Working with the AVR Assembly language

1. Assembly and C

Why work in Assembly?

The assembly code directly translates into AVR instructions to be executed by the microcontroller, without compiler or environment overhead. While performant C/C++ compilers may sometimes produce very efficient code in terms of speed or memory requirements, the assembly code gives the programmer full control over the binary code that is produced. Most of the time the assembly code is smaller, faster, more predictable in terms of time and memory requirements, and easier to debug.

Assembly directives

The assembly directives are not part of the assembly language (which is made by the opcodes for the instructions that can be directly executed by the target microprocessor), but they instruct the compiler in the process of code generation. Examples of uses:

- adjust the location of the program in memory
- define macros
- initialize memory

Some examples:

.byte	Reserve byte to a variable
.comm	declares a common symbol
.data	specifies the Data section of a program.
.ifdef	tells the assembler to include the following code if the condition is satisfied.
.else	tells the assembler to include the following code if the if condition is false
.include	include supporting files
#in-clude	include supporting files
.file	start of a new logical file
.text	assemble what follows after this directive, defines the code section
.global	defines or acknowledges a global variable or subroutine, often used with a variable specified by .comm, if the variable is used in more than one file
.extern	treats all undefined symbols as external
.space	allocates space (for an array)
.equ	define constants for use in a program
.set	define (local) variables used in a program

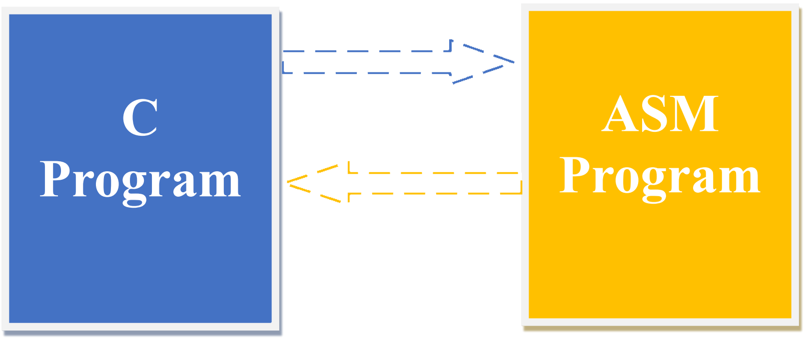
Remembering functions

If we want to combine the C/C++ language with the assembly language, we can do this by means of functions (procedures). Functions must be declared (in the function's prototype) and defined (in the function's body). The functions may or may not have inputs or outputs. Generally, if a variable is passed as an argument to a function, that variable is expected to remain unchanged when the function is executed. Global Variables may be changed inside a function.

Calling conventions determine where the arguments and the return values are stored. When a function is called, it needs to know where to look for the arguments, and where to store the return value.

The CALL of a function pushes the return address onto the stack, and the arguments and the return value are passed in accordance to the calling convention (in registers, on the stack, etc). If the function uses global variables, they need to be declared as such, and initialized with the proper values.

After the function is executed, it finishes with the RET instruction, which pops the return address off the stack.



Values are passed based on GCC convention
C functions can only “return” one value.

Functions and the stack

When a function is called, an activation record is “pushed” on the stack. When the function returns, that activation record is “popped” from the stack Activation record is whatever data needed to be remembered to resume the process when the function returns. It typically consists of local variables (which are usually stored in registers) and the return address. Stack is a memory area in SRAM pointed to by the 16 bit stack pointer (SP). At the AVR microcontrollers, the stack pointer is decreased when data is pushed on the stack, and therefore the stack must be initialized to the highest available data memory address (which at AtMega2560 is 0x21FF).

Global variable use

The global variables are accessible anywhere in your code, no matter if the code is C or assembly.

Global variables can be declared in the .ino file, but it is better to declare them in a header file. The globals should have equivalents in the .S assembly file (declared with the assembly directives .comm and .global)

Example declaration of the global variables in the .ino file, or in a header file:

```
extern "C" int8_t var8b;
extern "C" int16_t var16b;
extern "C" uint32_t var32b;
```

The declaration of these global variables in the assembly (.S) file:

```
.data
.comm var8b, 1
.global var8b
.comm var16b, 2
.global var16b
.comm var32b, 4
.global var32b
```

In C (Arduino) a global variable is just used as it is, whereas in ASM you may have to access one byte at a time.

Example:

Arduino:

```
void setup()
{
longvar = 0xAABBCCDD;
func1();
}
```

Assembly:

```
.align 2
.comm longvar, 4
.global longvar
.text
.global func1 ;
func1:
lds r18, longvar
lds r19, longvar+1
lds r20, longvar+2
lds r21, longvar+3
...
ret
```

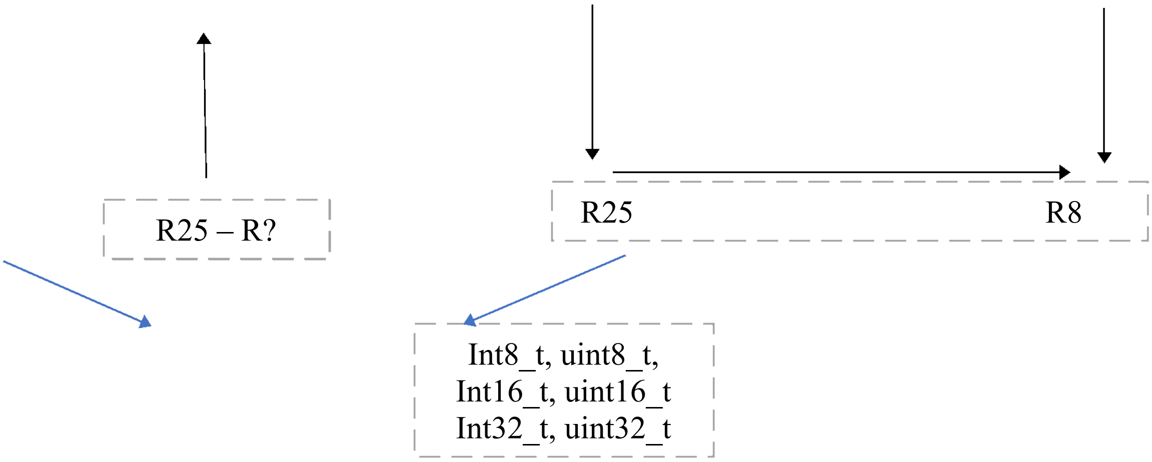
When working with more than 1 byte variables, the C/C++ compiler usually expect them to be aligned in memory. For example, a 2 byte variable must start at an address multiple of 2, while a 4 byte variable (such as the longvar above) must start at an address multiple of 4. The .align compiler directive, which receives as argument a power of 2 (2^2 = 4 in the example above) , ensures the alignment.

Parameter calling convention

The instructions call (two-word, jump farther) and rcall (one-word, jump shorter) cause the content of the PC register to become the address of the function being called. A call instruction also pushes the return address (the address of the next instruction after the call) on the stack. The instruction ret pops the return address off the stack and places it into the PC. The function parameters are stored in registers r25 … r8, the first byte being stored in r24. If the function has more parameters, and these registers are not enough, they are placed on the stack before the call is executed. The code inside the function must read them from the stack, and the caller code must remove them from the stack after the procedure is executed.

Function Prototype

(datatype) myfunc((datatype) var1, ...,(datatype) varn)



To access the parameters in the stack frame (activation record), you need to copy the stack pointer SP to the Y pointer register:

```
in r28, SPL
in r29, SPH
```

If we have a function with 11 1-byte arguments, the first nine will be passed in the even-numbered registers from r24 down to r8, and the last two will be passed on the stack. If we use the Y pointer for accessing these values, we must save it first, so the beginning of our function should look like:

```
push r28
push r29
in r28, SPL
in r29, SPH
```

You can access arguments 10 and 11 by:

```
ldd r7, Y+5
ldd r7, Y+6
```

The use of registers inside functions

The 32 registers of the AVR microcontroller are given various roles and treatment by the gcc compiler. If your project contains pure asm code, you can use the registers as you like. If you combine asm and C, you have to follow the rules described in the following table:

0x00	R0	“free” register, can be changed freely without need of restore

0x01	R1	Must always holds the value of 0, do not change.
0x02	R2	These must be left unchanged by a function or saved and restored before return.
…	…	
0x0D	R13	
0x0E	R14	
0x0F	R15	
0x10	R16	
0x11	R17	
0x12	R18	R18 to R27 are freely available for use in functions. You are to expect their values to be changed in a function.
…	…	
0x1A(XL)	R26	The X pointer, freely available to use in functions, not required to be saved.
0x1B(XH)	R27	
0x1C(YL)	R28	Frame pointer, Y. Can be used inside a function, but must be saved and restored before exiting.
0x1D(YH)	R29	
0x1E(ZL)	R30	The Z pointer, freely available to use in functions, not required to be saved.
0x1F(ZH)	R31	

Return values

The return value is passed in r25-r18, depending on the size of the return value (maximum return value size: 8 bytes). If the return value is 1 byte, it is placed in r24. r25 is either all 0's (positive return value) or all 1's (negative return value).

Declared Output	**Output location**
Byte, Boolean, int8_t, uint8_t	r24 (r25 = 00,FF)
int, uint, short, char, unsigned char int16_t, uint16_t	r25:r24
long, ulong, int32_t, uint23_t	r25:r22

Development tools

For developing assembly code that works with our Arduino Mega boards, we have the following options:

1. Using the Arduino IDE
   a. "Inline" Assembly – small pieces of assembly code inserted in the C++ code
   b. Assembly source files containing functions called from the .ino main file
2. Using Atmel Studio IDE

In this lab we will explore solutions 1.b and 2.

2. Using the Arduino IDE

a. *A simple blink*

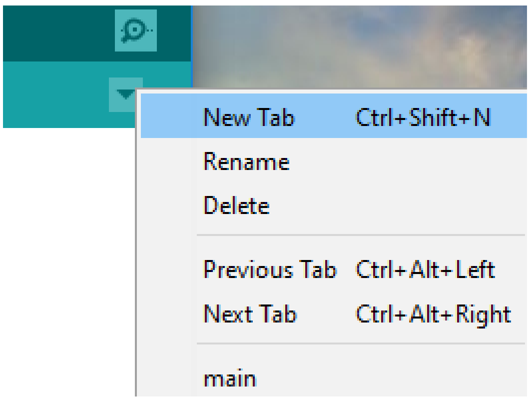
Combining assembly and C code using the Arduino IDE is pretty trivial. In the first example, we'll replicate the functionality of the first Arduino program, "Blink", by using assembly language functions for port bits manipulation.

Open the Arduino IDE and paste the following code:

```
extern "C" void setpin();
extern "C" void turnon();
extern "C" void turnoff();

void setup() {
        setpin();
}

void loop() {
        turnon();
        delay(1000);
        turnoff();
        delay(1000);
}
```

In the above code snippet, the functions setpin (which will configure pin 13 as output), turnon (which will turn the LED on), and turnoff (turn LED off) will be implemented in assembly.

To include assembly code, create a .S file by clicking in the upper right arrow in your IDE window, name it as you like, but don't forget to set its extension to ".S". Use CAPITAL S, not lowercase s, otherwise the compiler will not see the file.



Name this tab asm_functions.S. Paste the code snippet given bellow in this new file. The .S and .ino files have to be in the same folder.

```
#include "avr/io.h"

.global setpin

setpin:
  sbi _SFR_IO_ADDR(DDRB), 7 ; sets bit 7 of DDRB to 1 - output
  ret

.global turnon
```

```
turnon:
  sbi _SFR_IO_ADDR(PORTB), 7  ; sets bit 7 of PORTB to 1
  ret

.global turnoff
turnoff:
  cbi _SFR_IO_ADDR(PORTB), 7  ; sets bit 7 of PORTB to 0
  ret
```

The above code manipulates the value of bit 7 of port B, which is connected to digital pin 13 of the Arduino Mega board (see https://www.arduino.cc/en/Hacking/PinMapping2560 for other pin to port correspondences). First, the bit must be set to output by writing a ‘1’ to the corresponding position in DDRB, and then its value will be set by changing the bit in PORTB.

Warning: the Arduino compiler assumes that each port name/symbol refers to the Data Memory space address, and not to the I/O address. For example, for port B, we have from the datasheet two addresses: 0x05 in the I/O space, and 0x25 in the Memory space. In order to make the compiler use the I/O address, use the _SFR_IO_ADDR macro.

PORTB – Port B Data Register

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
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	

Compile the program and upload it to the board. It should perform the blinking function.

b. Using a function with parameters

We’ll try to achieve the same behavior as in the first example, but instead of using two functions, one for turning the LED on, and another for turning it off, we’ll use a single function and have the LED state passed as a parameter.

The c++ code will be changed to this:

```
extern "C" void setpin();
extern "C" char turnspecified(char c);

void setup() {
       setpin();
}

void loop() {
       turnspecified(1);
       delay(1000);
       turnspecified(0);
       delay(1000);
}
```

And the assembly code to this:

```
#include "avr/io.h"
```

```
.global setpin

setpin:
  sbi _SFR_IO_ADDR(DDRB), 7 ; sets bit 7 of DDRB to 1 - output
  ret

.global turnspecified
turnspecified:
  tst r24                  ; r24 will hold the parameter of the function, test it for zero
  breq set0                       ; if zero, go set the pin to 0
  sbi _SFR_IO_ADDR(PORTB), 7      ; otherwise set it to 1
  rjmp finish
 set0:
  cbi _SFR_IO_ADDR(PORTB), 7      ; set to zero
 finish:
  ret
```

The parameters of an assembly function will be passed in registers r25 down to r8. **A 8 bit parameter will be passed in register r24**, a 16 bit parameter in registers r25:r24, and so on.

c. Using the serial interface in Assembly

This example will display a message via the Serial interface. The message will be stored in the program flash memory as a null-terminated string of characters.

The Arduino c++ code is the following:

```
extern "C" void Serial_Setup();
extern "C" void Print_Hello();

void setup() {
  Serial_Setup();
}

void loop() {
  Print_Hello();
  delay(500);
}
```

The assembly code is this:

```
#include "avr/io.h"

.global Serial_Setup
Serial_Setup:

    ; Configure the parameters of serial interface 0
    clr r0
    sts   UCSR0A, r0
    ldi   r24, 1<<RXEN0 | 1 << TXEN0  ; enable Rx & Tx
    sts   UCSR0B, r24
    ldi   r24, 1 << UCSZ00 | 1 << UCSZ01  ; asynchronous, no parity, 1 stop, 8 bits
    sts   UBRR0H, r0
    ldi   r24, 103
    sts   UBRR0L, r24
    ret

.global Print_Hello
Print_Hello:
```

```
    ; load the starting address of the string in the Z pointer
    ldi ZL, lo8(the_message)         ; r30
    ldi ZH, hi8(the_message)         ; r31
    lpm r18, Z+             ; Load the first character of the string in r18

Loop:
    lds   r17, UCSR0A
    sbrs  r17, UDRE0        ; test the data buffer if data can be transmitted
    rjmp  Loop
    sts   UDR0, r18         ; send data contained in r18
    lpm r18, Z+             ; load the next character
    tst r18                 ; check if 0 - the string ends
    brne Loop
    ret

the_message:                ; the message itself, followed by LF and CR, and 0
.ascii "Assembly is fun"
.byte 10, 13,0
```

The first assembly function configures the parameters of the UART0 interface (the Serial interface of Arduino), and the second function sends a message stored in the program memory via this interface. Open the Serial Monitor tool to see the message being displayed.

Check the AVR ATMega2560 datasheet (http://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf) and the slides of Lecture 6, in order to understand the settings and operation of the UART interface.

d. Using C arrays in assembly functions

In this example we will write a function in assembly that adds the elements of an array, which is declared in the main Arduino file. The function is called form the Arduino code. Create three files: sum_array.ino, external_functions.h, arsum.S. The contents of these files are detailed below.

.ino file

```
#include "external_functions.h"

void setup() {
compute();
uint8_t val = result;
Serial.begin(9600);
Serial.println(val);
}
void loop() { }
```

external_functions.h file

```
#include <stdint.h>
extern "C" uint8_t result;
extern "C" void compute(void);
extern "C" uint8_t myarray[10]={1, 30, 3, 4, 5, 6, 7, 8, 10, 11};
```

arsum.S file

```
.file "arsum.S"
.data
.comm result, 1
.global result

.text
.global compute

compute:
      ldi r30, lo8(myarray)
      ldi r31, hi8(myarray)
      ldi r18, 0
      ldi r21, 0

looptest:
      ld r22, z+
      add r21, r22
      inc r18
      cpi r18, 10
      brlo looptest

out:
    sts result, r21
    ret
```

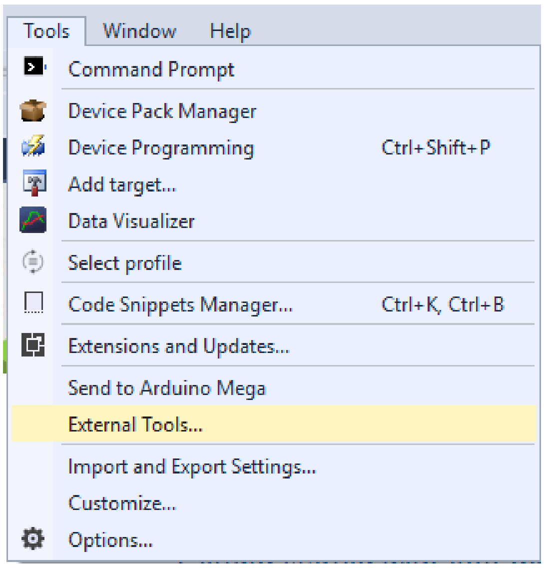
3. Using Atmel Studio

Setting up Atmel Studio

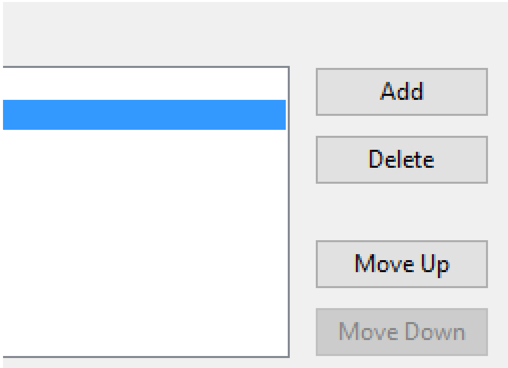
Atmel Studio 7 is the integrated development platform (IDP) for developing and debugging multiple microcontroller applications (including AVR). The Atmel Studio 7 IDP gives you an easy-to-use environment to write, build and debug your applications written in C/C++ or assembly code. It also connects together the debuggers, programmers and development kits that support AVR devices.

First of all, we have to set up the environment, in order to be able to upload the code on our Atmega2560 board.

1. Open up Atmel Studio 7
2. Go to the **Tools** menu
3. Select **External Tools**



4. Click the Add button from the window



5. A new entry will appear. Fill in the following data in the corresponding input text boxes.
   Title: Send to Arduino Mega
   Command : C:\Program Files (x86)\Arduino\hardware\tools\avr\bin\avrdude.exe
   Arguments: -v -C"C:\Program Files (x86)\Arduino\hardware\tools\avr\etc\avrdude.conf" -p atmega2560 -c wiring -P COM5 -b 115200 -D -U flash:w:$(TargetDir)$(TargetName).hex:i

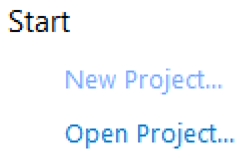
Please take note that this setting assumes that the Arduino tools are installed in C:\Program Files (x86). If Arduino is installed somewhere else, replace this folder with the correct path. If you have installed Arduino as a Windows 10 app, uninstall it and install it normally, otherwise Atmel Studio will not work with it.
Please note that in the Arguments string the COM port is written as a constant. Please check the port of your ATMega board, and replace the above red highlighted text with the correct COM port.

Please ensure that the “Use Output Window” check box is checked, press Apply and Ok.
After completing the above steps, a menu option “Send to Arduino Mega” will appear in the Tools menu.

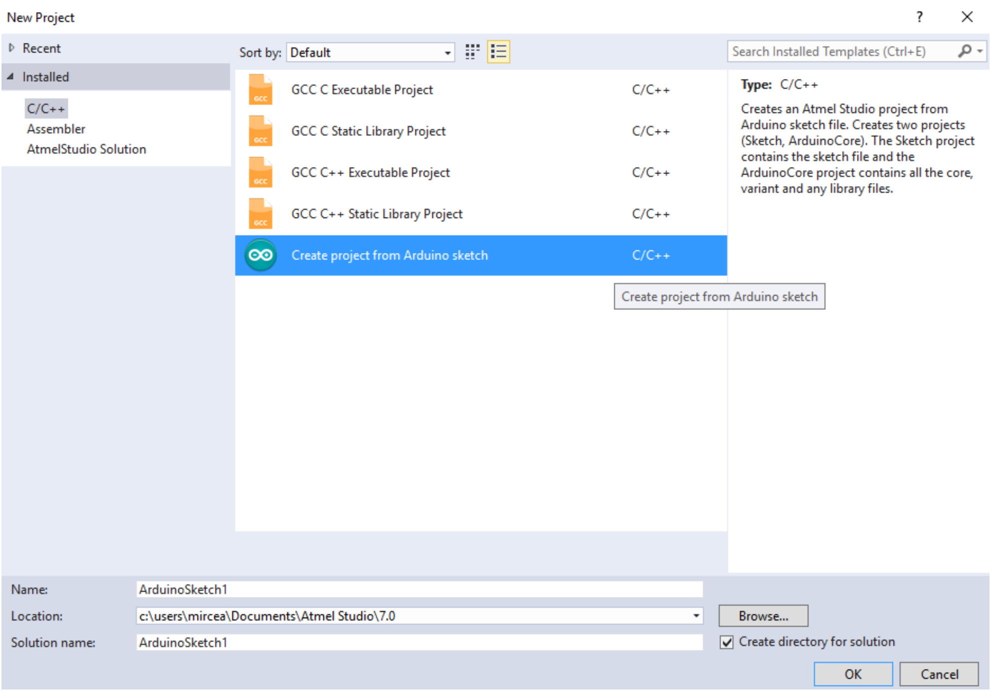
Creating a project from an Arduino sketch

Atmel Studio allows us to transform an Arduino project (sketch) into a Studio project. You have to perform the following steps:

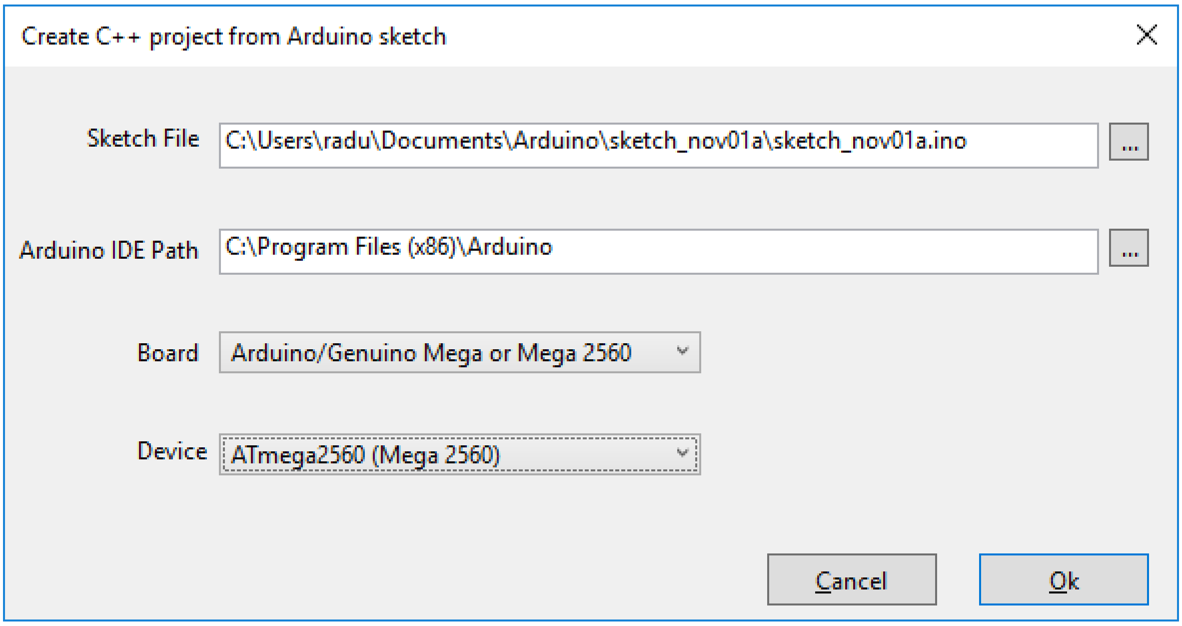
1. In Atmel Studio click on new project.



2. Click on the Create project from Arduino sketch and fill in the desired name, location and solution name.

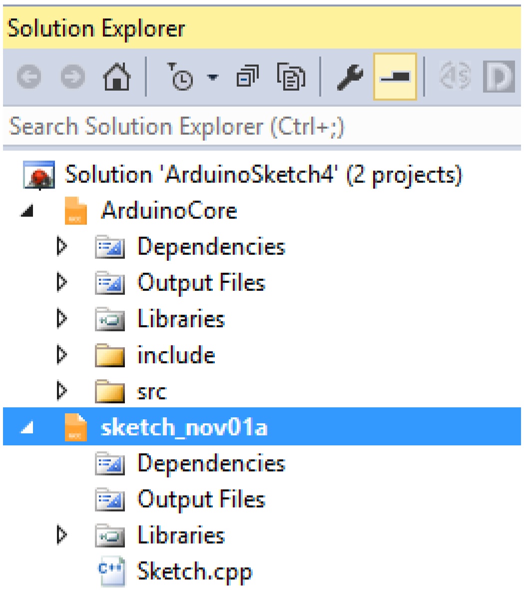


3. In the new window browse to the location of your Arduino sketch, select the path of your IDE and the development board used and press Ok.

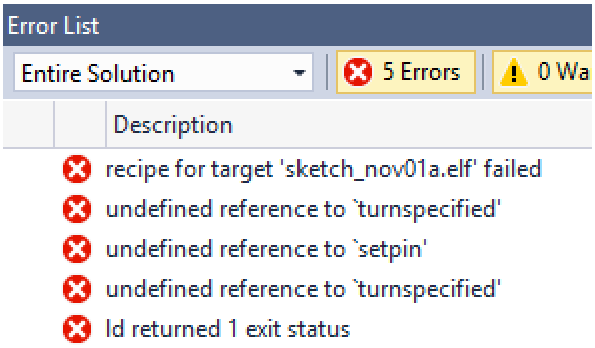


In this example, we will use the second blink program written in assembly and c, which uses an asm function to set the state of the LED pin 13.

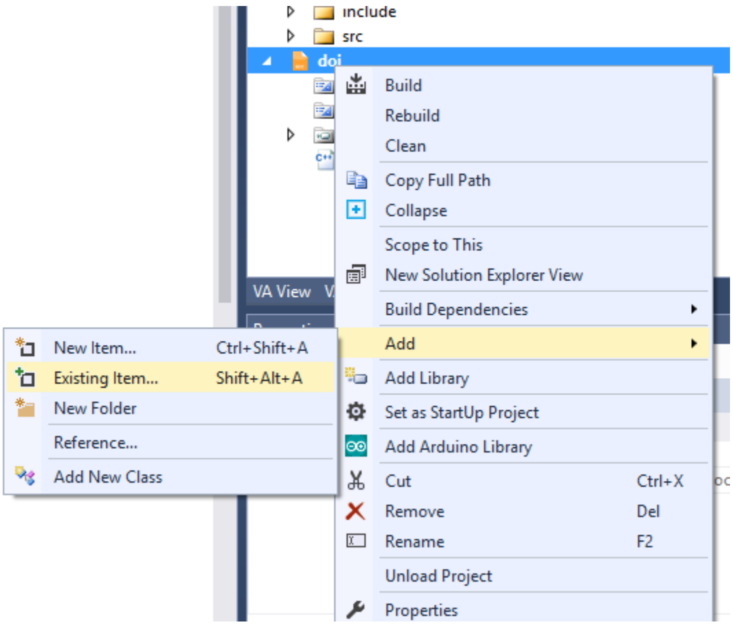
After importing the Arduino sketch the solution tree looks like in the image bellow.



If we will compile the program we will see that we get the following errors.



This happens due to the fact that Atmel Studio does not add automatically the references to any external library. To solve this issue right click on the project name and select Add Existing Item from the menu.



Browse to the location of the assembly file asm_functions.S in the Arduino folder where you have created them, and add the file to the solution.

Rebuild the solution and observe that the build succeeds this time.

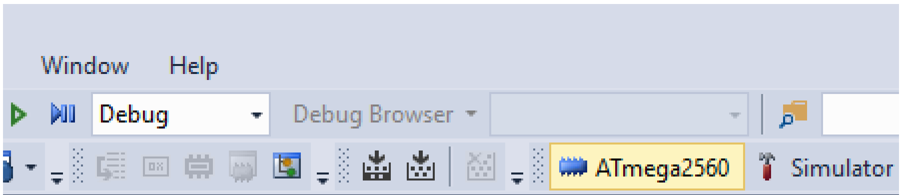
In order to upload the program to the board click on the Tools menu and select Send to Arduino Mega. If you have implemented the above steps correctly the program should upload on the Arduino mega board without any issue. You will see the following message in the output window in case the program is uploaded successfully.

```
Reading | ################################################## | 100% 0.13s

avrdude.exe: verifying ...
avrdude.exe: 882 bytes of flash verified

avrdude.exe: safemode: lfuse reads as FF
avrdude.exe: safemode: hfuse reads as D0
avrdude.exe: safemode: efuse reads as FF
avrdude.exe: safemode: Fuses OK (E:FF, H:D0, L:FF)

avrdude.exe done.  Thank you.
```

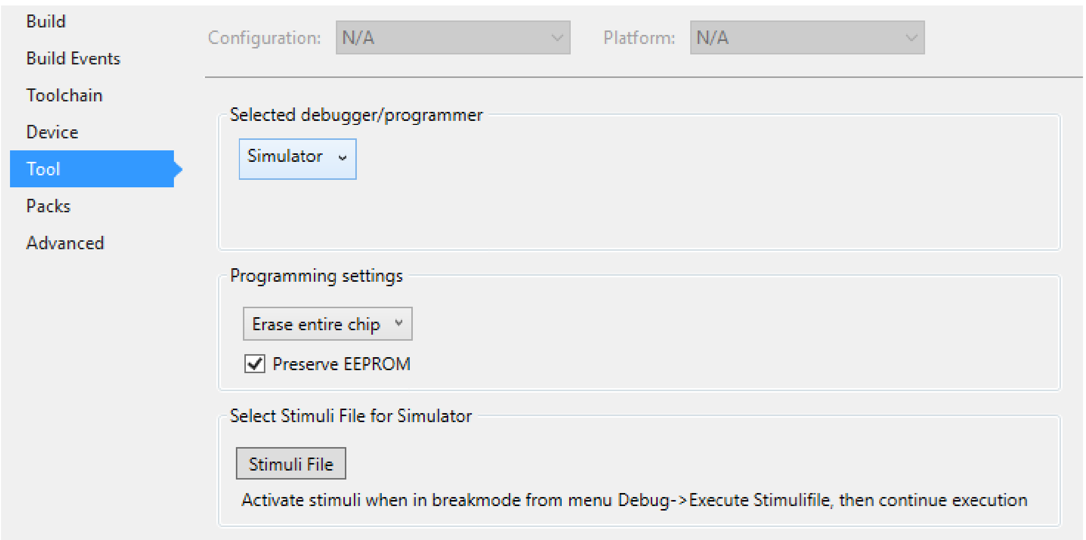
Debugging using Atmel Studio

A powerful feature of Atmel Studio is the simulation-based debugger. This debugging mechanism allows us to analyze the program behavior, monitor the registers, ports and memory, even without having a development board near us.

To set up the debugging environment first select the **Debug** option from the roll down menu and the **ATmega2560** board from the main menu strip.



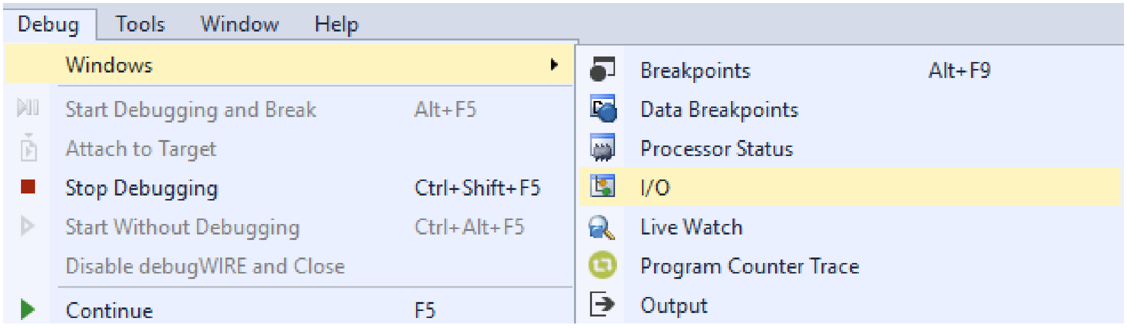
After pressing the ATmega2560 option, a new window will appear. Select the **Tool** option and from the **Select debugger / programmer** option select the **Simulator** option.



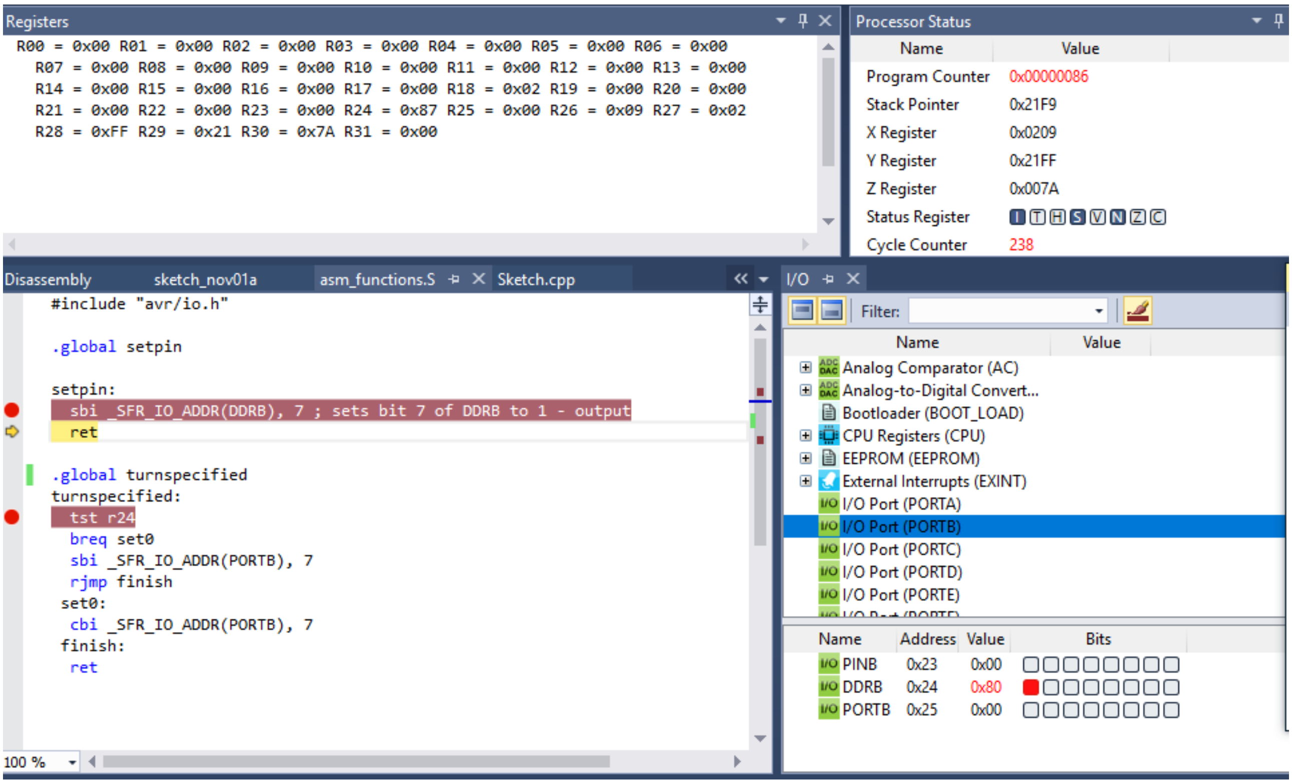
After setting up the specified options, come back to the main program by clicking the program sketch tab. We can use the debugger to analyze the behavior of the blinking program that we have already imported from Arduino. For debugging, it is recommended that you comment out the delays in the code, as they will take a very long time in simulation.

You can set breakpoints by clicking on the left gray part near the line where you want to include the breakpoint or by positioning on a line and from the **Debug** window select the **Toggle Breakpoint** option (or by pressing F9). You can set breakpoints in the c++ file, or in the assembly file.

To start debugging press the button near the Debug drop down menu or press the Alt+F5 button combination then press F5 once. To view the available AVR input/output registers (ports), from the **Debug** menu select **Windows** and then **I/O.** From the same Debug/Windows menu you can select to view other information, such as Registers (content of the 32 registers), Memory (content of the program flash, data memories, EEPROM), Disassembly, and so on.

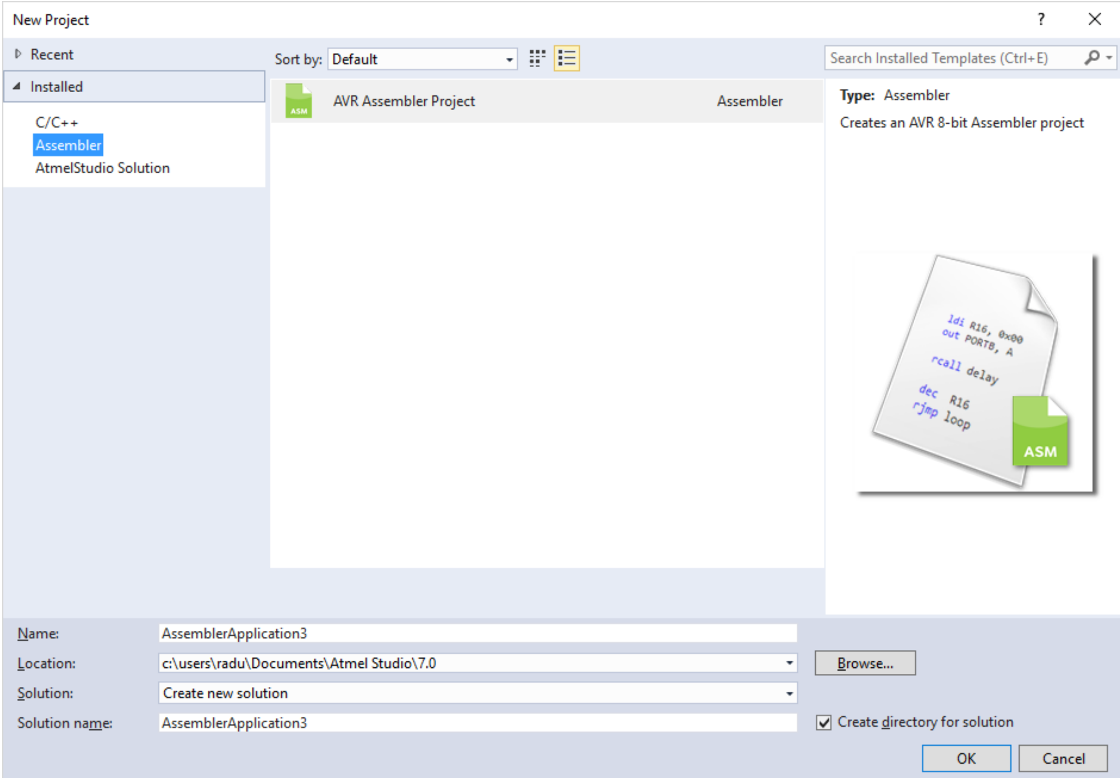


From the I/O window, select PORTB. You will see, at each step of the program, the contents of the associated registers, DDRB, PORTB, and the input PINB.



Working with assembly-only projects

Atmel Studio allows you to write assembly-only solutions. From the **File** menu, select **New Project**. When the New Project window opens, select from the left panel the “Assembler” option, as shown in the figure below. Name your project, and click OK.

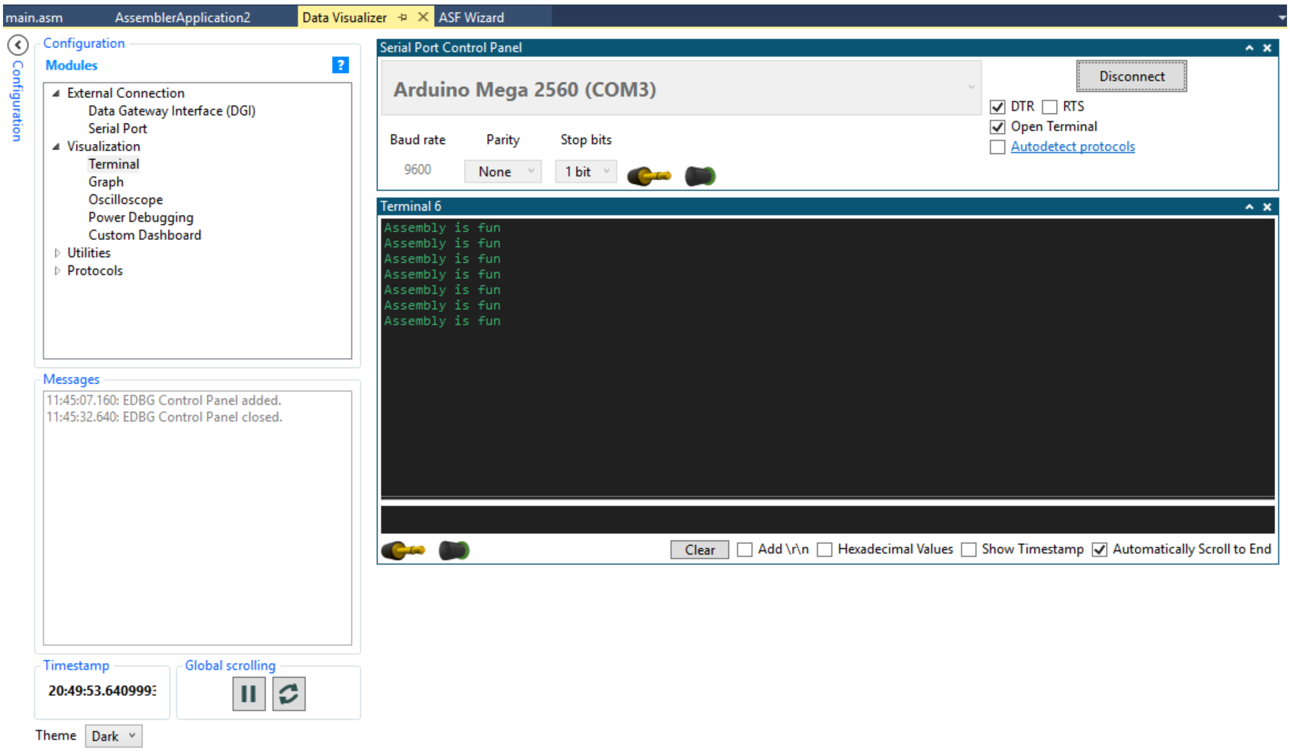


The environment will generate a main.asm file, with a dummy asm code. Replace this code with the one below, which will send to the serial interface the message “Assembly is fun”:

```
; Main program
main:
    rcall asm_setup

main_loop:
    rcall asm_loop
    rjmp main_loop


asm_setup:
; Init the serial interface
    clr r0
    sts   UCSR0A, r0
    ldi   r24, 1<<RXEN0 | 1 << TXEN0     ; enable Rx & Tx
    sts   UCSR0B, r24
    ldi   r24, 1 << UCSZ00 | 1 << UCSZ01 ; asynchronous, no parity, 1 stop, 8 bits
    sts   UBRR0H, r0
    ldi   r24, 103
    sts   UBRR0L, r24
    ret

asm_loop:
    ; print and wait
    rcall Print_Hello
    rcall wait
```

```
    ret


Print_Hello:

    ; loading address and size of array
    ldi ZL, LOW(2*array)         ; r30
    ldi ZH, HIGH(2*array)        ; r31
    lpm r16, Z+                  ; Load the character pointed by Z registers (r30/r31)

Loop:
    lds   r17, UCSR0A
    sbrs  r17, UDRE0              ; test the data buffer if data can be transmitted
    rjmp  Loop
    sts   UDR0, r16               ; send data contained in r16
    lpm r16, Z+                   ; point to the next character
    tst r16                       ; check for string end - 0
    brne Loop
    ret


; simple function to wait for aprox 1 second by idle counting
wait:

   ldi  R17, 0x53
  LOOP0:  ldi  R18, 0xFB
  LOOP1:  ldi  R19, 0xFF
  LOOP2:  dec  R19
   brne LOOP2
   dec  R18
   brne LOOP1
   dec  R17
   brne LOOP0
   ret

; string to be written, stored in the program memory
array:
.db "Assembly is fun",13,10,0
```

Build your solution using the Build menu, and send it to the Arduino Mega board. For seeing the serial output, you can use either the serial monitor of Arduino, or you can use the terminal of Atmel Studio. For this, from the Tools menu, select Data Visualizer. From the left panel of the tool, select Visualization/Terminal, and from the central panel select the serial port of your board and click Connect. The terminal should open and display your message, as shown in the following figure.



You can also use the debugger to analyze the step by step execution of assembly programs. The steps to be performed are the same as in the case of c/c++ projects.

Individual work:

1. Implement the examples provided in this lab. Use the debugger as often as possible, to see the behavior of the program. Can you see the message string in the program flash memory?
2. Using the datasheet and the information contained in Lecture 6, write a document explaining the settings and the operation of the serial interface as shown in the third example.
3. Write an assembly function that will return a value (the return values of functions start with registers r25:r24). Write the .ino program that will call this function and use its output. *Hint: you can read a port.*
4. Modify the example that displays the "Arduino is fun" message to display any string (char array, null terminated) declared in the c++ program. *Hint: the string is stored in the data memory.*
5. Analyze the assembly code for serial communication of the Arduino project, and compare it to the assembly code of the pure assembly project. Describe the differences and similarities.

References

https://docslide.us/documents/lecture-12-5600350816ac8.html
https://forum.arduino.cc/index.php?topic=490065.0
https://www.youtube.com/watch?v=8yAOTUY9t10