

**Technische Universität  
München**

**Fakultät für Informatik  
Forschungs- und Lehrinheit Informatik VI  
Robotics and Embedded Systems**

**Lab Course  
“Microcontroller Programming”**

Michael Geisinger  
geisinge@in.tum.de

Jia Huang  
huangji@in.tum.de

WS 2009/10



## Introduction

### Content

1. **Introduction:** organisational issues, microcontroller ATmega8515, IDE AVRStudio
2. **Simple I/O:** digital input and output, LEDs and switches, applications (~2 sessions)
3. **Serial communication:** U(S)ARTs, interrupts, ISRs, serial communication, applications (~3 sessions)
4. **Timers:** timers and timer resolution, prescalers, oscillators, PWM, applications (2 session)
5. **DCF77 time signal:** DCF77 purpose and history, signal encoding, applications (2 sessions)
6. **Analog signal acquisition:** ADCs, intelligent sensors, applications (1 session)
7. **Displays (optional):** LCDs, SPI, applications (1 session)
8. **Final exercise:** distributed systems, communication, applications (~3 sessions)

### Organisational issues

#### Time

Day	Time	Room
Thursday	13:15 - 17:45	MI 03.05.012

#### Organizers

Name	E-mail	Room	Telephone
Michael Geisinger	geisinge@in.tum.de	03.07.042	18111
Jia Huang	huangji@in.tum.de	03.07.042	18111

#### General conditions

- Please sign your name on the attendance list. Only participants that regularly attend the lab course will get a certificate!
- Please fetch necessary hardware from our office, sign your name on the list of borrowed hardware and bring the hardware back to our office when you leave.
- Your solutions must include appropriate documentation.
- Expect to be asked some questions about your implementation.
- Save each of the solutions to the exercises in this lab course as a separate *AVRStudio* project and send all the projects together with the answers to the questions to <geisinge@in.tum.de> (subject: “[MCP] Group <GroupNumber>, Exercise <ExerciseNumber>”).

#### Materials

You can find the most recent versions of the materials for the lab course at the network share P:\.



## Exercise 1: Introduction and Digital I/O

### 1 Microcontrollers

Over the last decades, microcontrollers became ubiquitous in everyday life. From small consumer electronic products like MP3 players and digital cameras to huge industrial applications like cars, aircrafts and power plants, microcontrollers play a key role in performing control and signal processing tasks. In contrast to microprocessors used in standard PCs and servers, microcontrollers are designed for small or dedicated applications. Those special-purpose computer systems constructed with microcontrollers and necessary peripherals are also known as *embedded systems*.

In this section we will present you some fundamental knowledge of microcontrollers, starting from introducing their basic elements.

#### Basic Elements of a Microcontroller

In general, a microcontroller is more than just a Central Processing Unit (CPU). It is typically a CPU integrated with memory and many other peripherals. This integration drastically reduces the number of chips and the amount of pins and wirings in the circuit board, and thus reduces the cost. Microcontrollers have proved to be highly popular in embedded systems since their introduction in the 1970s.

A microcontroller is a single integrated circuit, commonly with the following features:

- CPU (4/8/16/32/64 bit).
- Clock generator, often a crystal or realtime clock (RC) oscillator.
- ROM, EEPROM, or flash memory for program/configuration storage.
- RAM for data storage.
- General purpose input/output pins, configurable in software.
- Timers and Pulse-Width Modulation (PWM) generators.
- Communication ports, like Universal Asynchronous Receiver/Transmitter (UART), Universal Serial Bus (USB), Controller Area Network (CAN), etc.
- Analog-Digital Converter (ADC) and Digital-Analog converter (DAC).
- In-circuit programming/debugging facilities (e.g., JTAG port).

#### Programming Model

The programming model represents the programmer’s view of the device. In case of microcontrollers, this typically includes the set of accessible registers, the instruction set, the programming convention and so on. In microcontrollers, each register has a predefined and unique address (usually declared in a header with a meaningful name). Reading/writing to this address will result in accessing the corresponding register.

Those registers can be briefly divided into two categories: special purpose and general purpose registers. Registers in the first category are usually used to store the configuration or status of the device. Take the 8-bit microcontroller ATmega8515 for example, there is a register called *CKSEL* (clock selection). By writing '0001' to it, we can select the system clock to be 1 MHz provided by internal RC oscillator. Alternatively, by writing '0100' to this register, the system clock is set to be 8 MHz. The meaning of the bit fields in the configuration registers is predefined and needs to

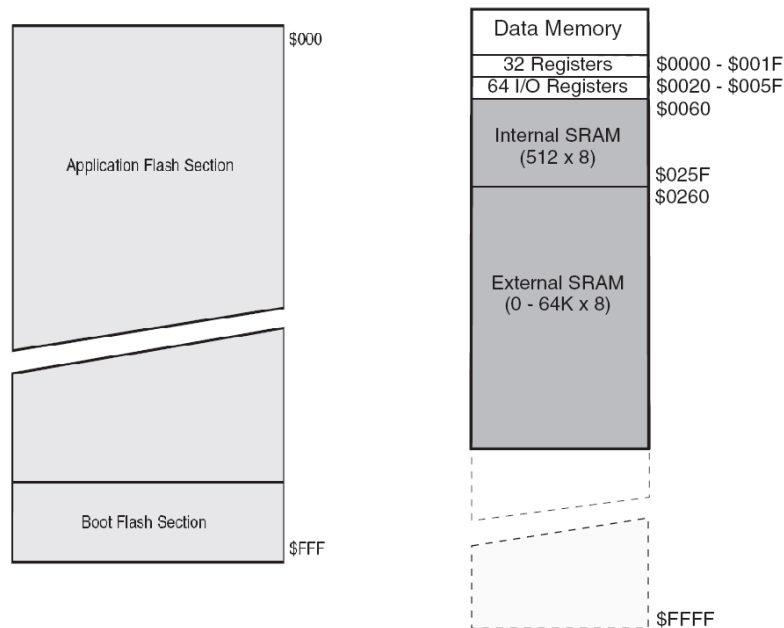


Figure 1: Program/Data Memory Map of ATmega8515

be looked up in the manual. The general purpose registers are used to store temporary values. ATmega8515 includes 32 8-bit general purpose registers (R0 to R31). The set of general purpose registers is often referred to as register file.

Microcontrollers were originally programmed only in assembly language, but various high-level programming languages are now also in common use to target microcontrollers. These languages are either designed specially for the purpose, or versions of general purpose languages such as the C programming language. Compilers for general purpose languages will typically have some restrictions as well as enhancements to better support the unique characteristics of microcontrollers. Some microcontrollers have environments to aid developing certain types of applications (e.g., *AVRStudio* in this lab course).

## Address Space and Memory Mapped I/O

The entire address range that is accessible from the CPU is called address space. Virtually, everything is just an address from the CPU’s point of view. Registers, memories, and peripherals may share the same address space and special care needs to be taken to guarantee absence of ambiguity. *Memory-mapped I/O* uses the same address bus to address both memory and I/O devices (e.g. pins and buffers), and the CPU instructions used to access the memory are also used for accessing devices. In order to accommodate the I/O devices, areas of CPU’s addressable space must be reserved for I/O.

Figure 1 illustrates the program and data memory mapping of ATmega8515. As a separate bus is used for each memory, they belong to different address spaces and therefore can both start from address zero. In program memory the lower address space is used for the 8 KB on-chip flash memory for program storage. The boot flash section contains the boot loader program (instructions that can be used to program the on-chip flash memory from the microcontroller itself, e.g., over a serial communication channel) and resides in the highest location. This section is usually locked for normal usage.

Now let’s see an example. We would like to read the value of port ‘B’ of ATmega8515, an 8-bit

general purpose I/O port. First, the address of PORTB is defined:

```
#define PINB    (*(volatile uint8_t*)(0x16))
#define DDRB    (*(volatile uint8_t*)(0x17))
#define PORTB   (*(volatile uint8_t*)(0x18))
```

In a real program, this definition is of course included from a header file written by the microcontroller manufacturer, so we do not need to care about the concrete addresses. We added it here just for illustration purposes. Each 8-bit general purpose I/O port has three registers associated to it, the PINx, the DDRx and the PORTx register. For detailed description of those registers, please refer to the manual<sup>1</sup>. With this definition, the current value on port 'B' can be obtained by performing a normal assignment operation:

```
/* Read port B */
char a = PINB;
```

The `volatile` C-language key word in the definition of the register names is essential here. It tells the compiler that the value in this address (e.g., 0x18 for PORTB) may change spontaneously even if the software itself does not modify it. For example, assume the following code:

```
/* Read port B */
char a = PINB;
/* Do something without writing to PORTB */
while(...) { ... }
/* Read port B again */
char b = PINB;
```

If we only look at the instructions, the variable *a* and *b* should have the same value, since no one writes to PINB between the two read operations. The 'smart' compiler can of course detect this fact and optimize the code by skipping the second read operation. This is correct in general but *not* for memory mapped I/O addresses. The value in PINB represents the current status of the corresponding pins in port B and it varies in real time (e.g., we can modify the value in port B easily by connecting supply voltage/ground the pins). The `volatile` key word prevents the compiler from applying optimizations and forces it to read the value every time when specified in the source code. Basically `volatile` should always be added to memory mapped I/O addresses. There are also some other occasions where we need the `volatile` key word. We will revisit this in the exercise about interrupts.

## Firmware

Microcontrollers are usually used to perform predefined tasks in a periodic manner. In this case, the program running on a microcontroller is commonly called firmware, representing a feature that it is somehow fixed. Naturally, there is no strict, or well defined, boundaries between firmware and software, both are quite loose descriptive terms. However, firmware is typically involved with very basic low-level operations in a device, without which the device would be completely non-functional. Because of its periodical behavior, the firmware is usually implemented as an endless loop. An example may look like:

```
int main() {
    /* Initialize the system */
    do_init();

    while(1) {
        /* The periodic task */
        do_my_task();
    }
}
```

---

<sup>1</sup>ATmega8515 manual: [http://atmel.com/dyn/resources/prod\\_documents/doc2512.pdf](http://atmel.com/dyn/resources/prod_documents/doc2512.pdf)

```
}

/* If we get here, the device will halt until being reset :-( */
/* We usually do not want this to happen, hence the while loop. */
return 0;
}
```

## 2 ATmega8515

ATmega8515(L) is a low-power 8-bit microcontroller developed by Atmel<sup>2</sup>. For the key features of this product, see the first page of the manual<sup>3</sup>. For easy interaction with the microcontroller, we are using the *STK500*<sup>4</sup> development board, which allows us to directly access all relevant signals of the microcontroller. The development board features the following components:

- A socket for the microcontroller (already mounted).
- Connectors for all general purpose input/output (I/O) pins of the microcontroller.
- Serial interfaces for communication with the host PC.
- 8 Light Emitting Diodes (LEDs) for visualizing the current state of the microcontroller.
- 8 switches for interacting with the program on the microcontroller.

Please make sure that the following cable connections are set up on the development board (see also figure 2). This is the initial configuration for all subsequent lab course exercises.

- Programming: Connect the serial interface of the PC to the RS232 CTRL interface of the board and use the six-wire pin to connect ISP6PIN to SPROG3 (please be sure to connect pin 1 on one connector to the respective pin 1 on the other connector).
- LEDs: Connect PORTB to LEDS using a ten-wire cable.
- Switches: Connect PORTA to SWITCHES using another ten-wire cable.
- Power supply: Connect the power supply to the respective connector on the board.

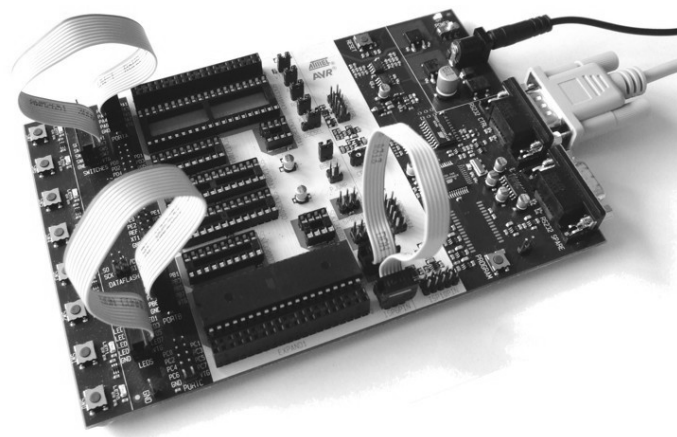


Figure 2: Initial Setup of STK500

---

<sup>2</sup>Atmel homepage: <http://www.atmel.com/>

<sup>3</sup>ATmega8515 manual: [http://atmel.com/dyn/resources/prod\\_documents/doc2512.pdf](http://atmel.com/dyn/resources/prod_documents/doc2512.pdf)

<sup>4</sup>STK500 manual: [http://atmel.com/dyn/resources/prod\\_documents/doc1925.pdf](http://atmel.com/dyn/resources/prod_documents/doc1925.pdf)



## AVRStudio

AVRStudio is a Integrated Development Environment (IDE) provided by Atmel for writing and debugging applications. AVRStudio itself is not shipped with a tool chain for ATmega8515 (compiler, assembler, linker, etc.), rather, it relies on external packages to provide those tools to compile a C program. In our case, we use *WinAVR*, which is a free GCC based tool set.

### Setting up an AVRStudio Project

1. Run AVRStudio by selecting *Atmel AVR Tools* → *AVR Studio 4* in the start menu. You might want to copy the shortcut to your desktop. Create a new *AVR-GCC* project.
2. Let AVRStudio create the project directory for you, but without an initial file.
3. Choose *AVR Simulator* as *Debug Platform* and *ATmega8515* as *Device*.
4. Adapt the project settings (right click on the project name, *Edit Configuration Options...*):
  - *General* → *Frequency*: 8000000 Hz
  - *General* → *Optimization*: -O0
  - *Custom Options* → *External Tools*: Make sure the *Use WinAVR* checkbox is checked.
5. Copy your source and header files to the directory you just created (or write them).
6. Use drag & drop from *Windows Explorer* to add the source and header files in the project directory to the project.

### Uploading a Program to the Target

1. In the menu, select *Tools* → *Program AVR* → *Connect...* Select *STK500* and the COM port the device is attached to. To find out the correct COM port number, use the *Windows Device Manager* or a program like *HyperTerminal*.
2. In the *Main* tab, select the correct device ATmega8515 as well as *ISP mode*. *ISP Frequency* can be set to 1,843 MHz (for faster programming).
3. Choose the compiled *Input HEX file* at the *Flash* area under the *Program* tab. When building a project from within *WinAVR*, the \*.hex file should be located in the *default* subdirectory of your project. Click the *Program* button to upload it to the controller.

<p><b>Warning:</b> Do not modify the settings in the other tabs, especially not the <i>Fuses</i> or <i>LockBits</i>. You could irrevocably lock the microcontroller so that it can not be programmed any more!</p>
--

### Exercise 1.1

- a) Familiarize yourself with the AVRStudio environment. Create a new project and add the file `blink.c` (located in `P:\exercise01`) to your project. Compile the code into a firmware image (\*.hex) and upload it to the device. LED1 on the board should start blinking.
- b) Analyze the program you have just compiled. Why is the `avr/io.h` header file needed?
- c) What is the effect of the instruction `DDRB = 0x02;` (look it up in the manual)?
- d) In the function `wait()`, we used the type `int32_t`. What is the meaning of this data type? Which other similar data types are defined in `<WinAVRPath>\avr\include\stdint.h`?
- e) On which factors does the blinking frequency of the LED depend?

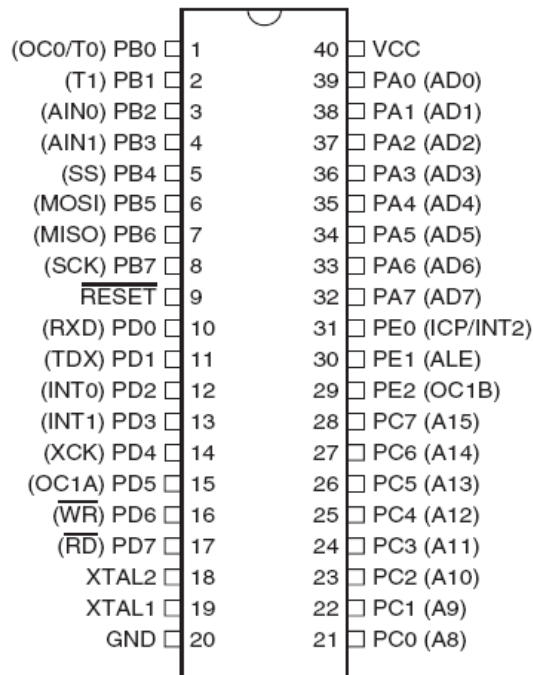


Figure 3: Pin Configuration of ATmega8515

## GPIO on ATmega8515

Figure 3 shows the pin configuration of ATmega8515. Read the manual including the sample C code to understand how to deal with reading/writing general purpose I/O ports.

### Exercise 1.2

- What is a general purpose I/O port? Which registers are involved? How many general purpose ports does ATmega8515 have? How can they be read? And how can they be set?
- LED blinking: Develop a fancy application to toggle one or more LEDs on/off. You might want to use the buttons on the board, make the blinking frequency adjustable, or similar. Document your program so that it is clear what it does.
- Counter: Develop an application to count the number of times that a button is pressed and display the current count using the LEDs. For each button press, switch one more LED on (up to 8). Reset the count to zero if it is larger than 8.
- Do you see any problems with the current button press counter implementation? How “accurate” is it?
- Try to handle a continuously pressed button (like what happens when you press a key on the keyboard for a long time).

### Hints

- Note that the LED/switch logic of the development board is a bit different from normal understanding. To light up an LED, you have to write a *zero* to the corresponding pin. The same applies to the buttons, the corresponding pin is *low* when a button is pressed.