Embedded Systems Laboratory Work

Introduction to embedded software
Abstract

In this laboratory work the main objective is to familiarize students with embedded system design. The laboratory work concentrates on a hardware environment and software development tools needed in the design of the embedded systems. This laboratory work is one of the requirements in the embedded systems course which is based on a course book [1] written by Arnold S. Berger. Students are highly recommended to read up on this book before participating in the laboratory work. This laboratory work is divided into two parts: a home read and a laboratory exercise. Prerequisite to complete laboratory exercise without anxious moments is to survey the home read. The home read gives an overview on the embedded systems design, particularly introducing some software development tools and hardware architecture. The laboratory exercise consists of a practise of the development tools and a programming of a simple program using the C language. Basics in a C programming are recommended but not required. In order to make the approach easier to the software development of the embedded systems with C language, some special characteristics of the C programming in the embedded systems design are given at the begin of the home read part. The laboratory exercise concentrates, particularly, on the major characteristics and difficulties encountered on the embedded systems.
Contents

1. Special Characteristics of the C language ......................................................... 4
   1.1. Pre-processor ................................................................................. 4
   1.2. Variable types ............................................................................ 4
   1.3. Bit masking ................................................................................ 5
   1.4. Pointers ...................................................................................... 5
   1.5. Global vs. local variables .............................................................. 6
   1.6. Storage class modifier volatile ....................................................... 7
   1.7. Inline functions .......................................................................... 7
   1.8. Addressing the registers ............................................................... 7
   1.9. Source code files ......................................................................... 7

2. Development environment ............................................................................. 8

3. Microcontroller ............................................................................................. 8
   3.1. I/O ports .................................................................................... 9
   3.2. Memory spaces .......................................................................... 10
   3.3. Interrupts .................................................................................... 11
   3.4. Timers ....................................................................................... 13
   3.5. USART ....................................................................................... 15
   3.6. JTAG Interface and On-chip Debug System ..................................... 17

4. Software Development Tools ......................................................................... 19
   4.1. Compilation process ................................................................... 19
   4.2. Debugging .................................................................................. 20
   4.3. Files in compilation process ......................................................... 22

References ........................................................................................................ 23
1. Special Characteristics of the C language

This section introduces some special characteristics of C language emerged particularly in the embedded systems software development.

1.1. Pre-processor

Pre-processor is directed using # symbol at the beginning of the command. Pre-processor is used to combine header files to main file and to replace defined macros and named constants in a source code files, for example. The following code examples present directives to preprocessor.

Include

```c
#include <avr/io.h> // General header files
#include <avr/interrupt.h>
#include "uart.h" // User’s own header files
#include "lcd.h"
```

Define

```c
#define BAUD_RATE 9600 // Named constant
#define SET_BIT(port, bit) ((port) |= (1 << (bit))) // Defined macro for setting a bit of a port.
```

1.2. Variable types

In C language different variable types are used to instruct compiler to divide and use data values in particular registers and in particular ALU (Arithmetic Logic Unit) operations. These types represent distinct byte (8-bit) values. The Atmega128 is a fixed-point microcontroller which means that all ALU operations (summation, subtraction, multiplication, division etc.) are performed using integer values. All integer types can be presented both in signed and unsigned in which unsigned values represent only positive numbers, and signed values represent both positive and negative numbers. Different types for WinAVR are presented below.

- **signed char**: 1-bytes integer values between -128 – 127
- **unsigned char**: 1-bytes integer values between 0 – 255
- **signed int**: 2-bytes integer values between -32 768 – 32 767
- **unsigned int**: 2-bytes integer values between 0 – 65 535
- **signed long int**: 4-bytes integer values between -2 147 483 648 – 2 147 483 647
- **unsigned long int**: 4-bytes integer values between 0 – 4 294 967 295

In C language different type definitions can be used to simplify and shorten variable definitions. The following code example presents type definition and using new types in variable definitions.

Type definitions

```c
typedef signed char s08; // New definition for signed char
typedef unsigned char u08; // New definition for unsigned char
u08 unsignedValue; // Define variables
s08 signedValue;
unsignedValue = 100; // Variables can be used as with
signedValue = -100; // standard definitions.
```
In C language numerical values can be presented with different radices. Default radix is decimal which can be presented in numbers between 0 – 9. Other valid radices are binary numbers presented with 0b prefix and hexadecimal numbers presented with 0x prefix. In this laboratory exercise, hexadecimal numbers are used to clarify register operations since value in 8-bits registers can be presented with two hexadecimal numbers. The following code example presents using different radices to express the same register operation.

Write logical one to pins 4, 5, 6 and 7 in PORTA

```c
PORTA = 0b11110000; // Binary number
PORTA = 240; // Decimal number
PORTA = 0xF0; // Hexadecimal number
```

### 1.3. Bit masking

Bit masking is an operation where single bits of a variable or register are changed. Bit masking is a common way to control microcontroller’s peripherals (I/O) since single bits in microcontroller’s registers represent some change to the state of the system, for example, enabling interrupts. Logical operators AND (&), OR (|), XOR (^) and COMPLEMENT (~) are used to change single bits in a register without modifying the rest bits of the register value. In addition, SHIFT (<<, >>) operators are used to simplify changing the known bit in the register. The following code example presents changing of bit values to attain the required register value.

Set and clear single bit

```c
PORTA = PORTA | 0x20; // Set bit 5 (bits 0, 1, ..., 7)
PORTA |= 0x20;  // Same operation written more briefly
PORTA = PORTA & ~0x20; // Clear bit 5
PORTA &|= ~0x20;  // Written more briefly
PORTA |= (1 << 5); // Set bit 5 using shift operator
PORTA |= _BV(5); // Same as previous but uses a macro from library
PORTA &|= ~_BV(5); // Clear bit 5 using shift operator
PORTA &|= ~_BV(5); // Same as previous but uses a macro from library
```

Set and clear multiple bits

```c
PORTA |= 0x50;   // Set bits 4 and 6
PORTA &= ~0x50;   // Clear bits 4 and 6
PORTA |= (1 << 4) | (1 << 6);  // Set bits 4 and 6 using shift operator
PORTA |= _BV(4)| _BV(6); // Same as previous but uses a macro from library
PORTA = (PORTA << 4);  // Shifting PORTA 4 bits to left
```

### 1.4. Pointers

Pointers are variables which enable addressing the memory implicitly. This is advantageous since pointers obviate using of large data structures, like, arrays or structures. In C language, at the right side of the code equation is a source, and at the left side is a target to where value is to be written, for example, PORTA(address) = 0xFF(value). Pointers are variables which contain memory addresses of other variables or registers, and which are used in addressing the memory implicitly. In C language pointer is defined with prefix *. In addition, pointer is of a particular data type which describes data types that can be written to address to where pointer refers. Prefix & returns the address of a variable while * writes to address defined by pointer. The following code examples present how to use of pointers.

Pointers

```c
unsigned char importantData;  // Define variable
unsigned char *dataPointer;  // Define pointer
```
dataPointer = &importantData;  // Set pointer value to
// address of importantData.
*dataPointer = 50;   // Write value 50 to
// address of importantData.
// After these operations variable
// importantData contains integer value
// 50.

1.5. Global vs. local variables

In C language variables can be used as global or local variables. Local variables are local for a particular function. Local
variables cannot be used out of the function they are defined. They can still remain their previous value between function
calls using static modifier. Instead, global variables are global for all functions in the source file they are modified. Static
and global variables reserve memory permanently; instead, local variables reserve memory only within function execution.
Global variables must be modified after pre-processor directives and before functions, whereas local variables must be
defined at the beginning of the function. In order to enable global variables out of the source file where they are defined,
these variables must be modified as extern in every source file they are supposed to be used. The following code examples
present definitions of global and local variables. In addition, extern and static definitions are presented.

File main.c

//preprocessor directives
#include <avr/io.h>
#include <avr/interrupts.h>
#define DEFAULT_VALUE 100

// Global variable definitions
unsigned char global;

// The main program starts here
void main (void)
{
    // Infinite loop
    while (1)
    {global = getLocalValue();
    }

    // Function for getting value
    unsigned char getLocalValue(void)
    {
        // Define local variable
        static unsigned char local = 0;

        // Increments variable at every function call
        local++;

        // Return a value of the local variable
        return local;
    }
1.6. Storage class modifier volatile

Inside functions, code is often optimized by compiler to keep variables temporarily in general purpose registers without updating values of the variables into memory. This cause information loss if another function updates the same variable into the memory and the new value is supposed to be used in the function mentioned before. In order to avoid information loss, variables can be modified as volatile which ensures that each time a variable is used a value of the variable is read from a memory in place of a register. The following code example presents a reliable delay using the volatile modifier.

Volatile modifier

```c
// Modify variable
volatile unsigned char timeCounter;

// The main program starts here
void main(void)
{
    // Reset timeCounter
    timeCounter = 0;
    // Wait for 100 interrupts */
    while (timeCounter < 100);
}

// Interrupt routine for time counting
void interruptRoutine(void)
{
    timeCounter++;
}
```

1.7. Inline functions

In order to increase performance, some functions may be required to work inline. Inline is done by replacing the function call with the function body. As well as eliminating the need for a call and return sequence, it might allow the compiler to perform certain optimizations between the bodies of both functions.

1.8. Addressing the registers

All the ATmega128 microcontroller’s registers are memory mapped to internal memory space which can be accessed directly. These addresses can be found from ATmega128 specifications\(^1\). In order to simplify register addressing, all register addresses are defined by name in iom128.h library. These names are descriptive and similar to register names found in ATmega128 specifications, for example, PORTA, DDRA, . . .

1.9. Source code files

.h files: These files contain header information of C source code files. Header files are files that are included in other files prior to compilation by the C pre-processor. Some, such as stdio.h, are defined at the system level and must be included by any program that uses the standard I/O library. Header files are also used to store data declarations, function prototypes and macros that are needed by more than one program.

c files: These files are often referred to as source files. These files contain all source codes of programs. The suggested order of sections for a source file is as follows:

- First in the file is a prologue that tells what is in that file
- Any header file includes should be next
- Any defines andtypedefs that apply to the file as a whole are next
- Next come the global data declarations
- The functions come last, and should be in some sort of meaningful order. Programs starts running from main(void) function, and this function can only be in one source file.

2. Development environment

In this laboratory work we do not consider the hardware/ software partitioning problem [1], instead, we use a commercial microcontroller and a circuit board designed in advance. However, in order to give a basic concept of the versatile characteristics and constraints of the microcontrollers, the major characteristics of the hardware used in this work (microcontroller) are presented. General development environment structure in the embedded systems design consists of programming environment, a microcontroller and an operating environment (figure 1). In this laboratory work software is written with a PC and debugged using a JTAG-ICE (in-circuit emulator) interface.

An emulator enables run-time debugging of the software which makes a program development easier. ATmega128 microcontroller controls and communicates with the operating environment by performing the software. In this laboratory exercise, operating environment consists of a LCD screen, leds and a RS232 serial interface which enable communication with the PC. Following sections describe more specifically microcontroller architecture and the software development tools.

![Image 1 Block diagram. Development environment for embedded systems consist of programming](image)

3. Microcontroller

The ATmega128 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATmega128 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed. Figure 2 presents a block diagram of the ATmega128 microcontroller. For more detailed description of the ATmega128 microcontroller’s architecture can be found in specification[2].

3.1. I/O ports

ATmega128 microcontroller includes 7 I/O ports (6 x 8 pins + 5 pins = 53 pins) which enable general digital I/O operations and alternate port functions, like pulse width modulation (PWM), using special function registers. Each port pin consists of three register bits: DDxn, PORTxn, and PINxn. The DDxn bits are accessed at the DDRx I/O address, the PORTxn bits at the PORTx I/O address, and the PINxn bits at the PINx I/O address. The DDxn bit in the DDRx register selects the direction of this pin. Structure of these registers is presented in figure 3.
If DDXn is written logic one, Pxn is configured as an output pin. If DDXn is written logic zero, Pxn is configured as an input pin. If PORTxn is written logic one when the pin is configured as an input pin, the pull-up resistor is activated. To switch the pull-up resistor off, PORTxn has to be written logic zero or the pin has to be configured as an output pin. If PORTxn is written logic one when the pin is configured as an output pin, the port pin is driven high (one). If PORTxn is written logic zero when the pin is configured as an output pin, the port pin is driven low (zero). Independent of the setting of data direction bit DDXn, the pin state can be read through the PINxn. Port configurations are presented in figure 4.

<table>
<thead>
<tr>
<th>DDXn</th>
<th>PORTxn (in SFOR)</th>
<th>I/O</th>
<th>Pull-up</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>Input</td>
<td>Pull-up</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>Input</td>
<td>Yes</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td>Input</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Output</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>Output</td>
<td>No</td>
</tr>
</tbody>
</table>

Image 4 Port configuration states.

Code examples for I/O port operations:

```c
DDRA = 0xFF; // Logic one to every pin of port A
PORTA = 0x0F; // Logic zero to first 4 pins of port A
value = PINA; // Value of port A
```

### 3.2. Memory spaces

AVR architecture has two main memory spaces, the data memory and the program memory space. In addition, the ATmega128 features an EEPROM memory for data storage. All three memory spaces are linear and regular.
Program memory: ATmega128 contains 128K bytes on-chip in-system reprogrammable flash memory for program storage. Since all AVR instructions are 16 or 32 bits wide, the flash is organized as 64K x 16. For software security, the flash program memory space is divided into two sections, boot program section and application program section. The program memory can be programmed in SPI, JTAG, or parallel programming mode. In this laboratory work, due to simplicity and emulation capabilities, flash is programmed in JTAG mode.

Data memory: In normal mode, the first 4352 data memory locations address the register file, the I/O memory, extended I/O memory, and the internal data SRAM. The first 32 locations address the register file, the next 64 location the standard I/O memory, then 160 locations of extended I/O memory, and the next 4096 locations address the internal data SRAM. An optional external data SRAM can be used with the ATmega128. This SRAM will occupy an area in the remaining address locations in the 64K address space. This area starts at the address following the internal SRAM.

EEPROM memory: ATmega128 contains 4K bytes of data EEPROM memory. It is organized as a separate data space, in which single bytes can be read and written. The EEPROM access registers are accessible in the I/O space. The access between the EEPROM and the CPU is performed using the EEPROM address registers EEARH and EEARL, the EEPROM data register EEDR, and the EEPROM control register EECR.

Code examples for EEPROM read and write operations:

Write

```
EEAR = 0x000; // Address is 0x000
EEDR = 0x08; // Data is 0x08
EECR |= (1 << EEMWE); // Start writing data 0x08
EECR |= (1 << EEWE); // to address 0x000.
```

Read

```
EEAR = 0x000; // Address is 0x000
EECR |= (1 << EERE); // Start reading data
value = EEDR; // Save data to variable
```

3.3. Interrupts

AVR provides several different interrupt sources. These interrupts and the separate reset vector each have a separate program vector in the program memory space (figure 5). All interrupts are assigned individual enable bits which must be written logic one together with the global interrupt enable bit (I-bit) in the status register (SREG) in order to enable the interrupt. This may be done editing directly SREG register or using WinAVR command sei() . Disabling interrupts may be done same way, editing register or using command cli(). When an interrupt occurs, the global interrupt enable I-bit is cleared and all interrupts are disabled. The user software can write logic one to the I-bit to enable nested interrupts. All enabled interrupts can then interrupt the current interrupt routine. When returning from an interrupt I-bit is automatically set and instruction RETI is executed. Some of the most relevant registers related to this laboratory work for setting interrupts are presented in a table 1.
<table>
<thead>
<tr>
<th>Vector No.</th>
<th>Program Address&lt;sup&gt;(2)&lt;/sup&gt;</th>
<th>Source</th>
<th>Interrupt Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$0000$&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>RESET</td>
<td>External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset</td>
</tr>
<tr>
<td>2</td>
<td>$0002$</td>
<td>INT0</td>
<td>External Interrupt Request 0</td>
</tr>
<tr>
<td>3</td>
<td>$0004$</td>
<td>INT1</td>
<td>External Interrupt Request 1</td>
</tr>
<tr>
<td>4</td>
<td>$0006$</td>
<td>INT2</td>
<td>External Interrupt Request 2</td>
</tr>
<tr>
<td>5</td>
<td>$0008$</td>
<td>INT3</td>
<td>External Interrupt Request 3</td>
</tr>
<tr>
<td>6</td>
<td>$000A$</td>
<td>INT4</td>
<td>External Interrupt Request 4</td>
</tr>
<tr>
<td>7</td>
<td>$000C$</td>
<td>INT5</td>
<td>External Interrupt Request 5</td>
</tr>
<tr>
<td>8</td>
<td>$000E$</td>
<td>INT6</td>
<td>External Interrupt Request 6</td>
</tr>
<tr>
<td>9</td>
<td>$0010$</td>
<td>INT7</td>
<td>External Interrupt Request 7</td>
</tr>
<tr>
<td>10</td>
<td>$0012$</td>
<td>TIMER2 COMP</td>
<td>Timer/Counter2 Compare Match</td>
</tr>
<tr>
<td>11</td>
<td>$0014$</td>
<td>TIMER2 OVF</td>
<td>Timer/Counter2 Overflow</td>
</tr>
<tr>
<td>12</td>
<td>$0016$</td>
<td>TIMER1 CAPT</td>
<td>Timer/Counter1 Capture Event</td>
</tr>
<tr>
<td>13</td>
<td>$0018$</td>
<td>TIMER1 COMPA</td>
<td>Timer/Counter1 Compare Match A</td>
</tr>
<tr>
<td>14</td>
<td>$001A$</td>
<td>TIMER1 COMPB</td>
<td>Timer/Counter1 Compare Match B</td>
</tr>
<tr>
<td>15</td>
<td>$001C$</td>
<td>TIMER1 OVF</td>
<td>Timer/Counter1 Overflow</td>
</tr>
<tr>
<td>16</td>
<td>$001E$</td>
<td>TIMER0 COMP</td>
<td>Timer/Counter0 Compare Match</td>
</tr>
<tr>
<td>17</td>
<td>$0020$</td>
<td>TIMER0 OVF</td>
<td>Timer/Counter0 Overflow</td>
</tr>
<tr>
<td>18</td>
<td>$0022$</td>
<td>SPI, STC</td>
<td>SPI Serial Transfer Complete</td>
</tr>
<tr>
<td>19</td>
<td>$0024$</td>
<td>USART0, RX</td>
<td>USART0, Rx Complete</td>
</tr>
<tr>
<td>20</td>
<td>$0026$</td>
<td>USART0, UDRE</td>
<td>USART0 Data Register Empty</td>
</tr>
<tr>
<td>21</td>
<td>$0028$</td>
<td>USART0, TX</td>
<td>USART0, Tx Complete</td>
</tr>
<tr>
<td>22</td>
<td>$002A$</td>
<td>ADC</td>
<td>ADC Conversion Complete</td>
</tr>
<tr>
<td>23</td>
<td>$002C$</td>
<td>EE READY</td>
<td>EEPROM Ready</td>
</tr>
<tr>
<td>24</td>
<td>$002E$</td>
<td>ANALOG COMP</td>
<td>Analog Comparator</td>
</tr>
<tr>
<td>25</td>
<td>$0030$&lt;sup&gt;(3)&lt;/sup&gt;</td>
<td>TIMER1 COMPC</td>
<td>Timer/Counter1 Compare Match C</td>
</tr>
<tr>
<td>26</td>
<td>$0032$&lt;sup&gt;(3)&lt;/sup&gt;</td>
<td>TIMER3 CAPT</td>
<td>Timer/Counter3 Capture Event</td>
</tr>
<tr>
<td>27</td>
<td>$0034$&lt;sup&gt;(3)&lt;/sup&gt;</td>
<td>TIMER3 COMPA</td>
<td>Timer/Counter3 Compare Match A</td>
</tr>
<tr>
<td>28</td>
<td>$0036$&lt;sup&gt;(3)&lt;/sup&gt;</td>
<td>TIMER3 COMPB</td>
<td>Timer/Counter3 Compare Match B</td>
</tr>
<tr>
<td>29</td>
<td>$0038$&lt;sup&gt;(3)&lt;/sup&gt;</td>
<td>TIMER3 COMPC</td>
<td>Timer/Counter3 Compare Match C</td>
</tr>
<tr>
<td>30</td>
<td>$003A$&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>TIMER3 OVF</td>
<td>Timer/Counter3 Overflow</td>
</tr>
</tbody>
</table>

*Image 5 Interrupt vectors.*
Code examples for masking interrupts:

Enable

\[
\text{SREG} \leftarrow (1 << I); \quad \text{// Global interrupts}
\]

\[
\text{TIMSK} \leftarrow (1 << \text{OCIE0}); \quad \text{// Timer0 output compare match interrupt}
\]

\[
\text{UCSR0B} \leftarrow (1 << \text{RXCIE0} | (1 << \text{TXCIE0})); \quad \text{// Receive and transmit complete interrupt.}
\]

Disable

\[
\text{SREG} \leftarrow \neg(1 << I); \quad \text{// Global interrupts}
\]

\[
\text{TIMSK} \leftarrow \neg(1 << \text{OCIE0}); \quad \text{// Timer0 output compare match interrupt}
\]

\[
\text{UCSR0B} \leftarrow \neg((1 << \text{RXCIE0}) | (1 << \text{TXCIE0})); \quad \text{// Receive and transmit complete interrupt.}
\]

<table>
<thead>
<tr>
<th>Register</th>
<th>Bit(s)</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SREG</td>
<td>I</td>
<td>Global Interrupt</td>
</tr>
<tr>
<td>EIMSK</td>
<td>INT0–INT7</td>
<td>External Interrupts</td>
</tr>
<tr>
<td>TIMSK</td>
<td>OCIE0,OCIE1A,OCIE1B,OCIE2,TOIE0-TOIE2,TICIE3</td>
<td>Timer Interrupts</td>
</tr>
<tr>
<td>ETIMSK</td>
<td>OCIE3A, OCIE3B, OCIE3C, OCIE1E1C, TOIE3, TICIE3</td>
<td>Timer Interrupts</td>
</tr>
<tr>
<td>UCSR0B</td>
<td>RXCIE0, TXCIE0, UDRIE0</td>
<td>USART Channel0 Interrupts</td>
</tr>
<tr>
<td>UCSR1B</td>
<td>RXCIE1, TXCIE1, UDRIE1</td>
<td>USART Channel1 Interrupts</td>
</tr>
</tbody>
</table>

Table 1 Interrupt mask registers

### 3.4. Timers

Atmega128 contains two 8-bit timer/counters with separate prescalers and compare modes. In addition, Atmega128 contains two expanded 16-bit timer/counters with separate prescaler, compare mode and capture mode. In this laboratory work, we are using timer0 which is a general purpose, single channel, 8-bit timer/counter module. Timer0 is controlled via 3 registers: TCCR0, TCNT0 and OCR0. Structure of these registers is presented in figure 6.

**Image 6 Timer registers.**

In TCCR0 register, setups between different waveforms, compare output modes and clock prescalers can be selected. Bit modes for different waveform generation are presented in figure 7. In addition, compare mode selection is presented in figure 8, and prescaler options in figure 9.
In Clear Timer on Compare or CTC mode (WGM01:0 = 2), the OCR0 Register is used to manipulate the counter resolution. In CTC mode the counter is cleared to zero when the counter value (TCNT0) matches the OCR0. The OCR0 defines the top value for the counter, hence also its resolution. This mode allows greater control of the compare match output frequency. It also simplifies the operation of counting external events.

In this laboratory work, CTC mode is used to generate interrupts at certain time intervals. This enables an accurate time counting, for example, in control systems or in sampling applications. In the formula below is presented how OCR0 register can be used to obtain required time interval.

\[
\Delta t = \frac{N \cdot (1 + OCR0)}{f_{clk}}
\]

<table>
<thead>
<tr>
<th>Mode</th>
<th>WGM01 (CTC0)</th>
<th>WGM00 (PWM0)</th>
<th>Timer/Counter Mode of Operation</th>
<th>TOP</th>
<th>Update of OCR0 at</th>
<th>TOV0 Flag Set on</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Normal</td>
<td>0xFF</td>
<td>Immediate</td>
<td>MAX</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>PWM, Phase Correct</td>
<td>0xFF</td>
<td>TOP</td>
<td>BOTTOM</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>CTC</td>
<td>OCR0</td>
<td>Immediate</td>
<td>MAX</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>Fast PWM</td>
<td>0xFF</td>
<td>TOP</td>
<td>MAX</td>
</tr>
</tbody>
</table>

Image 7 Waveform generation in TCCR0 register.

Image 8 Compare mode selection in TCCR0 register.

\[
OCR0 = \frac{\Delta t \cdot f_{clk}}{N} - 1
\]

where

\(\Delta t\) variable represents the time interval

\(N\) variable represents the prescale factor (1, 8, 32, 64, 128, 256, or 1024)

\(f_{clk}\) variable represents the clock frequency of the MCU

Following code example initializes timer0 to generate an interrupt at 1 ms intervals. Clock frequency is presumed to be 8 MHz.

Interrupt

```c
SREG |= (1 << I); // Enable global interrupts
TIMSK |= (1 << OCIE0); // Enable timer0 output compare match interrupt
```

Setups

```c
TCCR0 |= (1 << WGM01); // CTC mode
TCCR0 |= (1 << CS02); // Prescaler to CLK / 64
```

Start-up
OCR0 = 124; // 1ms time interval
TCNT0 = 0; // Reset counter

<table>
<thead>
<tr>
<th>CS02</th>
<th>CS01</th>
<th>CS00</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>No clock source (Timer/Counter stopped)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( \text{clk}_{\text{T0}} ) (No prescaling)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>( \text{clk}_{\text{T0}} / 8 ) (From prescaler)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>( \text{clk}_{\text{T0}} / 32 ) (From prescaler)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( \text{clk}_{\text{T0}} / 64 ) (From prescaler)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( \text{clk}_{\text{T0}} / 128 ) (From prescaler)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( \text{clk}_{\text{T0}} / 256 ) (From prescaler)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>( \text{clk}_{\text{T0}} / 1024 ) (From prescaler)</td>
</tr>
</tbody>
</table>

Image 9 Prescaler options in TCCR0 register.

3.5. USART

Universal Synchronous and Asynchronous serial Receiver and Transmitter (USART) is a highly flexible serial communication device. The main features are:
- Full duplex operation (Independent serial receive and transmit registers)
- Asynchronous or synchronous operation
- Master or slave clocked synchronous operation
- High resolution baud rate generator
- Supports serial frames with 5, 6, 7, 8, or 9 data bits and 1 or 2 stop bits
- Odd or even parity generation and parity check supported by hardware
- Data overrun detection
- Framing error detection
- Noise filtering includes false start bit detection and digital low pass filter
- Three separate interrupts on TX complete, TX data register empty, and RX complete
- Multi-processor communication mode
- Double speed asynchronous communication mode

ATmega128 has two USARTs, USART0 and USART1. In this laboratory work, we utilize only USART0. The USART has to be initialized before any communication can take place. Initialization process normally consists of setting the baud rate, setting frame format and enabling the transmitter or the receiver depending on the usage. For interrupt driven USART operation, the global interrupt flag should be cleared (and interrupts globally disabled) when doing the initialization. The USART transmitter is enabled by setting the transmit enable (TXEN) bit in the UCSRB Register. Similarly, the USART receiver is enabled by writing the receive enable (RXEN) bit in the UCSRB Register to one. When the transmitter or the receiver is enabled, the normal port operation of the TxD or RxD pin is overridden by the USART and given the function as the transmitter’s serial output or the receiver’s input. The baud rate, mode of operation and frame format must be set up once before any serial transmissions or receptions can be done. Frame size and stop bit selections are presented in figures 11 and 12.

In this laboratory work, we use an asynchronous serial interface. The asynchronous serial data transmission is realized using 2 serial lines, RxD and TxD, one for the transmission and the other for reception. The asynchronous data transmission is controlled via 6 registers: UDR0, UCSRA, UCSRB, UCSRC, UBRR0L and UBRR0H. Structure of these registers is presented in figure 10.

The formula below describes UBRR0 register settings for particular baud rate.
\[ UBBR0 = \frac{f_{clk}}{16 \cdot BAUD} - 1 \]

where

- BAUD variable represents the BAUD rate (bps)
- \( f_{clk} \) variable represents the clock frequency of the MCU
Following code example presents initialization, receive and transmit procedures of the USART serial interface.

**Initialization**

```c
UBRR0H = 0; // Set BAUD rate 9600 bps,
UBRR0L = 51; // while clock frequency = 8MHz.
UCSR0B |= (1 << RXEN0) | (1 << TXEN0);
UCSR0C |= (1 << USBS0) | (1 << UCSZO0) | (1 << UCSZO1);
```

**Receive**

```c
while (!(UCSR0A & (1 << RXC0))); // Wait for data to be received
value = UDR0; // Save received data to variable
```

**Transmit**

```c
while (!(UCSR0A & (1 << UDRE0))); // Wait for empty transmit buffer
UDR0 = data; // Put data (8-bit) into buffer
```

<table>
<thead>
<tr>
<th>UCSZn2</th>
<th>UCSZn1</th>
<th>UCSZn0</th>
<th>Character Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5-bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7-bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8-bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9-bit</td>
</tr>
</tbody>
</table>

Image 11 Frame size selection in UCSRnC register (n=0,1).

<table>
<thead>
<tr>
<th>USBSn</th>
<th>Stop Bit(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1-bit</td>
</tr>
<tr>
<td>1</td>
<td>2-bits</td>
</tr>
</tbody>
</table>

Image 12 Figure 12: Stop bits selection in UCSRnC register (n=0,1).

### 3.6. JTAG Interface and On-chip Debug System

The AVR IEEE std. 1149.1 compliant JTAG interface can be used for testing PCBs by using the JTAG Boundary-scan capability, for programming the non-volatile memories, fuses and lock bits, and for on-chip debugging. Figure 13 shows a block diagram of the JTAG interface and the On-chip Debug system. The TAP Controller is a state machine controlled by the TCK and TMS signals. TAP Controller selects either the JTAG Instruction Register or one of several Data Registers as the scan chain (Shift Register) between the TDI input and TDO output. The Instruction Register holds JTAG instructions controlling the behaviour of a Data Register.
ID-Register, Bypass Register, and the Boundary-scan Chain are data registers used for board-level testing. The JTAG Programming Interface (actually consisting of several physical and virtual Data Registers) is used for serial programming via the JTAG interface. Internal Scan Chain and Break Point Scan Chain are used for On-chip debugging only.

**Programming via JTAG** Programming of AVR parts via JTAG is performed via the four pin JTAG port, TCK, TMS, TDI, and TDO. These are the only pins that need to be controlled / observed to perform JTAG programming (in addition to power pins). It is not required to apply 12V externally. The JTAGEN fuse must be programmed and the JTD bit in the MCUCSR register must be cleared to enable the JTAG Test Access Port. The JTAG programming capability supports:

- Flash programming and verifying
- EEPROM programming and verifying
- Fuse programming and verifying
- Lock bit programming and verifying

**On-chip debugging**: Hardware support for on-chip debugging consists mainly of a scan chain on the interface between internal AVR CPU and internal peripheral units, of a break point unit, and of a communication interface between the CPU and JTAG system. The JTAGEN fuse must be programmed to enable the JTAG test access port. In addition, OCDEN fuse must be programmed and no lock bits must be set for the on-chip debug system to work.

In this laboratory work, on-chip debugging and JTAG programming are realized using AVR Studio. The AVR Studio enables the user to fully control execution of programs on an AVR device with On-chip Debug capability, AVR In-Circuit Emulator, or the built-in AVR Instruction Set Simulator. AVR Studio supports source level execution of Assembly programs assembled with Atmel Corporations AVR Assembler and C programs compiled with third party vendors' compilers.
All necessary execution commands are available in AVR Studio, both on source level and on disassembly level. User can execute the program, single step through the code either by tracing into or stepping over functions, step out of functions, place the cursor on a statement and execute until the statement is reached, stop the execution, and reset the execution target. In addition, the user can have an unlimited number of code break points (using the BREAK instruction) and up to two data memory break points, alternatively combined as a mask (range) break point.

4. **Software Development Tools**

Software development process of the embedded systems follows a flowchart presented in a figure 14. The software development consists of four different tasks: code editing, compilation, programming and debugging. In this laboratory work, we use different development tools for these tasks. These tools are presented in the following chapters.

![Image 14 Flowchart of a typical software development cycle in the embedded systems design](image)

**4.1. Compilation process**

In this laboratory work, the compilation process is realized using WinAVR package. WinAVR is a suite of open source software development tools for the Atmel AVR series of RISC microprocessors hosted on the Windows platform. It includes the GNU GCC compiler for C and C++. WinAVR includes AVR Libc, which is a free software project whose goal is to provide a high quality C library for use with GCC on Atmel AVR microcontrollers. AVR Libc package provides a subset of the standard C library for Atmel AVR 8-bit RISC microcontrollers. WinAVR may be downloaded from [http://sourceforge.net/projects/winavr/](http://sourceforge.net/projects/winavr/), and comprehensive documentation of supported functions may be found from [http://hubbard.engr.scu.edu/embedded/avr/doc/avr-libc/avr-libcuser-manual/](http://hubbard.engr.scu.edu/embedded/avr/doc/avr-libc/avr-libcuser-manual/).

To compile with WinAVR, you must have in your project directory project source codes and makefile (see explanation below). Compilation is done from command prompt entering in the project directory:

```
make all
```
Cleaning up the project (deletes files produced by compiler) is realized typing:
make clean

4.2. Debugging

The main objective in the debugging process is to find logical errors (bugs) from the source code. This is done by executing program step by step and inspecting variable contents, execution flow, registers and so on. In this exercise we will use AvrStudio debugging tool which is an Integrated Development Environment (IDE) for debugging AVR applications in Windows environment. AVR Studio provides a project management tool, chip simulator and In-circuit emulator interface for the AVR 8-bit RISC family of microcontrollers, AVR Studio is available for free download from: http://www.atmel.com/dyn/products/tools_card.asp?tool id=2725

**Simulator:** In the simulator, program is executed completely in PC which simulates target processor execution and environment. Simulator is advantageous, since there is no need for special hardware. Disadvantage is that external impulses may be difficult to simulate. In addition, there is no visual feedback from target device, like LCD or LEDs.

**Emulator:** Emulator uploads program into the target device. The program uploaded is executed in the target device while execution information and status is monitored using PC. Target processor may include emulation logic integrated (like all ATmega chips) or whole chip can be replaced with special emulator circuit. When starting emulating session:
1. Compile project with make all to produce .elf file
2. Start AVR Studio
3. Select Project ->Project Wizard
4. From opening window select Open and find previously generated .elf file from project folder
5. Click save to save AVR Studio project (.aps )
6. From opening dialog select JTAG ICE for emulating session (AVR simulator to simulate) and from the right select target controller type (ATmega128)
7. Check emulator cables and that there is power on!
8. If everything goes right AVR studio uploads program to the target device and emulating session may begin!

**Image 15 Important emulating functions**

The most important debugging buttons in the AVR studio are presented in the figure 15.

1. Start debugging: AVR Studio uploads program to the target device
2. Stop debugging: AVR Studio stops debugging session and releases target device
3. Run: Emulator starts executing code until break button is pressed or breakpoint occurs
4. Break: Halts code execution
5. Reset: Puts program back to the beginning of main routine
6. Step into: executes single step of the code
7. Run to Cursor: Executes program until it comes to the line where blinking cursor was when button was pressed
8. Toggle Breakpoint: Sets breakpoint on the line with cursor. If breakpoint is already set on the same line, breakpoint is removed. You can set up to three breakpoint same time when using JTAG ICE.
9. Remove all breakpoints: Removes all breakpoint from project

**Image 16 Integrated compiler functions**

10. Compile all source code files
11. Compile all source code files and start debugging
12. Compile active file
13. Delete all compiler generated files
14. Edit makefile options with GUI
Figure 17 shows typical view of AVR Studio session

1. **Code section**: Executed code is displayed here. The next line to be executed is marked with yellow arrow. Breakpoints are marked with red circle on the left side.

2. **Register view**: State of the processor registers are shown here. State is updated when program execution is break. If the value has changed since the last break, the register appears colour coded. The most important section is I/O View. I/O registers are labelled in the same way as you can use with WinAVR to access them.

3. **Message**: The Messages window is the common window for all modules of the AVR Studio to present messages to the user. Messages are colour coded. Most messages are plain messages of no significant priority. They have no colour. Warnings signalling potential problems are yellow. Errors are red.

4. **Memory**: The Memory View is able to display all the different memory types associated with the AVR devices. The code executed lies in the program memory, the variables are located in SRAM (mainly), the I/O registers are mapped into I/O memory area, and the EEPROM is yet another memory area.

5. **Watch**: Debugging high level languages such as C/C++ you need to watch variables. With tools like AVR Studio 4 this is an easy task, click on the variable you want to watch and drag and drop it into the watch window. If the variable you
selected is a struct or an array a [+ ] symbol will occur in front of the variable, indicating that it can be expanded in the view. When the execution breaks, the variable automatically will be updated - if it is in scope. To be in scope simply means that the variable actually exist in memory at the location where the program stopped its execution. If you compile project again under emulating session, just click Break from AvrStudio and it will ask if you want to refresh session.

4.3. Files in compilation process

Different files are required and produced in the compilation process of the embedded systems. Software can be compiled into .hex, .coff or .elf files, depending on the requirements of the software development. Makefile contains compilation instructions. These files are described more specifically in the following paragraphs.

.hex file: This is a compiled binary file which may be programmed to the target microcontroller. It however doesn’t include any kind of information of source code so finding bug from this file is very messy.

.elf file: This file contains compiled binaries as above but in addition it includes instructions for emulator program how to link source code lines and machine language instructions together. So user may follow C source codes instead of cryptic assembly instructions. In this exercise we will use this file when working with AVRstudio.

makefile This file contains information for compiler how to compile the program. Some most important lines in makefile:

MCU = : This line tells what is the target processor type. May be, for example, atmega128, at90s2313 or whatever from Atmel’s AVR-series.
TARGET = : This is the name of project target file without .c extension. If you, for example, have main source code file “main.c” you may then enter TARGET = main
SRC = : If you have more than one source code file, all of them must be listed here. For example, SRC = lcd.c, uart.c
F_CPU = : This line defines the clock frequency of MCU in Hz. For example, F_CPU =8000000

NOTE! All important makefile options are implemented under settings dialog in AVR Studio and can be changed easily with few mouse clicks. So, editing makefile directly has come unnecessary!
References