Embedded Systems Laboratory

Fangning Hu

Course description

Microcontrollers are core components of modern devices. Designed to handle sensor data and to control actuators, equipped with considerable computational power at relatively low cost and with limited power consumption, they are enablers of our rapidly growing technological environment when it comes to mobile systems.

We are going to use the AVR/ARM processer based on the RISC-architecture which becomes more and more popular used in smartphones, tablets, and various forms of embedded systems due to its small size and low power consumption. The course provides a sound introduction to these almost ubiquitous devices and guides the students in an application-oriented manner through a series of design tasks. The list of topics includes the basic architecture of a microcontroller with its ALU, timer/counter, memory, I/O interface; the concepts of working registers, interrupt vectors, program counter etc.; necessary programming tools like embedded C, and assembler, as well as several implementation problems like reading/controlling various sensors/actuators, processing internal/external interrupts, generation of PWM signals, and AD/DA conversion. At the end of the course, students should be able to develop and implement their own solutions for typical applications on AVR/ARM based microcontrollers.

In this course, we are going to use the AVR 8-bits Atmega328 microcontroller on an Arduino Uno Board.

Prerequisite

Electronics, Circuit Design, C Programming, Computer Architecture

Grading

25% Lab Performance
25% Lab Report
50% Final

Chapter 1 Microcontroller Basics

A microcontroller is a single chip containing at least a CPU, non-volatile memory, volatile memory, a timer, and an I/O control unit.

Section 1 Components of a Microcontroller

Central Processor Unit (CPU) includes the Arithmetic Logic Unit (ALU), Control Unit, Instruction Decoder, some General-Purpose Working Registers and some Special Registers (Stack Pointer, Status Register, Program Counter, etc.). ALU is the only unit in the microcontroller to perform arithmetic and logic operations between a general-purpose working register and another register or constant.

Volatile Memory is a memory used for temporary data storage. This type of memory requires power to maintain the data. Dates will be lost when the power is switched off. Static random-access memory (SRAM) and dynamic random-access memory (DRAM) are examples of this type of memory. AVR microcontrollers utilize SRAM to store data.

Non-Volatile Memory, on the contrary to the volatile memory, does not require power to maintain the data, which is typically used to store programs, especially the booting program. The data on this type of memory will not be lost when power is switched off. Memory in this category includes read only memory (ROM), erasable programmable read only memory (EPROM), electrically erasable programmable read only memory (EEPROM) and Flash memory. Flash memory is basically a type of EEPROM but with fast erasing speed. The AVR microcontrollers utilize Flash memory to store programs, thus it is also sometimes called flash program memory or simply program memory.

Timer Module are usually composed of one or several independent counters which are used to counting clock cycles or generating PWM signal in the microcontroller.

Interrupt Module enables the microcontroller to monitor certain events in the background while executing and application program and react to the event if necessary, pausing the original program.

Digital I/O Module allows digital/logic communication with the microcontroller and the external world. Communication signals are that of TTL or CMOS logic.

A microcontroller apart from the above-mentioned components usually also include, but not limited to, serial communication capabilities and analog I/O capabilities.

Analog I/O Modules are used to input/output analog information from/to the external world.

Serial Modules used for serial communication with the external world. An example is the USART peripheral which utilizes the RS232 standard.

Section 2 Difference from a PC and a Microprocessor

Difference from a PC

Basically, a microcontroller can be described as a computer on a chip. The difference between a microcontroller and a regular PC is that the PC is a general-purpose computer while a microcontroller is a computer that usually only runs a single program performing dedicated task(s).

Difference from a Microprocessor

A microcontroller contains on chip CPU, input/output interface, memory, clock, timer, and other necessary peripherals. A Microprocessor on the other hand is just a CPU.

Compared to the microprocessor product, a microcontroller product has many components on one chip and so is more compact. With a microprocessor product one has to add several other chips. That is also the reason that a microcontroller product is cheaper than a microprocessor product.

Questions:

1. Can you list the main components of a typical microcontroller?
2. What is the function of each component?
3. What type of memory the USB belong to?
4. Is an FPGA a microcontroller? Why?
5. What is the difference between a microcontroller and a PC?
6. What is the difference between a microcontroller and a microprocessor?

Chapter 2 The AVR Atmega328 Microcontroller

The following contents are mainly from the Atmega328 datasheet. For your convenience, I summarize the important information in the AVR Memories Chapter and the AVR CPU Core Chapter in the datasheet.

During the whole course, the Atmeage328 datasheet is one of the most important resources you should refer to, please read it carefully.

Section 2.1 Datasheet Chapter AVR Memories

The ATMega328 microcontroller contains three blocks of memory: Data Memory (SRAM), Program Memory (Flash), and EEPROM Memory.

Section 2.1.1 Program Memory Contains:

32K bytes Flash Memory (boot section at the bottom and application section on top) is organized as 16K 16-bits (16K words) space. Please check the datasheet Fig 8-2.

Another important thing in the program memory is that the spaces from 0x0000 to 0x0032 are reserved for interrupt vector table (check the Chapter Interrupts in the datasheet), your application programs should always be stored starting or after the address 0x0034. Note that the 0x prefix means the following number is a hexadecimal number and a binary number is defined using the 0b prefix.

Section 2.1.2 Data Memory Contains:

32 8-bits (bytes) General Purpose Register
64 8-bits (bytes) Input/Output Registers
160 8-bits (bytes) Ext I/O Registers
2048 8-bits (bytes) SRAM space

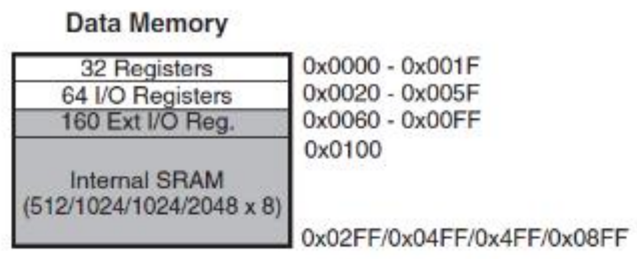


Fig. 2.1 The Data Memory Map of Atmega328

The 32 General Purpose Working Registers (datasheet Chapter CPU Core)

This 32 working Registers can be directly read, written and operated by the ALU, that is why they are called *working* registers. The addresses of these registers as well as their names are listed in the following diagram.

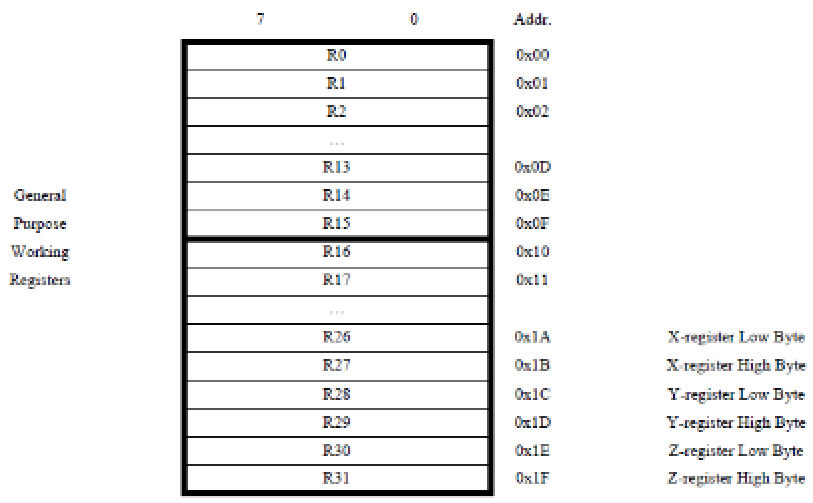


Fig. 2.2 The 32 General Purpose Working Register of Atmega328

Among them, the most interesting registers are R16 to R31. A constant value can immediately load into these registers by the assembly language instruction LDI. For example:

```
  LDI R16, 0xFF      ; load a Hexadecimal value FF into R16
```

Store any value into an SRAM address (k) need to use STS instruction:

```
  STS 0x6F, R16   ; store the value in R16 to address 0x6F
```

All the assembly language instructions with short explanation are summarized in the datasheet Chapter Instrution Set Summary.

The structure of 64 I/O Registers

The addresses as well as their names are listed in a table in the datasheet Chapter Register Summary. Please read the NOTE at the bottom of the table to find out how to access them by assembly instructions.

EEPROM Memory Contains: 1K bytes EEPROM space.

Section 2. Datasheet Chapter AVR CPU Core

The AVR CPU are composed of an ALU, 32 general purpose working registers which can be directly accessed by the ALU, some special registers (Program Counter, Stack Pointer, Status Registers, etc.) and necessary control units.

The CPU control unit reads the instruction at the address defined by Program Counter (PC) from the flash program memory, load the corresponding data into the general purpose working registers. ALU then implements the corresponding instructions such as ADD, AND, MOV, etc. on the working registers and write the results back to the working registers. After finish one instruction, the Program Counter (PC) automatically increases 1 or 2 depends on the different instructions. Please consult the datasheet Chapter Instruction Set Summary to find out how PC increases by implementing different instructions.

The following is an AVR Atmega328 microcontroller CPU Core Architecture.

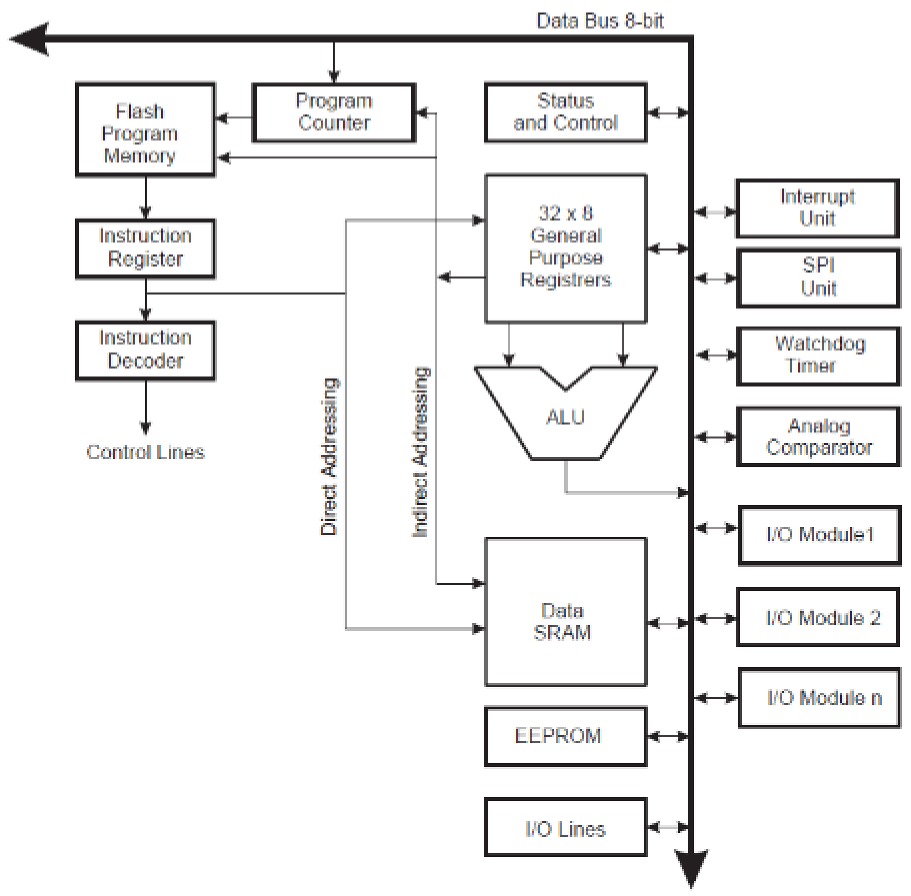


Fig. 2.3 The CPU Core Architecture of Atmega328

Among all the special registers, the most interesting ones are Status Register (SREG) and Stack Pointer (SP). Both are located within the 64 I/O memory addresses.

The Status Register (SREG) is defined as:

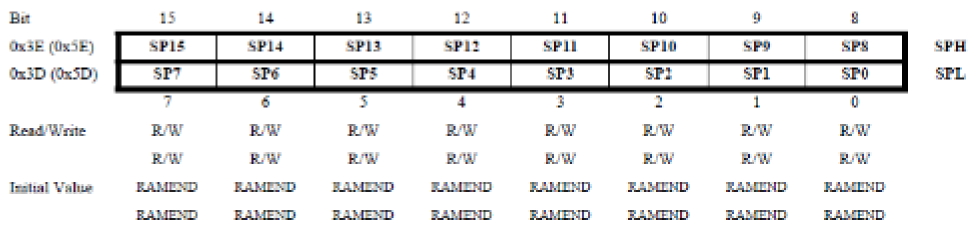
Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

where the bit 7 - I (the Global Interrupt Enable bit) is the most important one. In order to enable any interrupts, this bit has to be set to 1. This I-bit can be set to one or cleared to zero by assembly instruction SEI and CLI.

Other bits may be affected by some certain instructions. Please check the Instruction Set Summary in the datasheet for detail information.

The Stack Pointer (SPL and SPH)

The Stack Pointer in AVR works as two 8-bits register defined as SPL (contains the lower 8-bits) and SPH (contains the high 8-bits). This 16-bits SP contains the address of the top of the Stack. The Stack is mainly used to store temporary data. The initial address of the Stack is always the last address of the SRAM which has a special name: RAMEND



Bit	15	14	13	12	11	10	9	8	
0x3E (0x5E)	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
0x3D (0x5D)	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	
	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	RAMEND	

The Stack Pointer has to be defined before you call a subroutine or an interrupt. For more detail information of the Stack Pointer, please refer to the datasheet Chapter AVR CPU Core – Stack Pointer.

Questions:

1. Where can you find the information of the memory map of a microcontroller?
2. How can you load a data to a working register?
3. What does the instruction SEI do?

Chapter 3 The Basic of the AVR Assembly Language

The Assembly language is an alphanumeric representation of machine code. Below is an example of a AVR assembly code written in assembly language.

```
        ;Col_1    Col_2    Col_3      Col_4
         CLI                         ; clear the I-bit
         ADD      R16,     R17       ; Add value in R16 to value in R17
         DEC      R17                ; Minus 1 from the value in R17
         MOV      R12,     R16       ; Copy the value in R16 to R12
END:     JMP      END                ; Jump to the label END
```

Column 1 (Col_1) is used for labels. Labels are basically markers use by the programmer when indicating to the microcontroller to jump to a specific location in the code.

Column 2 (Col_2) is used for the microcontroller instructions.

Column 3 (Col_3) is used for arguments operated on by the instruction in column 2.

Column 4 (Col_4) is used for comments. Any statement after a semi-colon ";" in the same line is a comment.

Assembler Directives

Apart from the microcontroller instruction set that is used in writing an AVR assembly code the AVR Assembler support the use of assembler directives. Assembler Directives are used by the programmer to assist in adjusting the location of the program in memory. A list of the AVR assembler directives available to the AVR assembler is given below.

Some AVR Assembler Directives	
Directives	**Description**
BYTE	Reserve a Byte for a Variable
CSEG	Define the Code Segment Section
DB	Define Constant Byte(s)
DEF	Define a Symbolic Name
DSEG	Define the Data Segment Section
DW	Define Constant Word(s)
ENDMACRO	Signifies the End of a Macro
EQU	Set a Symbol Equal to an Expression
ESEG	Define the EEPROM Section
INCLUDE	Read Source Code from a File
MACRO	Signifies the Beginning of a Macro
ORG	Set Program Origin
SET	Set Symbol to an Expression

The assembler directives are always start with a dot "." as show below:

```
.include "m328def.inc"
.org 0x0000
          RJMP begin      ; jump to begin
.org 0x0034                ; set the initial program address at 0x0034
begin:    ADD R16, R17     ; Add R17 to R16
          MOV R17, R18      ; Copy R18 to R17
```

In order to use the register names such as R16, R17, SREG, etc., a header files "m328def.inc" has to be included in the beginning of the code.

As already mentioned in early chapter, the application program should be stored in the program memory after the address 0x0034. The org directive is used to set the initial program memory address.

Writing Subroutines

A subroutine is the assembly equivalent of a function in C. Subroutines are generally use when there are blocks of code that are executed more than once. Subroutines allow for the block of code to be written once then call upon when it is required.

Very Important

There are two very important points you always need to remember when using subroutines:

1. When a subroutine is called the microcontroller needs to save the address of the next instruction. This is necessary as the microcontroller needs know where to return after the execution of the subroutine. The microcontroller automatically saves this address on the stack. As such the programmer **MUST** initialize the stack. This is done by storing the starting address of the stack in the stack pointer. The AVR assembly code below shows an example, here the stack pointer of an

ATMega328 AVR microcontroller is being loaded with the last address in SRAM memory.

```
.include "m328def.inc"
.org 0x0000
                RJMP begin      ; jump to begin
.org 0x0034          ;Initialize the microcontroller stack pointer
 begin:   LDI    R16, low(RAMEND)
                OUT    SPL, R16
                LDI    R16, high(RAMEND)
                OUT    SPH, R16
```

2. When executing a subroutine, the microcontroller needs to know where the end of the subroutine is in other to return to the point in the code where it was before the subroutine was called. The programmer indicates this by putting the **RET** at the end of the subroutine. The following AVR assembly program toggles the logic value on the pins of portD of an ATMega328AVR microcontroller with a delay of 2 CPU clock cycles after each change. Here the delay is provided by the **"Delay"** subroutine.

```
.include "m328def.inc"
.org 0x0000
          RJMP   begin      ; jump to begin
.org 0x0034
 begin:      CLI
          LDI    R16, low(RAMEND)
          OUT    SPL, R16
          LDI    R16, high(RAMEND)
          OUT    SPH, R16

          LDI    R16, 0xFF
          OUT    DDRD, R16

          LDI    R16, 0xFF
          OUT    PORTD, R16
          RCALL  Delay
          LDI    R16, 0x00
          OUT    PORTD, R16
          RCALL  Delay


Delay:    LDI  R17, 0x02
loop:     DEC  R17
          BRNE loop
          RET
```

Resources

Atmega328 datasheet, Chapter Instruction Set Summary

Atmel AVR 8-bit Instruction Set

Development Tool (*Microchip Studio*):

We are going to use the Microchip Studio to simulate, debug and upload our codes to the microcontroller. Please download it at

[Microchip Studio for AVR® and SAM Devices | Microchip Technology]()

Download [arduino.zip]() from the course webpage directory 'Resources', unzip it and you will find two files: avrdude.exe and avrdude.conf. Put them in to your computer directory "C:\arduino\".

Start Microchip Studio, in the drop down menu, find 'Tool->External Tool' and set the parameters as follows:

```
Command: C:\arduino\avrdude.exe
Arguments: -v -v -v -v -patmega328p -carduino -P\\.\COM3 -b115200 -D -
Uflash:w:$(ProjectDir)Debug\$(TargetName).hex:i
```

Chapter 4   The Basic of AVR C Programming

Bits and Bytes

A bit represents one of two possible states: 1 or 0 (aka: on/off, set/clear, high/low). Several bits together represent numerical values in binary, where each bit is one binary digit. An AVR microcontroller groups 8 bits together to form one byte with the Least Significant Bit (LSB) on the right. Each bit is numbered, starting from 0, at the LSB.

Consider the decimal number 15 for example. 15 represented in 8-bit binary is 00001111. The 4 least significant bits 0, 1, 2, and 3, are set.

The following C code shows 3 ways in which a variable might be initialized to the decimal unsigned integer of 15.

```
uint8_t a = 15;        /* decimal */
uint8_t b = 0x0F;      /* hexadecimal */
uint8_t c = 0b00001111; /* binary */
```

Bitwise Operations

Since individual bits often have a significant meaning when programming AVR microcontrollers, bitwise operations are very important.

A single ampersand character (&) is the bitwise AND operator in AVR C.

```
uint8_t a = 0xAA; /* 10101010 */
uint8_t b = 0x0F; /* 00001111 */
uint8_t = a & b;  /* 00001010 */
```

A single pipe character (|) is the bitwise OR operator in AVR C.

```
uint8_t a = 0xAA; /* 10101010 */
uint8_t b = 0x0F; /* 00001111 */
uint8_t = a | b;  /* 10101111 */
```

The caret character (^) is the bitwise XOR operator in AVR C.

```
uint8_t a = 0xAA; /* 10101010 */
uint8_t b = 0x0F; /* 00001111 */
uint8_t = a ^ b;  /* 10100101 */
```

A tilde character (~) is the NOT operator in AVR C.

```
uint8_t a = 0xAA; /* 10101010 */
uint8_t b = ~a;   /* 01010101 */
```

Two less-than symbols (<<) is the left shift operator and two greater-than symbols (>>) is the right shift operator in AVR C. The right side of the operator is the number of bits to shift.

```
uint8_t a = 0x99; /* 10011001 */
uint8_t b = a<<1; /* 00110010 */
uint8_t c = a>>3; /* 00010011 */
```

Clearing and Setting Bits

Setting and clearing a single bit, without changing any other bits, is a common task in AVR microcontroller programming. You will use these techniques repeatedly.

When manipulating a single bit, it is often necessary to have a byte value in which only the bit of interest is set. This byte can then be used with bitwise operations to manipulate that one bit. Let's call this a **bit value mask**. For example, the bit value mask for bit 2 would be 00000100 and the bit value mask for bit 6 would be 01000000.

Since the number 1 is represented in binary with only bit 0 set, you can get the bit value mask for a given bit by left shifting 1 by the bit number of interest. For example, to get the bit value mask for bit 2, left shift 1 by 2.

To **set a bit** in AVR C, OR the value with the bit value mask.

```
uint8_t a = 0x08; /* 00001000 */
                  /* set bit 2 */
a |= (1<<2);      /* 00001100 */
```

Use multiple OR operators to set multiple bits.

```
uint8_t a = 0x08;   /* 00001000 */
                    /* set bits 1 and 2 */
a |= (1<<2)|(1<<1); /* 00001110 */
```

To **clear a bit** in AVR C, NOT the bit value mask so that the bit of interest is the only bit cleared, and then AND that with the value.

```
uint8_t a = 0x0F; /* 00001111 */
                /* clear bit 2 */
a &= ~(1<<2);     /* 00001011 */
```

Use multiple OR operators to clear multiple bits.

```
uint8_t a = 0x0F;      /* 00001111 */
                    /* clear bit 1 and 2 */
a &= ~((1<<2)|(1<<1)); /* 00001001 */
```

To **toggle a bit** in AVR C, XOR the value with the bit value mask.

```
uint8_t a = 0x0F; /* 00001111 */
                /* toggle bit 2 */
a ^= (1<<2);      /* 00001011 */
A ^= (1<<2);      /* 00001111 */
```

The Bit Value Macro _BV()

AVR Libc defines a the _BV() macro which stands for "bit value". It is a convenience macro to get the bit value mask for a given bit number. The idea is to make the code a little more readable over using a bitwise left shift. Using _BV(n) is functionally equivalent to using (1<<n).

```
/* set bit 0 using _BV() */
a |= _BV(0);

/* set bit 0 using shift */
a |= (1<<0);
```

Which method you choose is entirely up to you. You will see both in wide use and should be comfortable seeing either. Since the _BV(n) macro is unique to gcc, it is not as portable to other compilers as the (1<<n) method.

AVR C Library

In order to use the register names defined in Atmega328 such as PORTx, SREG, SPH, SPL, etc., specific header files such as <avr/io.h> <util/delay.h> <avr/interrupt.h> have to be included. For a detail description of all the header files in the AVR C Library, please check the online document at

http://www.nongnu.org/avr-libc/user-manual/modules.html

Create a new C project in Atmel Studio, please follow the instructor´s instruction. Do not forget to select the Atmega328 as a target device!!