

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

میکرو کنترلرهای AVR برنامه نویسی به زبان C

دانشکده برق و رباتیک
دانشگاه صنعتی شاهرود

حسین خسروی

۱۳۹۰-۹۱

Why C?

- C is a high-level programming language: C code is easier to understand compared to other languages.
- C supports low-level programming: We can access every hardware components of the microcontroller with C.
- C has standard libraries for complex tasks: data type conversions, standard input/output, long-integer arithmetic.
- The Atmel AVR instruction set has been designed to support C compilers: C code can be converted efficiently to assembly code.

C Tools

- **Atmel AVR Studio**
- **An integrated development environment for Atmel AVR microcontroller:**
 - ❑ **Editor, assembler, emulator, HEX file downloader and C compiler (from version 5.0)**
 - ❑ **Available from Atmel website:**
http://www.atmel.com/tools/ATMELAVRSTUDIO5_0.aspx
 - ❑ **Its Free**

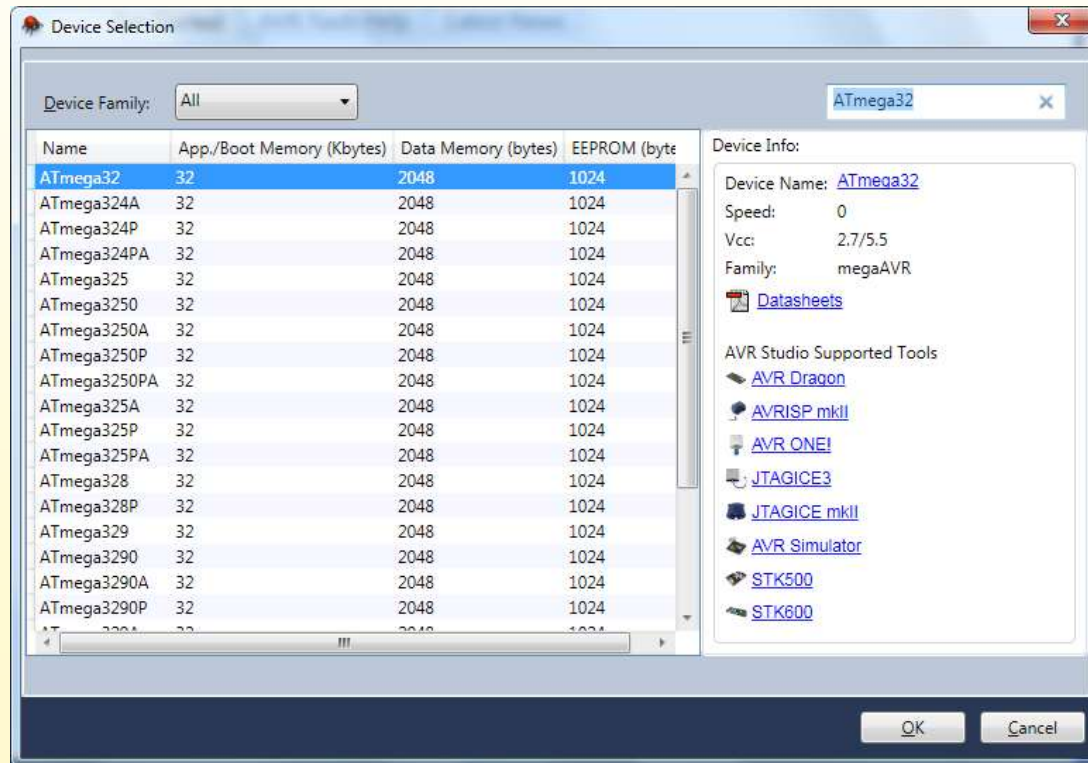


- **Code Vision AVR 2.05.3**
 - ❑ Simple and Great wizards
 - ❑ Good libraries
 - ❑ Easier than AVR Studio
 - ❑ **Lack of Debugger**
 - ❑ **Commercial (Not for Us!)**



Development Cycle for C in AVR Studio – Step 1

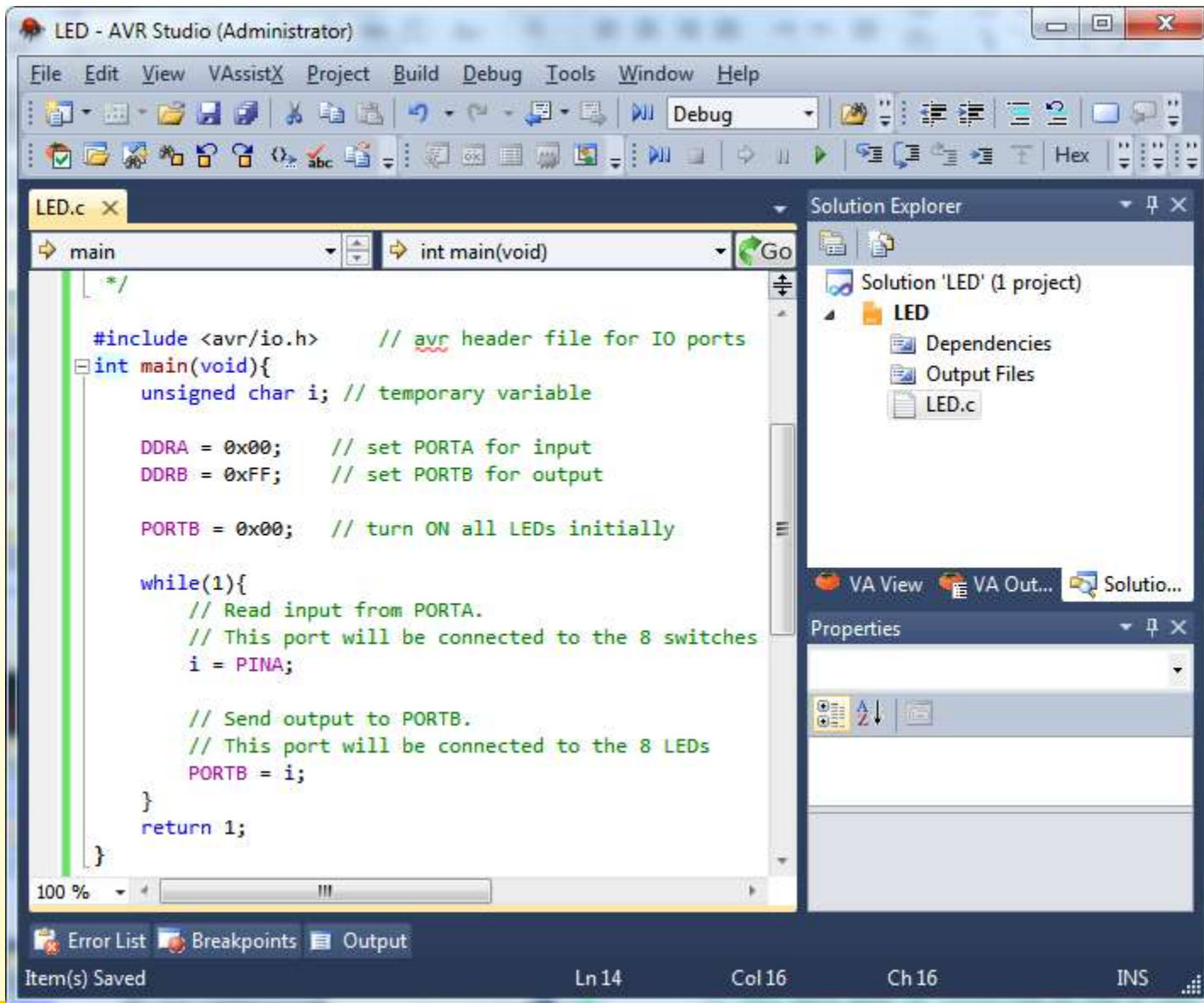
- New Project → AVR GCC → C Executable Project
- Enter a **meaningful** name for the project (not the first characters you find in keyboard like **asdf!**)
- Select destination AVR e.g. ATmega32 → OK



Development Cycle for C in AVR Studio

- **Step 2:** Enter a C program
- **Step 3:** Compile the C program to produce Hex file
 - ❑ Correct possible syntax errors
- **Step 4 (Optional):** Simulate on Proteus
- **Step 5:** Download and test the HEX file on Atmel AVR microcontroller

Sample Program – Toggle LED using switches



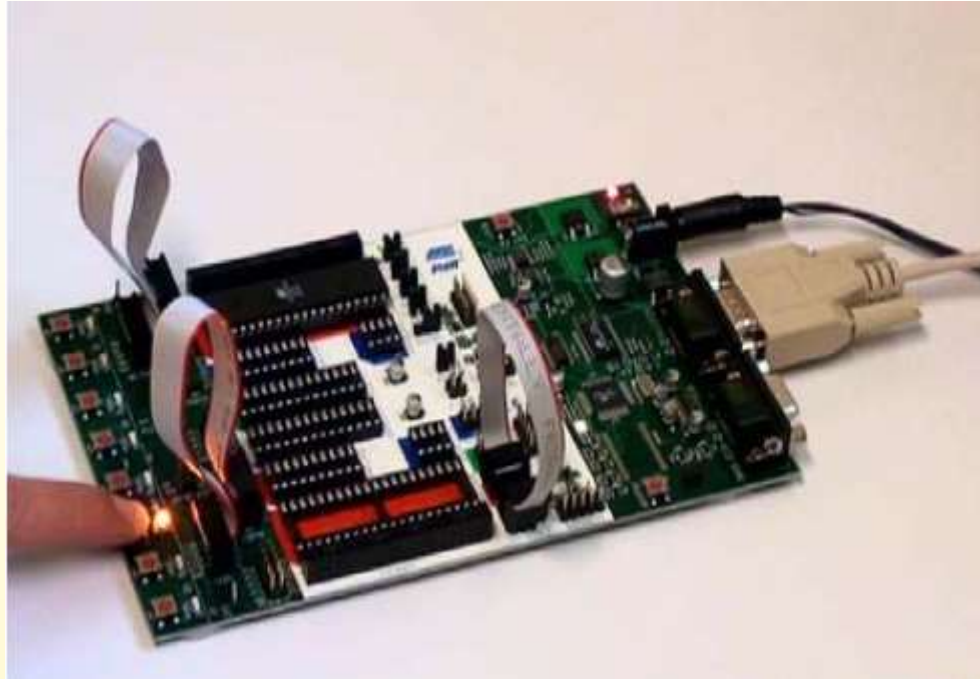
Sample Program

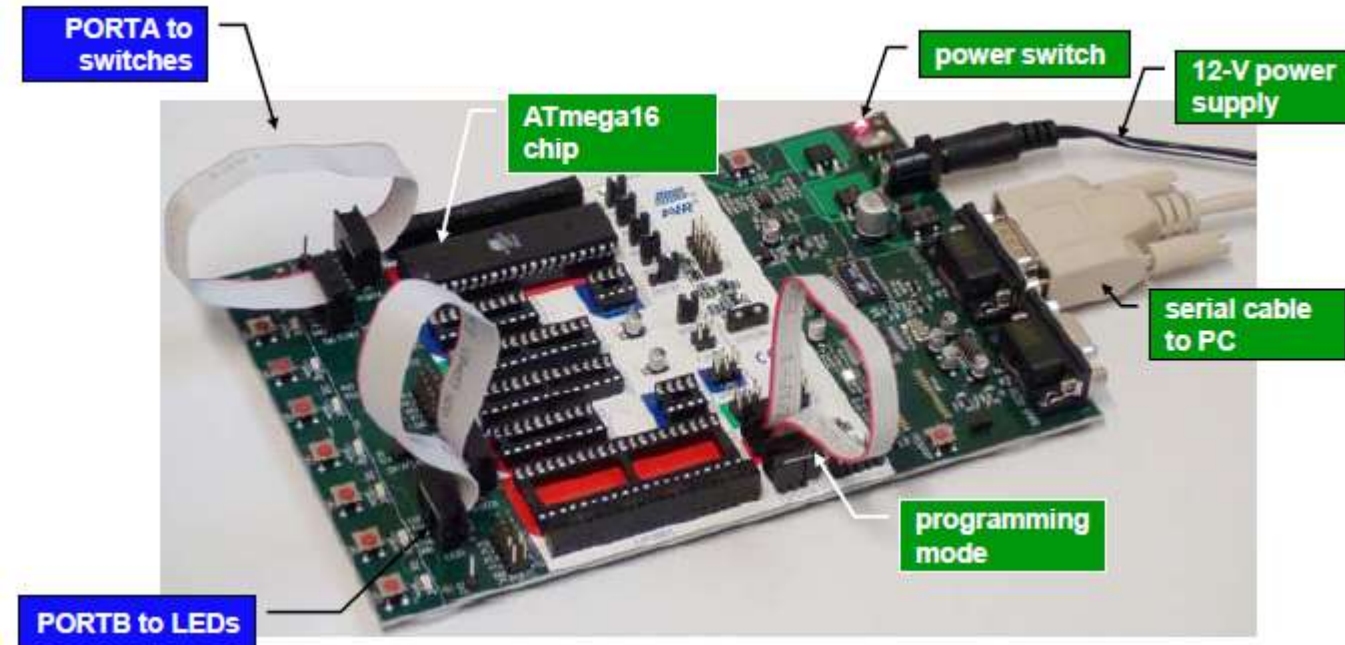
Toggle 8 LEDs using 8 switches

```
#include <avr/io.h>      // avr header file for I/O ports
int main(void){
    unsigned char i; // temporary variable
    DDRA = 0x00;      // set PORTA for input (8 switches)
    DDRB = 0xFF;      // set PORTB for output (8 LEDs)
    PORTB = 0x00;     // turn ON all LEDs initially

    while(1){
        // Read input from PORTA (switches).
        i = PINA;
        // Send output to PORTB (LEDs).
        PORTB = i;
    }
    return 1;
}
```


Result on STK500





Hardware setup for LED sample program.
Connections to PORTA & PORTB are only for this example.

Digital IO in ATmega32

- ATmega32 has four 8-bit digital IO ports:
 - ❑ PORT A,
 - ❑ PORT B,
 - ❑ PORT C, and
 - ❑ PORT D.

- Each port has 8 data pins.
- Every port is **bi-directional**.
- Each of the 8 pins can be individually configured as
 - ❑ input (receiving data into microcontroller), or
 - ❑ output (sending data from microcontroller).

PORT names

- To access registers
 - ❑ PORTx to access port register
 - ❑ PINx to access port input register
 - ❑ DDRx to access data direction registers
 - ❑ x could be A,B,C or D

- To access special bit of registers
 - ❑ There is no standard method
 - ❑ Each compiler has its own syntax
 - ❑ Use bitwise operators which works everywhere
 - ❑ To make pin5 as output and left others unchanged:
 - ❑ `DDRC = DDRC | 0b0010000`

I/O Ports

➤ 3 I/O registers per port, bitwise configuration

□ **DDRx: Data Direction Register (1: out, 0: in)**

- Every pin can be configured as input or output

□ **PORTx:**

- **DDR = 0xFF → output data**
- **DDR = 0x00 → floating or pullup resistor**
 - If PORTxn is written logic one when the pin is configured as an input pin, the pull-up resistor is activated

□ **PINx: Port Input (Read Only)**

- **DDR=0xFF → PORTx (with 1 clk latency)**
- **DDR=0x00 → input data**

Examples

- To set Port A pins 0 to 3 for input, pins 4 to 7 for output, we write C code:

```
DDRA = 0b11110000; // configure pins
```

- To write a binary 0 to output pin 6, binary 1 to other pins of Port A, we write C code:

```
PORTA = 0b10111111; // write output
```

- To read the input pins of Port A, we write C code:

```
unsigned char temp; // temporary variable  
temp = PINA;       // read input
```

SFR Definition

➤ Where do the C names PINA, PORTA, DDRA come from?

➤ **<avr/iom32.h>**

□ .\AVR Studio 5.0\AVR ToolChain\avr\include\avr\iom32.h

```
/* Port D */
```

```
#define PIND      _SFR_I08(0x10)
```

```
#define DDRD      _SFR_I08(0x11)
```

```
#define PORTD     _SFR_I08(0x12)
```

➤ Where does iom32.h come from?

➤ **<avr/io.h>**

```
#if defined (__AVR_ATmega32__)
```

```
#include <avr/iom32.h>
```

AVR header file

- To access all AVR microcontroller registers, your program must include the header file `<io.h>`
 - ❑ `#include <avr/io.h>`
- Depending on which device selected in your project, file 'io.h' will automatically redirect to a specific header file.
- **Example**
 - ❑ For ATmega32, the specific header file is `<avr/iom32.h>`
 - The header file lists the C names for all registers in ATmega32, and their memory locations.
 - We always use the C names in our code.

I/O Ports

```
DDRC = 0xFF;
PORTC = 0x12;
PINC = 0x22;
```

I/O	PORTC		
I/O	PORTD		
+	SPI		
+	TIMER_COUNTER_0		
+	TIMER_COUNTER_1		
+	TIMER_COUNTER_2		
Name	Address	Value	Bits
I/O	PINC	0x33	0x12

```
DDRC
PORTC
PINC
```

```
DDRC = 0xFF;
PORTC = 0x12;
PINC = 0x22;
```

PINC is **read only**
So this is useless

```
PINC
```

```
DDRC = 0xFF;
PORTC = 0x12;
PINC = 0x22;
```

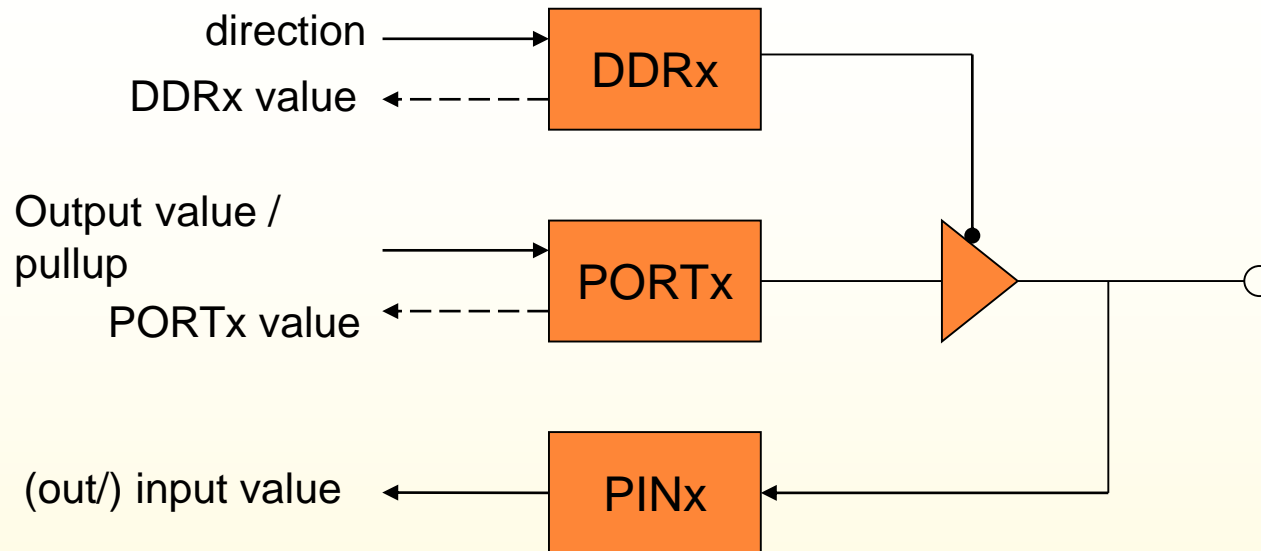
```
DDRC = 0x00;
PORTC = 0x73;
PINC = 0x22;
```

I/O	PORTD		
+	SPI		
+	TIMER_COUNTER_0		
+	TIMER_COUNTER_1		
+	TIMER_COUNTER_2		
Name	Address	Value	Bits
I/O	PINC	0x33	0x12
I/O	DDRC	0x34	0xFF
I/O	PORTC	0x35	0x12

I/O	PORTD		
+	SPI		
+	TIMER_COUNTER_0		
+	TIMER_COUNTER_1		
+	TIMER_COUNTER_2		
Name	Address	Value	Bits
I/O	PINC	0x33	0x12
I/O	DDRC	0x34	0x00
I/O	PORTC	0x35	0x73

PORTC is input so PINC is not related to PORTC

I/O ports (simple view)



DD _{xn}	PORT _{xn}	PUD (in SFIOR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	P _{xn} will source current if ext. pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output Low (Sink)
1	1	X	Output	No	Output High (Source)

General I/O Port Configuration

PUD: Pull-Up Disable

WDx: Write DDRx

RDx: Read DDRx

WPx: **W**rite **P**ORTx

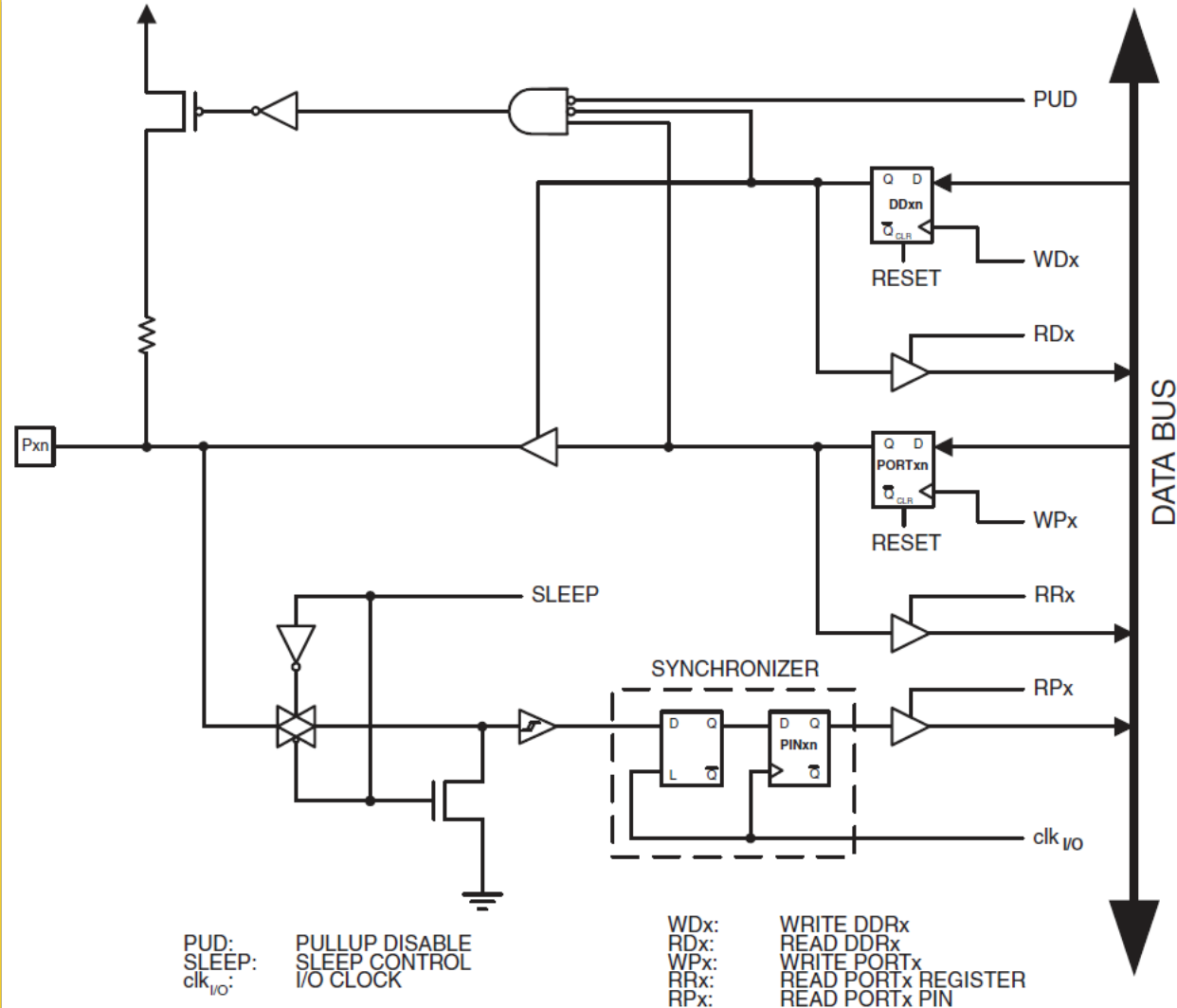
RRx: Read PORTx Register

RPx: Read PINx

Note:

WPx, WDx, RRx, RPx, and RDx are common to all pins within the same port. clk_{I/O}, SLEEP, and PUD are common to all ports.

These are internal to AVR and we don't have access to them and never use them except PUD (SFIOR D2)



Review of C Programming

- Most students in this class learnt C programming in their first year.
- **Is it learned or passed?**
- Here, we review briefly major aspects of the C programming language.
 - ❑ Structure of a C program
 - ❑ Data types and operators
 - ❑ Flow control in C
 - ❑ C functions
- In all lectures, C code examples will be used extensively.

Structure of a C program

- A C program typically has two main sections.
- **include** section: to insert header files.
- **main()** section: code that runs when the program starts.
- In the example below, **<avr/io.h>** is a header file that contains all register definitions for the AVR microcontroller.

```
#include <avr/io.h> // avr header file for IO ports
int main(void){
    unsigned char i; // temporary variable
    DDRA = 0x00;      // set PORTA for input
    DDRB = 0xFF;      // set PORTB for output
    PORTB = 0x00;      // turn ON all LEDs initially
    while(1){
        // Read input from PORTA (switches).
        i = PINA;
        // Send output to PORTB (LEDS).
        PORTB = i;
    }
    return 1;
}
```

C Comments

- Comments are text that the compiler ignores.
- For a single-line comment, use double back slashes

```
DDRA = 0x00; // set PORTA for input
```

- For a multi-line comment, use the pair `/*` and `*/`

```
/* File: led.c
```

```
Description: Simple C program for the ATMEL  
AVR(ATmega32 chip)
```

```
It lets user turn on LEDs by pressing the switches  
on the STK500 board
```

```
Author: Hossein Khosravi (1390/12/11)
```

```
*/
```

- Always use comments to make program easy to understand.

C statements and blocks

➤ C Statements

- ❑ C statements control the program flow.
- ❑ They consist of keywords, expressions and other statements.
- ❑ A statement ends with **semicolon (;)**.
- ❑ `DDRB = 0xFF; // set PORTB for output`

➤ C Blocks

- ❑ A C block is a group of statements enclosed by braces {}.
- ❑ Usually, a C block is run depending on some logical conditions.

```
while (1){  
    // Read input from PORTA - switches  
    i = PINA;  
    // Send output to PORTB - LEDs  
    PORTB = i;  
}
```

Data types and operators

➤ The main data types in C are

`char`: 8-bit integer

`int`: 16-bit integer

`long int`: 32-bit integer

`float`: 32-bit floating point

➤ The above data types can be modified by keyword `unsigned`

`char a`; // a value range -128, ..., 0, ..., 127

`unsigned char b`; // b value range 0, 1, 2, ..., 255

`unsigned long int c`; // c value range 0,..., $2_{32} - 1$

`unsigned float f`; // illegal. float is always signed

➤ Some examples of variable assignment

`a = 0xA0`; // a stores hexadecimal value of A0

`b = '1'`; // b stores ASCII code of character '1'

`c = 2000ul`; // c stores an unsigned long 2000

C Operators

- **C has a rich set of operators**
 - ❑ **Arithmetic operators**
 - ❑ **Relational operators**
 - ❑ **Logical operators**
 - ❑ **Bit-wise operators**
 - ❑ **Data access operators**
 - ❑ **Miscellaneous operators**

Arithmetic Operators

Operator	Name	Example	Description
*	Multiplication	$x * y$	Multiply x times y
/	Division	x / y	Divide x by y
%	Modulo	$x \% y$	Remainder of x divided by y
+	Addition	$x + y$	Add x and y
-	Subtraction	$x - y$	Subtract y from x
++	Increment	$x++$	Increment x by 1 after using it
		$++x$	Increment x by 1 before using it
--	Decrement	$x--$	Decrement x by 1 after using it
		$--x$	Decrement x by 1 before using it
-	Negation	$-x$	Negate x

Relational operators

Operator	Name	Example	Description
>	Greater than	$x > 5$	1 if x is greater than 5, 0 otherwise
>=	Greater than or equal to	$x >= 5$	1 is x is greater than or equal to 5, 0 otherwise
<	Less than	$x < y$	1 if x is smaller than y, 0 otherwise
<=	Less than or equal to	$x <= y$	1 is x is smaller than or equal to y, 0 otherwise
==	Equal to	$x == y$	1 is x is equal to y, 0 otherwise
!=	Not equal to	$x != 4$	1 is x is not equal to 4, 0 otherwise

Logical operators

- These operate on logical variables/constants.

Operator	Name	Example	Description
!	Logical NOT	!x	1 if x is 0, otherwise 0
&&	Logical AND	x && y	1 if both x and y are 1, otherwise 0
	Logical OR	x y	0 if both x and y are 0, otherwise 1

Bit-wise operators

- These operate on individual bits of a variable/constant.

Operator	Name	Example	Description
<code>~</code>	Bit-wise complement	<code>~x</code>	Toggle every bit from 0 to 1, or 1 to 0
<code>&</code>	Bitwise AND	<code>x & y</code>	Bitwise AND of x and y
<code> </code>	Bitwise OR	<code>x y</code>	Bitwise OR of x and y
<code>^</code>	Bitwise XOR	<code>x ^ y</code>	Bitwise XOR of x and y
<code><<</code>	Shift left	<code>x << 3</code>	Shift bits in x three positions to the left
<code>>></code>	Shift right	<code>x >> 1</code>	Shift bits in x one position to the right

Data-access operators



- These operate on arrays, structures or pointers.
- We'll learn more about these operators later.





Operator	Name	Example	Description
[]	Array element	x[2]	Third element of array x
.	Member selection	x.age	Field 'age' of structure variable x
->	Member selection	p->age	Field 'age' of structure pointer p
*	Indirection	*p	Content of memory location pointed by p
&	Address of	&x	Address of the memory location where variable x is stored

Miscellaneous operators

Operator	Name	Example	Description
()	Function	_delay_ms(250)	Call a function to create delay of 250ms
(type)	Type cast	char x = 3; (int) x	x is 8-bit integer x is converted to 16-bit integer
?	Conditional evaluation	char x; y=(x>5)?10:20;	This is equivalent to if (x > 5) y = 10; else y = 20;

Flow control in C

- By default, C statements are executed sequentially.
 - To change the program flow, there are six types of statements
 - if-else statement
 - switch statement

Conditional
 - while statement
 - for statement
 - do statement
- 
- 
- Iterative
- goto statement
- 
- 
- Should be avoided!

“if-else” statement

➤ General syntax

```
if (expression)
    statement_1;
else
    statement_2;
```

➤ Example code

```
char a, b, sum;
a = 4; b = -5;
sum = a + b;
if (sum < 0)
    printf("sum is negative");
else if (sum > 0)
    printf("sum is positive");
else
    printf("sum is zero");
```

“switch” statement

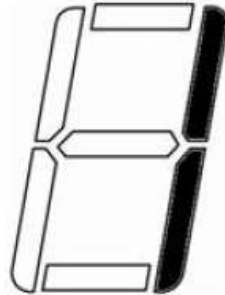
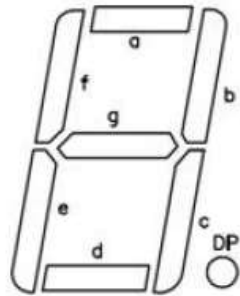
➤ General syntax

```
switch (expression)
{
case constant_1:
    statement_1;
    break;
case constant_2:
    statement_2;
    break;
...
case constant_n:
    statement_n;
    break;
default:
    default_statement;
    break;
}
```

Use **break** to separate different cases.

“switch” statement – Example

Find the bit pattern to display a digit on the 7-segment LED



(a) 7 segments of the LED

Bit number:	7	6	5	4	3	2	1	0
Purpose:	DP	g	f	e	d	c	b	a

- Bit pattern for digit '1': 0 0 0 0 0 1 1 0
- Bit pattern for digit '2': 0 1 0 1 1 0 1 1

“switch” statement – Example

```
unsigned char digit = 2;
unsigned char led_pattern;
switch (digit)
{
    case '0':
        led_pattern = 0b00111111;
        break;
    case '1':
        led_pattern = 0b00000110;
        break;
    case '2':
        led_pattern = 0b01011011;
        break;
    //you can complete more cases here...
    default:
        break;
}
PORTB = led_pattern; // send to PORTB and 7-segment LED
```

“while” statement

➤ General syntax

```
while (expression){  
    statements;  
}
```

➤ Example code: Compute the sum of 1 + 2+ ...+ 100

```
int sum, i;  
i = 1; sum = 0;  
while (i <= 100){  
    sum = sum + i;  
    i = i + 1;  
}
```

“for” statement

➤ General syntax

```
for (expression1; expression2; expression3)
{
    statements;
}
```

- ☐ expression1 is run before the loop starts.
- ☐ expression2 is evaluated before each iteration.
- ☐ expression3 is run after each iteration.

➤ Example code: Compute the sum of 1 + 2+ ...+ 10

```
int sum;
sum = 0;
for (int i = 1; i <= 10; i++){
    sum = sum + i;
}
```

“do” statement

➤ General syntax

```
do {  
    statements;  
} while (expression);
```

➤ Example code: compute the sum of $1 + 2 + \dots + 10$

```
int sum, i;  
i = 1; sum = 0;  
do{  
    sum = sum + i;  
    i = i + 1;  
} while (i <= 10);
```

“break” statement in loop

- The **break** statement inside a loop forces early termination of the loop.
- What is the value of **sum** after the following code is executed?

```
int sum, i;  
i = 1; sum = 0;  
while (i <= 10){  
    sum = sum + i;  
    i = i + 1;  
    if (i > 5)  
        break;  
}
```


“continue” statement in loop

- The **continue** statement skips the subsequent statements in the code block and forces the execution of the next iteration.
- What is the value of **sum** after the following code is executed?

```
int sum, i;  
i = 1; sum = 0;  
while (i <= 10){  
    i = i + 1;  
    if (i < 5)  
        continue;  
    sum = sum + i;  
}
```

C Arrays

- An array is a list of values that have the same data type.
- In C, array index starts from 0.
- An array can be one-dimensional, two-dimensional or more.
- This code example creates a 2-D array (multiplication table):

```
int a[8][10];  
for (int i = 0; i < 8; i++)  
    for (int j = 0; j < 10; j++)  
        a[i][j] = i * j;
```

- An array can be initialized when it is declared:

```
int b[3] = {4, 1, 10};  
unsigned char keypad_key[3][4] = {{ '1', '4', '7', '*' },  
                                   { '2', '5', '8', '0' },  
                                   { '3', '6', '9', '#' } };
```

C functions

- C functions are sub-routines that **can be called** from the main program or other functions.
- A C function can have a list of **parameters** and produce a **return value**.
- Functions enable modular designs, code reuse, and hiding of complex implementation details.
- Let us study C functions through examples.

Functions – Example 1

Write a function to compute the factorial $n!$ for a given n .

```
// factorial is the name of the custom function
// it accepts an input n of type int, and returns an output of type int
int factorial(int n){
    int prod = 1;
    for (int i = 1; i <=n; i++)
        prod = prod * i;
    return prod;    // return the result
}

int main(void){
    int n = 5;    // some example value of n
    int v;        // variable to storage result
    v = factorial(n); //call the function, store return value in v
    return 1;
}
```

Functions – Example 2

Write a function to compute the factorial $n!$ for a given n .

```
// factorial is the name of the custom function
// it accepts an input n of type int,
// it stores output at memory location by int pointer p
void factorial(int n, int* p){
    int prod = 1;
    for (int i = 1; i <=n; i++)
        prod = prod * i;
    *p = prod; // store output at memory location pointed by p
}

int main(void){
    int n = 5;      // some example value of n
    int v;          // variable to storage result
    factorial(n, &v); // call the function, store return value in v
}
```

Guidelines on C coding and documentation

- Optimize the C code for efficiency and length.
- Delete unnecessary lines of code.
- The C code must be properly formatted.
- Use indentation to show the logical structure of the program.
- Use a blank or comment line to separate code sections.
- Use meaningful variable names and function names.
- Use comments concisely to explain code.
 - ❑ Specially cite the purpose of the code, author name and date of generation

Summary

➤ What we learnt in this lecture:

- ❑ The tools and the steps for programming the Atmel AVR.
 - ❑ AVR Studio
 - ❑ Development Cycle for C Language
- ❑ Basics about C programming.
- ❑ Programming the digital I/O ports of Atmega32.

➤ References:

- ❑ ATmega32 datasheet
- ❑ Lecture notes of Dr. Lam Phung
 - ❑ University of Wollongong, Australia, 2011 (elec.uow.edu.au)