Objectives

- Review AVR I/O in C
Objectives

- Review AVR I/O in C
- Implement a demo AVR C program on the AVR Butterfly
PORTx and DDRx Review

- Summary of control signals for port pins

<table>
<thead>
<tr>
<th>DDxn</th>
<th>PORTxn</th>
<th>PUD (in MCUCR)</th>
<th>I/O</th>
<th>Pull-up</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>Input</td>
<td>No</td>
<td>Tri-state (Hi-Z)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Input</td>
<td>Yes</td>
<td>Px will source current if ext. pulled low.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Input</td>
<td>No</td>
<td>Tri-state (Hi-Z)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Output</td>
<td>No</td>
<td>Output Low (Sink)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>Output</td>
<td>No</td>
<td>Output High (Source)</td>
</tr>
</tbody>
</table>
Micro-controller Specific Constants/Defines

- All programs will have a line of code to include various utility functions, as well as various value definitions and processor specific definitions

```c
#include <avr/io.h>

or

#include <avr/iom169p.h>
```
Micro-controller Specific Constants/Defines

- All programs will have a line of code to include various utility functions, as well as various value definitions and processor specific definitions

  # include <avr/io.h>

  or

  # include <avr/iom169p.h>

- The files are located at "C:\Program Files (x86)\Atmel\Atmel Toolchain\AVR8 GCC\Native\3.4.1056\avr8-gnu-toolchain\avr\include\avr"
Setting up the Direction bits

▶ To set the direction of all 8 pins of port D, assign a 8-bit value to DDRD

```
DDRD=0xFF;  // set all port D pins as outputs
```

```
DDRD=0x00;  // set all port D pins as inputs
```

```
DDRD=0b10101010;  // alternating pin directions
```

▶ To just set pin 2 of port D to output, not touching the others

```
DDRD=DDRD | 0b00000100;  // recommended!!!!
```

```
DDRD |= 0b00000100;  // recommended!!!!
```
Setting up the Direction bits

- To set the direction of all 8 pins of port D, assign a 8-bit value to DDRD
  
  ```c
  DDRD=0xFF; // set all port D pins as outputs
  DDRD=0x00; // set all port D pins as inputs
  DDRD=0b10101010; // alternating pin directions
  ```

- To just set pin 2 of port D to output, not touching the others
  
  ```c
  DDRD=DDRD | 0b00000100;
  
  Or just
  
  DDRD |= 0b00000100; // recommended!!!!
  ```
Outputting values to PORTx

- Do not set all 8 bits in register PORTD without regard for the directions of each individual pin, i.e. all the bits stored in DDRD

- Don't forget to set direction of pins first!
- Remember if pins are configured as inputs (DDRDn bit is 0) then the corresponding bit in PORTD (PORTDn) sets the pull-up status
Outputting values to PORTx

- Do not set all 8 bits in register PORTD without regard for the directions of each individual pin, i.e. all the bits stored in DDRD
- Use bit operations when possible to suggest use of I/O bit assembly commands and to avoid unintentionally setting extra bit values

Set one pin:
PORTD |= (1 << 3);
same as
PORTD |= (1 << PD3);

Clear one pin:
PORTD &= ~((1 << 3));
same as
PORTD &= ~(1 << PD3);
Outputting values to PORTx

▶ Do not set all 8 bits in register PORTD without regard for the directions of each individual pin, i.e. all the bits stored in DDRD
▶ Use bit operations when possible to suggest use of I/O bit assembly commands and to avoid unintentionally setting extra bit values

Set one pin:
PORTD |= (1 << 3);
    same as
PORTD |= (1 << PD3);
Clear one pin:
PORTD &= ~(1 << 3);
    same as
PORTD &= ~((1 << PD3));
Objectives

Review

Special Functions

Code

Outputting values to PORTx

- Do not set all 8 bits in register PORTD without regard for the directions of each individual pin, i.e. all the bits stored in DDRD

- Use bit operations when possible to suggest use of I/O bit assembly commands and to avoid unintentionally setting extra bit values

Set one pin:
PORTD |= (1 << 3);

same as
PORTD |= (1 << PD3);

Clear one pin:
PORTD &= ~((1 << 3));

same as
PORTD &= ~(1 << PD3);

- Don’t forget to set direction of pins first!
Outputting values to PORTx

- Do not set all 8 bits in register PORTD without regard for the directions of each individual pin, i.e. all the bits stored in DDRD.
- Use bit operations when possible to suggest use of I/O bit assembly commands and to avoid unintentionally setting extra bit values.

Set one pin:
PORTD |= (1 << 3);

Same as
PORTD |= (1 << PD3);

Clear one pin:
PORTD &= ~(1 << 3);

Same as
PORTD &= ~(1 << PD3);

- Don’t forget to set direction of pins first!
- Remember if pins are configured as inputs (DDRDn bit is 0) then the corresponding bit in PORTD (PORTDn) sets the pull-up status.
Setting multiple bits

- Let's say we need 0,2,4,6 pins to be as input and 1,3,5,7 as output

```c
DDRD = (1 << 1) | (1 << 3) | (1 << 5) | (1 << 7); // set all port D pins as outputs
```

Same as

```c
DDRD = (1 << PD7) | (0 << PD6) | (1 << PD5) | (0 << PD4) | (1 << PD3) | (0 << PD2) | (1 << PD1) | (0 << PD0);
```

// alternating pin directions
Setting multiple bits

- Let's say we need 0, 2, 4, 6 pins to be as input and 1, 3, 5, 7 as output

\[
\text{DDRD} = (1 << 1) | (1 << 3) | (1 << 5) | (1 << 7); \quad \text{// set all port D pins as outputs}
\]

Same as

\[
\text{DDRD} = (1 << 7) | (0 << 6) | (1 << 5) | (0 << 4) | (1 << 3) | (0 << 2) | (1 << 1) | (0 << 0);
\quad \text{// alternating pin directions}
\]

- PD7 is defined as 7 in the device include file. USING PD7 instead of 7 is arguably more self-documenting:

\[
\text{DDRD} = (1 << \text{PD7}) | (0 << \text{PD6}) | (1 << \text{PD5}) | (0 << \text{PD4}) | (1 << \text{PD3}) |
\quad (0 << \text{PD2}) | (1 << \text{PD1}) | (0 << \text{PD0});
\]

So we can output values to 1, 3, 5 and 7 pins

\[
\text{PORTD |=(1 << 1) | (1 << 3) | (1 << 5) | (1 << 7);}
\]

Or clear them

\[
\text{PORTD &}= \sim((1 << 1) | (1 << 3) | (1 << 5) | (1 << 7));
\]
Checking multiple bits

```c
flag = PIND & (0b00000001 | 0b01000000);

if (flag){
    // do something when flag is non-zero
}
```

- The following modification changes nothing but expresses intent more explicitly

```c
if (flag!=0){
    // do something when flag is non-zero
}
```
Special Functions

- You may also use the \_BV(x) macro defined in avr/sfr_defs.h which is included through avr/io.h as \# define \_BV(x) (1<<x)

```c
#include "avr/io.h"

int main(void) {
    DDRD &=~_BV(0); //set PORTD pin0 to zero as input
    PORTD |=_BV(0); //Enable pull up;
    DDRD |=_BV(1); //set PORTD pin1 to one as output
    PORTD |=_BV(1); //led ON
    while(1) {
        if (bit_is_clear(PIND, 0)){
            //if button is pressed
            while(1) {
                PORTD &=~_BV(1); //led OFF
                //LED OFF while Button is pressed
                loop_until_bit_is_set(PIND, 0);
                PORTD |=_BV(1); //led ON
            }
        }
    }
}
```
Using predefined bits

UCSR0B = _BV(TXEN0)|_BV(RXEN0); //enable RX and TX
Using predefined bits

UCSR0B = _BV(TXEN0)|_BV(RXEN0); //enable RX and and TX

Both RXEN0 and TXEN0 is predefined in iom169p.h
Software Delay Functions

- _delay_ms(double _ms)
Software Delay Functions

- `_delay_ms(double _ms)`
  - Requires `# include <util/delay.h>`

- F_CPU preprocessor symbol should be defined as MCPU frequency in Hz using `# define` or passed through the `-D` compiler option
  - In code: `# define F_CPU 8000000UL // 8 MHz`
  - Command line option: `-D F_CPU=8000000UL`

- Max delay is $4294967.295 \times 10^6$ ms (ex: 536871 ms for a 8MHz clock)

- Assumes the avr-gcc toolchain being used has `__builtin_avr_delay_cycles(unsigned long)` support

- Otherwise max delay is less and reduced precision is used for long delays (see documentation)

- Use multiple delay commands if needed

- Conversion of delay to clock cycles will be rounded up to the next integer to ensure at least the specified delay

- Alternatively, user can define `__DELAY_ROUND_DOWN__` and `__DELAY_ROUND_CLOSEST__` to round down and round to closest integer
Software Delay Functions

- \_delay\_ms(double \_ms)
  - Requires # include <util/delay.h >
  - F\_CPU preprocessor symbol should be defined as MCPU frequency in Hz using
    # define or passed through the -D compiler option

\* \*10^6 / F\_CPU ms (ex: 536871 ms for a 8MHz clock)

assumes the avr-gcc toolchain being used has __builtin\_avr\_delay\_cycles (unsigned long) support

Otherwise max delay is less and reduced precision is used for long delays (see documentation)

Use multiple delay commands if needed

Conversion of delay to clock cycles will be rounded up to the next integer to
ensure at least the specified delay

Alternatively, user can define __DELAY\_ROUND\_DOWN__ and
__DELAY\_ROUND\_CLOSEST__ to round down and round to closest integer
Software Delay Functions

- `_delay_ms(double _ms)`
  - Requires `# include <util/delay.h>`
  - `F_CPU` preprocessor symbol should be defined as MCPU frequency in Hz using `# define` or passed through the `-D` compiler option
    - In code: `# define F_CPU 8000000UL // 8 MHz`
Software Delay Functions

- _delay_ms(double _ms)
  - Requires # include <util/delay.h>
  - F_CPU preprocessor symbol should be defined as MCPU frequency in Hz using
    # define or passed through the -D compiler option
    - In code: # define F_CPU 8000000UL // 8 MHz
    - Command line option: -D F_CPU=8000000UL
Software Delay Functions

- `_delay_ms(double _ms)`
  - Requires `# include <util/delay.h>`
  - F_CPU preprocessor symbol should be defined as MCPU frequency in Hz using `# define` or passed through the `-D` compiler option
    - In code: `# define F_CPU 8000000UL // 8 MHz`
    - Command line option: `-D F_CPU=8000000UL`
  - Max delay is $4294967.295 \times 10^6 / \text{F\_CPU ms}$ (ex: 536871 ms for a 8MHz clock)
Objectives

Review

Special Functions

Code

Software Delay Functions

- `_delay_ms(double _ms)`
  - Requires `# include <util/delay.h>`
  - F_CPU preprocessor symbol should be defined as MCU frequency in Hz using `# define` or passed through the `-D` compiler option
    - In code: `# define F_CPU 8000000UL // 8 MHz`
    - Command line option: `-D F_CPU=8000000UL`
  - Max delay is $4294967.295 \times 10^6 / \text{F}_{\text{CPU}}$ ms (ex: 536871 ms for a 8MHz clock)
    - assumes the avr-gcc toolchain being used has `__builtin_avr_delay_cycles` (unsigned long) support
Software Delay Functions

- `_delay_ms(double _ms)`
  - Requires # include `<util/delay.h>`
  - F_CPU preprocessor symbol should be defined as MCPU frequency in Hz using # define or passed through the -D compiler option
    - In code: # define F_CPU 8000000UL // 8 MHz
    - Command line option: -D F_CPU=8000000UL
  - Max delay is $4294967.295 \times 10^6 / \text{F}_\text{CPU} \text{ ms}$ (ex: 536871 ms for a 8MHz clock)
    - assumes the avr-gcc toolchain being used has `__builtin_avr_delay_cycles` (unsigned long) support
    - Otherwise max delay is less and reduced precision is used for long delays (see documentation)
Software Delay Functions

- `_delay_ms(double _ms)`
  - Requires `#include <util/delay.h>`
  - F_CPU preprocessor symbol should be defined as MCPU frequency in Hz using
    # define or passed through the -D compiler option
    - In code: `# define F_CPU 8000000UL // 8 MHz`
    - Command line option: `-D F_CPU=8000000UL`
  - Max delay is \[ 4294967.295 \times 10^6 / F_{\text{CPU}} \text{ ms} \] (ex: 536871 ms for a 8MHz clock)
    - Assumes the avr-gcc toolchain being used has `__builtin_avr_delay_cycles`
      (unsigned long) support
    - Otherwise max delay is less and reduced precision is used for long delays
      (see documentation)
  - Use multiple delay commands if needed
Software Delay Functions

- _delay_ms(double _ms)
  - Requires # include <util/delay.h>
  - F_CPU preprocessor symbol should be defined as MCPU frequency in Hz using # define or passed through the -D compiler option
    - In code: # define F_CPU 8000000UL // 8 MHz
    - Command line option: -D F_CPU=8000000UL
  - Max delay is $4294967.295 \times 10^6 / F_{CPU}$ ms (ex: 536871 ms for a 8MHz clock)
    - assumes the avr-gcc toolchain being used has __builtin_avr_delay_cycles (unsigned long) support
    - Otherwise max delay is less and reduced precision is used for long delays (see documentation)
    - Use multiple delay commands if needed
  - Conversion of delay to clock cycles will be rounded up to the next integer to ensure at least the specified delay
Software Delay Functions

- `_delay_ms(double _ms)`
  - Requires `# include <util/delay.h>`
  - F_CPU preprocessor symbol should be defined as MCPU frequency in Hz using `# define` or passed through the `-D` compiler option
    - In code: `# define F_CPU 8000000UL // 8 MHz`
    - Command line option: `-D F_CPU=8000000UL`
  - Max delay is $4294967.295 \times 10^6 / F_CPU$ ms (ex: 536871 ms for a 8MHz clock)
    - assumes the avr-gcc toolchain being used has `__builtin_avr_delay_cycles` (unsigned long) support
    - Otherwise max delay is less and reduced precision is used for long delays (see documentation)
    - Use multiple delay commands if needed
  - Conversion of delay to clock cycles will be rounded up to the next integer to ensure at least the specified delay
    - Alternatively, user can define `__DELAY_ROUND_DOWN__` and `__DELAY_ROUND_CLOSEST__` to round down and round to closest integer
Software Delay Functions

- `_delay_us(double _us)"
Software Delay Functions

- `_delay_us(double _us)`
  - Same as before but max delay is 1000 times less: $4294967.295 \times 10^6 / F_{CPU}$ us (ex: 536871 us for a 8MHz clock)
AVR C Code

Download code from instructor website (c_example.c)