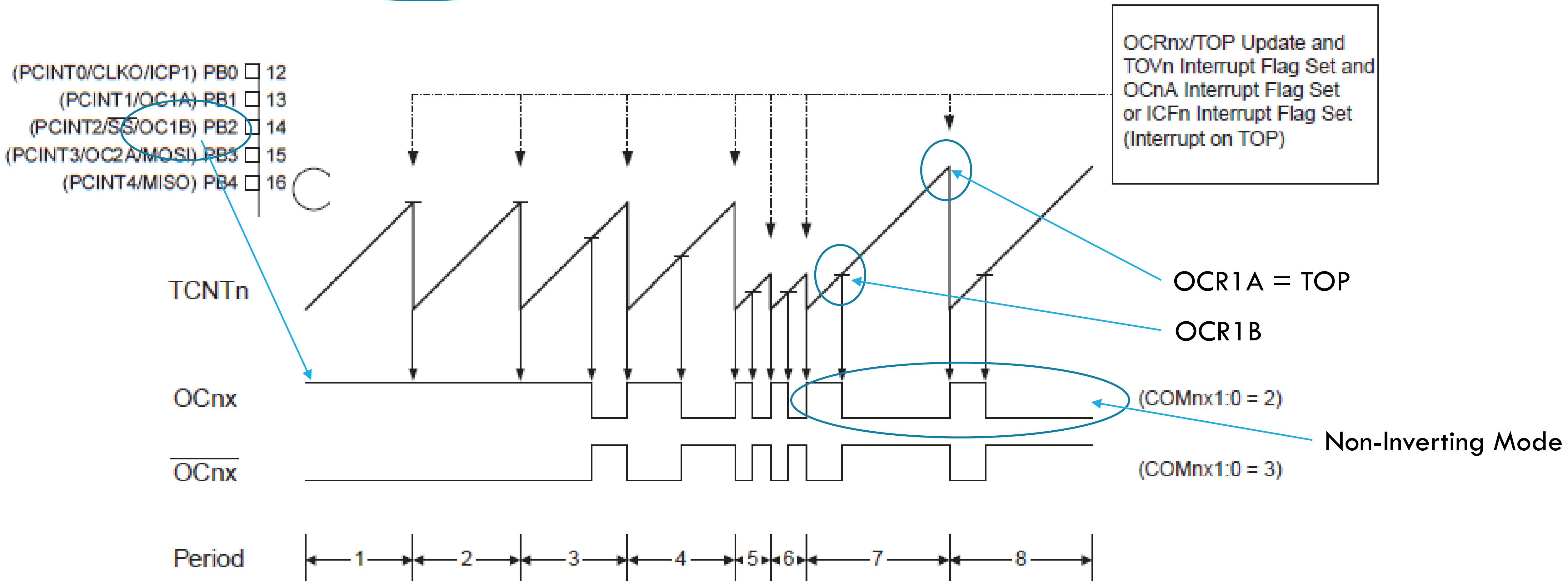
Review Session

Marten van Dijk, Syed Kamran Haider
Department of Electrical & Computer Engineering
University of Connecticut
Email: {vandijk, syed.haider}@engr.uconn.edu

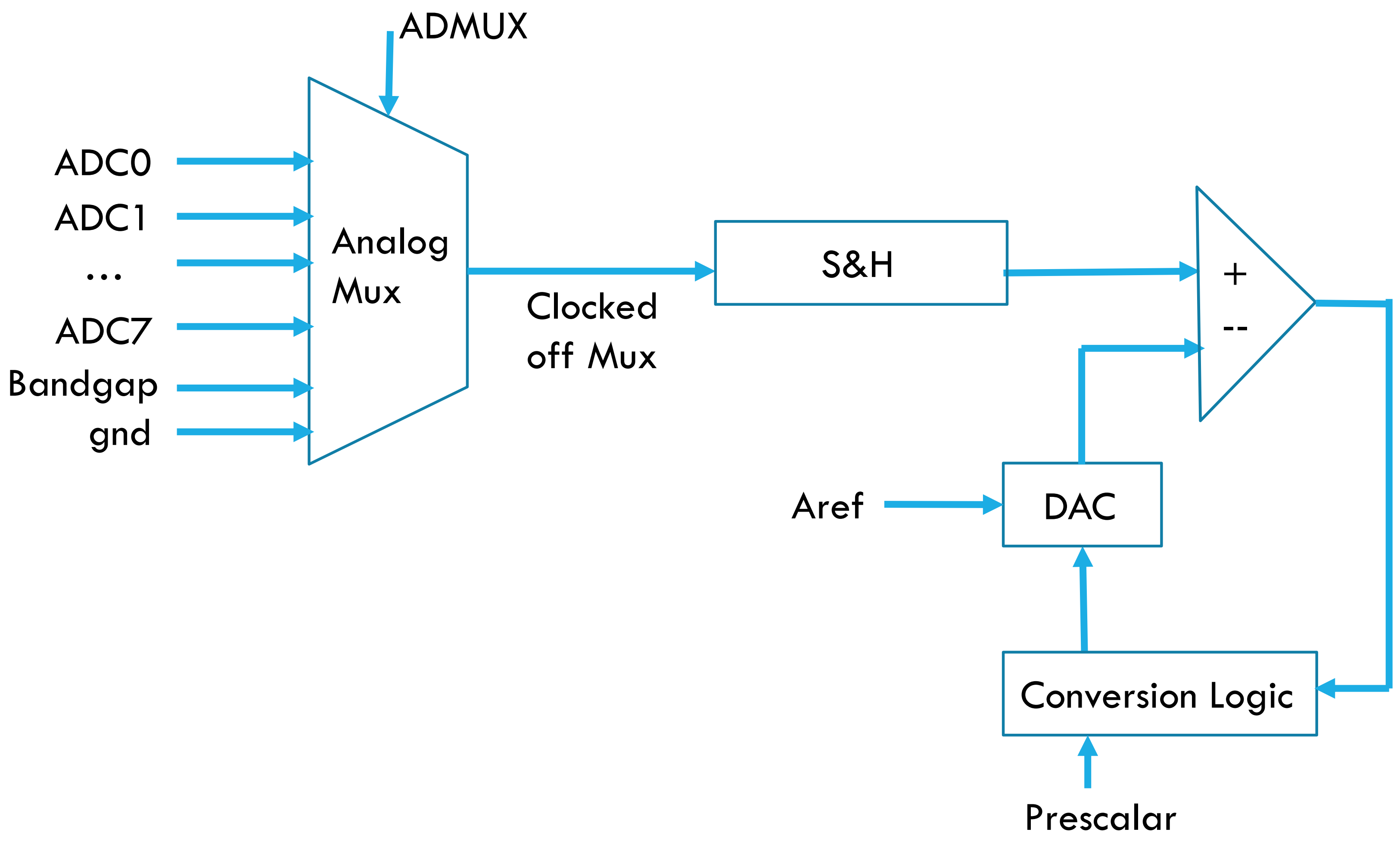
UCONN

Pulse Width Modulation using Timer 1

Figure 15-7. Fast PWM Mode, Timing Diagram



ADC


ADMUX
ADC0
ADC1
…
ADC7
Bandgap
gnd
Analog Mux
Clocked off Mux
S&H
+
--
Aref
DAC
Conversion Logic
Prescalar

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");
```

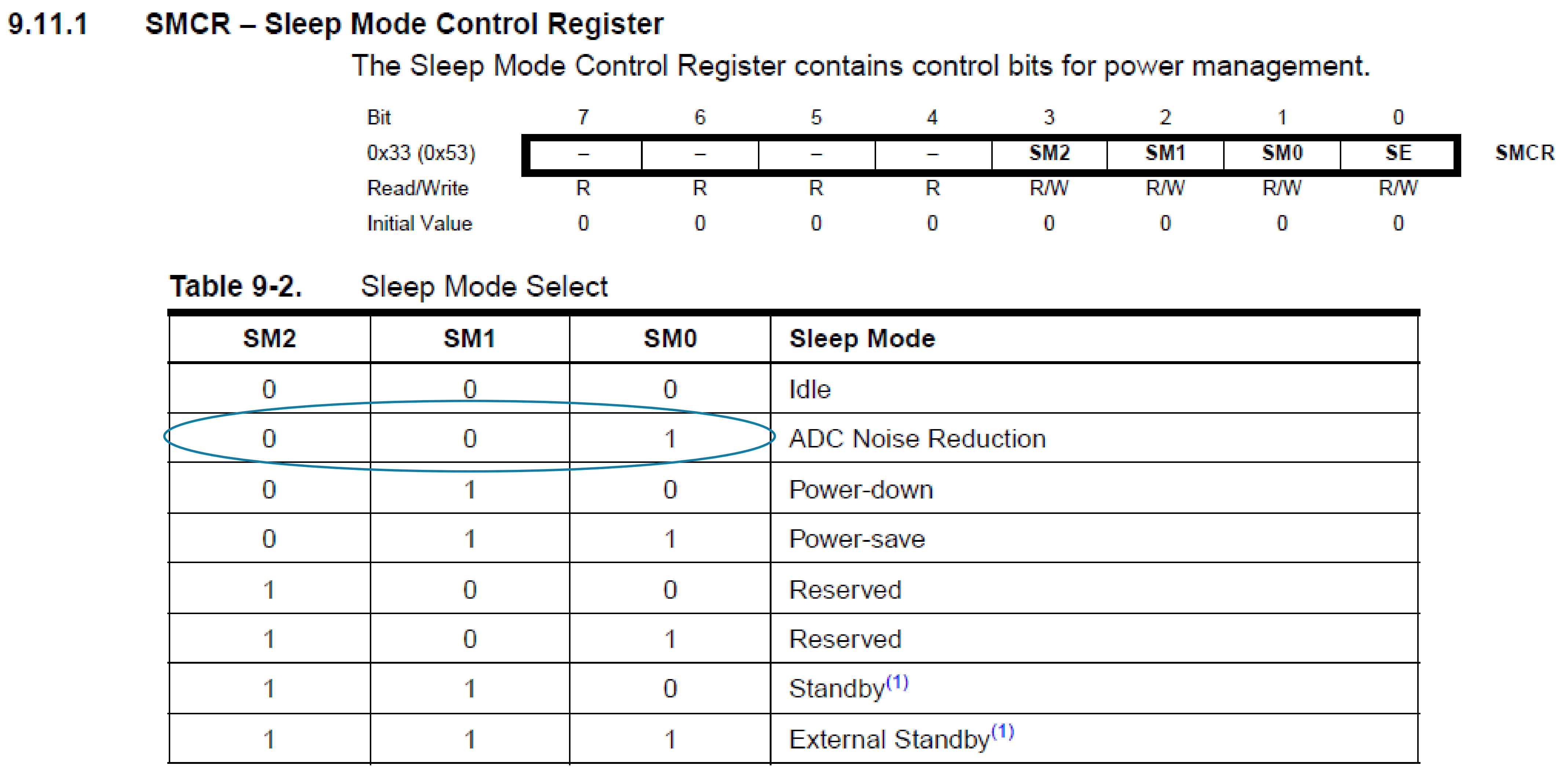
Takes more than 1ms, hence conversion will finish which takes 104us

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

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]

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.
}
```

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) { ..... }
}
```

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)

Scheduling Policies

Static Scheduling Schemes

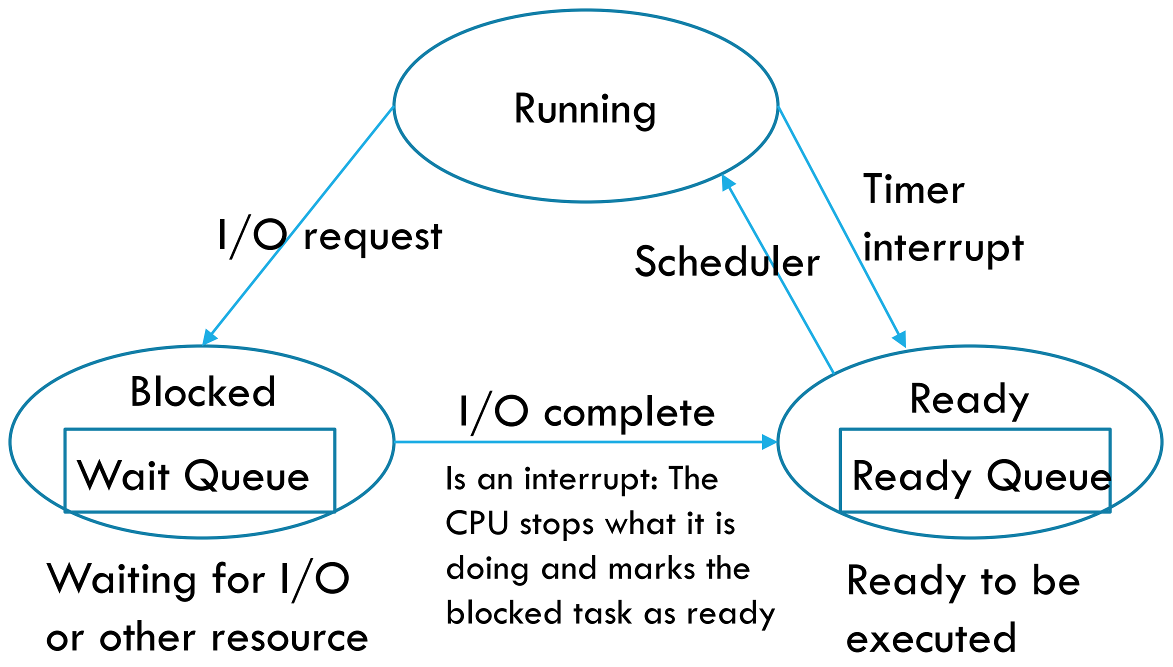
- Round-robin scheduling
- Rate-monotonic scheduling
- Deadline-monotonic scheduling
- Shortest Remaining Time First

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

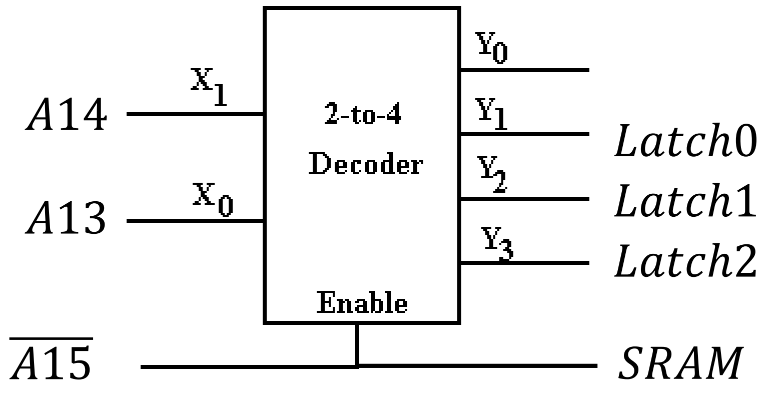


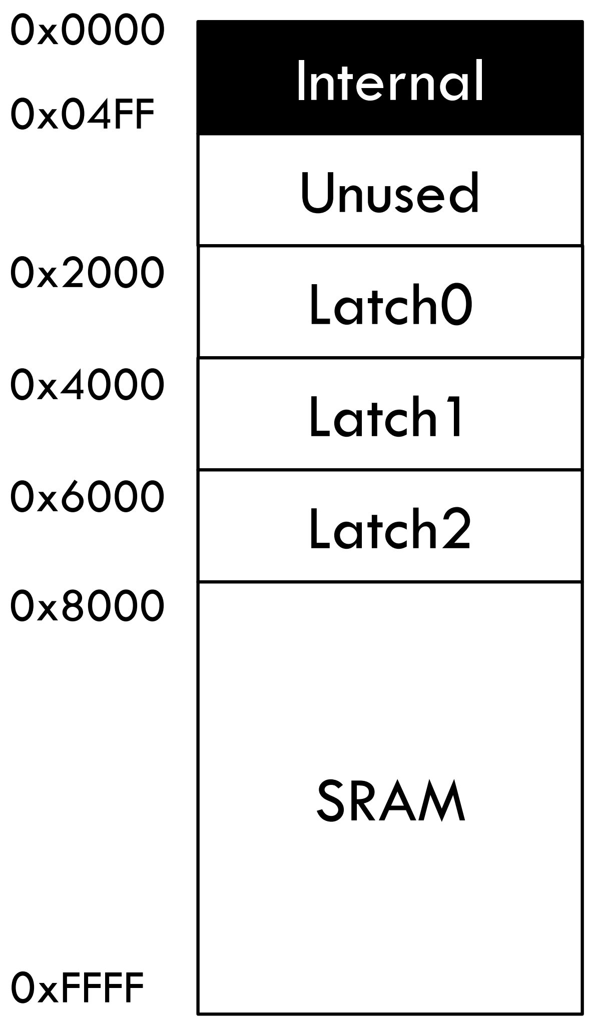
Running
I/O request
Scheduler
Timer interrupt
Blocked
Wait Queue
I/O complete
Ready
Ready Queue
Is an interrupt: The CPU stops what it is doing and marks the blocked task as ready
Waiting for I/O or other resource
Ready to be executed

- 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.

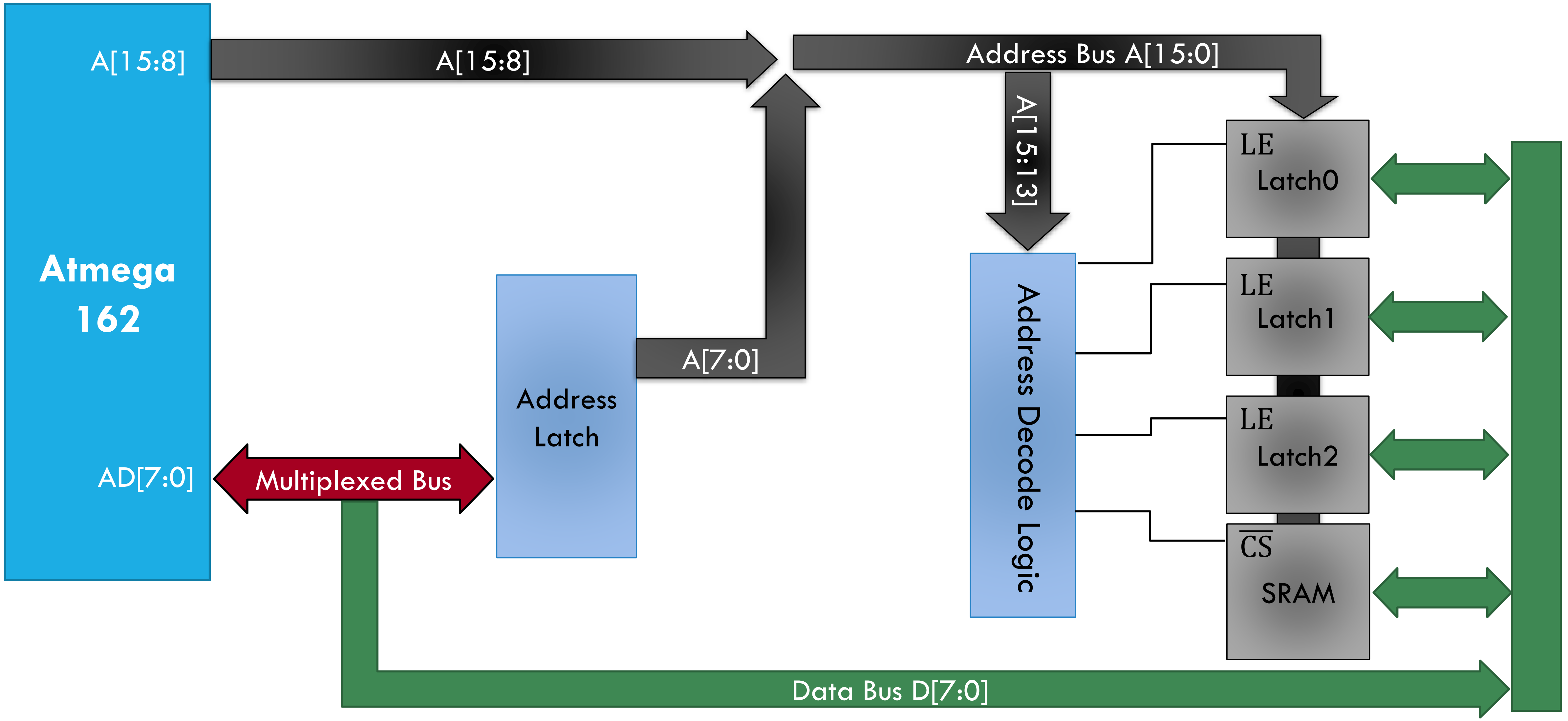
Address Decoding of selected Memory Map

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



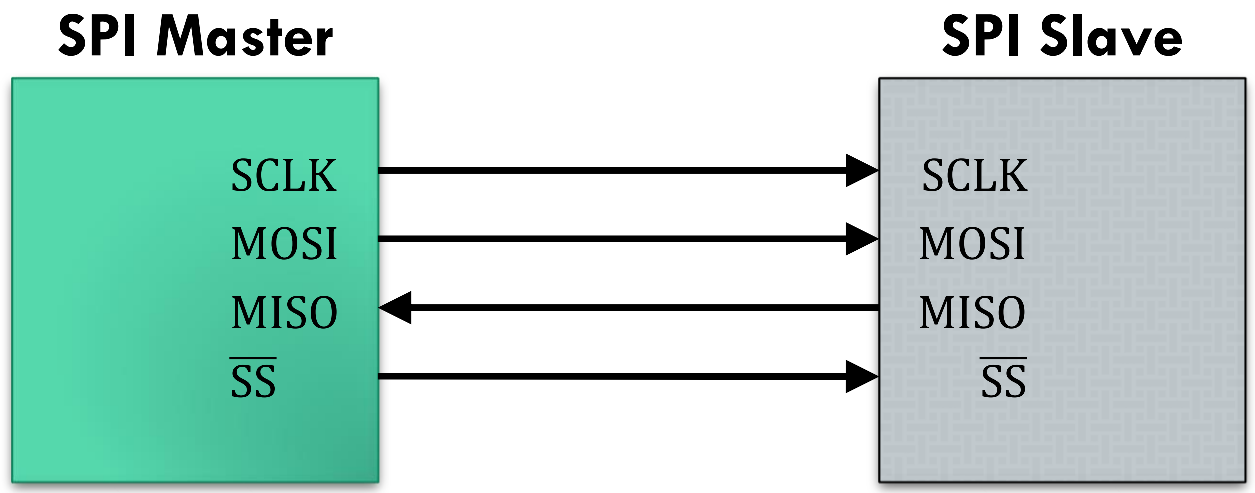


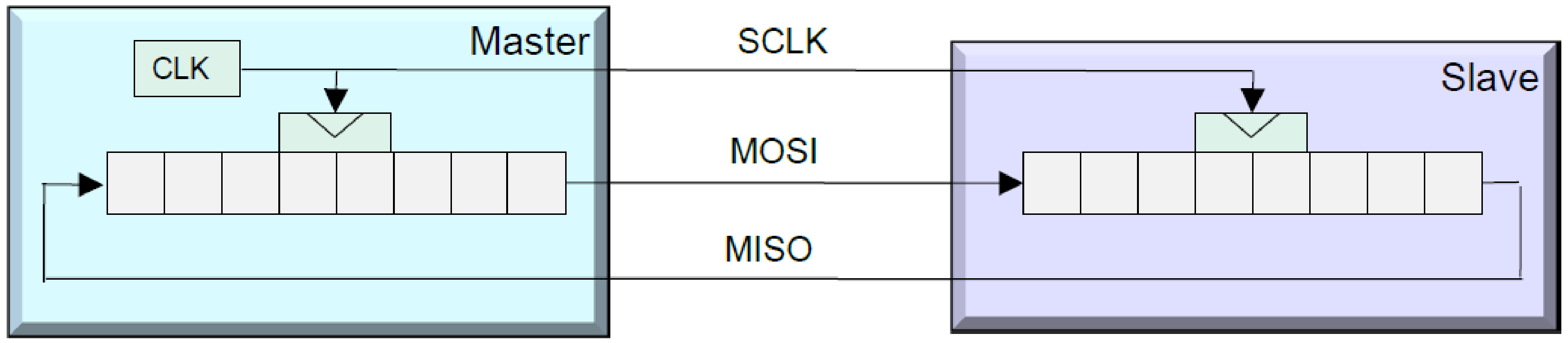
Bus Multiplexing & Address Decoding



SPI: Serial Peripheral Interface

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





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

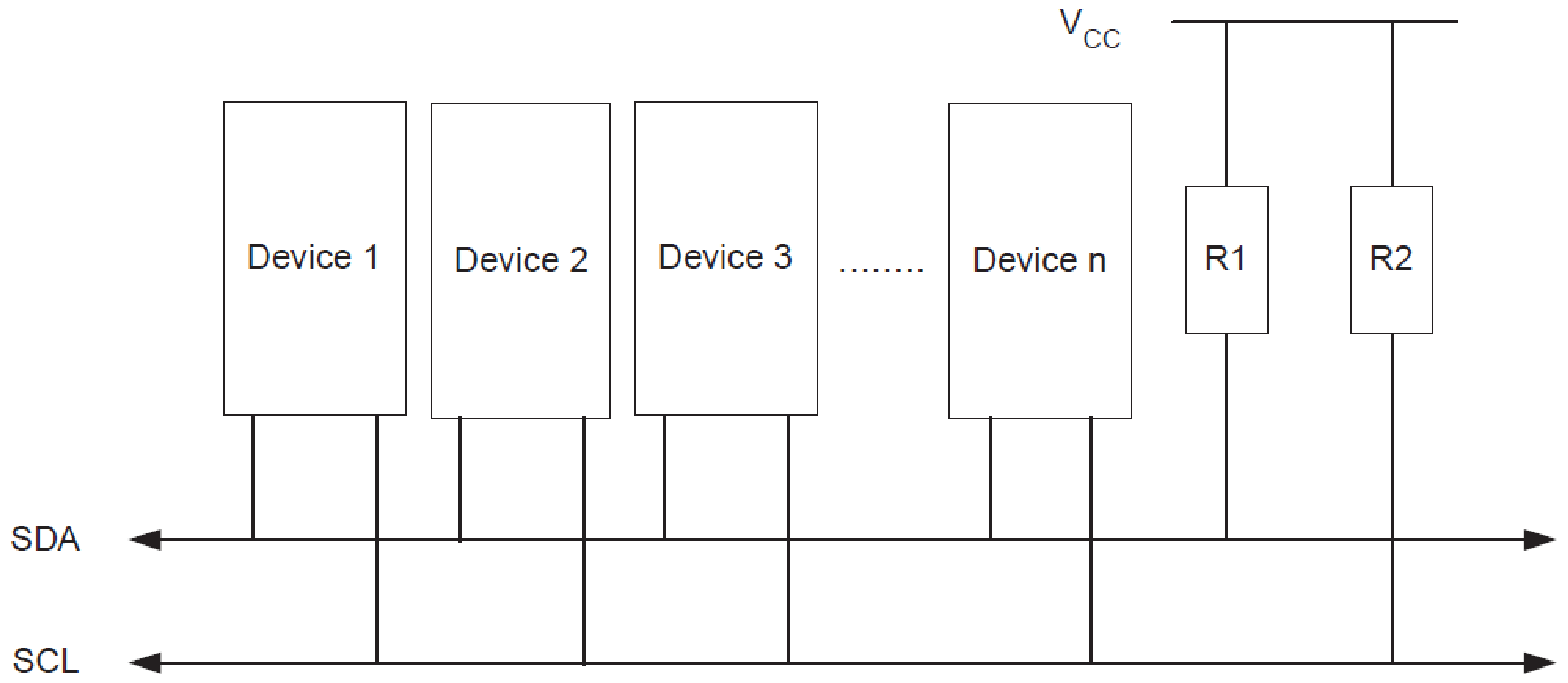
```
void SPI_SlaveInit(void)
{
		/* Set MISO output, all others input */
		DDR_SPI = (1<<DD_MISO);
		/* Enable SPI */
		SPCR = (1<<SPE);
}
```

```
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.

I^2C: Inter Integrated Circuit bus

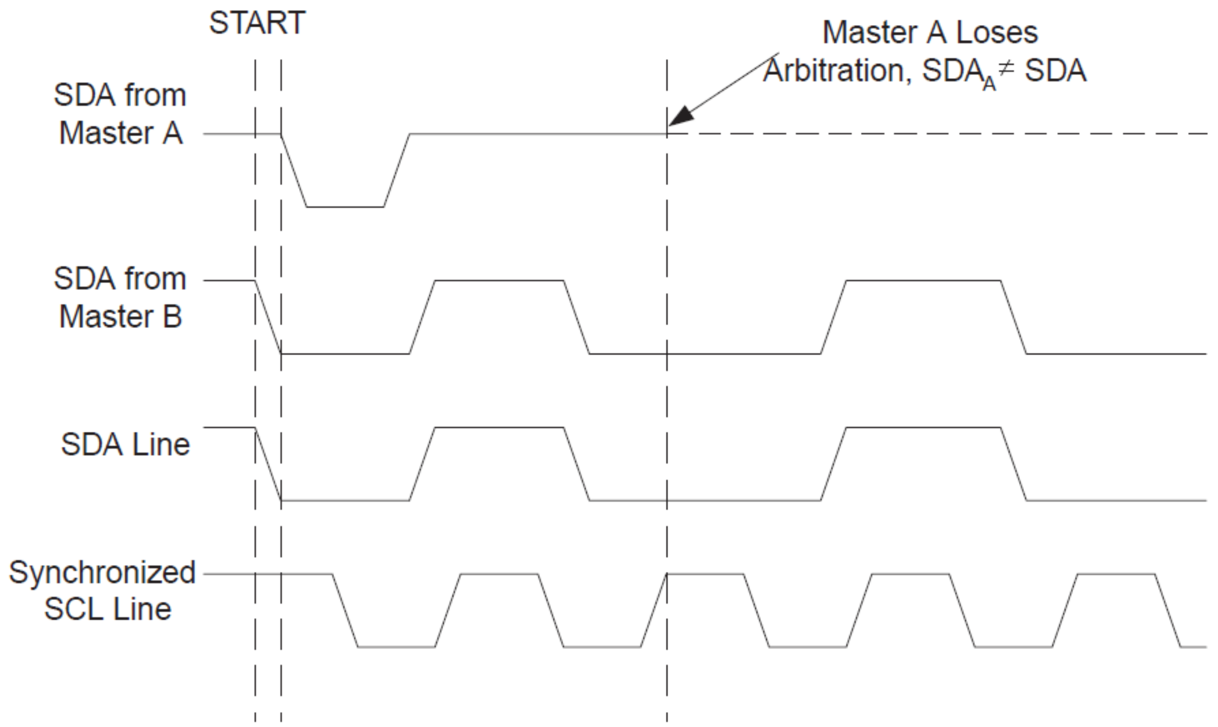
- Also known as Two Wire Interface (TWI)
- Allows up to 128 different devices to be connected using only two bi-directional bus lines, one for clock (SCL) and one for data (SDA).
- All devices connected to the bus have individual addresses.



I^{2}C 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


START
Master A Loses Arbitration, SDA$_A$ ≠ SDA
SDA from Master A
SDA from Master B
SDA Line
Synchronized SCL Line