SJVALLEY
ENGINEERING

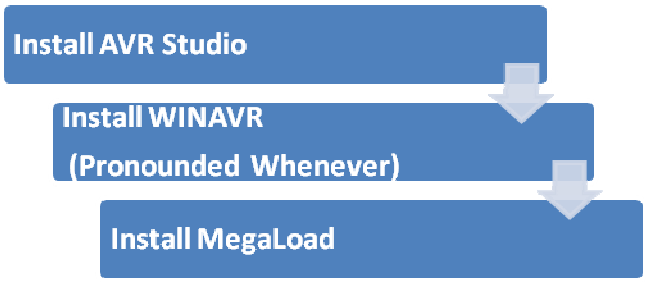
C PROGRAMMING IN AVR STUDIO

Your complete guide to C Programming on Atmel Microcontrollers using free, open-source Compiler for AVR |

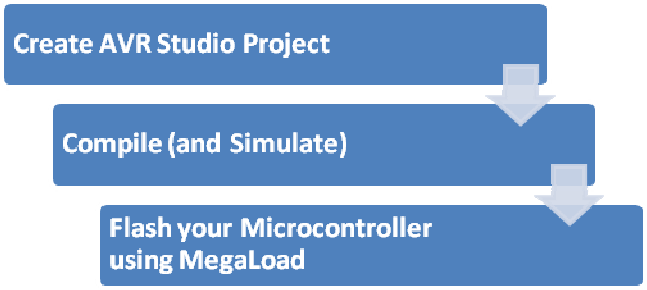
Table of Contents

Overview

To program in C using AVR Studio, first you need to install the necessary programs that will help you create the programming file:



Once you finish installing the programs, the following process is used to create a project and then load the programming file to your microcontroller.



Install Required Programs

To get started, install AVR Studio and WINAVR:

- AVR Studio: http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725
- WINAVR: http://winavr.sourceforge.net/download.html

Direct download links can be used from the following website, otherwise you will have to complete a free registration process under Atmel: http://www.societyofrobots.com/step_by_step_robot_step4.shtmlCreate AVR Studio Project

Install the BootLoader

All SJValley microcontroller kits come with a Bootloader preinstalled. Follow this section if you are having trouble programming or your microcontroller doesn't contain a Bootloader program. Installing the Bootloader is a one-time process that is performed such that your chip can program by itself. There is a one-time need of using a real-programmer (ISP programmer) and later the programs can be "flashed" to the microcontroller without this. Follow the

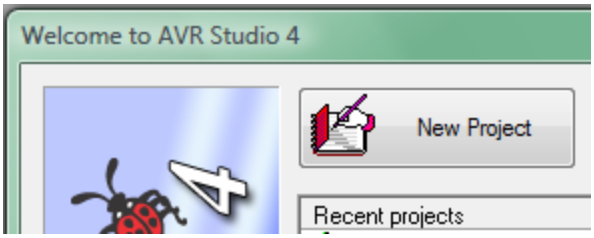
Flash Programming using MegaLoad section to load subsequent programs to your microcontroller.

Download the Bootloader tutorial from www.sjvalley.com by browsing to the Resources section. This separate tutorial will guide you on how to load the Bootloader software. Again, follow the Bootloader tutorial only if you are sure that your programming development kit didn't come preloaded with the Bootloader software.

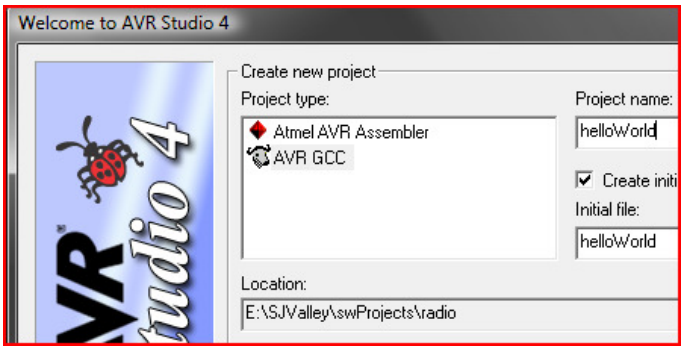
Create Project in AVR Studio

Follow the following Instructions:

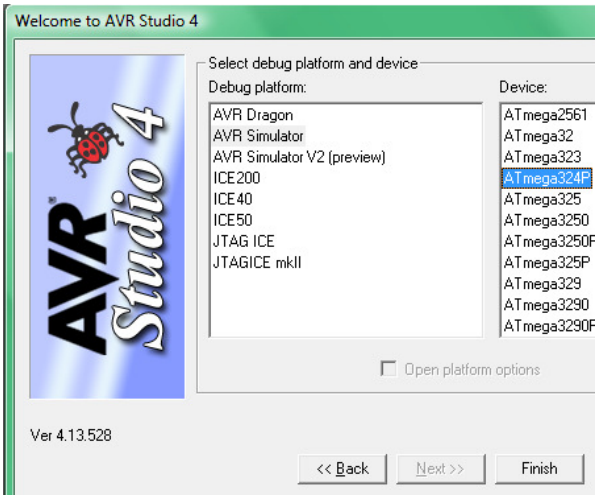
1. Open AVR Studio and click “New Project”



2. Choose a directory and name of your project. The directory path or the project name should not have a space.



3. Choose “AVR Simulator” and find your device. In the example below, Atmega324P is chosen:



4. Click “Finish” and your source code file will open up as an empty file.
5. Use the following “Hello World” Program of simple LED lights. Connect your PORTA to LED port and flash the program to test the basic board operation and programming operation.

```
#include <avr/io.h>
#include "serial.h"
#include <util/delay.h>
#include <stdio.h>

int main(void)
{
    initializeSerialBaudRate(9600);
    initializeSimpleRxTx();

    DDRA = 0xff;
    while(1) {
        PORTA = 0xaa;
        delay_ms(500);
        PORTA = 0x55;
        delay_ms(500);
    }
    return 1;
}
```

6. Follow the next section to configure your delay calls otherwise they will not be accurate. After writing the code above, press F7 to compile. Ensure there are no errors and then use MegaLoad to load your programming file to the board. The following screen shows successful build:

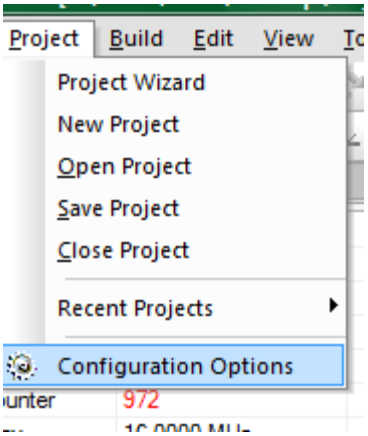
```
(.text + .data + .bootloader)

Data:        349 bytes (17.0% Full)
(.data + .bss + .noinit)


Build succeeded with 7 Warnings...
```

7. Locate programming hex file at your AVR Studio project directory to load it to your microcontroller.

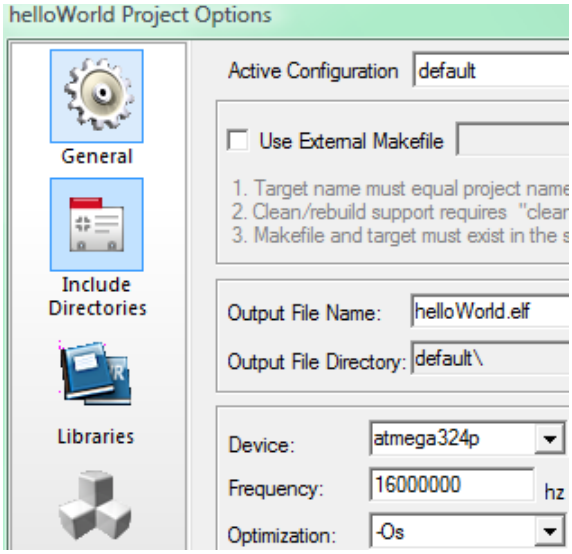
Configure delay function calls

The delay function in AVR Studio requires a couple of setup options. Follow the following steps to configure your delay functions:

1. Include <uitl/delay.h> in your program.
2. Go to Project Configuration



3. Specify your microcontroller frequency and turn on Compiler Optimizations to O1, O2, O3 or Os.



4. Os optimizes for size and O0 turns off optimization. O1 – O3 choose the optimization level in terms of speed. Choosing different optimization levels might let your program behave strangely ONLY when it is being debugged or simulated, but your program on the microcontroller should run perfectly.
5. Note that delay function parameter must be known at compile time to be accurate otherwise the delay time is not guaranteed. In other words, the delay functions are not dynamic delay functions.
6. Another important note is that delay_us() works up to 768uS/CPU_MHZ and delay_ms() works up to 262ms/CPU_MHZ so If you want to make your own delay that takes longer, try something like this:

```
void myDelay10ms(char count) {
        while(--count != 0)_delay_ms(10);
}
```

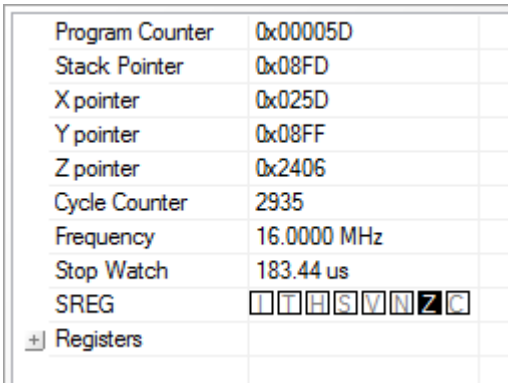
Debug and Simulate AVR Studio Project

Debugging your program is easy, but remember that AVR Studio doesn't support realtime debugging. For example, you cannot insert a breakpoint in the middle of a "live" program on your chip and bush a button on the programming board.

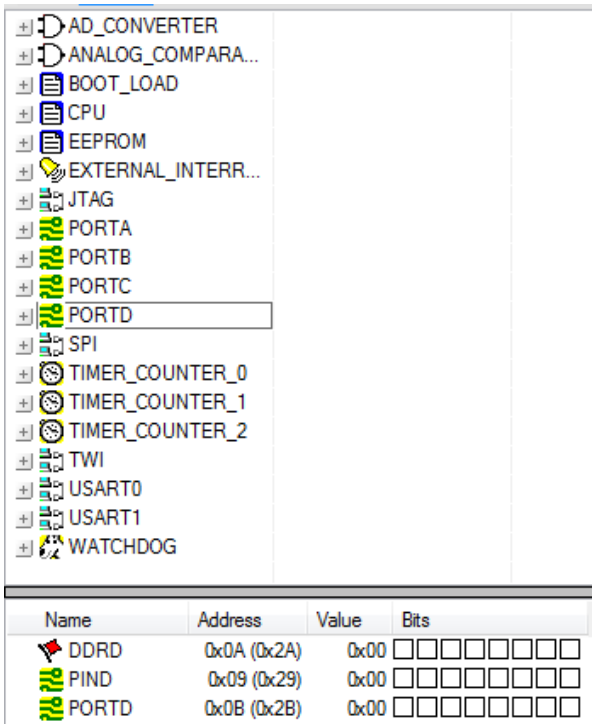
First, go to Debug → AVR Simulator options and specify your CPU frequency. Press F7 to compile your program then enter debugging mode by browsing to Debug → Start Debugging. A few warning notes are that if you use printf or similar functions that require a real-time flags to be set, then your debugger would hang at printf. Either use an interrupt-driven printf or comment out printf to debug other logic in your program.

The following is a walkthrough of the debugger.

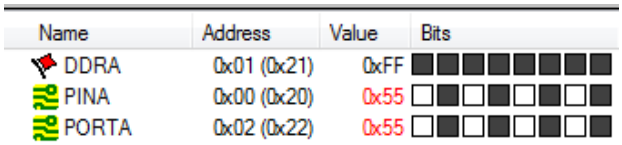
1. Enter Debugging mode (Debug → Start Debugging)
2. **You might want to disable compiler optimizations (-O0) because sometimes the code doesn't work in simulation if optimizations are on.**
3. In this example, the program is supposed to make DDRA = 0xff and then toggle PORTA
4. Note the left hand side pane that shows CPU speed, the timing and register contents:



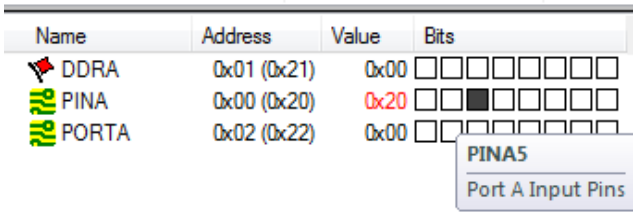
5. On the right side, you see your Atmega resources that you can modify:



6. Press F10 a few times and you will see PORT change on the right side:



7. Similarly, you can continue debugging your code by using options from debug menu. You can step into functions, step over functions etc. Try to use a few options from the Debug menu.
8. Another trick to consider is that if you wish to simulate you pressing a switch, expand the PORTx register and you can just click on any PINx to turn that pin level to High or Low. See the following screenshot:



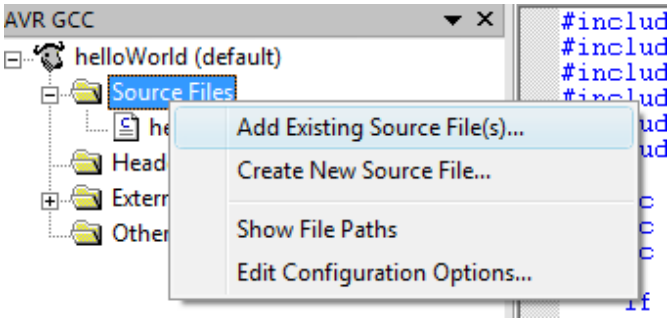
Similarly, you can manipulate other registers or interrupt flags to simulate an interrupt trigger.

Add Header and Source files to AVR Studio Project

1. Put the header and source file in the same directory as your project files.
2. #include your desired file:

```
#include <math.h>
#include "taskScheduler.h"
```

3. Add the source file to your project by right clicking on “Source Files”.

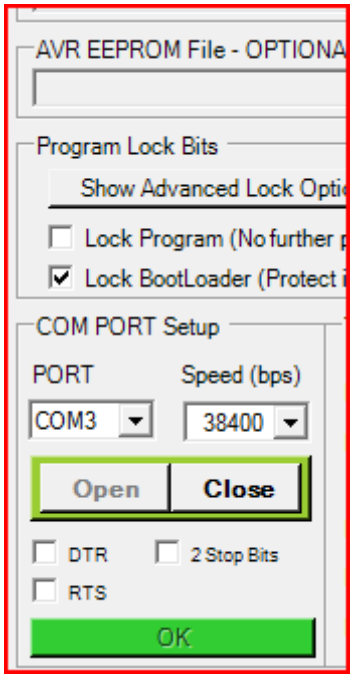


4. Add header files to your project by right clicking on “Header Files”.

Flash Programming using MegaLoad

After loading the BootLoader program to your microcontroller, the actual programming hex file can be loaded easily using the MegaLoad.Net program.

1. Compile your project in CodeVision or AVR Studio
2. Locate your program’s HEX file in your CodeVision or AVR Studio project directory
3. Connect your serial cable to microcontroller’s serial input which is PD0 and PD1. Reference your PCB manual for details on how to setup serial input and output.
4. Turn on your microcontroller.
5. Open MegaLoad.Net and click on “OPEN” for FLASH file and browse to the hex file from step 2. Do not open any EEPROM file.
6. Select your COM port and click “Open”



7. Your Atmega should have programmed automatically after the COM PORT opened up, if not, then hit the “reset” switch on the PCB.
8. To load new programming file, compile your project in your compiler and hit Atmega’s reset switch and MegaLoad.Net will automatically install the new file.
9. If the MegaLoad shows “Programming Failed” and doesn’t recover, your BootLoader is probably corrupted. You can reinstall the BootLoader to fix this problem.

References of MegaLoad:

http://www.microsyl.com/
http://www.imagecraft.com/

EEPROM in AVR Studio

Atmega chips contain three types of memory. The FLASH memory is ROM (Read-only memory) and this is where your program code resides. The contents can only be changed by BootLoader or a programmer. Contents in this memory remain on the chip across power-cycles which is why when your power-off and power-on your microcontroller, you do not need to reload the code.

The second memory type is SRAM. Simply-put, the SRAM contents are erased when the chip is power-cycled. This memory is used your program to execute your code. This finally gives way to EEPROM which inhibits characteristics from FLASH and SRAM. Contents written to this memory are preserved across power-cycles and the user can manipulate this memory easily as part of the program code.

Codevision compiler provides the simplest access to the EEPROM by declaring the variable as an eeprom type. All the EEPROM memory operations are performed "under the hood".

To use EEPROM in AVR Studio with WINAVR GCC, eeprom.h functions can be used. First, include the ***avr/eeprom.h*** file. Your chip datasheet contains the total memory size so do not address EEPROM outside of the available memory. There are functions at eeprom.h to read and write a byte or a word, but recommended function is write/read block, which works for any storage variable such as a char, an int, or even a structure.

- eeprom_read_block()
- eeprom_write_block()

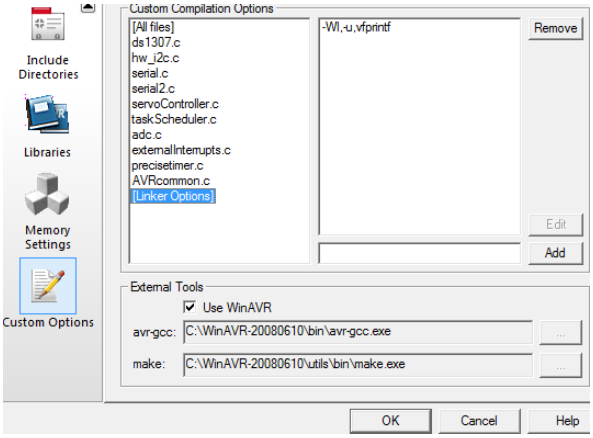
The variables that need to be written to EEPROM or read from the EEPROM need to declare their storage location as "EEMEM". This will help AVR Studio/WinAVR compiler organize the memory so you do not have to manage where things are stored in EEPROM memory. The address of these memories will be managed and given correctly to the eeprom functions. For example, when you use eeprom_write_block, you can pass along the address of the variable or structure to the function and the address will be unique and it will not overlap with other variables declared as EEMEM.

More functions can be located from WINAVR's online reference:
http://www.nongnu.org/avr-libc/user-manual/index.html

Here are some examples of reading and writing eeprom. Follow the syntax carefully otherwise your chip might keep resetting itself.

```
typedef struct TIME{
      unsigned char hour, min, sec;
}TIME;

// Declare storage areas for EEPROM
TIME EEMEM s_CURRENT;
unsigned int EEMEM s_myInt;
unsigned char EEMEM s_myString[50];

// Now declare actual variables
TIME CURRENT;
unsigned int myInt;
unsigned char myString[50];

int main(void) {

// To write EEPROM from SRAM
eeprom_write_block((const   void   *)&CURRENT,
(void *)&s_CURRENT, sizeof(CURRENT));
eeprom_write_block((const void *)&myInt, (void
*)&s_myInt, sizeof(myInt));
eeprom_write_block((const   void   *)&myString,
(void *)&s_myString, sizeof(myString));

// To read EEPROM back to SRAM
eeprom_read_block((void *)&CURRENT, (const void
*)&s_CURRENT, sizeof(CURRENT));
eeprom_read_block((void  *)&myInt,  (const  void
*)&s_myInt, sizeof(myInt));
eeprom_read_block((void   *)&myString,   (const
void *)&s_myString, sizeof(myString));


while(1);
}
```

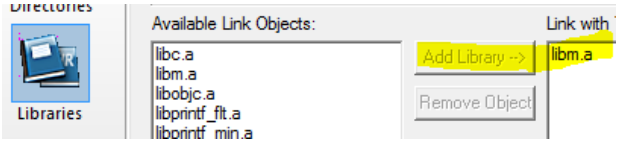
EEPROM should be read and written only occasionally. If you continue to read and write to EEPROM, the EEPROM memory may die because it has a limit of 100k R/w cycles. Usually, it should be read at the beginning of your program and written-back during your CPU shutdown or upon an occasional event.



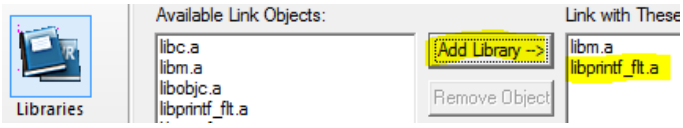
FAQ

- **AVR Studio Debugger doesn't track the code**
  Turn off Compiler optimizations

- **Math functions like sinA do not work.**
  Include the math library and then link the following library at Project → Configuration Options:



- **printf doesn't print floating point numbers.**
  Include the printf library that can printf the floating point numbers. Go to Project → Configuration Options and add the printf_ft.a



  Also, while you are at configuration options, go to "Custom Options" on the left side. Then click on "[LINKER OPTIONS]" and add the following statement: **-Wl,-u,vfprintf** and click OK. Now you can print floating point numbers.