

Getting Started with the AVR

(An 18-545 Walkthrough)

Legal Note:

AVR, ATAVRISP and STK500 are registered trademarks of Atmel Corporation.
All other company and product names are the possession of their respective owners.

All products and materials discussed in this tutorial are used solely for educational purposes. Should any product be misrepresented or this document in any way be in violation of trademark law, the course staff will be more than happy to make the appropriate corrections and can be contacted at ee545@ece.cmu.edu.

This document may be freely proliferated.

Text Formatting:

In this document all function calls, function names, paths and text commands are presented in `courier`. All other text is in Times New Roman.

Contents

Contents.....	2
What is the AVR?.....	3
Why Do I Care?.....	3
Interfacing With the AVR.....	5
Basic GPIO use.....	5
UART Setup.....	5
The Many Steps of Compilation.....	6
Tutorial 1: Simple Terminal Demo.....	7
Tutorial 2: SRAM Demo.....	10
Other Capabilities of the atmega8.....	12
References.....	13

What is the AVR?

The AVR is a family of 8-bit microcontrollers. In this course, we will be using the Atmel atmega8 AVR processor. This processor features 8KB of FLASH, 1KB SRAM, and a variety of built-in peripherals. It supports a proprietary 8-bit AVR ISA for which there is GCC support. The atmega8 has a maximum rated processor frequency of 16MHz and is little endian. The atmega8 lends itself extremely well to prototyping due to its DIP form factor (in the version we use) and its simple requirement of a 4.5V-5.5V power source.

Why Do I Care?

The AVR allows you to rapidly develop code for a very low-power, low-cost microprocessor with the advantages of GCC compiler support. The atmega8 has a decent amount of computational power and has excellent flexibility in the use of its IO pins.

The AVR family of processors has a very large following on the Internet allowing for rapid advice on development issues.

If that isn't enough, you are required to use this processor for the purposes of this class.

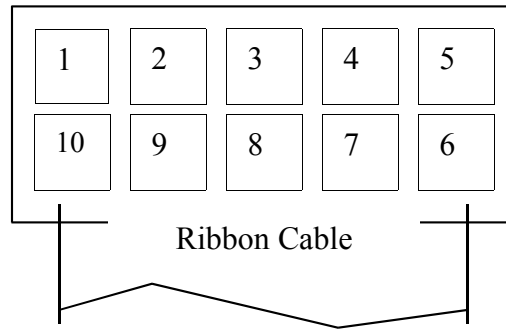
Programming the AVR

The AVR processor family, and in particular the atmega8 processor in its DIP format, are very easy to program through its In System Programming (ISP) interface. All programming is done via the ATAVRISP which connects easily to a COM port on the PC and to a 10 pin header on AVR. As a result, programming on the atmega8 processor is fast and easy.

Before any programming can take place, we must power the AVR and connect the UART lines on the atmega8 to the computer. This is not as simple as it seems, however. RS232 signals go between +12V and -12V and are inverted (-12V is a logical 1). Applying this kind of voltage to the AVR is very dangerous. Dealing with this problem requires the use of a very common chip: the MAX232. The MAX232 uses capacitor pumps to perform the necessary level shifting and provide two receive and two transmit channels per chip.

The AVR is programmed via an interface between a computer's serial port that goes to pins on the AVR. In this section, we will set up this interface. The ATAVRISP's (the programming interface) serial connector must be connected to the PC and we must setup an interface between the 10 pin connector on the ATAVRISP and the atmega8.

To ease this setup, we will establish a convention in which the bottom row of the 10 pin connector is defined as the row closest to the long part of the ribbon cable.



Use the following table to make connections between the AVR processor and the 10 pin header.

AVR Processor Pin	Programmer 10 Pin Header
17 (MOSI)	1
GND (any source will do)	2
1 (RESET)	3
19 (SCK)	4
18 (MISO)	5

One thing to note is that you must attach a 10K Ω resistor (brown – black – orange) to a 5V power source and pin 1 on the AVR processor. This pull-up resistor is necessary on RESET since the programmer is not strong enough to pull down anything beefier and the AVR does not have a pull up resistor on this pin internally.

Also, pin 10 on the 10 pin connector must be connected to a 5V power source. Be sure to do this step as it is VERY IMPORTANT.

To begin programming, the processor must have RESET and SCK pins set to 0. Then, specific Programming Enable serial instructions must be sent to the MOSI pin (more details in the atmega8 data sheet). However, this does not have to be done by the user as the ATAVRISP takes care of all of this functionality.

Now that we've setup the programming interface, it is relatively easy to compile and download a program.

Find the TerminalDemo program located at C:\545\demo\code\avr\ and type make. This will create a few files (whose creation and meaning will be discussed later in this document). Of most interest to us is the TerminalDemo.hex file. This file contains all the information necessary to run the program. To download this program to the AVR, type in

```
STK500.EXE -dATmega8 -e -ifTerminalDemo.hex -pf -vf
```

Since we are just testing to ensure that the programmer is properly setup, no actual output will be seen. However, if you receive an error message, ensure that the green light on the ATAVRISP is on and stable. If this is ok, check the orientation of your cable.

Interfacing With the AVR

Basic GPIO use

Most of IO pins on the atmega8 can be configured to act as GPIOs or to serve an alternate function. Most pins have two alternate functions and the available functions vary by pin. The value of the GPIOs is stored in memory as a few 8-bit integer with each bit corresponding to the appropriate pin. Information about specific pins can be extracted through bit-hacking (e.g. to find the value of Port C pin 4 you might write something like `((PORTC >> 4) & 1)` . A large variety of macro's and access to pin configuration information is located in `iom8.h`. It is highly recommended that you look through and understand the contents of this file.

UART Setup

This section describes how to set up the MAX232 chip and connect it to the AVR to enable UART transmissions between the AVR and the PC.

Let's setup the power lines to the chip first. After placing it onto the breadboard, place a 5V wire to pin 16 and place GND to pin 15 of the MAX232.

Next, obtain 4 1 μ F capacitors. Connect the negative end of one of the capacitor (the one with the stripe) to pin 3 and the positive end to pin 1. With another capacitor, connect the negative end to pin 5 and positive end to pin 4. Connect the 3rd capacitors negative end to a 5V column and the positive end to pin 2. Connect the last capacitors negative end to pin 6 and the positive end to a ground column. These connections are summarized in the table below.

Capacitor	Negative End	Positive End
1 μ F	Pin 3 – MAX232	Pin 1 – MAX232
1 μ F	Pin 5 – MAX232	Pin 4 – MAX232
1 μ F	5V	Pin 2 – MAX232
1 μ F	Pin 6 – MAX232	GND

Now it's time to connect the MAX232 to the AVR UART pin headers. Place a 5x2 pin header onto the board. Be careful not to place it in columns powered by another device. Make the following connections (use the picture of the pin header from **2.1.2** to guide you).

10 pin Header Connection	MAX232 Connection
Pin 1	GND
Pin 3	Pin 13 (R1 _{IN})
Pin 4	Pin 14 (T1 _{OUT})

Lastly, we must connect the MAX232 to the AVR. This involves two simple connections.

MAX232 Connection	AVR Connection
Pin 12 (R1 _{OUT})	Pin 2 (RXD)
Pin 11 (T1 _{IN})	Pin 3 (TXD)

The Many Steps of Compilation

(this section is borrowed entirely from Sean Pieper's excellent guide of the TinyARM)

GCC's default behavior masks many details of the compilation process from the average user. As a result, most programmers remain unfamiliar with the behavior of the linker which is crucial for cross-compilation. To go from source to binaries, there are a variety of steps involved.

These can be summarized as:

- Preprocessing
- Compilation
- Assembly
- Linking
- Object Translation

Preprocessing, compilation, and assembly flow as expected. As is normal, for speed, you may want to create individual object files for each input file. When we get to the linking stage, however, things become exciting.

To start with, when you're programming the atmega8, you're operating without any of the benefits of an OS. This means that you need to take care of a variety of low-level tasks that, as a programmer, you aren't typically concerned with. We provide you with a small assembly routine that will perform the most basic initialization, but if you need to install interrupt handlers, you will be forced to modify this code to set the appropriate jump points and distinguish between different interrupt sources.

For those of you who are unfamiliar with the functions performed by the linker (ld), it resolves labels across a number of input files, and then maps them to various segments of memory. There are segments for code, read-only variables, initialized global variables,

uninitialized global variables, and a variety of stacks. To some extent, these mappings are controlled by the memory layout of the atmega8 (i.e. code, read-only variables, and the initial values of initialized globals need to live in the part of memory reserved for FLASH whereas space is reserved for initialized globals to be copied into, for uninitialized variables, and for stacks in the the RAM.

Finally, once the linking has been performed, you need to convert the output .elf file to a .hex file that can be loaded directly into the processor's flash. This operation is performed by the objcopy utility.

Because of the numerous steps involved, it is strongly recommended that you take one of the provided Makefiles and modify that to fit your needs. If you are experienced in the dark arts of creating Makefiles, there is a lot of room for improvement in the provided file and the course staff would greatly appreciate any useful tweaks to an admittedly ugly, albeit functional, Makefile.

Tutorial 1: Simple Terminal Demo

In the C:\545\demo0\code\avr\ source directory, find and open the files TerminalDemo.c, TerminalDriver.c, TerminalDriver.h and makefile.

Let's start with the TerminalDriver.h file:

This file includes many terminal screen controlling functions that you may find useful. You may want to pay special attention to these lines:

Line (TerminalDriver.h)	Explanation
#include <avr/io.h>	Includes various architecturally dependent defines.
void Term_Initialise()	Function that sets up UART communications on the AVR with the appropriate baud rate.
void Term_Send(unsigned char data)	Function to send one char across the UART.
unsigned char Term_Get()	Function to retrieve one char from the UART. This function is blocking.

Next, let's look at the TerminalDemo.c file:
This file holds the main program.
You may want to pay special attention to these lines:

Line (TerminalDemo.c)	Explanation
#include <stdio.h>	Standard IO library
FILE *ser;	Create a stream pointer to be used by various IO functions.
ser=fdevopen(Term_Send, Term_Get, 0);	Function that establishes that all IO functions should use the UART driver as their primitive input output functions. Also assigns the stream.

Lastly, let's look at the makefile:

Line (Makefile)	Explanation
# MCU name MCU = atmega8	This specifies the name of the device we are using. Do not change this.
# Output format. (can be srec, ihex, binary) FORMAT = ihex	The format used by the STK500 to download the program onto the AVR.
# Target file name (without extension). TARGET = TerminalDemo # List C source files here. (C dependencies are automatically generated.) SRC = \$(TARGET).c # If there is more than one source file SRC += TerminalDriver.c	TARGET specifies the file that contains main. SRC must include the target and any other C file you wish to compile.

Note that a complete explanation for the lines above can be found in the AVR reference manual and on the AVR-LIBC webpage (<http://www.nongnu.org/avr-libc/user-manual/index.html>).

Basically, this program sets up the UART routines Term_Send and Term_Get as the basic atomic functions to be used by printf, putc, getc, etc. It then prints out "Hello World" onto a terminal screen and then echos whatever characters you enter into the terminal back to you. Since there are dedicated IO ports reserved for the UART, no special port configuration goes on in this program. The only configuration is via fdevopen to setup the screen functions. Basic information on how the UART works can be found in the atmega8 data sheet.

Before downloading the program to the AVR, open Tera Term. Configure Tera Term to use the port not occupied by the ATAVRISP and set it to a baud rate of 2400, no parity, one stop bit and no flow control.

Now that we know how the program works, let's test it out. Refer to the above sections as to how to compile the program and download it to the AVR. Once the program is downloaded, you should see "Hello World" printed out and be able echo characters to the terminal.

At this point you should be prepared for a slightly more ambitious project.

Tutorial 2: SRAM Demo

This section describes how you will connect 32KB of external SRAM to the AVR processor. This will be accomplished by using the AVR processor built-in input/output ports. Some background about the ports and functions is first given. Then the connections you need to make are described along with ideas for testing. Note, this part of Demo 0 is separate from the rest of Demo 0. As soon as you complete this section, please have one of the TA's check you off so that you can continue with the rest of the Demo 0.

AVR Processor Ports

The AVR offers two methods to communicate with its GPIO's. One method offers a separate IO address space that can only be addressed with specific IO instructions (in, out, etc.). The other method, which you maybe more familiar with, is memory-mapped IO which maps the entire IO address space onto the regular physical address space of the AVR, allowing port's to be read from and written to using simple C-style pointers. Both formats are described here and both formats are supported by the version of GCC we are using for this course.

Before setting up input and output commands, we must include the io.h file in our programs. This file simply includes the real IO header file for the system you are using. In our case, since we are using the atmega8, io.h includes the file iom8.h. Review this file so you know its contents.

With every available port comes 3 8-bit memory-mapped IO registers. These registers serve the following purpose:

Register	Purpose
PINx	Actual physical connection to pin. Use this register for reading!!
DDRx	Establish the direction of data. 1 for output, 0 for input.
PORTx	If DDR is set to 1, this is the output data. If DDR is set to 0, this indicates float or pull-up

The interaction between various settings of PORTx and DDRx are shown below:

	DDR bit = 1	DDR bit = 0
Port bit = 1	Output High	Input Pull-Up
Port bit = 0	Output Low	Input Floating

Sample code to read from and write to PORTC follows.

```
#include <io.h>

int main(void) {
    PINC = 0xAA;      //Bit pattern of 10101010, setting every other bit as input
    DDRC = 0xAA;      //Mimic the pattern of PINC to turn on the pull-up registers.

    //Two ways to read data
    volatile int read;
    read = PINC;       //Be careful to read only those bits which are inputs
    read = inb(PINC);  //Same as above

    //Two ways to write 1's to all of the output ports
    PORTC = 0xAA;
    outb(PORTC, 0xAA);

    return 0;
}
```

Thus, communication with the ARM can be fashioned via usage of the above commands. Note that your communications protocols will most likely have to poll certain bits of the cntl lines. This can be accomplished through while loops or through special macro's in the iom8.h. Note also that it is probably in your best interest to write a few macros of your own to expedite certain repeated code and to avoid little bugs that may result from accidentally writing to a wrong port.

Connecting the SRAM

First, place the SRAM chip above or below the AVR microcontroller (for ease of wiring). Refer to the SRAM datasheet as an aid for the rest of this section.

The table below shows the recommended connections to connect the SRAM to AVR. Note, that we are only using the first 9 address bits which limits our SRAM to be 4Kb due to restricted pin bandwidth on the AVR. Also, we dual use the AVR programmer lines for the same reasons.

AVR Connection	SRAM Connection
5V (Vcc) on breadboard	Vcc
Ground on breadboard	Vss
PD5	WE
PD6	OE
PD7	CE
PC0-PC5	A0-A5
PD2-PD4	A6-A8
Ground on breadboard	A9-A14
PB0-PB7	IO0-IO8

Software to Communicate with the SRAM

We provide you with some software. Look it over! Make and upload the sramtest.c. The outlook should look as follows:

```
Hello World
Starting SRAM Tests
erase completed successfully
AA write/read completed successfully
55 read/write completed successfully
SRAM successfully tested
```

Other Capabilities of the atmega8

The atmega8 has so many alternate functions available that it would be an amazingly time consuming task to create support for all of them. The chip has several PWM outputs, two internal timers, capture compare ports, and a real-time clock. The best way to learn about these features is to read the manual. The documentation for this chip is particularly thorough and well written and includes many example code blocks to quickly integrate and test functionality within your programs.

References

The atmega8 Data Sheet: http://www.atmel.com/dyn/resources/prod_documents/doc2486.pdf

The AVR ISA Reference Manual:
http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf

The MAX232 Data Sheet: <http://pdfserv.maxim-ic.com/en/ds/MAX202E-MAX241E.pdf>

Solderless Breadboard walkthrough: Solderless Breadboarding for Fun and Profit.doc
Getting Started with the TinyARM walkthrough: getting started with the TinyARM.doc