APPENDIX C FOR LABORATORY 1 SHEET

Using Atmel Studio to Program in C Language

Other documents that are referred within this document are located in the link
https://www.dropbox.com/sh/s16jri4eol3agl5/AAAzn_W3p7F0dJS-W1-XCnNqa?dl=0

C.1. The AVR-Libc

The Avr-libc is the Standard C Library project for the AVR platform on the Windows Operating System for AVR Studio. AVR-Libc provides many of the same functions found in a regular Standard C Library and many additional library functions that is specific to an AVR. Some of the Standard C Library functions that are commonly used on a PC environment have limitations or additional issues that a user needs to be aware of when used on an embedded system. AVR-Libc also contains the most documentation about the whole AVR toolchain.

While using and abiding ANSI C/C++ programming Language, avr-libc has keywords and function that support the specific ATMELE device that is selected when a C/C++ project is or created or opened. The Avr-libc recognise keyword of programming modes/identifications/names that are declared in the respective reference Manual (like Register name, location names, bitwise location name e.g PORTA, PINB, DDRA7 or TIMER0) that are valid in AVR Assembler.

Datatypes

Atmel Studio C/C++ compiler (GCC) supports the following standard ANSI datatypes:

- (signed/unsigned) char - 1 byte
- (signed/unsigned) short - 2 bytes
- (signed/unsigned) int - 2 bytes
- (signed/unsigned) long - 4 bytes
- (signed/unsigned) long long - 8 bytes
- float - 4 bytes (floating point)
- double - alias to float

Alternate datatypes outlined in the C99 standard, and made available to GCC. These use the convention of a "u" to denote the signedness (no "u" to denote signed), "int" to denote that it's a integer and not a float, the number of bits in the int and a trailing ".t".

Examples:

```
int8_t - signed char
uint8_t - unsigned char
int16_t - signed int
uint16_t - unsigned int
uint32_t - unsigned long
int32_t - signed long
uint64_t - unsigned long long
```

In Laboratory 1 we will use standard ANSI datatypes but in Laboratory 2 we will use C99 standard datatypes.

Using “volatile” qualifier when declaring variables.

However, when declaring variable it is necessary in certain condition (there are conditions where it can be ignored) to add the “volatile” qualifier to the datatype declared or else the program may accessing an invalid data. The following as an example for the reason why (excerpts from http://stackoverflow.com/questions/4437527/why-do-we-use-volatile-keyword-in-c site):

Consider this code,

```
int some_int = 100;
while(some_int == 100)
{
  //your code
```

Appendix C For SKEE3732 Laboratory 1 Sheet (Rev12072017).docx (changes on Page 1 highlighted in Red font)
When this program gets compiled, the compiler may optimize this code, if it finds that the program never makes any attempt to change the value of some_int, so it may be tempted to optimize the while loop by changing it from while(some_int == 100) to simply while(true) so that the execution could be fast (since the condition in while loop appears to be true always). *(if the compiler doesn't optimize it, then it has to fetch the value of some_int (if it's not loaded on a register) and compare it with 100, each time which obviously is a little bit slow.)*

However, sometimes, optimization (of some parts of your program) may be undesirable, because it may be that someone else is changing the value of some_int from outside the program which compiler is not aware of, since it can't see it; but it's how you've designed it. In that case, compiler's optimization would not produce the desired result!

So, to ensure the desired result, you need to somehow stop the compiler from optimizing the while loop. That is where the volatile keyword plays its role. All you need to do is this, volatile int some_int = 100; //note the 'volatile' qualifier now!

In other words I would explain this as follows:

volatile tells the compiler that,

"Hey compiler, I'm volatile and, you know. I can be changed by some XYZ that you're not even aware of. That XYZ could be anything. Maybe some alien outside this planet called program. Maybe some lighting, some form of interrupt, volcanoes, etc can mutate me. Maybe. You never know who is going to change me! So O you ignorant, stop playing an all-knowing god, and don't dare touch the code where I'm present. Okay?"

Well, that is how volatile prevents compiler from optimizing code.

Using “static” qualifier when declaring variables.

In some case too the “static” qualifier is needed to be added to the volatile qualifier of a variable, if during a single-step command of a debugging process, a certain statement in the program omits a statement that uses the particular variable! Example as follows:

```c
static volatile int some_int = 100
...
some_int++; //without the “static” qualifier, this statement is omitted
```

The Avr-libc provides pre defined libraries that in play very important role in compiling a program and significantly reduce the code size for the same. Complete AVR Libc Reference Manual can be found at [http://www.atmel.com/webdoc/AVRLibcReferenceManual/ch20.html](http://www.atmel.com/webdoc/AVRLibcReferenceManual/ch20.html). The following are some examples:

**a. avr/io.h**

Even a single line of code would require this particular header file. It resolves the problem of handling the registers and provides a convenient way to treat the registers as variables. This makes it simple to assign values to them. E.g. to write data into port B data direction register, the register can be addressed using the variable ‘DDRB’. In code, you can write an expression like this one:

```c
DDRB = 0xff; // 0x prefix stands for an integer represented in hexadecimal format in C
```

You can treat all other valid I/O register names of the device selected in the project (refer Figure A.5 (b) on in APPENDIX A FOR SKEE3732 LABORATORY 1 SHEET) in similar fashion. There are some exceptions and complexities in some cases. But such things are beyond the scope of this Laboratory.
b. util/delay.h

In many microcontroller programs, delay loop is essential. This header file defines 2 delay
loops.

```c
void _delay_ms (double __ms) //Perform a delay of __ms milliseconds.
```

The macro F_CPU is supposed to be defined to a constant defining the CPU clock
frequency (in Hertz).

The maximal possible delay is 262.14 ms / F_CPU in MHz.

When the user request delay which exceed the maximum possible one, _delay_ms() provides
a decreased resolution functionality. In this mode _delay_ms() will work with a
resolution of 1/10 ms, providing delays up to 6.5535 seconds (independent from CPU
frequency). The user will not be informed about decreased resolution.

```c
void _delay_us (double __us) //Perform a delay of __us microseconds
```

The macro F_CPU is supposed to be defined to a constant defining the CPU clock
frequency (in Hertz).

The maximal possible delay is 768 us / F_CPU in MHz.

If the user requests a delay greater than the maximal possible one, _delay_us() will
automatically call _delay_ms() instead. The user will not be informed about this case.

Example: Using _delay_us(DELAY_TIME)

It is a basic function to create a delay of ‘DELAY_TIME’ micro seconds. They are
implemented with basic stop and wait delay loop. Variables are not allowed to use as an
argument, instead you can use predefined constants. And to use this loop, you need to
specify the CPU operating frequency (use: #define F_CPU).

```c
#include util/delay.h
int main()
{
    #define F_CPU 1000000
    _delay_us(29);
    //-
}
```

or

```c
#include util/delay.h
int main()
{
    #define F_CPU 1000000
    #define DELAY 35
    _delay_ms(DELAY);
    //-
}
```

Example: Using _delay_ms(DELAY_TIME)

This loop is same as previous. Just one difference, it creates delay in milli seconds.
Rest of the facts which are applicable for ‘_delay_us()’, are applicable for
‘_delay_ms()’ too. E.g.
```c
#include avr/io.h
#include util/delay.h

int main()
{
    #define F_CPU 1000000
    DDRB=0x0f;
    PORTB=0x05;
    while(1)
    {
        _delay_ms(1000);
        PORTB ^= PORTB;
        //...
    }
}
```
c. `avr/interrupt.h` (will be used in Lab 2)

A micro controller has several **interrupt sources**. Each of them has separate **interrupt sub-routine**. In **ANSI C**, there are no interrupt handling schemes. But for micro controllers, interrupts are a matter of special significance! Many programs are very much dependent upon it! So to help users to implement subroutine codes more easily, there is a header file `avr/interrupt.h` It defines some functions and macros described below.

**sei()**

This function enables the global interrupt by setting the global interrupt mask.

**cli()**

This function disables the global interrupt by resetting the global interrupt mask.

**reti()**

Enables interrupts by setting the global interrupt mask. This function compiles into a single line of assembly code.

**ISR (INTERRUPT_vect)**

ISR stands for Interrupt Sub Routine. Using this macro, users can write up interrupt sub routine associated with interrupts ‘INTERRUPT’. In the place of the argument of the macro, some symbols are supplied. Here, symbols are named after the interrupt vectors they are representing. For a particular micro controller, some specific symbols are valid. For them, look at the **AVR GCC reference manual** that comes with the AVR Studio. Let’s see an example:

```c
#include avr/io.h
#include avr/interrupt.h
ISR(INT0_vect)
{
    PORTB = ~PORTB;
}
void initInterrupt(void)
{
    cli();
    GICR=0x40;
    MCUCR=0x03;
    sei();
}
int main()
{
    initInterrupt();
    DDRB=0xff;
    PORTB=0x55;
}
```
Table C.1(c). AtmelStudio Interrupt Vector Keywords for ATmega32/ATmega32A Vector Table (Related to Table 18 of “Atmega32 Reference manual.pdf”).

<table>
<thead>
<tr>
<th>Vector No.</th>
<th>Address</th>
<th>Source</th>
<th>Name</th>
<th>Location</th>
<th>ATmel Studio C/C++ Symbol Names Keywords</th>
<th>Interrupt Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$000</td>
<td>On Chip’s Pin</td>
<td>Reset</td>
<td>No needed since it is by default assigned by C/C++ compiler referencing to main() function</td>
<td>External Pin, Power - on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$002</td>
<td></td>
<td>INT0</td>
<td>INTO_vect</td>
<td></td>
<td>External Interrupt Request 0</td>
</tr>
<tr>
<td>3</td>
<td>$004</td>
<td></td>
<td>INT1</td>
<td>INT1_vect</td>
<td></td>
<td>External Interrupt Request 1</td>
</tr>
<tr>
<td>4</td>
<td>$006</td>
<td></td>
<td>INT2</td>
<td>INT2_vect</td>
<td></td>
<td>External Interrupt Request 2</td>
</tr>
<tr>
<td>5</td>
<td>$008</td>
<td></td>
<td>TIMER1 COMPA</td>
<td>TIMER1_COMPA_vect</td>
<td></td>
<td>Timer/Counter1 Compare Match A</td>
</tr>
<tr>
<td>6</td>
<td>$00A</td>
<td></td>
<td>TIMER2 OVF</td>
<td>TIMER2_OVF_vect</td>
<td></td>
<td>Timer/Counter2 Overflow</td>
</tr>
<tr>
<td>7</td>
<td>$00C</td>
<td></td>
<td>TIMER1 CAPT</td>
<td>TIMER1_CAPT_vect</td>
<td></td>
<td>Timer/Counter1 Capture Event</td>
</tr>
<tr>
<td>8</td>
<td>$00E</td>
<td></td>
<td>TIMER2 COMP</td>
<td>TIMER2_COMP_vect</td>
<td></td>
<td>Timer/Counter2 Compare Match</td>
</tr>
<tr>
<td>9</td>
<td>$010</td>
<td></td>
<td>TIMER1 COMPB</td>
<td>TIMER1_COMPB_vect</td>
<td></td>
<td>Timer/Counter1 Compare Match B</td>
</tr>
<tr>
<td>10</td>
<td>$012</td>
<td></td>
<td>TIMER1 OVF</td>
<td>TIMER1_OVF_vect</td>
<td></td>
<td>TIMER1 OVF Timer/Counter1 Overflow</td>
</tr>
<tr>
<td>11</td>
<td>$014</td>
<td></td>
<td>TIMER0 COMP</td>
<td>TIMER0_COMP_vect</td>
<td></td>
<td>Timer/Counter0 Compare Match</td>
</tr>
<tr>
<td>12</td>
<td>$016</td>
<td></td>
<td>TIMER0 OVF</td>
<td>TIMER0_OVF_vect</td>
<td></td>
<td>Timer/Counter0 Overflow</td>
</tr>
<tr>
<td>13</td>
<td>$018</td>
<td>Status bit on respective I/O Register which is transparent to programmer</td>
<td>SPI, STC</td>
<td>SPI_STC_vect</td>
<td></td>
<td>Serial Transfer Complete</td>
</tr>
<tr>
<td>14</td>
<td>$01A</td>
<td></td>
<td>USART, RXC</td>
<td>USART_RXC_vect</td>
<td></td>
<td>USART Rx Complete</td>
</tr>
<tr>
<td>15</td>
<td>$01C</td>
<td></td>
<td>USART, UDRE</td>
<td>USART_UDRE_vect</td>
<td></td>
<td>USART Data Register Empty</td>
</tr>
<tr>
<td>16</td>
<td>$01E</td>
<td></td>
<td>USART, TXC</td>
<td>USART_TXC_vect</td>
<td></td>
<td>USART, Tx Complete</td>
</tr>
<tr>
<td>17</td>
<td>$020</td>
<td></td>
<td>ADC</td>
<td>ADC_vect</td>
<td></td>
<td>ADC Conversion Complete</td>
</tr>
<tr>
<td>18</td>
<td>$022</td>
<td></td>
<td>EE RDY</td>
<td>EE_RDY_vect</td>
<td></td>
<td>EEPROM Ready</td>
</tr>
<tr>
<td>19</td>
<td>$024</td>
<td></td>
<td>ANA_COMP</td>
<td>ANA_COMP_vect</td>
<td></td>
<td>Analog Comparator</td>
</tr>
<tr>
<td>20</td>
<td>$026</td>
<td></td>
<td>TWI</td>
<td>TWI_vect</td>
<td></td>
<td>Two-wire Serial Interface</td>
</tr>
<tr>
<td>21</td>
<td>$028</td>
<td></td>
<td>SPM RDY</td>
<td>SPM_RDY_vect</td>
<td></td>
<td>Store Program Memory Ready</td>
</tr>
</tbody>
</table>

d. The stdio.h library (for information - will not be used in lab)

The term ‘stdio’ stands for ‘Standard Input Output’. This library defines many functions that definitely will not meet the I/O map of your hardware, so function in this library cannot be used.

Appendix C For SKEE3732 Laboratory 1 Sheet (Rev12072017).docx (changes on Page 1 highlighted in Red font)
In short: functions under this library should be avoided unless the standard input and output has been defined in the application. You will need to develop new routine or modify existing routine that uses the IU/O map of your hardware.

e. The string.h library (for information - will not be used in lab)

The header file ‘string.h’ defines some functions to make operations over strings. Like comparing, joining two strings, copying two strings, moving one string to some other location and few more. These functions make it easy to process strings. In many advanced programming tasks, these predefined functions make the job easier. (Refer Standard ANSI C/C++ reference or the file “Essential C.pdf”).

f. The math.h library (for information - will not be used in lab)

If you are opting to use some mathematical functions, this header file will reduce the labour required. It defines mathematical functions like sin(), cos(), tan(), exp() and much more. (Refer Standard ANSI C/C++ reference or the file “Essential C.pdf”).

g. Other libraries (for information - will not be used in lab)

Other ANSI C/C++ libraries may be use if you are developing complete system but in this lab they are omitted.

C.2. AVR Data in Program Space

The AVR is a Harvard architecture processor, where Flash is used for the program, RAM is used for data, and they each have separate address spaces. It is a challenge to get constant data to be stored in the Program Space, and to retrieve that data to use it in the AVR application.

The problem is exacerbated by the fact that the C Language was not designed for Harvard architectures, it was designed for Von Neumann architectures where code and data exist in the same address space. This means that any compiler for a Harvard architecture processor, like the AVR, has to use other means to operate with separate address spaces.

GCC has a special keyword, __attribute__ that is used to attach different attributes to things such as function declarations, variables, and types. This keyword is followed by an attribute specification in double parentheses. In AVR GCC, there is a special attribute called progmem. This attribute is use on data declarations, and tells the compiler to place the data in the Program Memory (Flash).

AVR-Libc provides a simple macro PROGMEM that is defined as the attribute syntax of GCC with the progmem attribute. This macro was created as a convenience to the end user, as we will see below. The PROGMEM macro is defined in the <avr/pgmspace.h> system header file.

It is difficult to modify GCC to create new extensions to the C language syntax, so instead, avr-libc has created macros to retrieve the data from the Program Space. These macros are also found in the <avr/pgmspace.h> system header file. Reference for avr/pgmspace.h is available at http://www.atmel.com/webdoc/AVRLibcReferenceManual/pgmspaceapgmspace_strings.html.

a. A Note On the const keyword

Using C’s keyword const as a means to tell the compiler that the data is to be “read-only” but not as a means of declaring data to be in Program Space. Doing this would be an abuse of the intended meaning of the const keyword.

The const keyword make it easier for the compiler to make certain transformations, or to help the compiler check for incorrect usage of those variables.

For example, the const keyword is commonly used in many functions as a modifier on the parameter type. This tells the compiler that the function will only use the parameter as read-only and will not modify the contents of the parameter variable.

The const keyword is not as a means to identify where the data should be stored.

b. Storing an array of byte size data in the Program Space

To store your data in Program Memory, use the PROGMEM macro found in <avr/pgmspace.h> and put it after the declaration of the variable, but before the initializer:
```c
#include <avr/pgmspace.h>
/*Declaring data that will be stored in Program Memory
Address location will be assigned by handled by C compiler*/

unsigned const char SS_table[10] PROGMEM = {0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d, 0x03, 0x7f,
0x6f};
unsigned const char numbers1[10] PROGMEM = {0x10, 0x1,0x12,0x3,0x14,0x15,0x6,0x7,0x18,0x9};

Two global array as table namely SS_table and numbers1 are declared in program memory in the statements above.

c. Retrieving a byte sized data in array identified as numbers1 the Program Space (as declared above)

Use the appropriate `pgm_read_*` macro, and the address of your data becomes the parameter to that macro:

```c
volatile char tempPGMdata;
tempPGMdata=pgm_read_byte_near(&numbers1[i]);
```
Step C4: In the ‘Device Selection’ dialog that appears (see Figure C2.1), search for ATmega32A and then click button OK.

**Note:** If you want to use other AVR chips such as ATMAGE8515, select it at this step. In this tutorial, we will use ATmega32A for both software simulation and hardware testing.

![Figure C2.1: Selecting device.](image)

Step C5: A project file will be created and Atmel Studio displays an initial file Lab1PreCversion.c (see Figure C2.2).

![Figure C2.2: The Atmel Studio with a project opened.](image)

Step C6: Enter the C code shown in Figure C3. It is not important to understand the code at this stage, but you can do that by reading the C comments. It is important that the program is not CUT from this document and PASTE on to the Atmel Studio. You will pick-up new
knowledge as you type in the statements while referring to the comments.

Do not copy the comments as-is given in Figure C3. You are required to develop your own original comments which can be done as you progressively type out and understand the sequence of statements.

Step C7: Click menu File | Save All to save all project files. Note that an Atmel Studio solution has extension `.atsln`; an Atmel Studio C project has extension `.cproj`.
/* Lab1PreCversion.c */
/* Created: 12/8/2016 7:15:47 PM
* Author: Dell5110
* This program lets the user in C that implement the same process specified in Lab1Pre.asm
* but with some added functions which you will be needed to analyse */

#include <avr/io.h>
#include <avr/pgmspace.h>

/*Declaring data that will be stored in Program Memory
 Address location will be assigned by C compiler*/
#define ten 10
unsigned const char SS_table[10] PROGMEM = {0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d, 0x03, 0x7f, 0x6f};
unsigned const char numbers1[10] PROGMEM = {0x10, 0x1,0x12,0x3,0x14,0x15,0x6,0x7,0x18,0x9};

//Main Program
int main(void) { //

    // Declaring data that will be stored in Data Memory
    //declaring num1 as 1 byte storage
    volatile char num1;
    //declaring num2 as 1 byte storage
    volatile char num2;
    //declaring data1 as 1 byte storage
    volatile char data1;
    //declaring data2 1 byte storage
    volatile char data2;
    //declaring numbers2 as a 7 byte array storage
    volatile char numbers2[7];
    //declaring endofnumbers2 whose address is immediately after numbers2 array
    volatile char endofnumbers2;

    //declaring numbers3 as a 6 byte array storage
    volatile char numbers3[5];
    //declaring numbers4 as a 16 byte array storage
    volatile char numbers4[10]={0};
    //declaring temp1
    volatile char temp1;
    //declaring temp2
    volatile char temp2;
    //declaring sum
    volatile char sum;

    //Interrupt vector address locations need not be reserved.
    //Address of Start of program need not be declared
    //All the above are assigned by C compiler

    /* Process No: 1.
    //SP need not be declared - assigned by C compiler */

}
/* Process No: 2.
Now we are going to initialise num1 with value $2F*/
num1=0x2a;
/* Process No: 3.
Now we are going to copy data in num1 into temp1*/
temp1=num1;
/* Process No: 4.
Now we are going to do the calculation of sum = num1 + temp1 + 5*/
sum = num1 + temp1 + 5;
/* Process No: 5.
Now we are going to initialise a pointer with address of numbers3
We do not concern of the pointer register to use because it is handled by C compiler*/
static volatile char* ptrnumbers3;//ptrnumbers3 is declares as a 6 byte array
ptrnumbers3=&numbers3;
Now we are going to store value $53 at address pointed by ptrnumbers3 which is
the address numbers3;*/
*ptrnumbers3=0x53;
/* Process No: 7.
Now we are going to store value $35 at address which is 3 byte after
address of numbers3 (displaced +vely) */
*(ptrnumbers3+3)=0x35;
/* Process No: 8.
Now we are going to initialise value:
$10, $11, $12, $13 and $14
in sequence in the array pointed by numbers3 (i.e address unknown -but can be obtain
from Watch View in Debug) consecutively. We do not concern of using the pointer
register because it is handled by C compiler when we use index pointer*/
volatile char tempdata=ten;
for (int i=0;i<5;i++)
{
    numbers3[i]= tempdata;
    tempdata++;
}
/* Process No: 9.
Now we are going to copy in sequentialy, the string of 1 byte data in
address numbers3+0, numbers3+1,numbers2+2 and numbers3+3
to array pointed by numbers2+3 in REVERSE ORDER (address handled by C)
Effectively sequentially, the following will happen:
(numbers2+3) ---(numbers3)
(numbers2+2) ---(numbers3+1)
(numbers2+1) ---(numbers3+2)
(numbers2) ---(numbers3+3)
*/
numbers2[3]=numbers3[0];
numbers2[2]=numbers3[1];
numbers2[1]=numbers3[2];
numbers2[0]=numbers3[3];
/* Process No: 10.
Now we are going to read 4 successive byte of data array from numbers1 in Program
Memory to array numbers4 in data memory*/
volatile char tempPGMdata;
```c
for (int i=0; i<4; i++)
{
    tempPGMdata = pgm_read_byte_near(&numbers1[i]);
    numbers4[i] = tempPGMdata;
}
/* Process No: 11.
wait forever...
here:
jmp here
*/
    while(1)
    {
        /* You cannot debug in this loop because there is no instruction to execute
        In This loop is where codes that will be repetitively executed is
        written.*/
    }

Figure C3: Program code Lab1PreCversion.c.

C.4. Compiling C code to HEX file

Step C8: Click menu Build | Build Solution to compile the C code (the hot-key for this is F7).

Step C9: If there is no error message, a file called Lab1PreCversion.hex will be produced (see Figure C11). This file contains the machine code that is ready to be downloaded to the ATmega32A microcontroller. The file is stored in sub-folder 'debug' or 'release' of your project.

Step C10: If there are error messages, check your C code. Most often, error messages are caused by typographical or syntax errors. Atmel Studio will show the line numbers where errors appear in the C code.
C.5. Debugging C program using the simulator

Debugging is an essential aspect in any type of programming. This section will show you how to debug a C program at source-code level, using Atmel Studio. Basically, you can execute a C program one line at a time, and observe the effects on the CPU registers, IO ports, and memory. This is possible because Atmel Studio provides a software simulator for many AVR microcontrollers, including the ATmega32A chip. The following steps in this section do not require the Gotronik ATmega32AA Target Board.

Continue with the example project Lab1PreCversion.cproj created in Section 3.2.

Step C11: Start the debugger by selecting menu Debug | Start Debugging and Break. Atmel Studio will require you to specify a debugger. Select ‘Simulator’ and unselect the “Preserve EEPROM” checkbox, as shown in Figure C11.
Figure C11: Specifying the debugger to be ‘Simulator’.
Step C12: The display shown in Figure C12 will be shown.

Figure C12
The yellow pointer shows that program stop at statement `num1=0x2a;`

The instruction is the first executable statement in the C program. Declarations of variables (e.g. “volatile char”) or constants (e.g. “def ten 10”) are not executable statements, of which C/C++ compiler will assigns addresses in memory locations or equates to a value the identifier respectively.

Similar as in Assembly Language Project in Section B.3 thru B.12 of Appendix B in “Lab1 20162017 Appendix B.pdf”, In C/C++ Project, while
in Debugging mode, Atmel Studio Simulator lets you examine the contents of CPU registers and IO ports. To enable these views, select the menu Debug|Windows and then select Processor View or I/O View. You can also select the Watch view to monitor the contents of data locations. The watch window requires you to enter the name of the identifier to the variable. The Watch windows display the absolute address of the identifier. Memory View though is not much applicable in debugging C/C++ program because when you write in C/C++ language, you do not use absolute memory address but just identifier.

You may Step Over through the program to inspect result of the execution through the Watch window. Since display the absolute address of the identifier so if it need be, you can monitor content of the address using the Memory window though normally it is not necessary. However you may use memory views to confirm whether the memory location of the data is in Program Memory, Data Memory or EEPROM memory. We are not touching EEPROM memory in this laboratory.

C.6. **Inspecting the Assembly Language Code Generated from the C/C++ code.**

**Step C13:** Open the disassembly by selecting Debug|Windows|Disassembly which will be as in Figure C13:
The yellow pointer shows that program stop at address 0x004E, the location for the instruction “LDI R24,0x2F” which is followed by instruction “STS 0x007E,R24”.

This two set of instructions is generated by Atmel Studio C/C++ compiler that implements the statement “num1=0x2f;”.

Note Content of Program Counter = 0x004E, the location where the program stop because when you have selected “Start Debugging and Break”, of which the execution breaks at the first instruction met in the program. In the program all the other statement above the statement “num1=0x2f;” are declarations of memory locations for data. The location will be assigned by the compiler which can be inspected from the
disassembly or the Watch window. These statements are not generated as instructions so that is why these statements will not be seen executed in the debugging flow.

Note also that on the Processor View that the value of PC=0x004E (four leading zeroes is truncated so that PC is presented as a 16 bit number which is the architecture for ATmega32A) and SP = 0x085D.


Step C14: Scroll up the window to the most upper top of the disassembly window which is shown in Figure C14.

Figure C14

Referring to Table A2 in “Lab1 20162017 Appendix A.pdf” and Table C.1(c) the RESET vector at address 0x0000 is “JMP 002F”.

This is the first instruction executed by the ATmega32 when a RESET occur. The the program will jump to execute the
instruction at address 0x002A. From here subsequent instructions which is generated by the C/C++ compiler are executed from address 0x002A sequentially until it reaches address “0x004E” where the program breaks at statement “num1=0x2f”. These sequence of execution of instructions initialised X, Y and SP register that is in red font on the Processor view of Figure C14. You will be needed to identify which instructions will initialise the value of SP.

You will also be needed to identify the Assembly Language Instructions generated from the C/C++ program for each of the process.

Note: On the Disassembly View, address and PC are displayed as 32 bit number. For ATmega32, Address and PC are 16 bit numbers, so to meet the architecture for ATmega32 I have truncated the 4 leading zeroes of 32 bit number to 16 bit number (values do not change). In Atmel Studio, address and PC are displayed in 32 bit numbers to accommodate devices like ATmega128 which has 32 bit address and PC.

C.8. “Watch”ing variable during debugging.

Step C15: Open a watch window by selecting Debug|Windows|Watch|Watch 1. Then insert the identifier num1, numbers3 and ptrnumbers3 under the Name column as shown in Figure C15. You may insert other identifier later. Under the value column of the num row is given value the content of the identifier Num1 which is 0x00. Under the Type column of the identifier num row is given the type, locations, and address of the identifier Num1 which are char, 0x00 and 0x007e respectively. This value will change and can be monitored when a statement (in this case the instruction “num1=0x2F”) is executed.

The identifier numbers3 is a 5 by 5 array, so you can expand it to monitor the content of each element in the array. The number in the square bracket is the index number in the array.

![Watch window](image)

Figure C8

Step C16: You can now change to the LAB1PreCversion.c View. You can now step over and inspect the result of execution of the program.
for from process 1 to process 11. You will need to add more identifiers in the “Watch 1” View as you proceed along.

Step C17: For every execution of statement (during tracing or “Step Over”) you must be aware of the source and destination of the statement, and monitor changes on the Watch 1 View. This will help you to understand the operation of the program, and pick up experiences. You can memories in your mind the changes, but for Laboratory report purpose, you will need to take a snapshot.