How to write code for ATxmega128A1U in C

NOTE: When debugging in C it is often helpful to turn off all optimizations to be able to step through your code. In order to do so, locate the device part number at the top right (look for the picture of the chip) of the Atmel Studio screen: **ATxmega128A1U → Toolchain → AVR/GNU C Complier → Optimization → Optimization Level: None (-O0)**. Don't forget to turn it back on when you are finished debugging.

Solutions to your labs 1 and 2, written in C are shown below.

Lab1

```
/* Atmel_in_C.c
 * Description: Lab 1 in C
 */
#include <avr/io.h> //this line replaces .include "ATxmega128A1Udef.inc"
#include "ebi_driver.h"

char  x[] = {'4','q','7','g','4','4','.','C','z','A','N','}',
          'C','x','H','~','A','N','|','G','E',' ','U','!'};
// Storing your table into flash (in C it automatically decides where (in memory)
// to store the table)
int main(void)
{
int address = 0x002700;      // Initializing a variable to hold the address of
                         // SRAM to where you want to store your filtered table
for(int i; i<sizeof(x);i++) //for integer i to the length of the character array
{
       if (x[i] < 0x5A)
       {
              __far_mem_write(address,x[i]); // Store that character into SRAM
              address++;                      // Increment your pointer to the next address
       }
}
return 0;
}
```

Lab2

NOTE: Only have one of the three parts uncommented at a time when running!

```
/* Atmel_in_C.c
 * Description: Lab 2 in C
 */
#include <avr/io.h>    // this line replaces .include "ATxmega128A1Udef.inc"
#include "delay.h"     // include header to be able to use _delay() function in code
#define F_CPU 2000000; // Define processor speed for _delay()
                       // This is necessary in order to be accurate.
void PORT_INIT()
{
       PORTE_DIRSET = 0x0F; //Initialize bits 3:0 as outputs; don't affect other bits
       PORTF_DIRCLR = 0xF0; //Initializes bits 3:0 as inputs; don't affect other bits
}
int main(void)              //Starting MAIN
{
       PORT_INIT();         //call to function to initialize the ports used

       //Lab2 Part A
       while(1)                           //Forever
       {
              PORTE_OUT = PORTF_IN;       //echo the input on PORTF to output on PORTE
       }

       //Lab2 Part B
       while(1)                           //Forever
       {
              PORTE_OUTGL = 1;     //Toggle bit on and off
              _delay_ms(1);        //with 1ms delays
       }


       //Lab2 Part C
       volatile uint8_t count = 255;                    //Declare variable to hold counter value
       volatile uint8_t rotate = 0x80;     //Declare variable to hold rotate
       while(1)
       {
              if((PORTF_IN)&0x10 == 0x10) //if bit4 of PORTF is set
              {
                     PORTE_OUT = count;    //output counter value to PORTE
                     _delay_ms(500);       //500ms = .5 second delay
                     count--;              //decrement count value by 1
              }
              else                        //if bit4 of PORTF is clear
              {
                     PORTE_OUT = rotate;  //output rotate value to PORTE
                     _delay_ms(500);      //500ms = .5 second delay
                     rotate = (rotate >> 1)|(rotate << 7);
       // shift once to the left and 'OR' that with the original number
       // shifted #bits-1 to the right
              }
       }
return 0;
}
```

In C everything is decided automatically

You don’t manually place anything at a certain address or in a certain register, and you don’t have to set up pointers; all this is done for you based on whatever you have coded.

All you have to do is initialize the proper control registers for your microprocessor and write the necessary code logic needed to perform the required task. This is done by the use of variables:

Variables – user-named register used for storing and manipulating data

Types: Char – character (i.e. ‘A’, ‘B’, ‘!’, ‘@’)
String – series of characters (i.e. “3744 is fun”)
Int – integer
Double/Float – decimal types (main difference is precision)
Array – a matrix-type variable type used to store lists of data (ints, chars, etc.)
Global – a variable defined before the main code, can be seen by the whole code
Define – like a global variable, initialized differently and has constraint (see below)

Scope:
Local variables - defined at the beginning of a function and only accessible inside that function

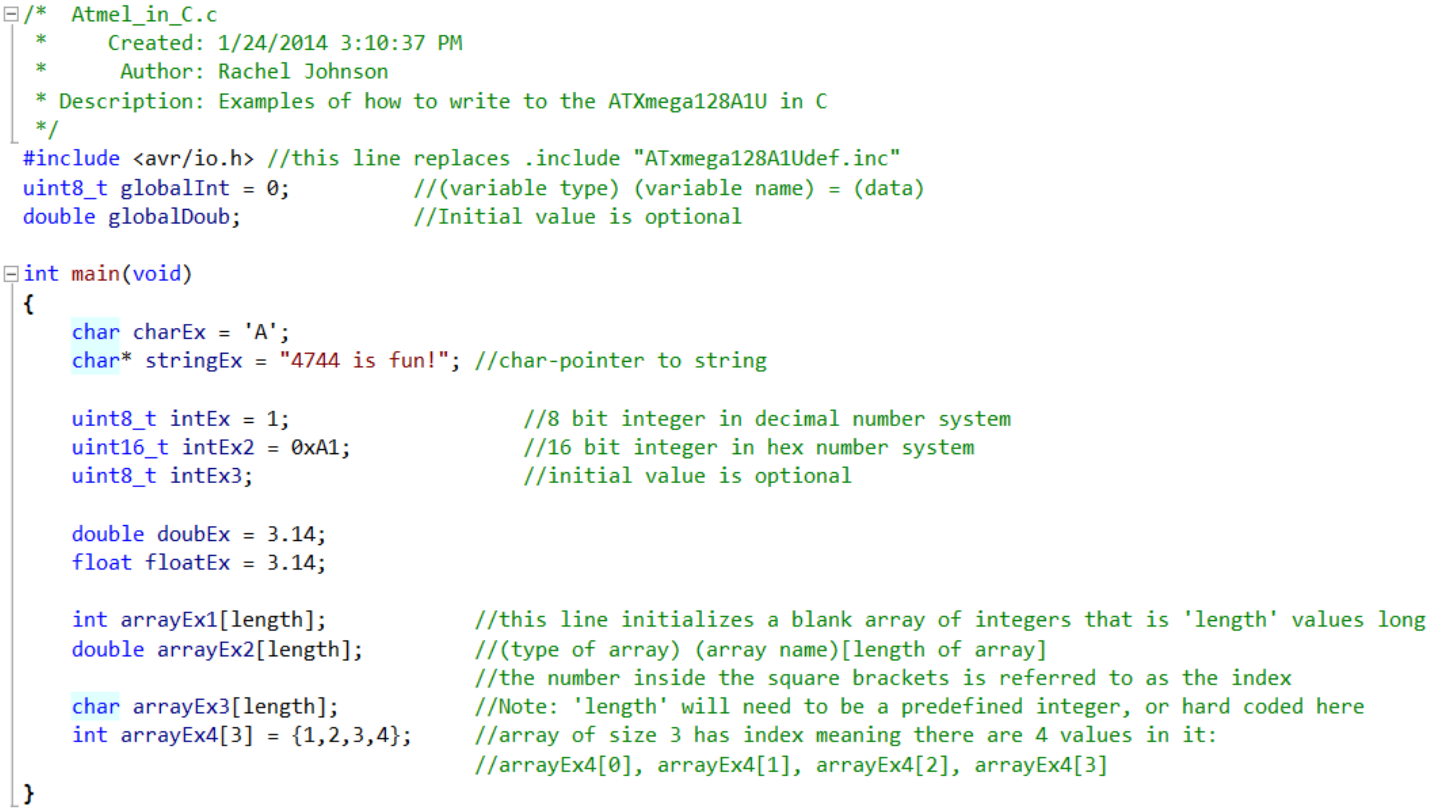
Global variables - defined at the top of .c file usually underneath the include files and accessible from anywhere in code.

Initializing variables

The standard format for initializing a variable is:

```
(variable type) (name for the variable) = (data)
```

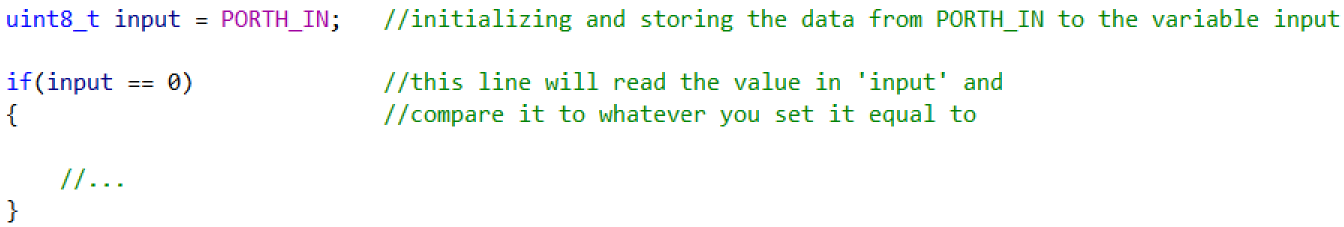
Most initializations are straightforward as shown in the screenshot below. However, because of how we have to communicate with our processor, the String variable has to be handled a little differently. Our processor can only transmit data 8 bits at a time, in other words, one character (char) at a time. For this reason, when defining strings we have to initialize it as a character pointer. We then will have to write a short function that will keep printing a char until it's reached NULL, or the end of the string. (Please see a TA for help with this if needed)



```
/*  Atmel_in_C.c
 *     Created: 1/24/2014 3:10:37 PM
 *      Author: Rachel Johnson
 * Description: Examples of how to write to the ATXmega128A1U in C
 */
#include <avr/io.h> //this line replaces .include "ATxmega128A1Udef.inc"
uint8_t globalInt = 0;          //(variable type) (variable name) = (data)
double globalDoub;              //Initial value is optional

int main(void)
{
    char charEx = 'A';
    char* stringEx = "4744 is fun!"; //char-pointer to string

    uint8_t intEx = 1;                  //8 bit integer in decimal number system
    uint16_t intEx2 = 0xA1;             //16 bit integer in hex number system
    uint8_t intEx3;                     //initial value is optional

    double doubEx = 3.14;
    float floatEx = 3.14;

    int arrayEx1[length];           //this line initializes a blank array of integers that is 'length' values long
    double arrayEx2[length];        //(type of array) (array name)[length of array]
                                    //the number inside the square brackets is referred to as the index
    char arrayEx3[length];          //Note: 'length' will need to be a predefined integer, or hard coded here
    int arrayEx4[3] = {1,2,3,4};    //array of size 3 has index meaning there are 4 values in it:
                                    //arrayEx4[0], arrayEx4[1], arrayEx4[2], arrayEx4[3]
}
```

Storing/reading with variables

Storing to a variable is as simple as setting the variable equal to whatever the required data or register, value. Reading the information in a variable is as simple as using the variable for comparison statements or using the variable in a calculation.



```
uint8_t input = PORTH_IN;   //initializing and storing the data from PORTH_IN to the variable input

if(input == 0)              //this line will read the value in 'input' and
{                           //compare it to whatever you set it equal to

    //...
}
```

A few notes on useful syntax

- Conditional expressions:
  - IF-Statements

```
if (expression) {
<statements>
}
else if (expression) {
<statements>
}
else {
<statements>
}
```

  - The expressions are comparisons using the following possible Relational and Boolean operators:

Relational Operator	Definition	Example (True results)
>	Greater than	47 > 37
>=	Greater than or equal to	47 >= 47
<	Less than	37 < 47
<=	Less than or equal to	37 <= 47
==	Equal to	47 == 47
!=	Not equal to	37 != 47

Boolean (comparator) Operator	Definition	Example (True results)
&&	AND two expressions	((47 >= 47) && (47 > 37))
||	OR two expressions	((37 != 47) | (37 > 47))
!	Complement expression	!(37 > 47)

  - While / do while : do while expression is true

```
while(expression) {
     <statements>
}

do {
     <statements>
}while(expression)
```

The only difference between the two is that the do{}while() runs through the <statements> once before checking the expression.

  - For-loop – allow for repetition while also iterating

```
for(start value; end condition; inc/dec value) {
     <statements>
}
```

- Switch Statements – acts as selection control, like if-statements

```
switch(variable) {
     case value1:
          <statements>
          break;
     case value2:
          <statements>
          break;
     default:
          <statements>
          break;
}
```

- Bitwise Operators:

Symbol	Bitwise Operation
&	AND
|	OR
^	Exclusive OR
<<	Left Shift
>>	Right Shift
~	1's Complement

- Bitmasks
  Instead of having to type out binary or hex numbers to set particular bits in the control registers for your processor, you have the option to use bit masking. The following example sets pins 2 and 4 of PORTB as outputs:

```
PORTB_DIRSET = PIN2_bm | PIN4_bm
```

Bit masks can also be used for the configuration registers and the available masks can be found in the include file (`iox128a1u.h`). The USARTC0_CTRLC register can be initialized in the following manner:

```
USARTC0_CTRLC = USART_CMODE_ASYNCHRONOUS_gc
                | USART_PMODE_DISABLED_gc
                | USART_CHSIZE_8BIT_gc;
```

- There must be a semi-colon at the end of each assignment line, function call, and initialization. The exception is high-level programming constructs that use braces.
- All programs must contain `int main(void) {  }`
- Defines do not use an equals sign for assignment, e.g., `#define F_CPU 2000000`

How to write code for ATxmega128A1U in C

Compared to Assembly

<table>
<tr><th>Assembly</th><th>C</th></tr>
<tr><td colspan="2"><b>Initializing Ports/EBI</b></td></tr>
<tr><td>ldi R16, 0xFF ;set all PORTK pins to be outputs
sts PORTK_DIR, R16

ldi R16, 0x17   ;Configure PORTH pins 0, 1, 4
sts PORTH_DIR, R16 ;These are WE, RE, CS0, ALE1

ldi R16, 0x13   ; Since they are active low
sts PORTH_OUT, R16 ; signals, we must set the
                   ; default output of WE, RE,
                   ; and CS0 to 1
ldi R16, 0x01   ; Store 0x01 in EBI_CTRL
sts EBI_CTRL, R16 ; to select 3 port EBI(H,J,K)
                   ; mode and SRAM ALE1 mode
ldi ZH, HIGH(EBI_CS1_BASEADDR);set base address
ldi ZL, LOW(EBI_CS1_BASEADDR)

ldi R16, 0x21 ; Set to 64K chip select space and
sts EBI_CS1_CTRLA, R16   ; turn on SRAM mode
                          ; address space of the
                          ; input port will be</td><td>PORTK_DIR = 0xFF;//Enable Address 7:0 (outputs)
PORTH_DIR = 0x17;//Enable RE, WE, CS0, ALE1
PORTH_OUT = 0x13;
EBI_CTRL = 0x01   //ALE1 multiplexing, 3 port
                   //configuration.

EBI.CS1.BASEADDR =
     (uint16_t) (CS1_START>>8) & 0xFFFF;

EBI_CS1_CTRLA = 0x21;</td></tr>
<tr><td colspan="2"><b>Function use for initializations</b></td></tr>
<tr><td>EBI_INIT:
 ;EBI initializations
ret

.org 0x200
MAIN:
       rcall EBI_INIT

DONE: rjmp DONE</td><td>void EBI_init(void) //defined or referenced
                    //before main
{
       //EBI initialization
}

int main(void)
{
      EBI_init(); //call function to initialize
      return 0;
}</td></tr>
<tr><td colspan="2"><b>Initializing an external interrupt from PORTD</b></td></tr>
<tr><td>.org PORTD_INT0_VECT ;place code at the interrupt vector for the PORTD_INT0 interrupt
       rjmp EXT_INT_ISR
       ;relative jump to our interrupt routine
                         .
                         .
                         .

EXT_INT_ISR:
       ;...
reti   ;return from the interrupt routine</td><td>ISR(PORTD_INT0_VECT) //Initializes external interrupt vector for PORTD_INT0
{
       //...
}</td></tr>
<tr><td colspan="2"><b>Include files</b></td></tr>
<tr><td>.include "ATxmega128A1Udef.inc"</td><td>#include <avr/io.h&gt;</td></tr>
<tr><td colspan="2"><b>Constants in Assembler/Compiler Directives</b></td></tr>
<tr><td>.equ Table_Size = 10</td><td>#define F_CPU 2000000</td></tr>
</table>

How to write code for ATxmega128A1U in C

How to think about Atmel in C

Includes and Defines

- In order to have access to the information in header and/or include files, they need to be included in your programs. Generally, the header (.h) and/or include files (.c or .asm) should be located in the same folder as your main file (or put into the search path used by the compiler). At the start of your program, use the syntax shown in the above table.
- Defines in C are compiler directives like equates (.equ) in assembly language, as shown in the above table.
- Main programs should generally do nothing more than a few initializations and then call functions.
  - Functions are organized schemes for holding code. Therefore, using functions to perform a single or a small number of independent tasks is recommended. This will help to keep your code clean, easier to understand, and easier to debug.
  - All functions require a function prototype. A prototype declares the type of each of the parameters passed into and out of the function and also the type of the returned value, e.g., the below example initializes a function to take the average of three integer values and return a floating point value.

    ```
    float average(int integer1, int integer2, int integer3);
    ```

  - If functions that have no input parameters, then the `void` input parameter will be used, e.g.,

    ```
    int main(void)
    ```

  - If functions that have no value to return, then `void` will be utilized, e.g.,

    ```
    void sendChar(char character)
    ```

  - Functions that have no parameters and no return, will use void twice, e.g.,

    ```
    void EBI_INIT(void) { … }
    ```

  - An Interrupt Service Routine (ISR) looks like a function whose parameter is the source of the interrupt. Explicit prototypes are not needed for ISR's in Atmel C, since they are defined in the interrupt header file. The interrupt header must be included if an ISR are required, i.e.,

    ```
    #include <avr/interrupt.h>
    ```

  - In combination with the ISR and local interrupt enables, you also need to enable and disable global interrupts, with `sei(); or cli();`, respectively.
  - ISRs functions are defined by the vector of that particular interrupt, e.g.,

    ```
    ISR(USARTC0_RXC_vect) { … }
    ```

  - When writing or read to or from external memory addresses, functions are provided (on our website) called `__far_mem_write` and `__far_mem_read(addr)`, respectively. The header file ebi_driver.h defines the syntax for the instruction. The syntax and an example for this instruction follows,

    ```
    __far_mem_write(addr, data)
    __far_mem_read(addr)
    __far_mem_write(0x370000, 0x55);
    uint8_t mem_value = __far_mem_read(0x370000);
    ```