Preface

This book was written as a practical guide for the students from the technical field of Computer Engineering and Information Technology, who want to be introduced in the fields of Digital Telecommunication, Digital Signal Acquisition and processing, and Embedded Systems Design and Development.

In this book, the readers will find practical aspects regarding GSM/GPRS protocols of telecommunication, digital signal acquisition, audio signal processing and embedded systems design and development. The readers will also be invited to gain practical skills in the fields mentioned above, by developing a series of applications like: a digital oscilloscope, a basic sound processing system or a digital alarm clock. All this applications being divided into relatively small practical assignments which have a gradual complexity level.

The development of the material which represents the core of this book and of the proposed assignments is the result of more than 4 years of work commitment of a team of researchers and teachers, members at the DSPLabs, from which the authors would like to thank eng. Dan Chiciudean and professor Mihai Micea for all the guidance and support. I would also like to thank dr. Ciprian Chirilă, eng. Cristian Cucuieț and professor Marius Marcu.

The authors

Table of contents

1 Digital Telecommunication

1.1 Introduction

1.1.1 General specifications

The Digital Telecommunication laboratory aims at introducing the attending students into the domain of GSM/GPRS communications. The main aspect of the laboratory is to provide the basics for designing and implementing applications that use GSM/GPRS communications. The finality of the laboratory is represented by a GSM/GPRS terminal that is able to display status information about the GSM modem as well as perform simple functions like reading and sending SMS messages or initiating voice calls. The user interface is represented by a touchscreen with an interface designed by the attending students.

In order to attend to this laboratory the students must have the following mandatory prerequisites:

- Strong C programming skills [1]
- Basic knowledge of embedded systems and embedded programming and debugging
- Basic knowledge of finite state machine theory

Although it is not mandatory, it is recommended for student to have basic knowledge about communication protocols like RS233 as well as basic knowledge about computer graphics.

1.1.2 The DSPLABS_DT_STK_V1 learning kit

The main hardware component of the laboratory is represented by the DSPLABS_DT_STK_V1 board. This board is design and developed internally by DSPLabs and is intended to be a learning kit for disciplines that involve any kinds of digital communications. During the laboratory applications intended for this discipline the attention will be focused on the GSM modem.

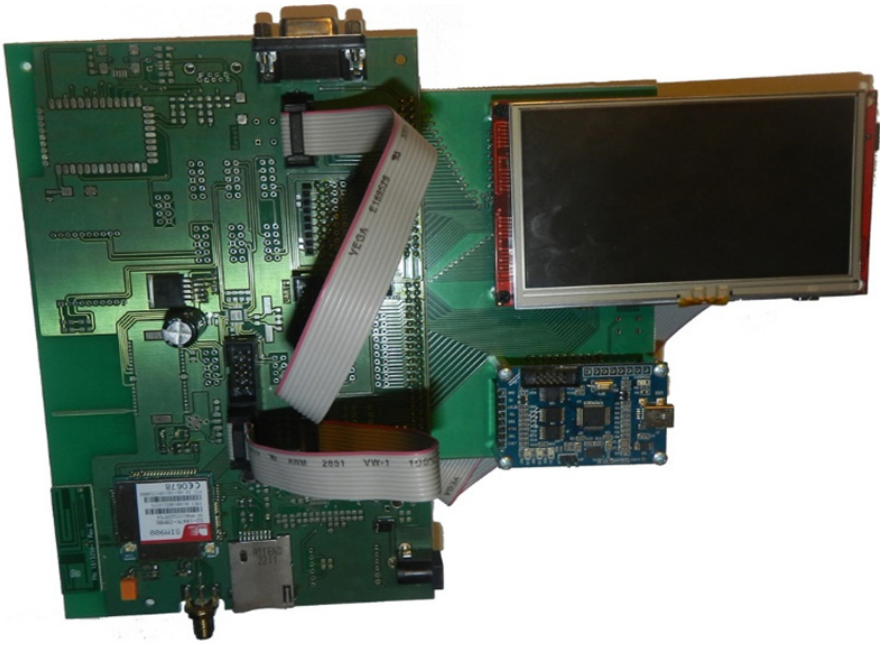

Fig. 1-1 DSPLABS_DT_STK_V1

The learning kit that is used for the laboratory applications is divided into 3 separate but interconnected boards as shown in Fig. 1-1. One of the boards is represented by the Olimex MOD-LCD4.3 board [2], displayed in Fig. 1-2, which is used as a main processing and control unit. This board contains an ARM Cortex M3 NXP LPC1788 microcontroller [3], a TFT LCD with a resolution of 480x272 24bit color pixels, backlight and touchscreen as well as 32 MB of SDRAM external for the microcontroller.

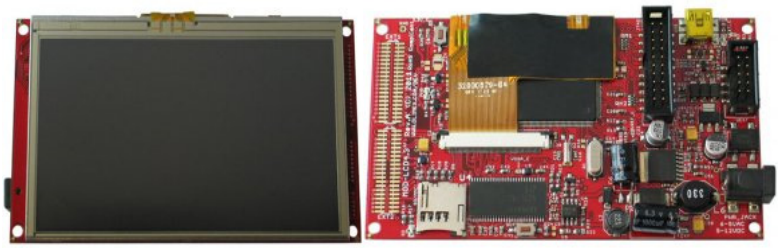

Fig. 1-2 Olimex MOD-LCD4.3 board [2]

The debugging and microcontroller programming is handled by the dedicated JTAG module CooCox CoLinkEx [4] which is directly integrated into the development environment.

Fig. 1-3 CooCox CoLinkEx Debugger Module [4]

The main component of the learning KIT is the DSPLABS_DT_STK_V1_COMM board which is connected to the Olimex MOD-LCD4.3 board and the CooCox CoLinkEx Debugger via a passive connection board. The DSPLABS_DT_STK_V1_COMM board contains various communication modules in order to be used for many disciplines involving digital communication. The board was designed to offer various means of communication using interfaces like XBee [5, 6], for wireless sensor networks, Bluetooth, Ethernet with TCPIP [7], GSM/GPRS [8] and many more as shown in Fig. 1-4.

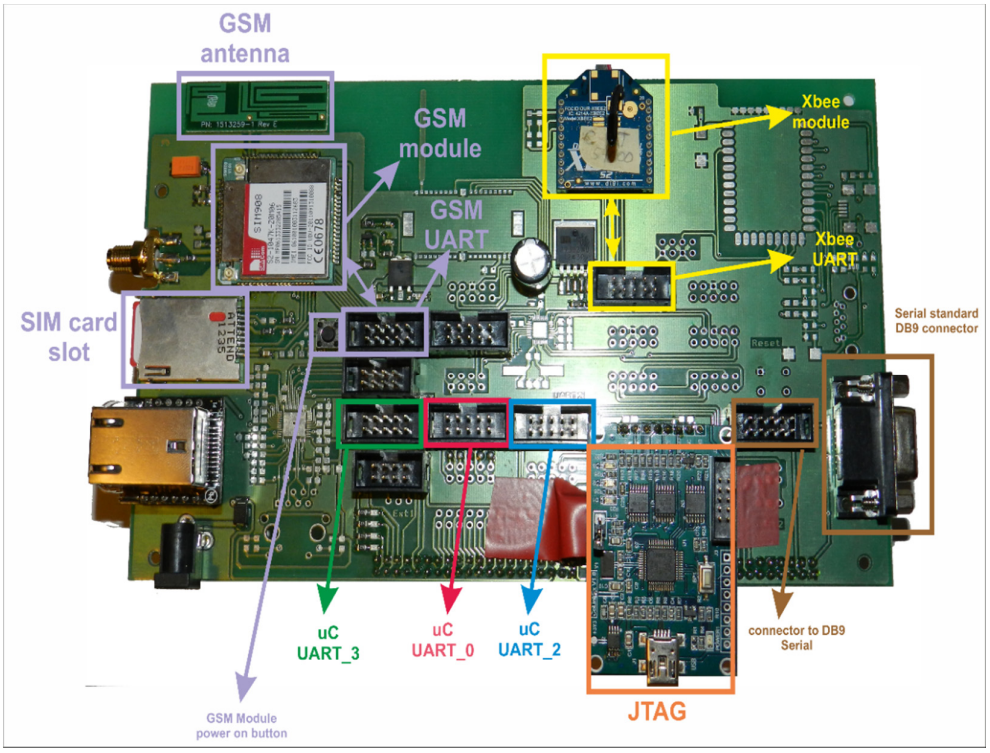

Fig. 1-4 DSPLABS_DT_STK_V1_COMM board component

One very important aspect of this board is that it was designed especially for laboratory lessons thus it is highly modular. Practically all of the interfaces can be connected in almost anyway using some provided connection cables. In Fig. 1-4 all the related components are highlighted in different colors which offer a base analysis of the board components.

The purple highlight is related to the GSM/GPRS interface which has as its main components: the SimCom SIM908 GSM/GPRS/GSM modem, the embedded GSM antenna, the SIM card slot, the GSM modem power on button and the GSM modem's UART interface 2x5 pin connector. It is very important to mention that the UART interface of the GSM modem is directed to the purple highlighted 2x5 pin connector and not further after this connector. Practically the UART signals from this GSM modem stop in the purple highlighted 2x5 pin connector. Any connection to any other interface can only be made via connection cables. This aspect is applicable to all the other communication interfaces. For example the XBee related components, highlighted by the yellow border, are the XBee module and its UART interface which is accessible though the associated 2x5 pin connector also highlighted connector. In this case also the XBee UART signals are taken only to the yellow highlighted 2x5 pin connector and not any further. Cables are required for any connections.

Similar procedure applies to the communication interfaces of the LPC1788 microcontroller of the Olimex MOD-LCD4.3 board. The NXP LPC1788 has 5 UART interfaces [9] and 3 of them are exported on the DSPLABS_DT_STK_V1_COMM and translated to 3 2x5 pin connectors in the following convention: the UART_3 of the LPC1788 is exported to green highlighted connector; UART_0 is exported to the red highlighted connector and UART_2 is exported to the blue highlighter connector.

In order to facilitate the design and development of application on this learning kit, an extern UART interface was added on the DSPLABS_DT_STK_V1_COMM. This interface exports UART signals outside the board via de standard RS-232 DB-9 connector. The TTL side of the signals are connected to the 2x5 dark brown highlighted connector. Using this feature any of the previous interfaces described may be connected to an outside terminal making the necessary connections. For example connecting purple highlighted UART connector to the dark brown highlighter connector may offer a way to interface the GSM modem to an external terminal like a PC.

The 2x5 pin connectors have the following pin numbering and orientation (the arrow designated pin 1):

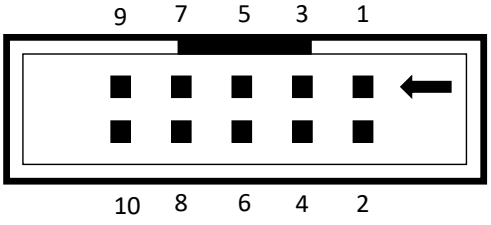

Fig. 1-5 2x5 pin connector orientation and pin numbering

All the 2x5 pin connectors have a serial UART interface connected to the pins. The signals have CMOS logic levels. Not all of them have the same pinout though. In the following table, the pinout is described for every serial interface.

Pin no	DB9	XBee	Bluetooth	GSM	GPS	LPC UART0	LPC UART2	LPCU UART3	LPC UART1
1	-	-	-	DCD	-	-	-	-	DCD
2	RX	TX	TX	TX	TX	TX	RX	RX	RX
3	TX	RX	RX	RX	RX	RX	TX	TX	TX
4	-	-	-	DTR	-	-	-	-	DTR
5	GND	GND	GND	GND	GND	GND	GND	GND	GND
6	-	-	-	DSR	-	-	-	-	DSR
7	RTS	CTS	CTS	CTS	-	-	-	-	RTS
8	CTS	RTS	RTS	RTS	-	-	-	-	CTS
9	-	-	-	RI	-	-	-	-	RI
10	-	-	-	-	-	-	-	-	-

Table 1 Pin mapping for peripheral UART interface connectors

Using the pin mapping presented above one has to be aware when interconnecting various interfaces. Special precautions need to be taken not to generate bus conflict. Bus conflict may appear for example when after a connection to TX signals are linked. This can be avoided only after an analysis of the application.

The minimum external connections that need to be made between the learning kit and a PC used for developing are the following:

- External power supply: 9-12 VDC, 1000 mA
- Connection using micro-USB cable between the PC and the CooCox JTAG module
- For external terminal usage: a connection between the DB9 connector of the DSPLABS_DT_STK_V1_COMM and the serial port of the external terminal (PC) using a DB9 Serial Cable

1.1.3 General specifications about the GSM/GPRS Modem

The applications intended for the laboratory lessons of this discipline are oriented on GSM/GPRS communication. As stated above, the GSM/GPRS components are purple highlighted: the GSM/GPRS SimCom SIM908 Modem, the embedded GSM antenna, the SIM card slot, the power-on, power-off button and the 2x5 connector which have the UART signals from de GSM modem. The SIM908 GSM modem also has an embedded GPS receiver which has a dedicated 2x5 pin header connector. According to Table 1 the GSM modem's serial interface may be connected to the following interfaces: DB9, LPC UART_2, LPC UART_3 and LPC UART_1.

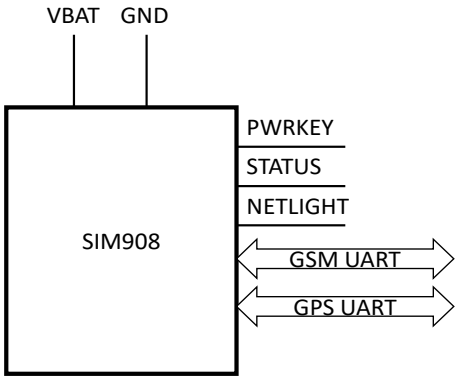

Fig. 1-6 2x5 Block diagram of the GSM modem

An important statement is that a GSM modem is not an integrated circuit. Practically a GSM modem is a small board that contains a microcontroller, with an upgradable firmware running on it, a radio module, some additional memory and a power module.

Giving the fact that the GSM modem is a very complex module, a much more simplified diagram is presented in Fig. 1-6. The idea is to eliminate all the pins that do not present interest for this laboratory applications.

The first pin that needs to be analyzed is the NETLIGHT pin. Its behavior defines the state of the GSM modem from the network registration point of view. In the case of the learning kit this pin is connected to a LED so that its behavior can be visually analyzed. In general, the following states define the way this line communicates the state of the GSM modem [8]:

- The LED connected to the NETLIGHT pin is off: The SIM908 modem is not running. In this case the modem will not communicate on any bus and needs to be power on using the POWERKEY pin.
- The LED connected is 64 ms On and 800 ms Off – In this situation the SIM908 is powered on but the modem is not registered to the mobile network.
- The LED connected is 64 ms On and 3000 ms Off – In this situation the SIM908 is powered on and the modem is registered to the mobile network.
- The LED connected is 64 ms On and 300 ms Off – In this situation the SIM908 is powered on, registered to the mobile network and the GPRS communication is established.

Note that all the behavior above presented for the NETLIGHT pin is not standard for all the GSM modems. Practically every manufacturer is free to use (or not use) this pin as it sees fit. But in general this behavior can be found in many of the present GSM modems.

In this presented case, the NETLIGHT mode was connected to a LED for a visual observation, but, in a much more practical case, the NETLIGHT signal may be connected to a microcontroller in order to be analyzed by the embedded software.

The most important signals that needs to be taken into consideration by the embedded programmer are presented in the above figure. One of the most important signal is represented by the Power Key signal (PWRKEY). This signal is responsible for the power

on and power off operation. An important observation needs to be made. Usually all the GSM modems after they receive the necessary voltage on the power supply pins they do not switch on all the modules described above. An assumption is that the only module switched on is the power module. The microcontroller and the power module do not begin their designated operations until a specific command. This command is provided through the PWRKEY pin. Almost all the GSM modems available on the market have a similar behavior regarding this aspect, but it is not standardized.

For a much easy usage of the GSM modem by the attending students, the DSPLABS_DT_STK_V1_COMM was designed so that the PWRKEY is handled by a push button according to the available hardware manual of the SIM908 GSM Modem.

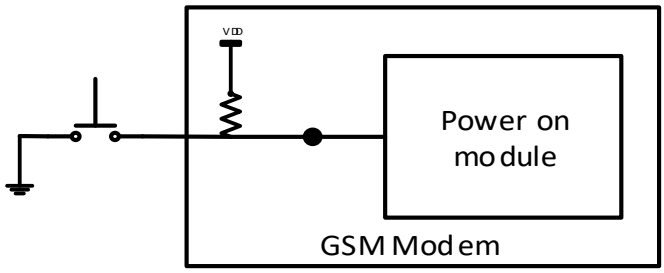

Fig. 1-7 GSM modem power on logic

After the DSPLABS_DT_STK_V1 learning kit is supplied with power, the GSM modem is not running. This can be identified by observing that the NETLIGHT LED on the board is not on. The GSM modem can be set into running mod by pressing the push button on the board for a minimum time of 1 second or until the NETLIGHT LED is turned on. After releasing the button, the LED needs to have the behavior described above and so identifying the state of the GSM modem.

It is important to be noted that the modem will not provide any answer on any of the serial interface busses until the modem is running. No “conversation” with the GSM modem can be established until the modem is fully powered on by using the button (the POWERKEY signal).

Again, this signal, the POWERKEY signal was connected to a push button in order to provide a didactical approach. In a much more practical scenario, this signal should be connected, according to the hardware reference manual, to a pin of a host microcontroller. In this way, the host microcontroller will be responsible for the power on or off state of the modem.

As stated above, only after the GSM modem is in running state it will accept any commands from a host station, let that be a microcontroller or a terminal on a PC. The protocol accepted by the modem is represented by a subset of an AT command language which will be presented later in this book.

The GSM modem has 2 serial interfaces that are used for communication. One of the serial bus is dedicated to the GPS received and the other serial bus is the main interface for communicating with the GSM modem. The serial interface is RS-232 compatible [10], in TTL/CMOS logic levels.

Immediately after the power has been switched into running state the serial interface is activated but, if using the modem with factory defaults, it will not be synchronized on any BAUD rate. A useful property of the serial interface is the AUTOBAUD capability where the serial interface will try to automatically detect the BAUD rate which the host will use to communicate based on a pre-established pattern. The host (PC or microcontroller) will have to send a specific pattern before any attempt of communication with the modem. When receiving the pre-established pattern the modem will empirically detect the BAUD rate used by the host. The pre-established pattern that the GSM modem awaits to perform AUTOBAUD is "*AT*" followed by the 2 special characters Carriage Return (*CR*) with hexadecimal ASCII code 0x0D and Line Feed (*LF*) with hexadecimal ASCII code 0x0A. These 2 special characters are usually obtained by the press of the Enter key on the computer keyboard. After a successful auto BAUDRATE operation the modem should respond with the text "OK". Two important observations need to be made:

1. Under no circumstances, after the modem is switched into running state (power on) the host should send anything different from the AUTOBAUD pattern.
2. The AUTOBAUD is an operation based on measurements and therefore may not be always successful. The first pattern may not be sufficient for AUTOBAUD for a number of reason. A number of AUTOBAUD retries need to be expected by sending several patterns until (not sooner than 1 second) the GSM Modem responds.

Only after a successful AUTOBAUD procedure the host may send other AT commands to the modem. The structure and the meaning of the AT commands accepted by the GSM modem can be found in the SIM908 AT Command Manual [11]. There are 4 types of AT commands as described in the manual and presented in the following table:

Command type	**Syntax**	**Meaning**
Test command	AT+[command]=?	The modem returns the command syntax, parameters and parameter value range
Read Command	AT+[command]?	The modem returns the values of the parameters currently set for the specified command
Write Command	AT+[command]=value	The modem writes the parameters specified by the command and return the result of the operation
Execution Command	AT+[command]	The modem returns non-writable variables that reflect internal processes of the GSM modem

Table 2 AT Command types

Not all the commands have of the 4 types. It depends on the meaning of the command which types are implemented. This information is found in the documentation for each AT command separately which has to be analyzed before using a command.

1.1.4 Provided materials

The attending students for the laboratory applications of this discipline will have online and offline access to the necessary documentation. In this paragraph the available materials will be presented.

First of all, the students will have access to a full hardware [8] and software [11] documentation about the GSM modem. This documentation is essential for establishing a first successful "discussion" with the GSM modem over the serial UART interface.

The learning kit will be made available for the attending students during the laboratory sessions as well with a software library and example project for the microcontroller.

From a software point of view, the scope of these laboratory applications is not for the students to low level configure the peripherals of the microcontroller available but to design higher level software. Having this as a first consideration, a fully working software library will be provided as well as a basic project for the Keil uVision developing environment along with the necessary code examples. The library has a full documentation available, later presented in this material, as well as an online *Doxygen* generated documentation oriented at code level explanations.

1.1.5 Laboratory applications planning

The practical aim for this laboratory is that the attending students to design and implement a software for basic operations using the GSM/GPRS network via a GSM/GPRS modem. Also, a theoretic aspect of the applications is to train the attending students to analyze, design and implement a communication protocol between a host processing unit and a peripheral. This methodology should help the attendees in designing, analyzing and implementing both GSM and non GSM automation related applications as well as to be able to design and implement various communication protocols.

The laboratory lessons begin with establishing a first communication with a GSM modem using a PC as a final terminal. Students need to "talk" to the GSM modem via the PC terminal software in order to obtain information about its status. Such communication will be supported by the AT command manual explaining the accepted commands by the GSM modem [11]. The first lesson also has a theoretical part where the attendees must analyze the syntax of the response of the GSM modem and establish the prime rules for designing the communication protocols. After the presentation of necessary steps for implementing the AT command response language, the students need to prepare, as a homework, a finite state machine that designed the previously analyzed protocol.

The second laboratory applications focuses on implementing the protocol designed a lesson before, in a standard C programming language developing environment (ex. Microsoft Visual Studio). The GSM modem will be simulated using text files that contain the necessary test cases. In a designated period of time, the students need to develop,

implement and test the protocol using simulated responses from the GSM modem in test text files.

There will have to be two finalities from these two laboratory applications. One finality will be a fully functional AT command response parser, without the protocol exceptions. The other finality will be a test that will have to evaluate the student's capacity in analyzing, designing and implementing in pseudo-code a communication protocol similar to the one used by communicating to a GSM modem.

The following laboratory applications will be directly focused on the provided learning kits. Therefore laboratory work 1.4 will introduce the Keil uVision development environment and the basics of the driver documentation will be presented. The students will have to understand how to use the provided documentation. They will also have to write the first basic programs, upload them on the board and debug if necessary. These first basic programs will be oriented on using the LED driver, the UART driver, the *printf* redirection and the software timer.

One of the most important application is laboratory work 1.5 . This work is crucial for successfully accomplish the coming laboratory works. This session aims at the integration of the implementation developed and tested earlier in laboratory work 1.3 into the board. The students will be trained in how to realize this integration. The final result of this session will have to be a program that configures the BAUD rate of the GSM modem, requests and prints the RSSI value on the debug terminal with a frequency of 1 Hz. More status information will be obtained from in modem in laboratory work 1.6 and all of them will be requested and printed on the debug terminal sequentially one after another maintaining the print frequency of 1 Hz.

More advanced operations will be presented in laboratory work 1.7 where the students will have to manage the SMS messages by printing the available messages currently saved on the SIM card and also by writing the necessary code to send SMS messages.

The final mandatory laboratory work, 1.8 , students will have to put everything together and implement a graphical terminal on the LCD and Touchscreen that performs simple functions.

In order to facilitate the execution of the laboratory applications the students should be organized in groups of 2. It will be mandatory for the students to maintain the same organized teams for the rest of the semester. The students that will form a team will be responsible as a whole for the entire laboratory applications and will be evaluated accordingly.

The proposed planning for the laboratory applications is presented in the next table:

week	Laboratory work	Observations
1	Establishing laboratory groups	
2	Introduction + Laboratory work 1.2	Establish groups of 2
3	Laboratory work 1.3	
4		Evaluation of laboratory work 1.3
5	Test: protocol design	Evaluation of test
6	Laboratory work 1.4	
7	Laboratory work 1.5	
8		
9	Laboratory work 1.6	Evaluation 1.5 and 1.6
10	Laboratory work 1.7	
11		
12	Laboratory work 1.8	
13		Evaluation 1.7 and 1.8
14		

Table 3 Laboratory applications planning

1.2 Laboratory work – GSM AT command language

This first application will address the following aspects:

- Initiate communication with the GSM modem using a PC application as a terminal
- Analyze the form of the responses of the GSM modem in order to extract the general syntax of the AT command response
- Based on the extracted AT command response form, the first steps in protocol design will be presented

Before this application can properly begin, the attendees need to process some initial documentation in order to know the basics. As a first pre-assignment please read the introduction of this manual focusing on paragraph 1.1.2 which presents the general aspects of the learning kit and paragraph 1.1.3 related to the GSM modem. That main key questions that the attendee must find an answer to, may be the following:

- What are the main components related to the GSM communication on the learning kit? Where can they be found?
- How can we visually identify the state of the GSM modem?
- What kind of bus the GSM modem uses to communicate?
- What are the necessary connections that need to be made in order to connect the GSM modem to an external terminal through the serial DB9 connector?
- What are the means the GSM modem can be power on or off?

- What is AUTOBAUD and what does the host have to do in order to successfully synchronize the GSM Modem's BAUD rate to the BAUD rate of the host?

ASSIGNMENT 1: Read the paragraphs suggested above and find the answers to the questions.

The first part of the application aims at starting the first communication with the GSM modem. In order to achieve this, the serial interface of the GSM modem must reach a host PC where a dedicated serial terminal will be used. First of all, the serial interface of the GSM modem will be connected to the external serial interface of the board. The link will be made by connecting the purple highlighted 2x5 header pin connector (the serial interface connector belonging to the GSM modem) to the dark brown highlighted 2x5 header pin connector (the external serial interface) using a provided cable. Another set of connections need to be made outside the board using a standard DB9 serial cable to connected the external serial interface of the board represented by the DB9 connector to a dedicated serial port on a host PC. Having all of these connections made, the serial interface signals from the GSM modem will be directly connected to a host PC.

A dedicated serial port belonging to a host PC may easily be accessed by using a serial terminal. The serial terminal is a dedicated software that is able to configure and open a serial port and can be used to send and receive data through one the serial interfaces of the host PC. The serial terminal that will be used in all the laboratory applications is *Docklight scripting*.

Docklight scripting is an easy to use but powerful serial terminal software. The main advantages of *Docklight scripting* are:

- possibility to have access to all the settings of the serial port
- can function as a TCP/UDP client or server
- offers the possibility to define and send macros over the line (serial or network)
- has scripting features in order to simplify protocol interpretation
- offers good representation of unprintable characters
- byte interpretation may be ASCII, hexadecimal, decimal and binary

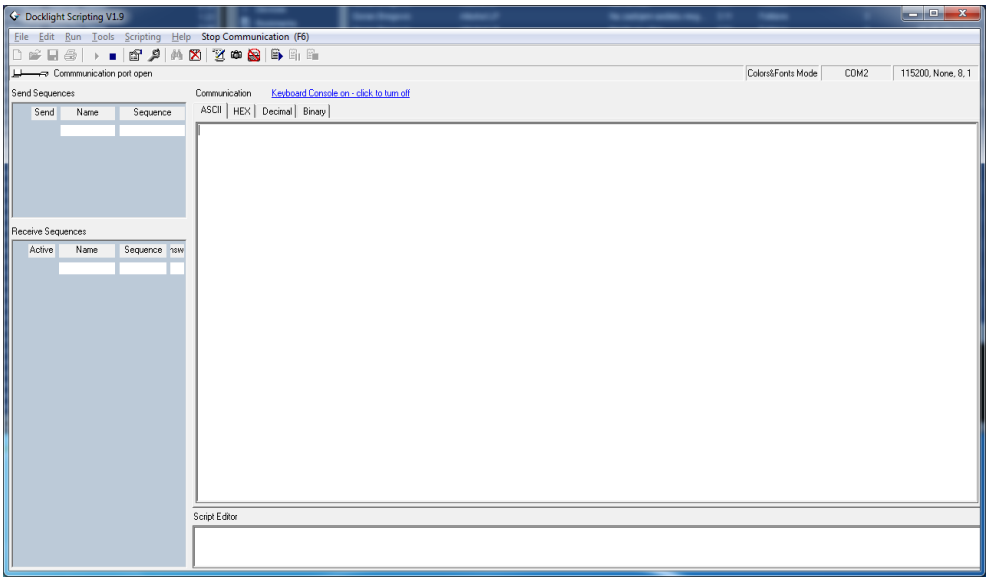

Fig. 1-8 Docklight main window

The main window of Docklight offers quick access to all of the features. The command bar contains practically all the necessary commands to configure, open, close and enable data write to the serial port.

Fig. 1-9 Docklight command bar

The active serial port along with its current configuration is displayed on the right side of the bar. In order to modify the COM port or the configuration a double click on the COM port name (ex COM 2 in Fig. 1-9). The configuration window is displayed in Fig. 1-10.

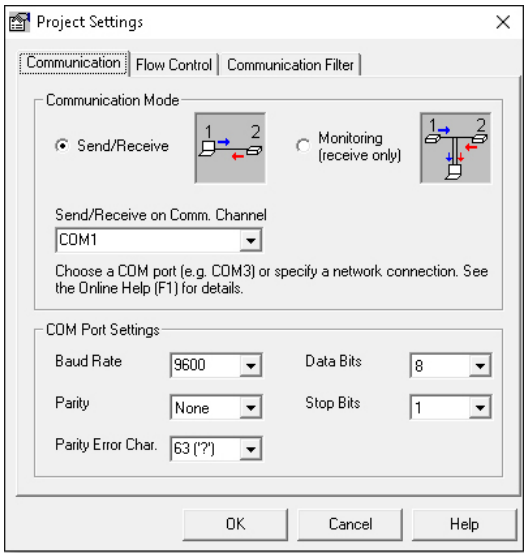

Fig. 1-10 Docklight COM port configuration window

The configuration of the serial port does not imply also the opening of the COM port for receiving and transmission. These operations are made using some of the buttons on the command bar in Fig. 1-9. The buttons that present the most interest are: *Start Communication*, *Stop Communication*, *Keyboard Console On* and *Clear Communication Window*. These commands are highlighted in this order in Fig. 1-11.

Fig. 1-11 Docklight most used commands

The most important commands found on the command bar are those responsible for opening and closing the serial COM port. The first two highlighted buttons in the above figure are responsible for these actions. The opening of the port is activated through the *Start Communication* button and the closing the port through the *End Communication* button. In the moment the COM port has been successfully opened the state is updated below the button bar and *Docklight scripting* is ready to receive data through the serial port which will be displayed in the main window in the currently selected format. The window may be cleared using the *Clear Communication Window*. It is important to mention that opening the communication window will only activate the receive process. Any typed data in the main window will be discarded. In order to activate the transmission of data using the keyboard the *Keyboard Console On* button must the accessed. The status bar will be updated accordingly.

The main window of *Docklight scripting* displays the received and transmitted data in a strictly defined format. Each operating is preceded by a full timestamp along with a

tag that specified whether it is a transmission ([TX]) or a reception ([RX]). Usually the transmitted data are colored in blue and the received data in red. The special characters are also displayed using a simple syntax: the definition of the special character according to the ASCII table between angle brackets. A sample of a short transaction displayed by *Docklight scripting* may be the following snippet.

Code listing 1-1 Docklight scripting communication sample

```
04.01.2016 12:21:51.575 [TX] - AT<CR><LF>

04.01.2016 12:21:52.455 [RX] - <CR><LF>
OK<CR><LF>
```

It is important to understand that the sequence <CR> for example is NOT a sequence of 4 characters. The actual meaning of the notation is the representation of the Carriage Return (CR) character (1 byte long of value 0x0D).

ASSIGNMENT 2: Open *Docklight Scripting*, configure the port for a BAUD rate of 115200 bps, 8 bit per character, 1 STOP bit and no parity. Open the COM port and activate the keyboard transmission feature.

In order to successfully initiate a communication with the GSM modem, as presented in paragraph 1.1.3 , the modem first needs to synchronize its communication port BAUD rate with the one of the host. The procedure, described in the previous sections, states that in order to initiate a proper AUTOBAUD the simple AT command, followed by Carriage Return and Line Feed characters (Enter) needs to be sent to the modem. The AUTOBAUD procedure is successful after an “OK” response from the modem. Such a sequence example is presented in Code listing 1-1. Only after a successful AUTOBAUD procedure, the host may communicate using other AT commands with the GSM modem. If the AUTOBAUD procedure fails (identified by a lack of response from the modem) the only applicable method to restart it is to power off and then power on again the GSM modem.

ASSIGNMENT 3: Power on the GSM modem and, using *Docklight scripting*, initiate a communication with the modem performing AUTOBAUD. Search the AT command manual for the necessary commands that can accomplish the following functions:

- retrieve the RSSI value in ASU
- retrieve the status of network registration
- retrieve the name of the currently selected operator
- retrieve the list of all the operator names known by the GSM modem
- retrieve the IMEI

Send the above found commands to the GSM modem and analyze the response from a syntax point of view.

In order to design and implement a library that is capable to parse and extract the data from the GSM modem, the communication protocol needs to be analyzed first. Only after a full analysis of the protocol, one can design the algorithm to implement it as long with the needed data structures.

First of all it is important to mention that, giving the fact that the transmission bus is a serial interface, the characters are transported one by one, making it mandatory for the parse algorithm to consider a character by character analysis. Considering this aspect the best solution for implementing the protocol would be using a finite state machine approach.

Another important observation is that, even though in the following paragraphs we will extract a general form of the AT command response, exceptions will also be present and will be treated separately. During the first stage of our analysis we will only consider the general form without any exceptions making it easier to design the finite state machine. The exceptions will only be treated when needed.

We will begin our protocol analysis by studying various AT command responses from the GSM modem. Only the responses will be analyzed, the transmitted commands are of no interest, mainly, because they are sent by a host and can be easily formatted accordingly. In the code listings below, several responses are presented. As stated before the TX line, the first line, will be ignored. Our analysis begins from the first RX line in each case.

Code listing 1-2 Command response for simple AT command

```
04.01.2016 12:21:51.575 [TX] - AT<CR><LF>

04.01.2016 12:21:52.455 [RX] - <CR><LF>
OK<CR><LF>
```

Code listing 1-3 Command response for AT+CSQ

```
04.01.2016 20:35:55.686 [TX] - AT+CSQ<CR><LF>

04.01.2016 20:35:57.018 [RX] - <CR><LF>
+CSQ: 27,0<CR><LF>
<CR><LF>
OK<CR><LF>
```

Code listing 1-4 Command response for AT+CREG

```
04.01.2016 20:51:19.717 [TX] - AT+CREG?<CR><LF>

04.01.2016 20:51:21.928 [RX] - <CR><LF>
+CREG: 1,1<CR><LF>
<CR><LF>
OK<CR><LF>
```

Code listing 1-5 Command response for AT+COPS

```
04.01.2016 20:52:36.940 [TX] - AT+COPS?<CR><LF>

04.01.2016 20:52:38.444 [RX] - <CR><LF>
+COPS: 0,0,"Vodafone RO",2 <CR><LF>
<CR><LF>
OK<CR><LF>
```

Code listing 1-6 Truncated command response for AT+COPN

```
04.01.2016 20:53:27.445 [TX] - AT+COPN<CR><LF>

04.01.2016 20:53:29.495 [RX] - <CR><LF>
+COPN: "001010","Test PA128-PA4"<CR><LF>
+COPN: "00101","Test PA128-PA4"<CR><LF>
+COPN: "20201","GR COSMOTE"<CR><LF>
+COPN: "20205","vodafone GR"<CR><LF>
+COPN: "310160","T-Mobile"<CR><LF>
+COPN: "31016","T-Mobile"<CR><LF>
+COPN: "310170","AT&T"<CR><LF>
+COPN: "356110","C&W"<CR><LF>
+COPN: "90115","OnAir"<CR><LF>
+COPN: "90117","Navitas1"<CR><LF>
+COPN: "90118","Maritime Wireless"<CR><LF>
<CR><LF>
OK<CR><LF>
```

Code listing 1-7 Command response for unknown or erroneous command

```
04.01.2016 20:57:30.058 [TX] - AT+ABC<CR><LF>

04.01.2016 20:57:32.476 [RX] - <CR><LF>
ERROR<CR><LF>
```

Taking a close look on the examples above, we can extract the following aspects:

- all responses from the GSM modem contain only printable characters, more specific from ASCII table the interval of characters begin from character 0x20 and ends with 0x7E. The CR and LF characters are also added to the interval.
- each response begins with a new line, more specific with the characters CR and LF;
- each response is concluded with OK followed by CR and LF or ERROR followed by CR and LF;
- there are commands where no data response lines are present, as presented in examples Code listing 1-2 Code listing 1-7;
- there are commands where one line of data response is present which ends again with CR and LF as presented in examples Code listing 1-3 Code listing 1-4 and Code listing 1-5;
- there are commands where more than one line of data response is present. Each data response line ends with CR and LF characters as presented in Code listing 1-6;
- each data response line begins with character '+';
- in the case of the commands containing one or more data response lines the sequence of response lines ends with CR and LF. Also giving the fact that a response line ends with the characters CR and LF, the whole sequence of data response lines ends practically with 2 sequences of CR and LF characters.
- each response line contains characters from ASCII table interval 0x20 – 0x7E but do not contain the CR and LF characters. The CR and LF characters are only used to mark the ending of a data response line.

Based on the observations above the general form of an AT command response may be following:

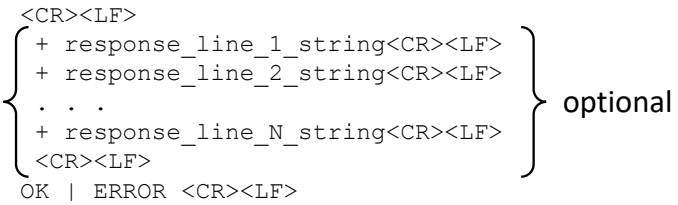

Fig. 1-12 General form of AT command response

The general form presented above may also be described as using a language grammar syntax form as following:

Code listing 1-8 General syntax of AT command response

```
AT_Command_response ::= \r\n at_command_respone_block at_command_final \r\n

at_command_final ::= "OK" | "ERROR"

at_command_response_block ::= response_line_list \r\n | eps

response_line_list ::= '+' response_line_string \r\n response_line_list  | eps

response_line_string ::= [a-zA-Z\!-\@\[-\`\{-\~ ]*
```

The next step in defining the architecture to follow for implementing our current protocol is to design the finite state machine capable in parsing the protocol. Practically everything is resumed to a lexical analysis which is widely used in compiler design [12]. The finite state machine should be oriented in having a successful design of a correct pattern of the protocol taking into account the character by character approach. Such a finite state machine that implement the AT command language may be the following:

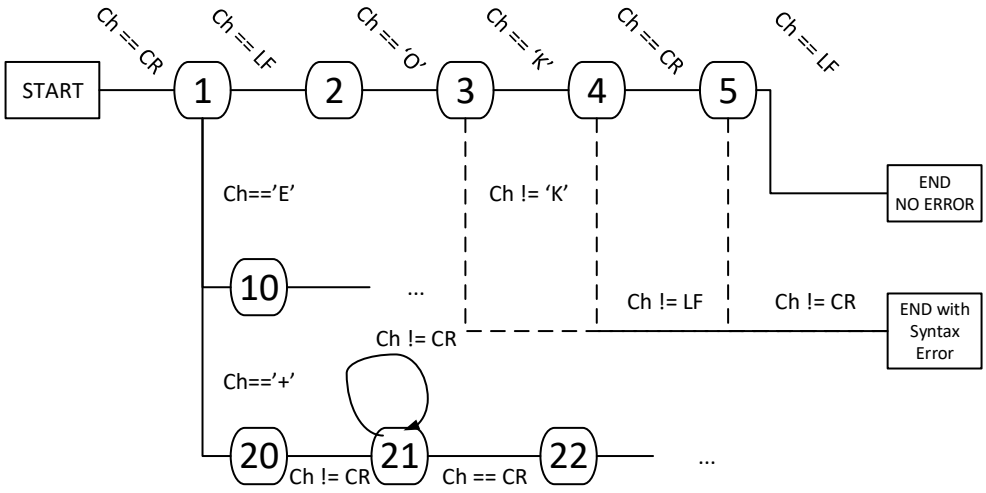

Fig. 1-13 Finite state machine for AT command response language

The finite state machine presented in Fig. 1-13 is incomplete and only gives an example of an approach. Not only that it does not implement all the situation but it also only handles a simple validation of the protocol.

Giving the fact that the result of an actual implementation should also extract the necessary information encapsulated within the protocol, in the next paragraph, we will analyze what data is useful for the end user and what could be the necessary data structures to support it. Taking another short look over the protocol syntax in Fig. 1-12 one can identify some important aspects regarding the usefulness of the data:

- The CR and LF characters are only used for encapsulation within the protocol and they are needed only from a syntactic point of view, but not as a data content component.
- Each response is concluded with the strings “OK” or “ERROR” which should be used not exactly as a string but more as a value to state that the transmitted command was executed successfully by the modem or not. A Boolean value should suffice.
- Most of the AT command responses contain a number of data response lines beginning with character ‘+’ and ending with the CR and LF characters. The actual content of these lines can be treated as string in the first phase. Practically these line contain the main information transmitted by the GSM modem.

Having this analysis made the following more actual data members may be used:

- A Boolean value designating whether the “OK” or “ERROR” strings were returned by the modem
- An array of strings that contain the actual data response lines without the ‘+’ character and without the ending CR and LF characters
- The number of response lines returned by the GSM modem

Given the fact that C programming language will be used for implementation throughout the laboratory, an example of the data structure, implemented in C is provided in in Code listing 1-9. The data types are the standard data types provided by the *stdint.h* library. Also, because the C programming language does not define a standard Boolean data type a 1 byte integer was used instead.

Code listing 1-9 AT Command proposed data structure

```
#define AT_COMMAND_MAX_LINES     100
#define AT_COMMAND_MAX_LINE_SIZE    128

typedef struct
{
  uint8_t ok;
  uint8_t data[AT_COMMAND_MAX_LINES][AT_COMMAND_MAX_LINE_SIZE + 1];
  uint32_t line_count;
}AT_COMMAND_DATA;
```

The finite state machine presented above should finally also extract the data described above, which is encapsulated within the protocol, and save it into a similar structure as the one presented in Code listing 1-9.

HOME ASSIGNMENT: Starting from the finite state machine from Fig. 1-13 write (on paper) a complete finite state machine that is capable of preforming the following functions:

- Parse and identify the correct form of the AT command response syntax from Fig. 1-12. The finite state machine must implement the whole syntax presented above.
- Add in the right places of the newly written state machine the necessary notations to describe where and how a data structure similar to Code listing 1-9 should be completed with the data received via the protocol.

Also write a pseudocode implementation of the finite state machine.

1.3 Laboratory work – GSM AT command language implementation in C

This laboratory application is oriented on the actual implementation of the finite state machine designed in the previous laboratory work. The language that will be used for implementation is ANSI C. The developing environment available on the laboratory computers will be a version of Visual Studio higher than Visual Studio 2010.

Practically the students will have to develop a library (presented as a separate couple of a C code file along with its C header file), which provides a function that implements the finite state machine. The function that will be exported outside the library, via header files, needs to be per character based, with minimum one parameter specifying de current character. This means that the function has its main functionality based on the current character. The function has to return the state of its execution. The most common states of function returns could be:

- the function has finished without an actual result and more characters are needed to finish the state machine
- the function has finished and has found a valid AT command response
- the function has finished and has exited with an error

The function will not be allowed to contain any cycles and its execution is needed to be per character.

An example function prototype along with an enumeration type return value may be the following:

Code listing 1-10 Proposed AT Command parse function prototype

```
typedef enum
{
  STATE_MACHINE_NOT_READY,
  STATE_MACHINE_READY_OK,
  STATE_MACHINE_READY_WITH_ERROR
}STATE_MACHINE_RETURN_VALUE;

STATE_MACHINE_RETURN_VALUE at_command_parse(uint8_t current_character);
```

The return values proposed in the enumeration type have the meaning described above and should ONLY REFLECT THE RETURN OF THE STATE MACHINE REGARDING SYNTAX. No confusion should be made between the return values of the function implementing the state machine and the OK/ERROR strings returned by the GSM modem. A command that finished with the OK or ERROR string may be correct from a syntax point of view.

The most used method to implement a finite state machine in the C programming language is using switch-case statements. An example of an implementation of the first 2 states presented in Fig. 1-13 may be the following:

Code listing 1-11 Proposed AT Command parse function implementation example

```
STATE_MACHINE_RETURN_VALUE at_command_parse(uint8_t current_character)
{
  static uint32_t state = 0;
  switch (state)
  {
     case 0:
     {
            if (current_character == 0x0D)
            {
               state = 1;
            }
            break;
     }
     case 1:
     {
            if (current_character == 0x0A)
            {
               state = 2;
            }
            else
            {
               return STATE_MACHINE_READY_WITH_ERROR;
            }
            break;
     }
     case 2;
     {
            if (current_character == 'O')
            {
               state = 3;
            }
            else
            {
               return STATE_MACHINE_READY_WITH_ERROR;
            }
     }
  }
  return STATE_MACHINE_NOT_READY;
}
```

Having a small analysis on the example above we can state that the parse function that needs to be implemented must work using a character by character approach. This parse function is called for each character. Furthermore, the function has no knowledge from where and how the character are acquired. Having this as a behavior of the function we can observe that a lot of variable need to retain their execution from one function call to another. Such variable may be: the current state, indexes in the string array, index of the character in the string within the string array, number of line, etc. This can be achieved either by using global variable in the code file containing the function or by using static variable declared inside de function similar to the state variable in the example above.

It is noted that the function does not return the structure containing the data extracted from the protocol mainly because returning such a big result on the stack is not recommended. Other approaches have to be considered. For example having the structure be transmitted (with pointer reference) to the function at each call or by using it declared

globally. The caller function that uses the parse function must know when to access the resulted data structure depending on the return value of the parse function.

The main idea of this laboratory is for the students to implement this function in C using a standard C compiler in order to be platform and system independent. The exported parse function, the data structures and all other functions needed to complete this task must be included in a library. The library in C needs to be formed out of 2 files: a C file and a header file which contains the external declarations. Having this approach assures that this library may be then included in any C environment, for example in a Keil uVision project, which will generate the microcontroller image needed on the learning kit.

Another important aspect is that the implementation may not contain anything outside the standard C libraries in order to be than used on a microcontroller. Furthermore it is highly recommended that the data types would be the ones defined by the standard C library *stdint.h*.

Students may use any standard C environment for developing console applications in order to develop and the AT command parse library. However, on the computers in laboratory only a version of Microsoft Visual Studio will be available.

In order to create a standard C console application in Visual Studio the following steps need to be taken: in the File Menu -> New -> Project; in the right tab select Visual C++ then Win32 Console Application in the middle window. The Win32 Application Wizard will open. In this stage click on Next (and not Finish) in order to configure the project. The best way to obtain a simple, standard project is to select Console Application and check the Empty project option as present in Fig. 1-14.

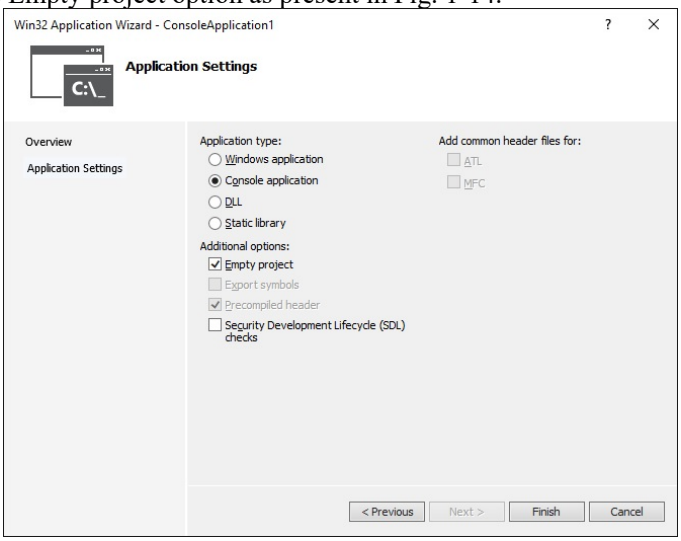

Fig. 1-14 Visual Studio Win32 Console C application project wizard settings

After clicking on the Finish button the Visual C++ Console Application blank project will be created without any source or header files added. A very important aspect is that when creating and saving a C code file the extension needs to be .c and NOT .cpp in order to force the compiler to treat the code using the ANSI C standard.

The main program, that will be developed, will be a tester for the parse function and practically it will read text files containing various AT command responses. The files will be read character by character and the program will call the AT command parse function for each character. The program will then test the return value and print the command that was just extracted from protocol when the state machine announces a normal finishing. In case the stream was incorrect an error message describing the situation will be printed. A possible state diagram of the test program is illustrated in Fig. 1-15.

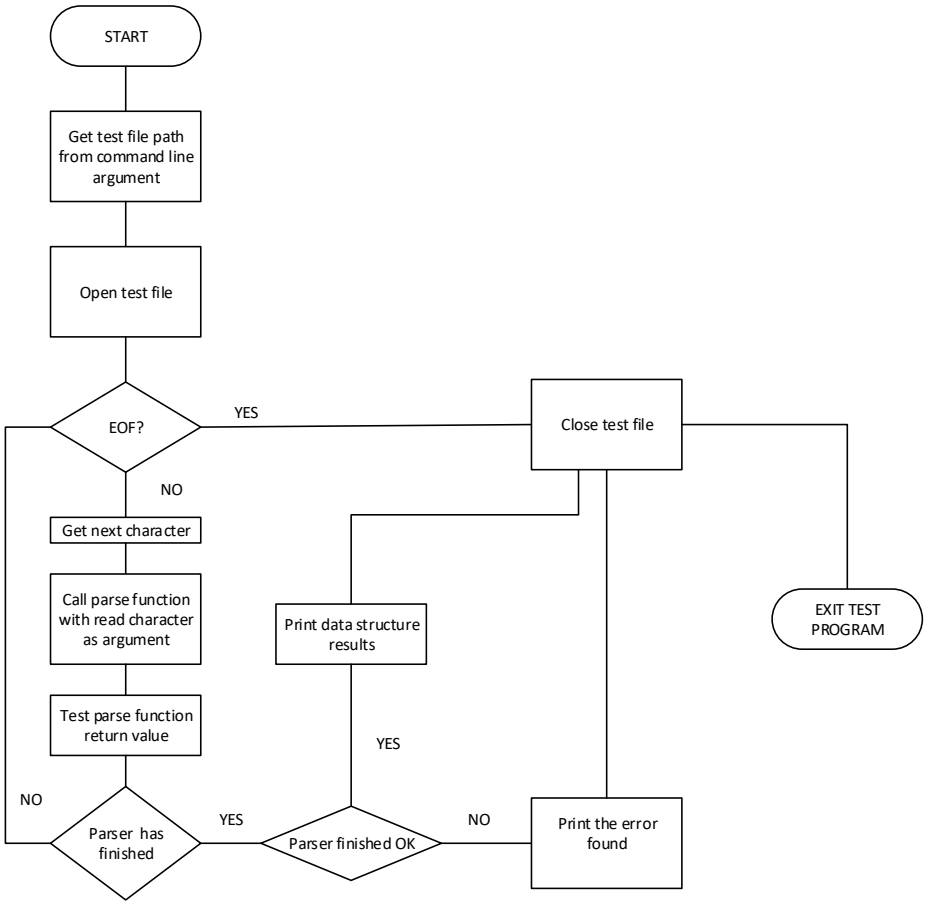

Fig. 1-15 AT command parser test program state diagram

The only aspect that needs to be clarified before an actual implementation can begin is the method on how can the test files be generated. One test file should only contain one single AT command response. It is advisable not to use one big test file containing all the test cases (numerous AT command responses). In order to create a test file the students may use either the actual responses from the GSM modem taken from a communication using *Docklight scripting* or they can use the examples presented in the previous

laboratory work: Code listing 1-2, Code listing 1-3, Code listing 1-4, Code listing 1-5, Code listing 1-6, Code listing 1-7.

One important question could be how to generate the special character CR and LF? Furthermore, how can we be assured that these characters are generated correctly? In a Windows system a new line, a press of the Enter key, usually generates a <CR><LF> sequence. However, in a UNIX system, the Enter key, only generates one of the 2 characters. In order to establish how the new line characters are generated in a text file, an abstract, encoded representation should be displayed.

A famous text editor, which is capable of displaying a notation for the special characters (similar to how Docklight scripting does), is Notepad++. In order to configure Notepad++ to display the special characters we need to select: in the menu bar -> View -> Show Symbol -> Show End of Line. Make sure that the Show End Of Line menu item is checked. An example of the response of AT+CREG command used for a test file in Notepad++ with the representation of the end of line characters can be found in Fig. 1-16.

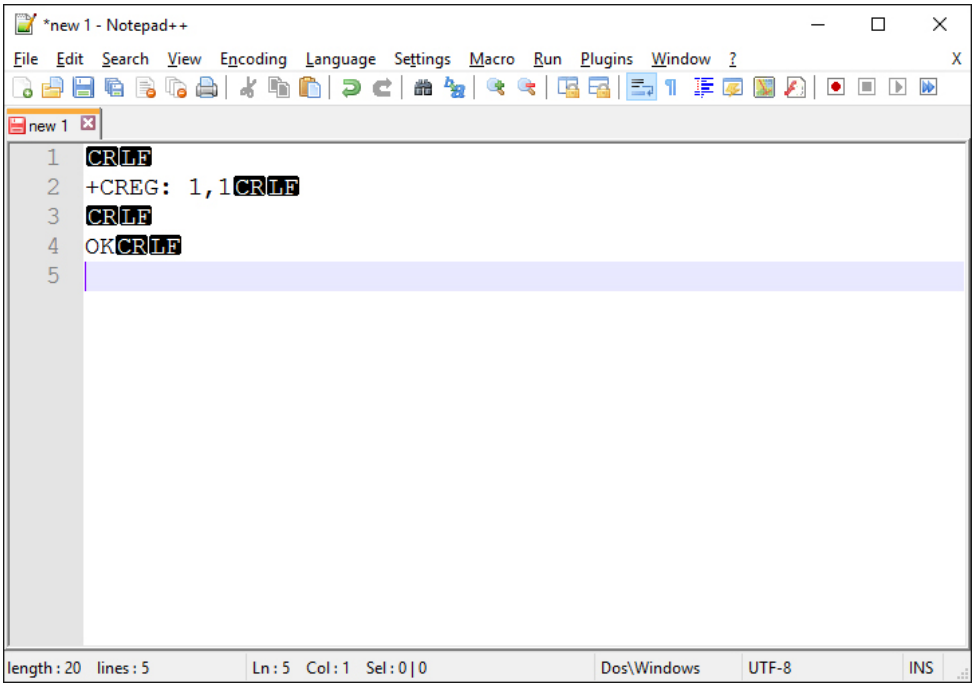

Fig. 1-16 AT command test file in Notepad++

ASSIGNMENT: Write an ANSI C library that implements an AT command parser and extracts the data transported by the protocol. Write an ANSI C standard program to test the AT command parser library using test files as suggested in this current laboratory work. Test files should cover all possible situations, both correct, and incorrect as described in the current and previous laboratory works.

The C program and library should not use any non-standard libraries. The test program should take the test file path as an argument and print the status. The library should be isolated from the test program as suggested. Furthermore, in order to obtain the exact data from the test files open the files in BINARY mode rather than text mode. In text mode, new line interpretation could cause significant issues to both the test program and the AT command parser library itself.

Each student group will have to make their OWN and UNIQUE implementation of the AT command parser library. The time interval for this assignment is 2 weeks.

HOME STUDY: For the next laboratory: review the documentation of the driver manual (presented and the end of the laboratory works in this document) and the *Doxygen* documentation in order to be able to write programs on the learning kit to perform the following: LED blinking, software timer and UART communication.

1.4 Laboratory work – Introduction to The Keil uVision environment

This laboratory work contains the first exercises that will be conducted directly on the learning kit. Furthermore, all the following laboratory works will be oriented on developing on the hardware target. All of the implementations, as stated before, will be made using ANSI C programming language and the developing environment will be Keil uVision. Keil is one of the most popular environment for application development on microcontrollers based on ARM architectures. It is a complex and stable environment and provides various tools for debugging, support for external debuggers and programmers as well as a simulation and emulation environment.

Before continuing with the laboratory work some connections on the learning kit need to be made. Using the information in paragraph 1.1.2 and a provided cable, make sure the following connections are made:

- The serial interface of the GSM modem is connected to the UART_3 interface of the microcontroller.
- The UART_0 interface of the microcontroller is connected to the external DB9 serial interface

The following paragraphs will describe some of the base functions of the Keil uVision development environment. The main window of the environment is presented in Fig. 1-17.

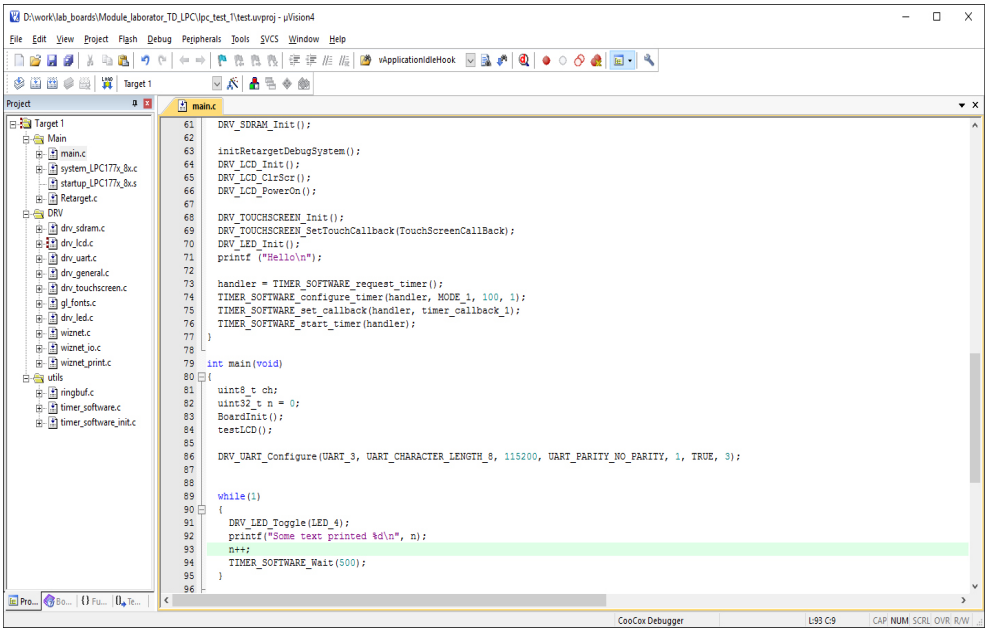

Fig. 1-17 Keil uVision environment

On the left side of the window, a project explorer which displays the project file tree is available while the code files are opened using tabs in the middle part of the window. The most important functions in Keil uVision are the following:

- Program compilation – it may be done by selecting Build Target from the Project Menu or by finding the correct button in the button bar
- Project clean – it may be done by selecting Clean Target from the Project or by finding the correct button in the button bar
- Download compiled program into the microcontroller – it may be done by selecting Download from the Flash menu. Usually after a download the microcontroller is being reset and begin the execution of the newly downloaded program
- Erasing the microcontrollers flash memory – from Flash menu selecting Erase
- Start simulation or emulation – from Debug menu select Start Stop Simulation. Same option is used for exiting simulation/emulation
- Watching variables – in simulation/emulation mode select the desired variable, right click and select Add variable to watch.

In both modes, simulation and emulation, the environment give the user access both to the microcontrollers core registers and also to the peripheral special function registers by accessing the menus.

One of the most important aspect of Keil is that it can work either as a simulator or as an emulation using a debugging hardware directly on the target. The

simulation/emulation mode is practically transparent to the user. In order to change between the 2 modes, a right click on the project name (in our case Target1) in the project explorer on the right will reveal a menu where Options needs to be selected. This selection will bring up the Project options window where the following settings can be found in the Debug tab:

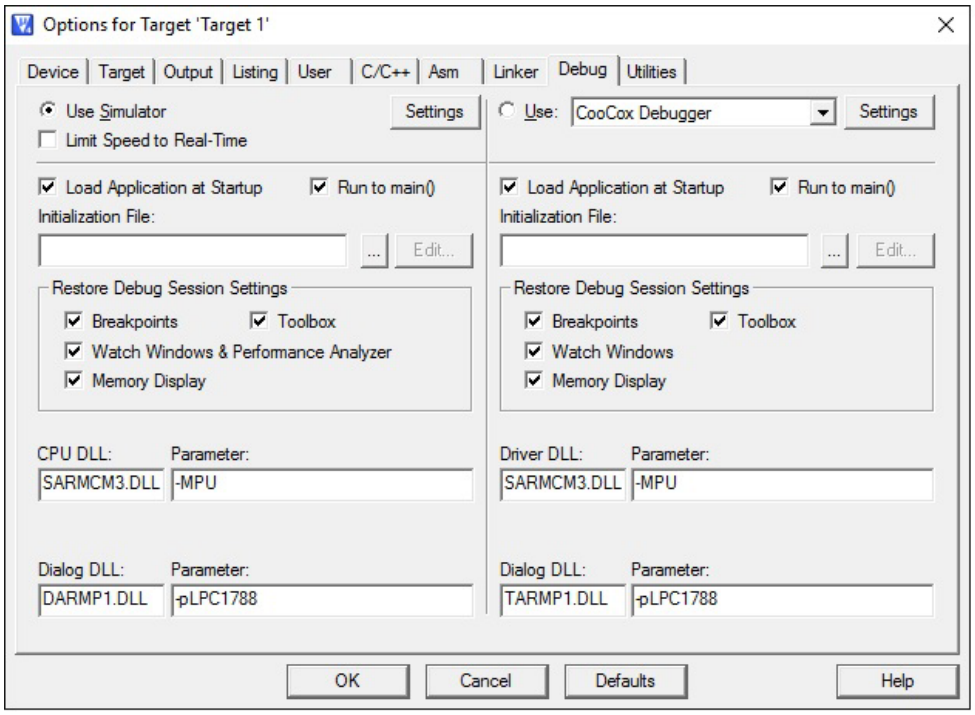

Fig. 1-18 Keil uVision project options

On the left side of the project window the Use Simulator setting selects the simulator. Selecting the corresponding option on the right side of the window will select the debugger and the emulation will be switched to, instead. In our case the debugger used will be CooCox Debugger.

ASSIGNMENT 1: Search through the menus of Keil uVision and find how the following function may be accessed: In debug mode: breakpoint management (adding, disabling, enabling, removing), access to core registers and special function registers of the periphery, adding watch for variables, run program, stop run program, step into, step out.

Giving the fact that the main idea of all the laboratory work sessions is to concentrate on the communication with the GSM modem and on the implementation of various functions of the GSM modem it is out of the scope of this laboratory to present the peripherals of the microcontroller. It is also out of scope to demand from the students to write drivers for the peripherals. Such drivers are needed in order to provide a stable

environment to implement all the requested tasks. Having this into consideration, a driver library is provided to the students. The most important drivers implemented by the driver library are the ones related to UART communication, LED management, LCD and TOUCHSCREEN management and printf redirection for debugging.

In order for the students to be able to use the library a full documentation is presented at the end of all the laboratory works explaining how every driver module may be used. Code examples are presented also. Furthermore, in order to facilitate using the driver libraries functions a Doxygen documentation is available online which presents a low level documentation of the manual.

Another provided material to the students is a basic fully configured Keil uVision project that contains both the driver library as well as examples. This basic project will be used from this point on for building the applications. The students should use this basic project as a starting point and not make their own project from scratch as this could prove to be difficult and time consuming which again is out of the scope of this discipline.

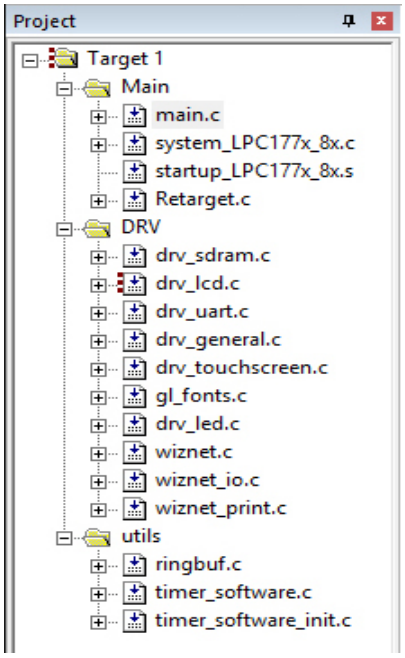

Fig. 1-19 Basic Keil project file structure

As presented in Fig. 1-19 the provided project is structured using 3 project folders. The utils folder contains the ring buffer data structure and the timer software. These 2 module are used by all of the other driver modules but may also be used by the students in their development, especially the software timer. Another folder, DRV, contains the actual driver modules. The main folder contains the microcontroller initialization files and the code file containing the main function. Students are encouraged to insert their code file either in this folder or in a newly created folder but NOT in the folders DRV and utils.

The main.c file contains all the initializations required by the driver library and microcontroller. One of the most important function that is called first in main is *BoardInit()*. This function is responsible for initializing all the driver modules as well as the software timer system and the printf redirection (UART 0 of the microcontroller). The next function called by main is *testLCD()* which displays a test on the LCD containing the Romanian National Flag and the Hello text. This call may be omitted thus it is only a test for the LCD. The rest of the main function content is made up of examples of usage for some of the driver library usage.

ASSIGNMENT 2: Write a short program using the basic Keil project provided, which blinks the LEDs on the board with a frequency of 1 Hz in an infinite loop. Use the LED driver module and the timer software wait module which uses an internal software timer to generate waits. Find the appropriate functions to accomplish the task.

ASSIGNMENT 3: Modify the program in the previous assignment to use a dedicated timer software. The program should poll the state of the software timer. Before using it, the program needs to request a software timer, configure it in the correct mode and then start the configured timer.

ASSIGNMENT 4: Modify the program in the previous assignment to use the software timer with a callback to announce the interrupt event. Note that the callback function is executed in processor interrupt context.

ASSIGNMENT 5: Modify the program in the previous assignment to send a message using printf each time the LED blinks. In order to see the message please open *Docklight scripting* on COM1 with the following settings: 8 bit per character, no parity, 1 stop bit and a BAUD rate of 115200 bps.

HOME ASSIGNMENT: Please make sure the AT command parser library is functioning properly and is ready for integration in the next laboratory session.

1.5 Laboratory work – Integration of the AT command parser in Keil

This laboratory work focuses on integrating the AT command parse library previously developed and tested in a standard ANSI C environment within the provided basic Keil project. The idea is to establish a short communication with the GSM modem, but from a program written on the microcontroller. The outcome of this laboratory work needs to be an infinite loop that requests the RSSI values from the modem and prints it on the debug console using *printf* once per second in both ASU and dBmW values.

In order to accomplish this task example, methods for integration will be presented along with state diagrams on how the program should work. The design method considered will be TOP-DOWN. All the methods presented in the following paragraph

are not mandatory to be used by the students. They are free to choose their own implementation.

On a hardware level the attendees need to make sure that the modem is power on by pressing the button on the board as described in chapters 1.1.2 and 1.1.3 .

First of all, the UART_3 interface of the microcontroller needs to be configured with the following parameters: a BAUD rate of 115200 bps, 1 STOP bit, 8 bits per character, no parity and buffered mode enabled. The main function, that needs to be studied for accomplishing the configuration of UART_3, is *DRV_UART_Configure*.

The next step, that needs to be taken, is to provide the necessary sequences in order to perform the modem's AUTOBAUD as described in chapters 1.1.2 and 1.1.3 . Practically an AT<CR><LF> sequence need to be sent to the modem prior of any other communication. It is advisable to send this sequence several times with a temporization of 1 second to insure the modem performs the AUTOBAUD. Sending 3 of this sequence may suffice. An example of how this can be accomplished may be found in following code:

Code listing 1-12 Example of sending AUTOBAUD sequence to modem

```
.....
const char at_command_simple="AT\r\n"
...
int main(void)
{
  ...
  DRV_UART_Write(UART_3, at_command_simple, strlen(at_command_simple);
  TIMER_SOFTWARE_Wait(1000);
  ...
}
```

For making things easier it is not necessary for the program on the microcontroller to wait for an answer from the GSM modem. We can assume that after 3 or 4 sequences sent to the modem, the AUTOBAUD is successful. Even if it is not successful the rest of the following commands will receive no answer for the modem.

Only after sending these sequences we can begin "talking" to the GSM modem. As stated above, considering a TOP-DOWN design method the first level, in our case, could be the infinite loop performing the signal request and print once per second on the debug terminal. A short example of C like pseudo-code can be found in the following code snippet:

Code listing 1-13 Top level infinite loop pseudo-code example

```
.....

AT_COMMAND_DATA data_structure;
const char at_command_simple="AT\r\n"
const char at_command_csq = "AT+CSQ\r\n"
timer_software_handler_t my_timer_handler;
...

int main(void)
{
  ...
  uint32_t rssi_value_asu;
  uint32_t rssi_value_dbmw;
  ...
  while (1)
  {
     if (TIMER_SOFTWARE_interrupt_pending(my_timer_handler))
     {
            ExecuteCommand(at_command_csq);
            if (CommandResponseValid())
            {
               rssi_value_asu = ExtractData(&data_structure);
               rssi_value_dbmw = ConvertAsuToDbmw(rssi_value_asu);
               printf(...);
            }
            TIMER_SOFTWARE_clear_interrupt(my_timer_handler);
     }

  }
  ...
}
```

In Code listing 1-13 an idea of how the top level infinite loop should look like is presented. All the necessary functions need to be implemented of course. The timer software in the above example is used to make a temporization. The execution of commands have to be made once per second. There could be another solution for temporization like using *TIMER_SOFTWARE_Wait* function but in this case the temporization would have been blocking which is not recommended. In the example below the configuration of the software timer was omitted thus it needs to be implemented.

The most important function, that needs to be addressed, is the *ExcuteCommand* function. This function has as main roles to send the command given as parameters and to wait for a correct response. Making this statement a possible pseudo-code function body may be:

Code listing 1-14 ExecuteCommand function pseudo-code example

```
void ExecuteCommand(const char *command)
{
  SendCommand(command);
  GetCommandResponse(command);
}
```

The SendCommand function has practically one important role which is the actual command string send over the UART line. The only observation here is that, in order to insure a clean buffer environment when initiating a new command, it would be best to first flush the buffers of the serial interfaces. Such an idea is presented in the following code snippet:

Code listing 1-15 SendCommand function pseudo-code example

```
void SendCommand(const char *command)
{
  DRV_UART_FlushRX(UART_3);
  DRV_UART_FlushTX(UART_3);
  DRV_UART_Write(UART_3, command, strlen(command));
}
```

The last and probably the most important function that needs to be implemented is the GetCommandResponse function. Its main role is to wait for a valid response from the GSM modem in a given period of time. A timeout mechanism using a software timer needs to be implemented here mainly because we need to consider that the GSM modem is not a reliable communication partner thus it may crash. In such a situation we need to insure that our software running on the microcontroller detects this situation and does not get into a blocking state. Giving the fact that a software timer will be used, a prior configuration before our main infinite loop needs to be taken into consideration. Also, the software timer handler needs either to be declared globally or sent to the GetCommandResponse function as parameter in order to have access to it.

Furthermore, it is very important to mention that the GetCommandResponse function will be the one responsible in calling our previously developed library for parsing and extracting data from the GSM modem.

An example of a pseudo-code implementation for the GetCommandResponse function may be found in the following code snippet:

Code listing 1-16 GetCommandResponse function pseudo-code example

```
...
timer_software_handler_t my_handler;
...
void GetCommandResponse()
{
  uint8_t ch;
  BOOLEAN ready = FALSE;
  TIMER_SOFTWARE_reset_timer(my_handler);
  TIMER_SOFTWARE_start_timer(my_handler);
  while ((!TIMER_SOFTWARE_interrupt_pending(my_handler)) && (ready == FALSE))
  {
     while (DRV_UART_BytesAvailable(UART_3) > 0)
     {
            DRV_UART_ReadByte(UART_3, &ch);
            if (at_command_parser(ch) != STATE_MACHINE_NOT_READY)
            {
               ready = TRUE;
            }
     }
  }
}
```

The rest of the functions presented in our TOP-DOWN analysis will not be discussed further, thus they will be designed by the attending students.

All that was analyzed and described above may be synthetized in a state chart diagram as the one presented in Fig. 1-20.

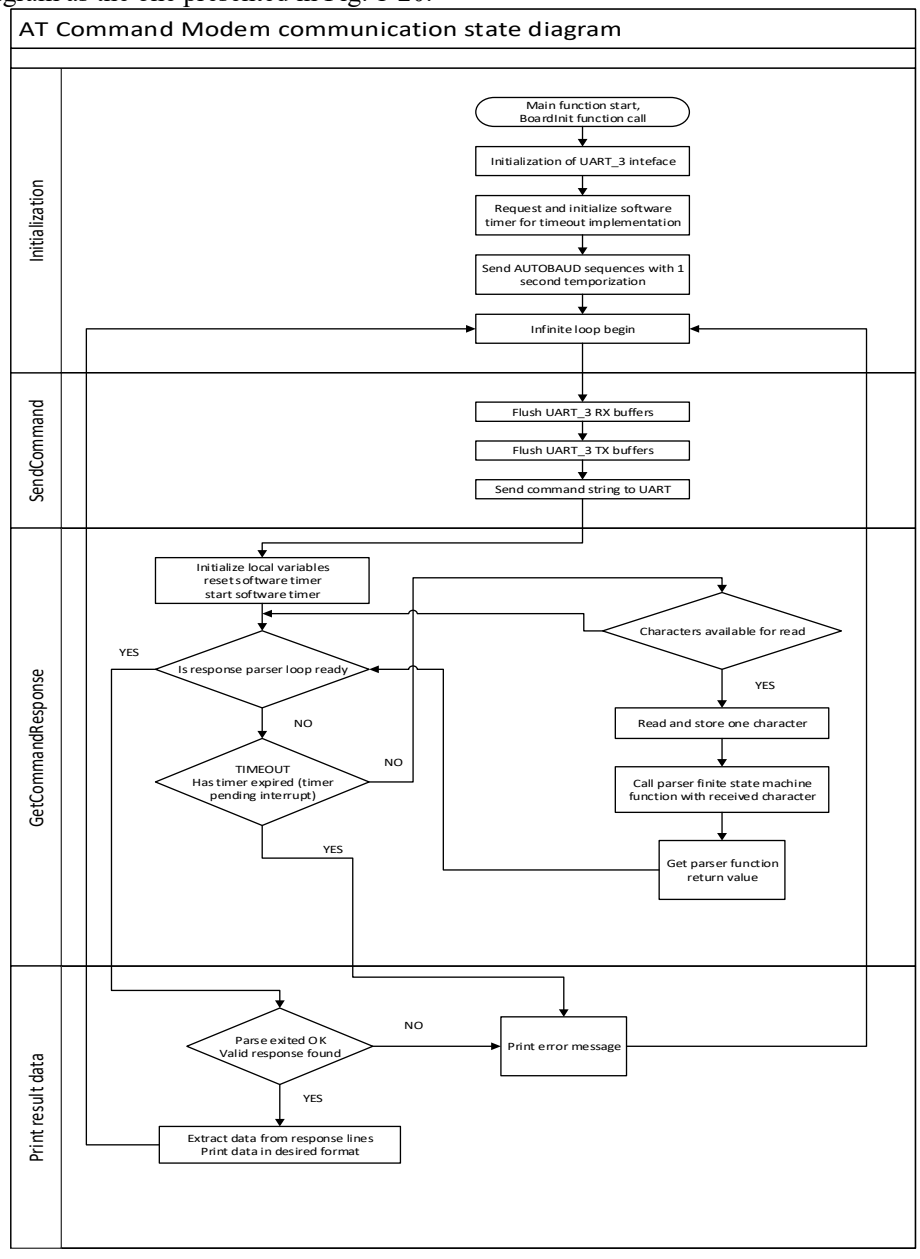

Fig. 1-20 GSM Modem AT command language communication software diagram

ASSIGNMENT: Integrate the AT command parser library developed and tested in the previous laboratory work into the basic Keil project provided, having the explanations in this laboratory work as a starting point. It is mandatory that the AT command parser library should not be copy-pasted into the main.c file but added as separated files (a .c and .h) in the main folder of the Keil project. The outcome of the program on the microcontroller should print the RSSI value received via the information return by the AT+CSQ command. Search the AT command manual for the syntax and how the result is presented. The RSSI value should be printed in both ASU units and dBmW units. The transformation from ASU units (as AT+CSQ returns) to dBmW can be made using the formula [13]:

$$dBmW = 2 \cdot ASU - 113 \quad (1\text{-}1)$$

The output of the printing should be for example:

Code listing 1-17 Example output of signal printing

```
04.01.2016 20:57:32.476 [RX] - GSM Modem signal 29 ASU -> -55 dBmW
04.01.2016 20:57:33.476 [RX] - GSM Modem signal 29 ASU -> -55 dBmW
```

NOTE: Please remember to power on the modem before anything! The GSM modem will not respond until a power cycle.

HOME ASSIGNMENT: Please find and read, from the GSM Modem AT Command Manual the syntax and usage for the following functions:

- State of network registration (AT+CREG)
- Name of network operator (AT+COPS)
- Modem IMEI (AT+GSN)
- Modem Manufacturer (AT+GMI)
- Modem Software Version (AT+GMR)
- Cell ID and Location ID (extended AT+CREG)

1.6 Laboratory work – Implementation of modem status commands

This laboratory work focuses on requesting more information from the GSM modem. Using the program implemented above where the signal RSSI value is printed on the debug terminal once per second, the student needs to add other prints in order to print along with the RSSI value the following information extracted from the modem using the necessary commands:

- State of network registration (AT+CREG)
- Name of network operator (AT+COPS)
- Modem IMEI (AT+GSN)
- Modem Manufacturer (AT+GMI)
- Modem Software Version (AT+GMR)

- Cell ID and Location ID (extended AT+CREG) - optional

At the first glance of the request of the laboratory work, things seem trivial. The only aspect that needs to be taken into consideration is that for some of the commands needed to extract the requested information the response syntax is a little different. While most of the command responses respect the syntax defined in the previous laboratory works, there are some commands where we should consider exceptions.

Let us consider the following response of the AT command that requests the Manufacturer Identification from the GSM modem:

Code listing 1-18 Command response for AT+GMI

```
04.01.2016 20:35:55.686 [TX] - AT+GMI<CR><LF>

04.01.2016 20:35:57.018 [RX] - <CR><LF>
SIMCOM_Ltd<CR><LF>
<CR><LF>
OK<CR><LF>
```

We can observe that in Code listing 1-18 the command response is slightly different from the responses presented so far. The difference is that the response line does not begin with the '+' character. Practically this is one the exceptions from the AT command language syntax. The limitation is that if such an exception is present there will be only ONE response line. No other cases shall be considered in this situation.

If the previously developed parser is executed on a syntax such as the one in Code listing 1-18, it will surly fail. The idea is that the parser should be modified in order to support such an exception. There are many ways to modify the parser. The simplest method is to inform the parser about the exception that it will encounter. This may be accomplished either by using a global flag that will be set prior to sending the AT+GMI command and reset after the response has been received or by using a parameter that will be sent to the parser.

OPTIONAL ASSIGNMENT: If needed, in order to test the output of the commands above make the connections of the board similar to the ones in the first assignment in laboratory work 1.2 in order to be able to connected to the modem directly with *Docklight scripting* as described in the previous laboratory works. After testing the commands come back to the previous connections on the board.

ASSIGNMENT: Add the commands above to the implementation and print the results on the debug terminal. Modify the AT command parser in order to support the newly introduced exceptions. Extract and interpret the data from the above commands in order to print a human readable status. The program should in final print once per second a status such as:

Code listing 1-19 Example output of signal printing

```
GSM Modem signal 29 ASU -> -55 dBmW
State of registration: Home network
Operator name: RO Vodafone RO
Modem IMEI: 123456789012345
Modem manufacturer: SIMCOM_Ltd
Modem software version: XXXXXXXXXXXXXXXXXXXX
Cell ID: 0x1234
Location ID: 0x1234
```

All the information should be printed in an infinite loop once per second. The Cell ID and Location ID are extract from the long version of the AT+CREG commands. Search the documentation on how to obtain this.

HOME ASSIGNMENT: Please study from the AT Command Manual of the modem what are the commands for managing the SMS messages. Identify how these commands should be used.

1.7 Laboratory work – SMS management

This laboratory work concentrates on introducing the SMS management to the attending students. The main aspects that will be addressed are how the SMS messages are stored in the SIM card, reading SMS messages, sending SMS messages and deleting SMS messages.

The SMS messages are stored in the memory of the SIM card. The GSM modem does not have any additional memory of storing external data and uses the SIM cards memory for this. The SIM card has an organized memory destined specially for storing SMS messages. The SMS memory is organized in 10 slots, one for each SMS message. Each SMS message is stored in one slot of the SIM cards SMS memory immediately after the SMS messages is received from the GSM network. The SMS messages is stored in the first free slot of the SIM card.

A problem appears when all the SMS slots in the SIM card are full. In this situation the GSM Modem, in most of the cases, ignores a newly arrived SMS message from the network. In many cases, in this situation, the GSM modem doesn't even announce that a new SMS message is pending. In order to avoid such a situation the host connected to the GSM modem must insure that an SMS memory slot is free all the time.

The first issue that we will address is the one related to reading all the SMS messages from the SIM card. The AT commands responsible for this action is AT+CMGL. If the documentation for this command is checked, one can find out that there are some parameters accepted for this command in order to filter the SMS message listing. In our case we will not use any filters because we intend to read all the SMS messages.

As found in the previous laboratory work we will encounter another exception from the general form of an AT command response. The exception will appear in the response

line string. As it can be found in Code listing 1-8 the response line string contains any alphanumeric characters togheter with punctuation signs and spaces but not CR and LF characters, thus these latter characters are used to identify the finishing of such a response line string. Practically the accepted characters are from character with HEX ASCII value 0x20 to character with HAX ASCII value of 0x7E. The exception is that, for the AT+CMGL command, when listing SMS messages, an extra CR LF sequence appears in the middle of a response line string which may be mistaken with a marking of such a line ending.

An example of the output of AT+CMGL command may be found in the following code listing:

Code listing 1-20 Example output of SMS listing command

```
08.01.2016 09:02:29.097 [TX] - AT+CMGL="ALL"<CR><LF>

08.01.2016 09:02:35.421 [RX] - <CR><LF>
+CMGL: 1,"REC READ","Notificare","","14/12/15,15:32:28+08"<CR><LF>
Creditul existent este insuficient pentru a trimite mesajul.<CR><LF>
<CR><LF>
+CMGL: 2,"REC READ","Notificare","","14/12/15,15:32:35+08"<CR><LF>
Creditul existent este insuficient pentru a trimite mesajul.<CR><LF>
<CR><LF>
+CMGL: 3,"REC READ","Vodafone","","14/09/30,15:49:49+12"<CR><LF>
i si iti activezi o optiune cu mai mult trafic inclus.<CR><LF>
<CR><LF>
+CMGL: 4,"REC READ","Vodafone","","14/09/30,15:49:51+12"<CR><LF>
Numarul Cartelei tale Internet este: 40732947310. Acest numar iti foloseste la
crearea contului Cartela Internet si la reincarcarea directa<CR><LF>
<CR><LF>
+CMGL: 5,"REC READ","Vodafone","","14/09/30,15:49:51+12"<CR><LF>
Bine ai venit!O data cu prima reincarcare se activeaza automat o optiune cu
trafic inclus de internet. Vei primi vesti!<CR><LF>
<CR><LF>
+CMGL: 6,"REC READ","Notificare","","14/12/15,15:32:42+08"<CR><LF>
Creditul existent este insuficient pentru a trimite mesajul.<CR><LF>
<CR><LF>
+CMGL: 7,"REC READ","Vodafone","","14/09/30,15:50:30+12"<CR><LF>
Oferta Internet cu 150MB trafic a fost dezactivata. Reincarca acum sa beneficiezi
de o optiune cu trafic inclus. Alege-ti oferta potrivita pe
www.vodafone.ro.<CR><LF>
<CR><LF>
+CMGL: 8,"REC READ","MyVodafone","","14/09/30,15:52:31+12"<CR><LF>
Codul pentru activarea contului MyVodafone este : 31395062.Acest mesaj este
confidential. Nu oferiti unor persoane necunoscute codul secret!<CR><LF>
<CR><LF>
+CMGL: 9,"REC READ","MyVodafone","","14/09/30,15:52:57+12"<CR><LF>
Contul tau MyVodafone a fost creat cu numele de utilizator 0732947310. Multumim
pentru alegerea de a utiliza serviciile noastre online.<CR><LF>
<CR><LF>
+CMGL: 10,"REC READ","+40726043403","","14/12/11,16:51:04+08"<CR><LF>
Blue screen<CR><LF>
<CR><LF>
OK<CR><LF>
```

In order to establish how to treat the newly introduced exception we should carefully examine the response presented above. At a first glance, there seems to be no exception, the response seems to be accordingly to the rule. Let us consider for example the last SMS message:

Code listing 1-21 One line of example output of SMS listing command

```
+CMGL: 10,"REC READ","+40726043403","","14/12/11,16:51:04+08"<CR><LF>
Blue screen<CR><LF>
```

The line begins accordingly with the '+' character followed by a string line. We can observe that the first string line contains a text designating whether the SMS message was read or not, then the phone number which sent the SMS message followed by the time stamp. After the timestamp a sequence of CR LF characters is found followed by the actual text message which end accordingly to the rule with the CR LF sequence. Practically the only exception that needs to be treated is that an extra CR LF sequence is present in the middle of the response line string.

Following the same idea as the one in the previous laboratory work the state machine is not mandatory to automatically detect this exception. The AT command parser can be informed that it must treat this exception trough a global flag or a parameter.

ASSIGNMENT 1: Modify the AT command parser to support the SMS listing command. Add to the 1 per second print sequence the printing of the SMS messages present in the SIM card under the following form:

SMS message <i> - phone_number – SMS message text

Code listing 1-22 SMS print example

```
SMS message 1 – 0722222222 – Some text message
SMS message 2 – 0733333333 – Some other thext message
```

Please extract only the needed data from the SMS listing strings in order to print as presented in the above example.

The issue to be addressed in this laboratory work is to implement the command responsible for the removal of a SMS message from its slot. This command does not introduce any additional issues.

ASSIGNMENT 2: Find the necessary information in the GSM Modem AT command manual in order to implement a function, which is able to delete one message from a given slot. The response from this command is known to respect the general form of the AT command response language grammar.

The last command that will be addressed in this laboratory work is the one capable of sending one SMS message. This command does not require special attention at its response but it is slightly different when sending it. A very small state machine needs to be implemented. An example of a communication involving sending an SMS message is presented in the following code snippet.

Code listing 1-23 SMS sending example

```
08.01.2016 09:52:56.328 [TX] - AT+CMGS="0722222222"<CR><LF>

08.01.2016 09:53:06.365 [RX] - <CR><LF>
>
08.01.2016 09:53:08.479 [TX] - some text to be sent
08.01.2016 09:53:14.825 [TX] - <SUB>
08.01.2016 09:53:14.857 [RX] - <CR><LF>
OK<CR><LF>
```

The next step should be to analyze the above snippet. First of all the AT+CMGS command is sent to the modem containing as a parameter the phone number the message should be sent to, between quotation marks followed by the CR LF sequence. Until this point no new aspect are being introduced. Further, the modem responses with a CR LF sequence as expected but then the '>' is sent. This character is practically a prompt and after this the modem expects to receive the actual text message. In the example below one can notice that after the prompt character the host PC sends the actual text message (in the TX line). The actual text message ends with a special, newly introduced character, which is the substitute (<SUB>) character with the HEX ASCII code of 0x1A. After the SUB character was received the modem tries to send the SMS message to the number given to the command as parameter. The response after this is perfectly "legal" which respects the AT command language grammar without any exception.

In order to implement the above presented sequence the receiving state machine doesn't need to be modified but a special send/receive state machine needs to be implemented in order to support the first part of the command. The last response of the command may be parsed with the already available AT command parser. A state chart on how the algorithm could be implemented may be the one in Fig. 1-21.

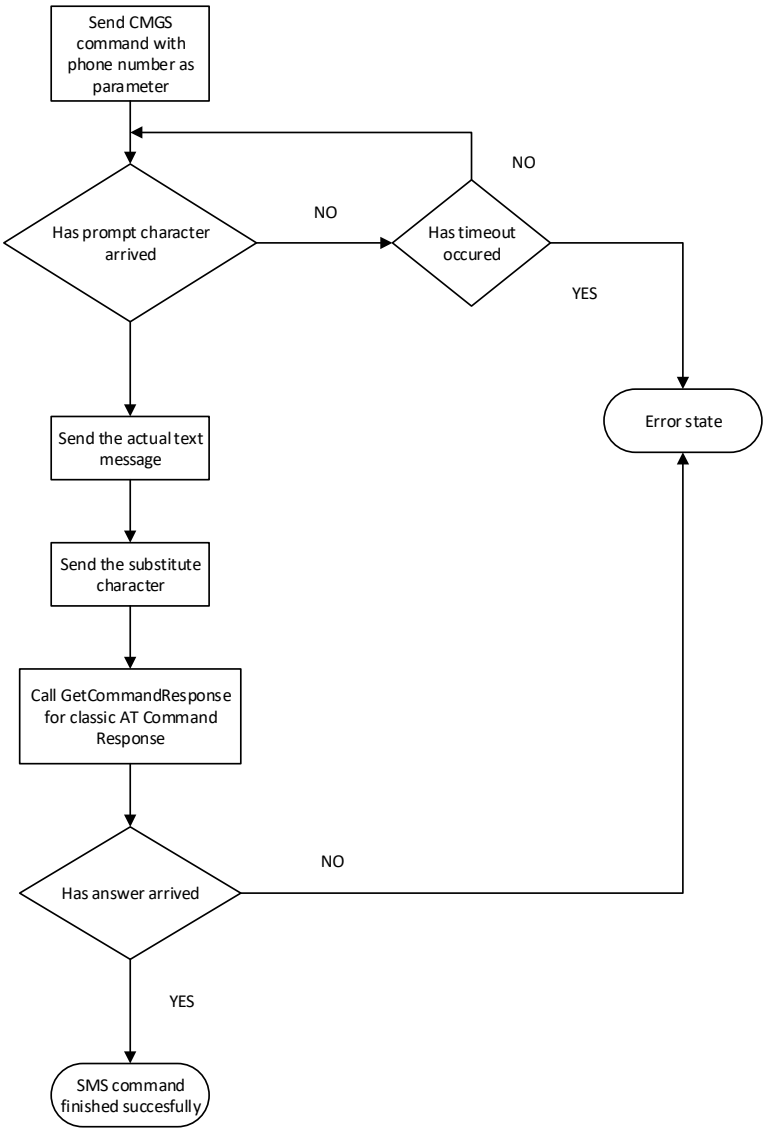

Fig. 1-21 SMS Send state chart

ASSIGNMENT 3: Implement a function capable of sending one SMS message to a phone number. The message and the phone number should be given as parameters to the function. Be careful as not to include the testing of this function in the infinite loop of the program as it will send a lot of SMS messages thus draining the credit from the SIM card.

HOME ASSIGNMENT: Please read the driver manual documentation and the Doxygen documentation regarding the modules for working with the LCD and TOUCHSCREEN.

1.8 Laboratory work – Management interface using LCD and touchscreen

All of the laboratory applications presented above have the debug console on *Docklight scripting* as output. This laboratory work concentrates on integrating the functionality available until now into a minimalistic graphical user interface. This can be accomplished by using the LCD and the TOUCHSCREEN. The starting point of the LCD and TOUCHSCREEN, coordinates (0, 0), is on the upper left corner of the display considering the picture in Fig. 1-1 from paragraph 1.1.2 .

For both the LCD and the TOUCHSCREEN a driver is available in the driver library. Furthermore, all the initializations required are done in the *BoardInit* which should be called the first function in main. Having this functional called the LCD is initialized and cleared. The next function that is usually called in main is *testLCD* which draws the Romanian National Flag and writes test messages. This call will not be needed anymore thus an interface needs to be displayed.

Information about the LCD, the prototypes, data structures and function calls may be found in the driver manual and in the Doxygen documentation. The LCD driver module supports the following functions:

- Clear screen
- Fill screen with a color
- Write a string on the screen with the given color and at the given coordinates using a small or big font
- Put a pixel of a given color at the designated coordinates.

Only minimal functions are supported by the library. Anything else needs to be further implemented.

Regarding the touchscreen, in the basic Keil project, the module is initialized and a callback is instantiated which offers as parameters the values of the coordinates of the pressed zone on the touchscreen. The callback function is called only when a touch is detected. The callback function implementation currently available in the example project is the following:

Code listing 1-24 Touchscreen callback function implementation example

```
void TouchScreenCallBack(TouchResult* touchData)
{
  printf("touched X=%3d Y=%3d\n", touchData->X, touchData->Y);
}
```

Furthermore, in order for this feature to work the touchscreen process function must be called in the main infinite while loop which contains no blocking calls. An adapted example from Code listing 1-13 which describes the suggestion above may be the one in Code listing 1-25. The integration of the touch screen process function is bolded at the end of the infinite loop.

Code listing 1-25 Example of touchscreen process integration in main loop

```
.....

AT_COMMAND_DATA data_structure;
const char at_command_simple="AT\r\n"
const char at_command_csq = "AT+CSQ\r\n"
timer_software_handler_t my_timer_handler;
...

int main(void)
{
  ...
  uint32_t rssi_value_asu;
  uint32_t rssi_value_dbmw;
  ...
  while (1)
  {
     if (TIMER_SOFTWARE_interrupt_pending(my_timer_handler))
     {
             ExecuteCommand(at_command_csq);
             if (CommandResponseValid())
             {
                rssi_value_asu = ExtractData(&data_structure);
                rssi_value_dbmw = ConvertAsuToDbmw(rssi_value_asu);
                printf(...);
             }
             TIMER_SOFTWARE_clear_interrupt(my_timer_handler);
     }
     DRV_TOUCHSCREEN_Process();

  }
  ...
}
```

ASSIGNMENT: Implement a graphical user interface that may offer a set of functions.

Regarding the way the interface should look, the students have the freedom to design the graphical interface. However, some minimal aspects need to be considered. The graphical interface should firstly contain the status of the modem which has the following information: signal RSSI value in dBmW, state of registration, operator name. Another part that needs to be present on the interface is a section where one SMS message will be displayed. Another section of the screen must contain buttons to perform the following functions:

- Previous SMS
- Next SMS
- Delete current SMS
- Send hardcoded SMS

The buttons may be designed by drawing a simple rectangle with some text in the middle. This is the suggested method to design the buttons mainly because it is easier to identify them when using the touchscreen.

NOTE: Be careful on how often the interface is updated. Having a much too fast update of the LCD may cause it to flicker. Usually the interface should be update only if some of the values change.

1.9 Laboratory work – GPRS, TCPIP stack and socket management

This laboratory work aims at studying much more advanced features of the GSM modem. We focus during this laboratory work to learn how to connect to the internet via GPRS and how to work with the integrated TCPIP stack. In order to be able to connect to the Internet with the GSM modem, a data connection activated SIM card is needed along with the necessary settings for an available APN provided by the network operator. It is important to mention that there is no significant difference in connecting to the Internet with a GSM modem or to connect to a VPN network provided by the operator. The only difference is the APN along with its credentials. For this laboratory work we will consider connecting to the Internet.

Before beginning to discuss anything regarding the GPRS registration, the host must ensure that the modem is properly registered to the GSM network which is mandatory, thus the GPRS network is just an additional module of the GSM network. The GSM network registration may be verified with the AT+CREG command as shown in Code listing 1-4 and consulting the AT command documentation in order to identify the state of the return result.

For making things simple we will consider that the modem will work in a Single IP non-transparent connection mode. The Single IP mode is configured using the AT+CIPMUX command with the parameter 1, presented in Code listing 1-26, and the non-transparent mode is made calling the command AT+CIPMODE with the parameter 0, presented in Code listing 1-27, all this according to the Modem AT command manual.

Code listing 1-26 Configuring Single IP mode for socket communication

```
08.01.2016 16:39:33.485 [TX] - AT+CIPMUX=1<CR><LF>

08.01.2016 16:39:35.438 [RX] - <CR><LF>
OK<CR><LF>
```

Code listing 1-27 Configuring non-transparent mode for socket communication

```
08.01.2016 16:39:33.485 [TX] - AT+CIPMODE=0<CR><LF>

08.01.2016 16:39:35.438 [RX] - <CR><LF>
OK<CR><LF>
```

These command should be sent to the modem prior to initiating any other communication regarding sockets and GPRS.

The first step in connecting to the Internet is to get ourselves assured that the GSM modem is registered in GPRS. This can be accomplished by interrogating the modem

using the AT+CGREG command. The result is similar to the AT+CREG command and both the explanations and syntax may be found in the GSM Modem AT command manual. Only after receiving a positive response similar to the one in Code listing 1-28 the procedure may continue.

Code listing 1-28 Interrogation result for a valid registration into the GPRS network

```
08.01.2016 16:39:33.485 [TX] - AT+CGREG?<CR><LF>

08.01.2016 16:39:35.438 [RX] - <CR><LF>
+CGREG: 0,1<CR><LF>
<CR><LF>
OK<CR><LF>
```

If this response is not received then the modem needs to be manually attached to the GPRS network with the help of the AT+CGATT command. A sample of this operating is shown below:

Code listing 1-29 Attaching to the GPRS network manually

```
08.01.2016 16:37:37.413 [TX] - AT+CGATT=1<CR><LF>

08.01.2016 16:37:40.430 [RX] - <CR><LF>
OK<CR><LF>
```

After the attach command was given, the host must verify that the modem has registered to the GPRS network using an interrogation such as Code listing 1-28. If even after several of these procedures the modem did not successfully register to the GPRS network, then a power cycle should be given to the modem and have the whole procedure restarted.

Immediately after the successful registration into the GPRS network the host must configure the APN into the GSM modem using the AT+CSTT command. The syntax of the command must be consulted in the AT command manual. An example of the AT+CSTT command which configures the GSM modem's APN to one provided by the Vodafone operator is shown in.

Code listing 1-30 Configuring the APN

```
08.01.2016 16:51:52.258 [TX] - at+cstt="live.vodafone.com","","<CR><LF>

08.01.2016 16:51:59.415 [RX] - <CR><LF>
OK<CR><LF>
```

After the APN was configured into the GSM modem, the actual connection needs to be brought up. This is practically easy and only requires the calling of AT+CIICR command without any parameters. Such a procedure is presented in Code listing 1-31.

Code listing 1-31 Bringing up the GPRS connection

```
08.01.2016 16:52:06.516 [TX] - at+ciicr<CR><LF>

08.01.2016 16:52:09.629 [RX] - <CR><LF>
OK<CR><LF>
```

Finally, after the connection was successful, the TCPIP stack needs to be activated. This is done by using the AT+CIFSR command, with no arguments. This command also has the role to return the IP address given by the operator.

Code listing 1-32 Activation of TCPIP stack

```
08.01.2016 16:52:15.406 [TX] - at+cifsr<CR><LF>

08.01.2016 16:52:17.218 [RX] - <CR><LF>
10.33.249.131<CR><LF>
```

An example of how the TCPIP stack gets activated is displayed in Code listing 1-32. Taking a look at the code snippet above, an extremely important observation needs to be made. The response of the command not only that it does not contain the '+' character at the beginning of the response line but also does not contain the OK/ERROR sequence at the end of the command.

The first AT command language exception: the lack of the '+' character, was encountered before in laboratory work 1.6 in the case of AT+GSN, AT+GMI and AT+GMR commands.

The second AT command language exception, which was not encountered before, is that the AT command does not conclude with the OK/ERROR statements. The AT commands parse needs to be modified, again, in order to support the newly added exceptions. Furthermore, the AT command parses needs to support 2 exceptions at once, in order to be able to proper validate end extract the response to AT+CIFSR command. Even in this case, it is recommended that the finite state machine should be notified about the existence of these exceptions. The finite state machine should not automatically detect these situations.

After the IP address was successfully obtained, then, the host can be sure by the fact that the GSM modem is connected to the network specified by the APN, in our case, the Internet.

ASSIGNMENT 1: Implement a function, which connects the Modem to the Internet and brings up the GSM stack. Modify the AT command parses in order to support the newly introduced exceptions. Adapt the graphical user interfaces implemented in the previous laboratory work in order to display the obtained IP in the modem status section.

In the moment the GSM modem is connected to the Internet (or another VPN network) TCP or UDP clients and servers may be configured. The GSM modem supports various modes of operation. While keeping things simple we will consider the situation where the modem will be configured to be a TCP client and make a simple transaction to a remote host. The chosen transaction is a simple communication to an email server.

Same as working with network sockets on a PC the socket firstly needs to be opened. We will open a socket to a public Google SMTP server. The command capable in opening a socket is AT+CIPSTART, which according to the documentation, accepts as parameters: the socket type (TCP or UDP), the remote host IP address and the remote host port. An example of opening a remote connection to the Google SMTP server can be found in the following code snippet:

Code listing 1-33 Opening a TCP socket

```
08.01.2016 17:02:53.886 [TX] - AT+CIPSTART="TCP","64.233.189.26","25"<CR><LF>

08.01.2016 17:03:14.921 [RX] - <CR><LF>
OK<CR><LF>
<CR><LF>
CONNECT OK<CR><LF>
```

An analysis on the previous response needs to be made. The response until the line OK<CR><LF> is perfectly correct according to the grammar presented in the previous laboratory works. However, the following lines are not. Another simple finite state machine needs to be implemented in order to validate the first part of the response. The new finite state machine needs only to identify the sequence following the “legal” answer, more exactly it needs to detect a “CONECT OK” statement. The “OK” statement of the AT+CIPSTART result states that the command was executed correctly. However it does not state that the socket was opened successfully. The fact that the modem was able to open the socket is signaled via the last line of the response presented in Code listing 1-33. Be aware that according to the documentation “CONNECT OK” is not the only possible response. Study the command syntax in order to find out all the possible situations! In the next figure a state diagram of how an open socket function should be implemented is presented:

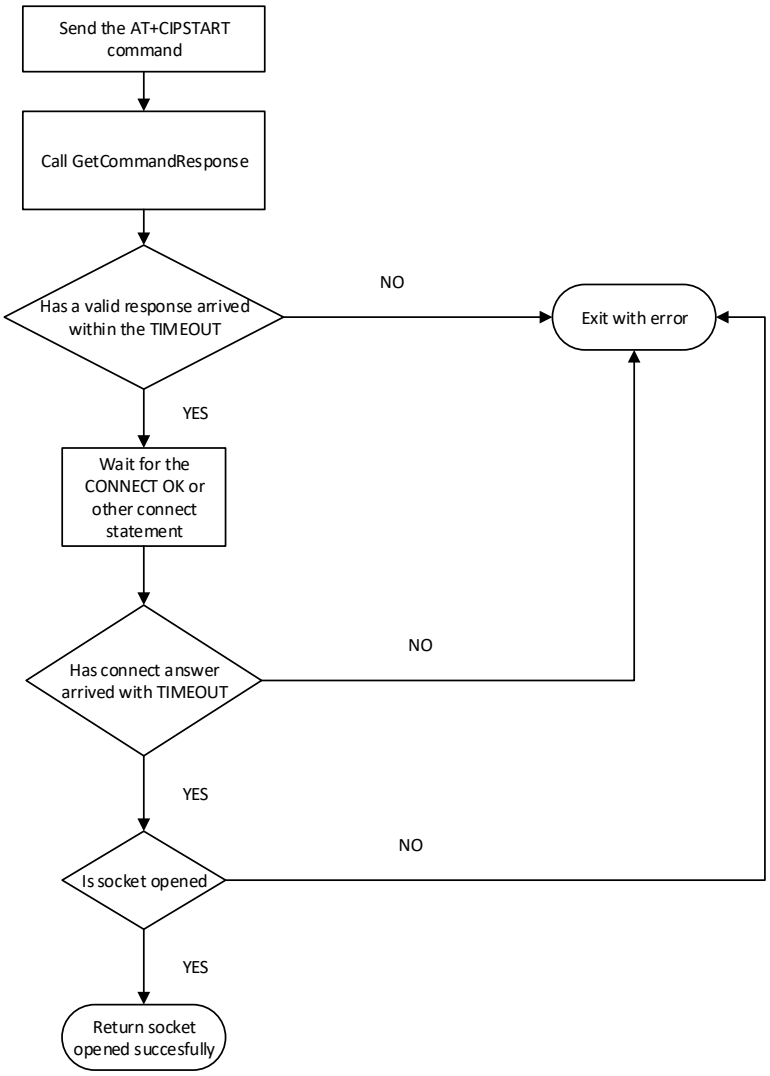

Fig. 1-22 Socket opening state diagram

It is important to mention that in non-transparent, single IP connection, after a socket is opened, any data received from the socket is sent in an unattended manner by the modem. A special and simple receiving state machine may need to be implemented in order to read this data. This state machine is practically a simple read from the UART buffer of the microcontroller when needed. Giving the fact that in the previous example we have opened a socket to an email server after the “CONNECT OK” statement the modem will send the greeting message received from the SMTP server immediately after the socket was opened. In conclusion, the complete communication log in this situation is the following:

Code listing 1-34 Full received data on socket open command

```
08.01.2016 17:02:53.886 [TX] - AT+CIPSTART="TCP","64.233.189.26","25"<CR><LF>

08.01.2016 17:03:14.921 [RX] - <CR><LF>
OK<CR><LF>
<CR><LF>
CONNECT OK<CR><LF>
220 mx.google.com ESMTP i19si28127789wmd.45 - gsmtp<CR><LF>
```

The difference between Code listing 1-33 and Code listing 1-34 is that after the "CONNECT OK" statement the SMTP server greeting received from the email server is sent by the modem unattended. A simple read from the microcontroller's UART interface after a few moments may obtain this message if needed.

The next aspect that will be discussed in this laboratory work is represented by the procedure for sending data over an opened socket. This procedure is practically identical to the procedure of sending an SMS message presented in laboratory work 1.7 . Such a procedure is displayed in the following example where we send an extended hello message (EHLO localhost) to the remote email SMTP server:

Code listing 1-35 Sending text data over a socket

```
08.01.2016 17:03:39.910 [TX] - AT+CIPSEND<CR><LF>

08.01.2016 17:03:43.838 [RX] - <CR><LF>
>
08.01.2016 17:03:45.480 [TX] - EHLO localhost<CR><LF>

08.01.2016 17:03:52.215 [TX] - <SUB>
08.01.2016 17:03:53.108 [RX] - <CR><LF>
SEND OK<CR><LF>
```

In Code listing 1-35 we can find a sequence of how text data can be sent over an opened socket. The first line represents the calling of the command responsible for sending data over the socket (AT+CIPSEND). Similar to the situation when sending an SMS message, after the send command the host needs to wait for the '>' prompt character (and the preceding CR LF characters). After the prompt character is received the actual data to be sent over the socket needs to be inputted. The finishing of the data to be sent again marked by the Substitute (SUB) character with 0x1A as the HEX ASCII value. Immediately after the sub character is received by the modem it begins to transmit the data over the socket. After the data was sent over the socket the modem responds (similar to the response in the case of sending SMS messages) with a CR LF sequence followed by the string "SEND OK" again with a CR LF ending.

Having this situation analyzed we can conclude that another exception has been found. The exception, in respect to the response of the SMS message sent command, is that in a successful situation "OK" is not the string being return but "SEND OK" replaces it.

After the successful sent of the data over the socket we should expect the answer from the remote host which should arrive shortly. After a user defined timeout we should read the microcontroller's serial interface in order to collect the remote host's answer. Such an answer followed after the send ok statement may be found in the next example:

Code listing 1-36 Sending text data over a socket with received answer from remote host

```
08.01.2016 17:03:39.910 [TX] - AT+CIPSEND<CR><LF>

08.01.2016 17:03:43.838 [RX] - <CR><LF>
>
08.01.2016 17:03:45.480 [TX] - EHLO localhost<CR><LF>

08.01.2016 17:03:52.215 [TX] - <SUB>
08.01.2016 17:03:53.108 [RX] - <CR><LF>
SEND OK<CR><LF>
250-mx.google.com at your service, [213.233.84.80]<CR><LF>
250-SIZE 35882577<CR><LF>
250-8BITMIME<CR><LF>
250-STARTTLS<CR><LF>
250-ENHANCEDSTATUSCODES<CR><LF>
250-PIPELINING<CR><LF>
250-CHUNKING<CR><LF>
250 SMTPUTF8<CR><LF>
```

The state chart presented for in the SMS sending algorithm in Fig. 1-21 may be adapted in order to support and describe the algorithm for making a simple transaction over an opened socket. By transaction we mean a successfully transmit of a request to the remote host and the reception of the response. Such a state chart describing the transaction algorithm may be found in the next diagram:

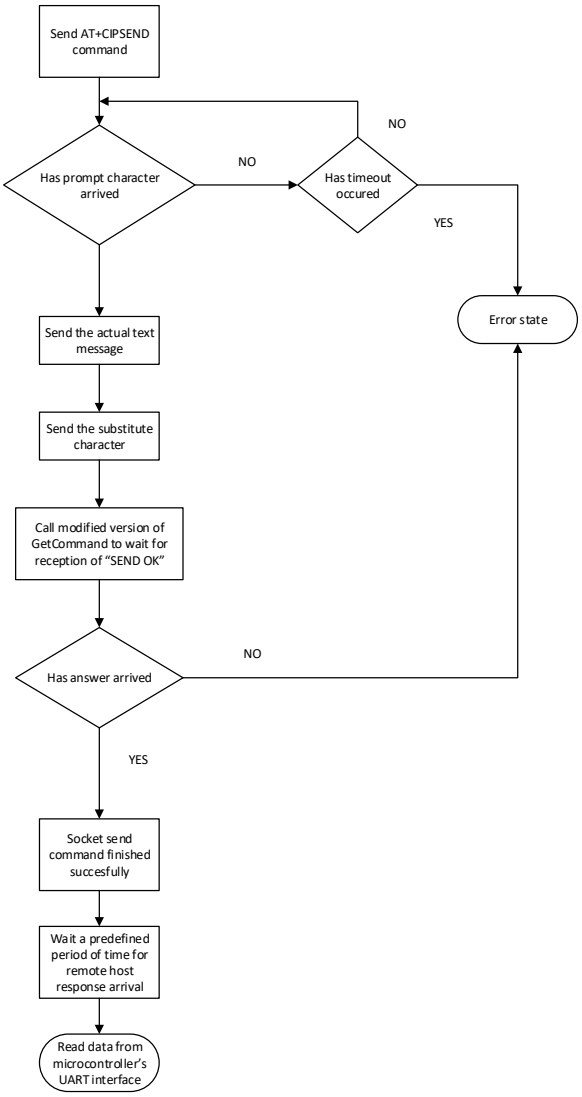

Fig. 1-23 Socket transaction state chart

Only one remark needs to be made, that after the work with the socket is finished, the host microcontroller should close the TCP socket by using the AT+CIPCLOSE command.

ASSIGNMENT: As described above, implement a transaction to a mail server and extract the hello message. Print the collected message either on debug console or on the LCD. In order to accomplish this, implement the necessary algorithms presented above and modify the AT command parser to support the newly introduced exceptions.

1.10 Driver manual

1.10.1 Introduction

1.10.1.1 General specifications

This document contains a description of the drivers that are given to the 4th year students attending to the Digital Telecommunications ("Telecomunicatii Digitale") laboratory in DSPLabs.

The main platform is a LPC1788 MOD-LCD 4.3 board from Olimex which is connected to a proprietary board with various communication modules like GSM, GPRS, GPS, WIFI, Ethernet TCP/IP, Bluetooth, and XBEE.

This document is also accompanied by the Doxygen documentation of the code of the drivers hosted at http://dsplabs.cs.upt.ro/~valys/td/driverdoc/index.html. Almost all the functions and the data types or variables are documented in Doxygen but not all of them are needed by the programmer. The functions that should not be of an interest to the programmer are marked with *Private*. It is highly recommended that the programmer doesn't call any of these private functions.

This document is structured as following:

- Introduction – presents the general specifications of the document and of the drivers which are related to DRV_GENERAL driver.
- Drivers – presents the description of the drivers
- Utilities – presents various structures and modules that help the development of applications within the laboratory and facilitate teaching and understanding.

All of the driver functions are named by the following rule: DRV, following by the name of the module, followed by the name of the function,all separated by underscore. Example: DRV_LCD_Configure – the function that configures the LCD module.

1.10.1.2 DRV_GENERAL

The module called DRV_GENERAL is not an actual driver. It contains general definitions and functions that are needed by the rest of the drivers.

One of the more important aspects is that it defines the STATUS enum which is used by most of the driver functions to return the status of the corresponding function call. The Doxygen documentation contains the actual description of this enum type.

This module also defines some general configuration for the rest of the drivers like the size of the buffers for each UART modules. It also defines some general data types like Boolean and some arithmetic macros: MIN, MAX, and ABS.

This module defines some functions that are used by the UART driver to calculate the baud rate divisors for any given baud rate. The calculations are made using the peripheral clock that clocks the UART modules. This module also contains the definitions for the size of all the buffers used by all of the other drivers.

1.10.2 SRAM Memory Driver

The LPC1788 MOD-LCD 4.3 board contains a 32 MB SRAM memory connected to the LPC1788 microcontroller's memory bus. The start address of the memory is 0xA0000000 defined by the SDRAM_BASE_ADDR macro. The programmer only has to configure the external memory controller of the microcontroller. The rest of the operations (read/write) are transparent for the programmer.

The only and most important function of the driver (that is exported to the programmer) is DRV_SDRAM_Init() which has to be called before using the external SDRAM memory (anything inside the address space 0xA0000000 – 0xA2000000).

1.10.3 LCD Driver

The LPC1788 MOD-LCD 4.3 board is equipped with a RGB LCD with touchscreen. This chapter describes only the driver responsible for the LCD, the touchscreen's driver being presented in the following chapter.

The LCD is landscape orientated with a resolution of 480x272 pixels. The driver exports the following methods (the complete description of these methods and their parameters and return value can be found in the Doxygen documentation of this driver library):

1. DRV_LCD_Init – This function has to be called prior to using any of the LCD related methods. This function initializes the LCD dedicated module of the microcontroller and also configures and powers on the LCD. Only after the call of this function the LCD may be used by the programmer. The function doesn't need any parameters.
2. DRV_LCD_PutPixel – This function is used to draw a pixel on the LCD at the specified location (by x and y coordinates). The color is defined by RGB coordinates. The full specification of this method may be found on the Doxygen documentation of the driver library.
3. DRV_LCD_TestFillColor – This function fills the whole LCD with a given color specified by RGB coordination.
4. DRV_LCD_ClrScr – This function clears the content of the LCD coloring its background in black.
5. DRV_LCD_Puts – This function draw a user string on the LCD at a given position, of the specified color. There are 2 fonts available in the driver to write text on the LCD: a small font and a large font. The desired font can be selected via a parameter of this function.

One important aspect of this library is that it doesn't work with special characters when writing text. It also doesn't do anything related to spacing. The user is responsible with the positioning of the text lines (whether these text lines overlap or not).

1.10.4 Touchscreen driver

Prior to using any method of the touchscreen driver, the *DRV_TOUCHSCREEN_Init()* function has to be called in order to initialize the touchscreen hardware and software components.

The driver of the touchscreen has 2 operating modes: a polling mode and a callback mode.

In the polling working mode, the user has to check whether the touchscreen has been touched and to acquire the coordinates of the touched position. This is done by calling *TouchGet* function which returns a Boolean value specifying whether a touch event has occurred. If such a touch event is present then in a non-null parameter the x and y coordinates are returned.

In the callback working mode, the driver itself calls a specified user function where it announces a touch event, the (x, y) location. In order to use this working mode, immediately after calling *DRV_TOUCHSCREEN_Init()* the user has to call *DRV_TOUCHSCREEN_SetTouchCallback*. This function takes a pointer to a function as an argument. This function will be called when a touch event occurs. The prototype of the function that this method accepts as parameters is the following:

Code listing 1-37 Prototype of touch screen callback function

```
void TouchScreenCallBack(TouchResult* touchData).
```

Also, in order to use this operation mode, in the programs main forever loop, the user has to include a call to *DRV_TOUCHSCREEN_Process()*.

Let's have the following example:

Code listing 1-38 Touchscreen driver usage with callback

```
#include <DRV/drv_general.h>
#include <DRV/drv_touchscreen.h>

void MyTouchScreenCallBack(TouchResult* tData)
{
  printf("Touch event at: X=%3d Y=%3d\n", tData->X, tData->Y);
}

void main(void)
{
  // user code
  DRV_TOUCHSCREEN_Init(); // initialize the touchscreen
  // set the touchscreen callback
  DRV_TOUCHSCREEN_SetTouchCallback(MyTouchScreenCallBack);
  // user code
  while(1)
  {
     // user code
     DRV_TOUCHSCREEN_Process();
     // user code
  }
  // user code
}
```

In the above example we initiate the touchscreen driver, configure it to use callback working mode and instantiate a callback for the touch event. In this situation whether a touch event occurs, the driver calls the user function *MyTouchScreenCallBack* where the user can process this event, in our case, the user prints the touched coordinates.

1.10.5 LED Driver

The laboratory board is equipped with 4 LEDs that are directly connected to the processor of the LPC1788 MOD-LCD 4.3 board on P0_0, P0_1, P0_10 and P0_11 of port 0. The purpose of this driver is to offer the programmer a way to easily interact with the LEDs on the board without knowing details about connections or port configurations. These 4 LEDs are identified by the LED enumeration type (check the doxygen documentation). This enumeration type has 4 possible values (LED_1, LED_2, LED_3, LED_4)

Prior to using the LEDs the programmer has to call the method *DRV_LED_Init()* in order to initialize both the driver and the GPIO system.

There are 3 methods that the programmer may use in order to interact with the leds.

- *DRV_LED_Off* – which takes the specified LED as an argument (defined by the *LED enumeration* type). This method turns off the led.
- *DRV_LED_On* – which takes the specified LED as an argument (defined by the *LED* enumeration type). This method turns on the LED.
- *DRV_LED_Toggle* – which takes the specified LED as an argument (defined by the *LED* enumeration type). This method toggles the LED: if the LED is off then it is turned back on and if the LED is on then a call of this function will turn off the specified LED).

1.10.6 UART Driver

The processor of the LPC1788 MOD-LCD 4.3 board has 5 UART modules designated as UART_0, UART_1, UART_2, UART_3, UART_4. This driver offers a unique way in accessing all of the 5 UART modules of the microcontroller by providing a parameters to each of the driver functions that identifies the UART module.

An important aspect of the driver is that it has an internal error reporting system similar to the *errno* variable in POSIX. After a call of every function provided by the driver, the programmer can check the type of the error (if present/needed) by using the *DRV_UART_GetErrno* function. The error types are found in the UART_ERROR_TYPE enumeration type.

The UART driver can be initialized using the *DRV_UART_Configure* (the full description of this method can be found in the Doxygen documentation). The programmer can specify the baud rate (9600, 57600, 115200, etc.…) the parity, the number of bits per character, the number of stop bits, the operating mode (which will be discussed later) and a read timeout value in milliseconds.

The UART driver has 2 operating modes: a blocking mode and a buffered mode. The blocking mode is the simplest way to use the UART modules via this driver. In this mode the processor waits indefinitely for every operation to finish. This is why this mode is called blocking (it blocks the execution of the processor until the current operation is complete). In order to configure the driver to work in this mode (blocking mode), the 6th parameters of DRV_UART_Configure function has to be false. Let's take the following example:

Code listing 1-39 UART driver mode usage in a blocking manner

```
#include <DRV/drv_general.h>
#include <DRV/drv_uart.h>
void main(void)
{
… // user code
  /*
  We configure and initialize the UART module UART_2 to use 8 bits/character, with
no parity at 9600 bps and to work in blocking mode
  */
  DRV_UART_Configure(UART_2, UART_CHARACTER_LENGTH_8, 9600, UART_PARITY_NO_PARITY,
1, false, 0);
  // here we may check the errno value (optional)
  If (DRV_UART_GetErrno(UART_2) != UART_ERROR_NO_ERROR)
  {
     // do something in case of error
  }
  uint8_t ch;
  while(1)
  {
     … // user code 1
     // we read a character from UART_2 and store it in ch
     DRV_UART_GetCharBlocking(UART_2, &ch);

     // we send the previously receive character
     DRV_UART_SendCharBlocking(UART_2, ch);
     … // user code 2

  }
}
```

In the previous example we configure the module UART_2 to work in blocking mode and implement a serial echo (the receive character is sent back to the transmitter). It is important to mention that the call to function *DRV_UART_GetCharBlocking* will freeze the execution of the forever loop until the reception of a character. Also the call *DRV_UART_SendCharBlocking* will freeze the execution of the forever loop until the character is transmitted. This will cause the user code designated as 1 and the user code designated as 2 to be actually executed only if a character is received. The driver also offers a way to send a whole buffer over a UART module in blocking mode this also freezing the user code execution until the whole buffer is transmitted.

The other operating mode of this driver is the buffered mode. In this mode the reception and the transmission is done using the interrupt system and a collection of buffers. In this manner the execution of the processor is not blocked until the reception/transmission operations are finished. The user only "schedules" a transmission or reception operation and the actual process is done in the background using the interrupt system. This can be seen similar to threading in a way. This operating mode is very complex offering a lot of flexibility to the user. A programmer can not only read/write to the UART port, but it can also check the amount of available data stored in the internal buffers and flush these internal buffers.

Another important aspect is that this driver also offers a callback system that the programmer can use if he or she wants to be announced when new data is ready or when data has been transmitted. The callback system is similar to the one in the touchscreen driver. In order for the UART driver callback system to function the user programmer has to include a call to *DRV_UART_Process()* within the programs forever loop.

In the next example we use the LEDs to demonstrate the usage of the buffered operating mode of the UART driver with callbacks. The program has the following behavior: inside the forever loop LED_1 is toggled; at the reception of character "a" LED_2 is toggled and at the reception of character "b" LED_3 is toggled.

Code listing 1-40 UART driver mode usage in a non-blocking manner using callbacks

```
#include <DRV/drv_general.h>
#include <DRV/drv_uart.h>
void main(void)
{
  // led configurations and other initialization code
  /*
  We configure and initialize the UART module UART_2 to use 8 bits/character, with
no parity at 9600 bps and to work in buffered mode.
  */
  DRV_UART_Configure(UART_2, UART_CHARACTER_LENGTH_8, 9600, UART_PARITY_NO_PARITY,
1, true, 0);
  DRV_UART_SetRxCallback(UART_2, MyCustomCallback);
  while(1) // program's main forever loop
  {
     // user code 1
     DRV_LED_Toggle(LED_1);
     DRV_UART_Process();
     // user code 2
  }
}

void MyCustomCallback(UART uart, uint32_t size)
{
  uint8_t ch;
  // process all the pending characters
  while (DRV_UART_BytesAvailable(UART_2) > 0)
  {
  // read a character and check the status of the call (status check is optional)
  if (DRV_UART_ReadByte(UART_2, &ch) == OK)
  {
  switch (ch)
  {
     case 'a':
     {
            DRV_LED_Toggle(LED_2);
            break;
     }
     case 'b':
     {
            DRV_LED_Toggle(LED_3);
     }
  }
  }
}
}
```

In the main function of this example, firstly, we initialize everything we need, especially UART_2 module. We configure the UART driver to use UART_2 module in buffered mode by calling DRV_UART_Configure. With the next function call we tell the driver that we want to use a custom callback when a character is received. The main loop of the program calls the mandatory function of the driver *DRV_UART_Process()*, the method to toggle LED_1 and other user code.

In this situation, the microcontroller will be busy running the forever loop without any blocking situations thus processing will not be significantly delayed. The behavior of the loop can be observed using a scope on LED_1.

Every time a character is received, the function *MyCustomCallback* will be called by the driver, interrupting the forever loop in order to let the user process this event. In our example we read all the available characters in the buffer and for each character we toggle LED_2 if the character is “a”, toggle LED_3 if the character is “b” and ignore the rest.

This operating mode has the advantage that the programmer doesn’t block the processor for unnecessary idle waits, for example when waiting for a character to arrive.

1.10.7 Driver utilities

1.10.7.1 Software timer (TIMER_SOFTWARE)

This module is a software implemented timer. The main advantages are that the user can define as many timers as needed, the only limitation being the memory. The software timer module is implemented using a hardware timer module of the microcontroller. The granularity of the software timer is 1 ms. Prior to using this feature, the function *TIMER_SOFTWARE_init_system()* has to be called. The user can specify the number of maximum timer software to be allocated at compile time by modifying the MAX_NR_TIMER macro.

Each software timer has 4 operating modes:

- MODE_0 – The software timer counts to the value given by period. When the counter is equal to period, the timer stops and generates an event.
- MODE_1 – The software timer counts to the value given by period. When the counter is equal to period the timer generates an event, resets the counter and restarts.
- MODE_2 - The software timer counts to the value given by period. When the counter is equal to period the timer generates an event and continues running
- MODE_3 – This operating mode is a free run mode. The counter just starts from 0 and keeps counting without generating any events.

The programmer may use this timer with event generation via a callback system or using polling methods via dedicated methods for checking pending events. Before using a timer, the programmers needs to acquire such a timer by calling *TIMER_SOFTWARE_request_timer* which returns a descriptor for the newly allocated timer. A timer may be released after the programmer finishes using it, by calling *TIMER_SOFTWARE_release_timer*. After a timer has been acquired by the programmer it has to be configured by specifying its operating mode and its counting period.

The library also offers a simple wait function which blocks the code execution for an amount of time given as argument.

The next example (led blinking) describes a simple usage of a software timer with event generation via callbacks:

Code listing 1-41 Timer software usage example with callbacks

```
#include <Utils/timer_software_init.h>
#include <Utils/timer_software.h>
#include <DRV/drv_led.h>

void main(void)
{
  //user code
  DRV_LED_Init();
  TIMER_SOFTWARE_init_system(); // initialize the software timer library
  timer_software_handler_t handler; // declare a software timer
handler(descriptor)
  handler = TIMER_SOFTWARE_request_timer(); // request a timer
  if (handler < 0) // check if the request was successful
  {
     // the system could not offer a software timer
  }
/* configure the requested timer to run in MODE_1 (reset and restart at match)
with a period of 100 ms
*/
  TIMER_SOFTWARE_configure_timer(handler, MODE_1, 100, true);
  // set a callback for the requested timer
  TIMER_SOFTWARE_set_callback(handler, MyTimerCallback);
  TIMER_SOFTWARE_start_timer(handler);
  while(1)
  {
  // user code
  }
}

void MyTimerCallback(timer_software_handler_t handler)
{
  DRV_LED_Toggle(LED_1);
}
```

In the previous example we can use the software timer to blink a LED with a period of 100 ms. .The first thing to do in the main function of the program is to initialize both the LEDs and the software timer system. After these initializations we declare a handler for the software timer we want to use and then, we request the timer. If the system could not offer a software timer (mainly because there are not software timers available) the value of the handler is negative. On the successful request of a system timer we configure the timer to work in mode 1 with a period of 100 ms. The next step is to instantiate a callback and finally we can start the timer. Our callback function (*MyTimerCallback*) will be executed in interrupt execution context, once every 100 ms where we can easily blink the LED. The user code is once again not affected.

There is also another way the programmer may use the software timer: without using a callback system thus using a polling method. Such a method is described in the following example which is similar to the previous one. The program also blinks the following LED at a period of 100 ms but without using callbacks and using event polling instead.

Code listing 1-42 Timer software usage example in a polling manner

```
#include <Utils/timer_software_init.h>
#include <Utils/timer_software.h>
#include <DRV/drv_led.h>

void main(void)
{
  //user code
  DRV_LED_Init();
  TIMER_SOFTWARE_init_system(); // initialize the software timer library
  timer_software_handler_t handler; // declare a software timer handler
  handler = TIMER_SOFTWARE_request_timer(); // request a timer
  if (handler < 0) // check if the request was successful
  {
     // the system could not offer a software timer
  }
/* configure the requested timer to run in MODE_1 (reset and restart at match)
with a period of 100 ms
*/
  TIMER_SOFTWARE_configure_timer(handler, MODE_1, 100, true);
  // set a callback for the requested timer
  TIMER_SOFTWARE_start_timer(handler);
  while(1)
  {
     // user code
     if (TIMER_SOFTWARE_interrupt_pending(handler) != 0)
     {
        DRV_LED_Toggle(LED_1);
        TIMER_SOFTWARE_clear_interrupt(handler);
     }
     // user code
  }
}
```

The difference between this example and the previous one is that in the latter we do not use a callback. Inside the forever loop of the program we check if and interrupt (event) has occurred. If so then we toggle the LED and clear the interrupt flag. In this situation this code is not executed in interrupt context.

1.10.7.2 Retarget debug system

The retarget debug system provides a simple way to send debug message via the UART_0 module of the microcontroller using the standard input/output (*stdio.h*). In this way the programmer may use functions such as *printf*, *sendchar*, *scanf, getchar,* etc. in order to debug the applications. It is not recommender to relay on these standard function in a normal operation of microcontroller applications mainly because these functions have extremely large, time consuming and unpredictable code. In order to use the retarget debug system the programmer has to call the *initRetargetDebugSystem()* method and connect the UART_0 port of the microcontroller to a computer terminal. A usage example can be found below:

Code listing 1-43 Initialization of the retarget debug system for printf

```
#include <stdio.h>
#include <retarget.h>

void main(void)
{
  initRetargetDebugSystem();
  printf ("Hello world");
  while(1);
}
```

1.11 Frequently used Modem AT commands

1.11.1 General specifications

The following chapters will summarize the most frequently used AT commands. For each command a description will be provided as well as all the available forms of the command: test command, read command, write command, execution command depending on which forms are supported.

For each form of the command a description will be provided along with the request and response syntax. Also all the necessary parameters for a command will be described but parameters that do not present interest for the laboratory works will be omitted.

Giving the fact that almost all commands have a test command form which usually is not used, the explanations for this form will be omitted. Check the AT Command documentation [11] for details about this form, if necessary

Only the actual command requests and responses will be detailed in this section, the actual AT protocol encapsulation will omitted.

1.11.2 Simple AT Command

Description: Simple AT command used only for protocol synchronization. Has no actual effect on the GSM modem

Available command type forms: Execution Command

Execution command

Syntax:

Request: AT
Response: OK

Description: No description available.

1.11.3 AT+CREG

Description: Command is used to obtain the registration status by the GSM Modem into the GSM network. The command has 2 response forms: a short response giving only its form number and the actual registration status and a long form giving its form number, the actual registration, the cell ID and the location ID.

Available command type forms: Read Command, Write Command

Read Command: short version

Syntax:

Request: AT+CREG?
Response: +CREG: <n>, <stat>

Description: The read command returns 2 important parameters: the first parameter is the form of the command. This parameter has the value equal to 1, if this short form is used or 2, if the long form is used. The second parameter <stat> contains a code representing the actual registration in the network of the GSM Modem. The <stat> parameter has the following values:

<stat> value	description
0	Modem is not registered in the network and is not searching for a network
1	Modem is registered to home network
2	Modem is not registered but it is currently searching for a network
3	Modem registration into the network was denied
4	Unknown modem registration state
5	Modem is registered to roaming network

Example:

Request: AT+CREG?
Response: +CREG: 1,2

Read Command: long version

Syntax:

Request: AT+CREG?
Response: +CREG: <n>, <stat>, <location_id>, <cell_id>

Description: The read command returns 4 important parameters. The first and second parameter were presented in the short form description. The third and fourth parameter are presented as 4 digit hexadecimal numbers, which represent the code of the location and the code of the cell.

Example:

Request: AT+CREG?
Response: +CREG: 1,2,4A5B,083A

Write Command:

Syntax:

Request: AT+CREG=<n>
Response: OK

Description: The write command accepts only one parameter having the same values and meaning as parameter <n> presented above. Writing this parameter selects which of the 2 read command version should be returned by the modem. Setting the <n> parameter to 1 will select the short version of the read command. Setting the <n> parameter to 2 will select the long version of the read command.

Example:

Request: AT+CREG=2
Response: OK

1.11.4 AT+CSQ

Description: This command is used to return the current value of the signal strength in ASU.

Available command type forms: Execution Command

Execution Command:

Syntax:

Request: AT+CSQ
Response: +CSQ: <rssi>,<ber>

Description: The execution command form of this command is the only one available and returns two important parameters: the RSSI (received signal strength indicator) value and the BER (bit error rate) value. The RSSI can be a value of 0 to 31, if a valid reading was made or a value of 99, if an unknown value was acquired.

The measuring unit for this value is ASU. In order to transform from ASU values to dBmW values the following formula may be used:

$$dBmW = 2 \cdot ASU - 113$$

The second parameter is the bit error rate value measured along with the RSSI value.

Example:

Request: AT+CSQ
Response: +CSQ: 27,0

1.11.5 AT+COPS

Description: This command is used to read information regarding the GSM network operator. This command in slightly complex, but only the most used syntaxes (Read Command) and parameters will be presented here. For more information check the SIM900 AT Command Manual

Available command type forms: Read Command, Write Command

Read Command:

Syntax:

Request: AT+COPS?
Response: +COPS: <mode>,<format>,<op_long>,<op_short>

Description: The read command syntax is used to interrogate the GSM modem about information regarding the network operator. This command returns a valid response only if the GSM modem has registered to the network. It should be used only if a positive registration response is given by AT+CREG command. The return value contains 3 parameters: <mode>, <format> and <op_short> are numerical parameters and <op_long> is a string parameter. The string parameter contains the name of the network operator. The rest of the parameters will not be discussed here.

Example:

Request: AT+COPS?
Response: +COPS: 0,0,”Vodafone RO”,2

1.11.6 AT+COPN

Description: This command is used to read the list of operators known by the GSM modem. This command retrieves a list of the supported operators. Even if an operator is not supported by the GSM modem it will still register with it if possible, but it will not display its name.

Available command type forms: Execution Command

Execution Command:

Syntax:

Request: AT+COPN
Response:
+COPN: <numeric1>,<alpha1> … +COPN: <numeric*n*>,<alpha*n*>

Description: The execution command returns the actual list of supported operators from the GSM modem's internal memory. There are 2 parameters for each entry of the list. The <numeric*i*> parameter contains the network operator identification string which is practically a hexadecimal number and the <alpha*i*> parameter contains the network operator's name.

Example:

Request: AT+COPN
Response:
+COPN: "001010","Test PA128-PA4"
+COPN: "00101","Test PA128-PA4"
+COPN: "20201","GR COSMOTE"
+COPN: "20205","Vodafone GR"
+COPN: "310160","T-Mobile"
+COPN: "31016","T-Mobile"
…

1.11.7 AT+GSN

Description: This command is used to retrieve the IMEI (International Mobile Equipment Identifier) number of the GSM Modem.

Available command type forms: Execution Command

Execution Command:

Syntax:

Request: AT+GSN
Response: <sn>

Example:

Request: AT+GSN
Response: 356938035643809

1.11.8 AT+GMI

Description: This command is used to retrieve the Manufacturer Identity from the GSM Modem.

Available command type forms: Execution Command

Execution Command:

Syntax:

Request: AT+GMI
Response: <manufacturer_identity>

Example:

Request: AT+GMI
Response: SIMCOM_Ltd

1.11.9 AT+GMR

Description: This command is used to retrieve the software revision number of the GSM Modem.

Available command type forms: Execution Command

Execution Command:

Syntax:

Request: AT+GMR
Response: Revision: <revision>

Example:

Request: AT+GMI
Response: Revision: 123456V1

1.11.10 AT+CMGF

Description: This command is used to manage the way the modem should tread SMS messages. There are 2 ways SMS messages can be interpreted: in text mode as it is usually done and in PDU mode where SMS message do not contain text data but raw bytes.

Available command type forms: Read Command, Write Command

Read Command:

Syntax:

Request: AT+CMGF?
Response: +CMGF: <mode>

Description: The read command retrieves the current setting available in the GSM modem on how it should interpret SMS message. The mode may be a value of 0 is PDU mode is to be used or 1 if interpretation is made for text mode

Example:

Request: AT+CMGF?
Response: Revision: +CMGF: 1

Write Command:

Syntax:

Request: AT+CMGF=<mode>
Response: OK/ERROR

Description: The write command is used to set the mode the modem should use to interpret SMS messages. The <mode> parameter has the same meaning as in the read command: value of 0 for PDU mode and value of 1 for text mode

Example:

Request: AT+CMGF=1
Response: Revision: OK

1.11.11 AT+CMGL

Description: This command is used to list the SMS messages that are stored in the SIM card memory.

Available command type forms: Execution Command, Write Command

Write Command:

Syntax:

Request: AT+CMGL=<stat>,[<mode>]
Response:
+CMGL:<index>,<stat>,<oa/da>[,<alpha>][,<scts>][,<tooa/toda>,<length>]<CR><LF><data><CR><LF>

Description: The write command form for this command, in contrast with the other write commands, does not write any parameters for the modem. It is practically a read command with filtering possibilities. This command returns a list of the SMS messages stored in the SIM card connected to the GSM modem based on the filter that is given as parameter. There are 2 parameters: <stat> and <mode>. The <stat> parameter is a string used for filtering the SMS messages. The possible values are:

String value	String meaning
"REC UNREAD"	Received unread messages
"REC READ"	Received read messages
"STO UNSENT"	Stored unsent messages
"STO SENT"	Stored sent messages
"ALL"	All messages

The <mode> parameter is a numerical values which specifies whether the call should affect (value 1) or not (value 0) the read/unread status of the messages that are displayed. The response of this command is a list of the SMS messages in the format presented in the syntax. The fields have the following meaning:

Field name	Field description
<index>	The slot number where the SMS is stored in the SIM card memory
<stat>	Read/unread status of the message
<oa/da>	Destination/source address (phone number)
<alpha>	Optional field. Alphanumeric representation of the destination/source address from phone addressbook
<scts>	Optional field. Timestamp of SMS message in string format
<data>	The string of the SMS message

Example:

Request: AT+CMGL=”ALL”

Response:

+CMGL: 5,"REC READ","Vodafone","","14/09/30,15:49:51+12"<CR><LF> Bine ai venit!O data cu prima reincarcare se activeaza automat o optiune cu trafic inclus de internet. Vei primi vesti!

+CMGL:6,"REC READ", "Notificare", "" , "14/12/15,15:32:42+08" <CR><LF> Creditul existent este insuficient pentru a trimite mesajul.

Execution Command:

Syntax:

Request: AT+CMGL

Response:

+CMGL:<index>,<stat>,<oa/da>[,<alpha>][,<scts>][,<tooa/toda>,<length>]<CR><LF><data><CR><LF>

Description: This execution command is a particular case of the Write command with the parameter <stat> set to “REC UNREAD”. No further information will be presented here thus this command being a particular case of Write command.

1.11.12 AT+CMGS

Description: This command is used to send a SMS message. This command has to stages. In the first stage a write command syntax is required where the destination phone number is specified. Then, after the command is processed a prompt will be received from the GSM modem asking for the content of the SMS message terminated by substitute character (hexadecimal ASCII code 0x1A).

Available command type forms: Write Command

Write Command:

Syntax:

Request: AT+CMGS=<da>
Response: >
Request: <text_message> 0x1A
Response: OK

Description: The only command syntax form accepted by this command is the write command. The syntax requests a parameter <da> which contains the destination phone number between quotation marks. The first response of the command is a prompt represented by a ">" symbol. After this prompt is received the actual SMS message has to be given. The SMS message text must end with the substitute character which is the character with ASCII code 0x1A.

Example:

Request: AT+CMGS="0722222222"
Response: >
Request: "sms message text" 0x1A
Response: OK

1.11.13 AT+CMGD

Description: This command is used to delete a SMS message from a memory slot of the SIM card.

Available command type forms: Read Command

Read Command:

Syntax:

Request: AT+CMGD=<index>
Response: OK

Description: This command is used to delete the SMS message from a memory slot of the SIM card. The index of the slot is given as parameter <index> to the command.

Example:

Request: AT+CMGD=5
Response: OK

1.11.14 AT+CIPMUX

Description: This command is used to set a feature of the GSM modem whether it should manage single IP connections or multiple IP connection at the same time. The idea is to specify if single socket or multi socket operations could be made at the same time. This setting also has effect over the mode of using the sockets

Available command type forms: Read Command, Write Command

Read Command:

Syntax:

Request: AT+CIPMUX?
Response: +CIPMUX: <n>

Description: This command syntax is used the retrieve the actual value of the parameter. The parameter is 0 for single IP connection and 1 for multiple IP connection

Example:

Request: AT+CIPMUX?
Response: +CIPMUX: 1

Write Command:

Syntax:

Request: AT+CIPMUX=<n>
Response: OK

Description: This command syntax is used the set the actual value of the parameter. The parameter is 0 for single IP connection and 1 for multiple IP connection.

Example:

Request: AT+CIPMUX=1
Response: OK

1.11.15 AT+CIPMODE

Description: This command is used to set the parameter of the GSM mode which defines the mode the sockets will be used: in transparent mode or in non-transparent mode. In transparent mode, when a socket is connected the modem relays all messages to or from the socket directly to the UART interface. In order to exit this mode special commands have to be used. In non-transparent mode the receiving and transmitting operations for the network sockets are made through specialized AT commands

Available command type forms: Read Command, Write Command

Read Command:

Syntax:

Request: AT+CIPMODE?
Response: +CIPMODE: <mode>

Description: This command syntax is used the retrieve the actual value of the parameter. The parameter is 0 for non-transparent mode and 1 for transparent mode.

Example:

Request: AT+CIPMODE?
Response: +CIPMODE: 0

Write Command:

Syntax:

Request: AT+CIPMODE =<mode>
Response: OK

Description: This command syntax is used the set the actual value of the parameter. The parameter is 0 for non-transparent mode and 1 for transparent mode.

Example:

Request: AT+CIPMODE=0
Response: OK

1.11.16 AT+CGREG

Description: Command is used to obtain the registration status by the GSM Modem into the GPRS network. The command has 2 response forms: a short response giving only its form number and the actual registration status and along form giving its form number, the actual registration, the cell id and the location id. This command is almost identical to AT+CREG but it referrers to the registration to the GPRS network.

Available command type forms: Read Command, Write Command

Read Command: short version

Syntax:

Request: AT+CGREG?
Response: +CGREG: <n>, <stat>

Description: The read command returns 2 important parameters: the first parameter is the form of the command. This parameter is 1 if this short form is used or 2 if the long form is used. The second parameter <stat> contains a code representing the actual registration in the network of the GPRS Modem. The <stat> parameter has the following values:

<stat> value	description
0	Modem is not registered in the network and is not searching for a network
1	Modem is registered to home network
2	Modem is not registered but it is currently searching for a network
3	Modem registration into the network was denied
4	Unknown modem registration state
5	Modem is registered to roaming network

Example:

Request: AT+CGREG?
Response: +CGREG: 1,2

Read Command: long version

Syntax:

Request: AT+CGREG?
Response: +CGREG: <n>, <stat>, <location_id>, <cell_id>

Description: The read command returns 4 important parameters. The first and second parameter were presented in the short form description. The third and fourth parameter are presented as a 4 digit hexadecimal number that represent the code of the location and the code of the cell.

Example:

Request: AT+CGREG?
Response: +CGREG: 1,2,4A5B,083A

Write Command:

Syntax:

Request: AT+CGREG=<n>
Response: OK

Description: The write command accepts only one parameter having the same values and meaning as parameter <n> presented above. Writing this parameter selects which of the 2 read command version should be returned by the modem. Selecting the <n> parameter to 1 will select the short version of the read command. Selecting the <n> parameter to 2 will select the long version of the read command.

Example:

Request: AT+CGREG=2
Response: OK

1.11.17 AT+CGATT

Description: Command is used to handle the attachment to the GPRS network.

Available command type forms: Read Command, Write Command

Read Command

Syntax:

Request: AT+CGATT?
Response: +CGATT: <state>

Description: The read command returns the actual state of the GPRS attachment. The <state> value may be 0 if the GSM modem is detached from GPRS or 1 if the GSM modem is attached to GPRS

Example:

Request: AT+CGATT?
Response: +CGATT: 1

Write Command

Syntax:

Request: AT+CGATT=<state>
Response: OK

Description: The write command is used to instruct the modem whether it should attach to the GPRS network. If the <state> parameter value is written to 0 then GPRS attachment is disable. If the parameter is written to 1 then the attachment to GPRS is enabled.

Example:

Request: AT+CGATT=1
Response: OK

1.11.18 AT+CSTT

Description: This command is used to manage the active APN and credentials to connect through the GPRS network.

Available command type forms: Read Command, Write Command

Write Command

Syntax:

Request: AT+CSTT=<apn>,<username>,<password>
Response: OK

Description: The write command is used to configure the APN and credentials to be used by the modem in order to connect to the data network. Three parameters are required**:** <apn> which holds the name of the APN between quotation marks which is given by the operator, <username> and <password> holds the credentials also between quotation marks and also given by the operator

Example:

Request: AT+CSTT=”internet.vodafone.ro”,”vodafone”,”vodafone”
Response: OK

Read Command

Syntax:

Request: AT+CSTT?
Response: +CSTT: <apn>,<username>,<password>

Description: The read command is used to retrieve the currently configured settings for the APN.

Example:

Request: AT+CSTT?
Response: +CSTT: ”internet.vodafone.ro”,”vodafone”,”vodafone”

1.11.19 AT+CIICR

Description: This command is used to bring up the wireless connection over GPRS. This command also initialized the TCP stack is the wireless connection was successfully activated over GPRS

Available command type forms: Execution Command

Execution Command

Syntax:

Request: AT+CIICR
Response: OK

1.11.20 AT+CIFSR

Description: This command is used to retrieve the IP address assigned to the GSM modem by the operator after the TCP IP stack has been configured. The call of this command is not optional, it is also used to configure the TCP IP stack, not only to retrieve the IP address

Available command type forms: Execution Command

Execution Command

Syntax:

Request: AT+CIFSR
Response: <ip_address>

Example:

Request: AT+CIFSR
Response: 192.168.0.1

1.11.21 AT+CIPSTART

Description: This command is used to open a socket connection.

Available command type forms: Write Command

Write Command

Syntax:

Request: AT+CIPSTART=<mode>,<ip_address>,<port>
Response: OK
Response: <state>

Description: The write command is used to open a socket connection to a TCP/UDP server. There are three parameters needed: <mode> which may be a string of value "TCP" or "UDP", <ip_address> is a string representing the ip address and <port> which is also a string representing the port number. The command has to return line. The first line returns the correctness of the syntax which may be OK or ERROR. The second return line is represented by the <state> parameter which may have the following values:

IP INITIAL
IP START
IP CONFIG
IP GPRSACT
IP STATUS
CONNECT OK
TCP CONNECTING/UDP CONNECTING/SERVICE LISTENING
CONNECT OK
TCP CLOSING / UDP CLOSING
TCP CLOSED/ UDP CLOSED
PDP DEACT

The only value that is important is CONNECT OK which signals that the socket was successfully connected to the host. Other values clearly return other situations and the connection was not successfully made.

Example:

Request: AT+CIPSTART="TCP","216.58.214.67","80"
Response: OK
Response: CONNECT OK

1.11.22 AT+CIPSEND

Description: This command is used to send data over an opened socket. Only the situation where one connection can be handled at the same time. The syntax and usage of this command is almost identical to AT+CMGS

Available command type forms: Write Command

Write Command

Syntax:

Request: AT+CMGS[=<length>]
Response: >
Request: <data> 0x1A
Response: OK

Description: The command accepts only one parameter which specifies the length of the data buffer to be sent over the socket. After the command is sent to the modem, similar to command AT+CMGS, the modem returns prompt represented by character ">". In this moment the modem waits for a number of bytes specified in the <length> parameter to be sent followed by the substitute character 0x1A. The length parameter may be omitted.

Example:

Request: AT+CIPSEND
Response: >
Request: "sms message text" 0x1A
Response: OK
Response: SEND OK

1.11.23 AT+CIPCLOSE

Description: This command is used to close the opened socket.

Available command type forms: Execution Command

Execution Command

Syntax:

Request: AT+CIPCLOSE
Response: OK

Example:

Request: AT+CIPCLOSE
Response: OK

2 Digital Signal Acquisition and Conditioning

2.1 Introduction

2.1.1 General specifications

The Digital Signal Acquisition and Conditioning laboratory introduces the student into basic methods to acquire and process digital signals by implementing simple applications. The laboratory works are structured in such a way that at the end of the semester a complex application will be built. The finality of the laboratory will be a two channel oscilloscope implemented using the provided hardware and software materials. An intermediate milestone will also be considered, which will be represented by a digital voltmeter.

In order to attend to this laboratory the students must have the following mandatory prerequisites:

- Strong C programming skills [1]
- Basic knowledge in electronic fundaments
- Capacity to interpret an electronic schematic

2.1.2 Provided materials

This laboratory will be oriented on signal acquisition and conditioning using embedded devices. The main component, which the students will use, is the Atmel ATMEGA16 microcontroller [14]. The microcontroller will be encapsulated on a header board, which the students can easily use to connect to other external components. Another important component provided to the students is a project board-based student learning kit formally designed by Freescale now currently maintained by NXP. The main advantage of this board is that it consists of a breadboard, which can be used to build prototype circuits and a peripheral board, which contains an important number of peripherals.

Beside the hardware components presented above, the students will also have access to a dedicated software, which will serve as an oscilloscope display and control along with the necessary developing tools and serial communication port terminals.

Moreover, in order to be able to test the projects, the students will also need access to oscilloscopes and signal generators.

In this chapter, the following subsections will be reserved for the description of the modules that will be provided to the students.

2.1.2.1 ATMEGA16 Microcontroller, header board and debugger

ATMEGA16 is an 8bit MEGA-AVR microcontroller designed around a RISC architecture core surrounded by peripheral devices. The microcontroller has 16 KB of Flash memory available for code along with 1 KB of SRAM and 512 bytes of EEPROM memory. The main peripheral devices available in the ATMEGA16 microcontroller are:

- 2 8-bit timers
- 1 16-bit timer
- Real time counter
- 4 channel of PWM
- SPI interface
- UART interface
- Analog to Digital Converter
- 4 8-bit General Purpose Input Output Ports

Although this microcontroller has very low performance comparing to the existing microcontrollers currently present on the market, it maintains its high didactical value thus being one of the most suitable microcontrollers for teaching. A strong argument to sustain this statement is that it only requires a power supply in order to run and it is available in 40 pin DIP capsule thus making it perfect for building small circuits on a breadboard. Another important advantage is that it can be clocked using an internal RC oscillator with a maximum frequency of 8MHz.

The pinout of ATMEGA16 is also very simple and well organized as presented in Fig. 2-1 [14]:

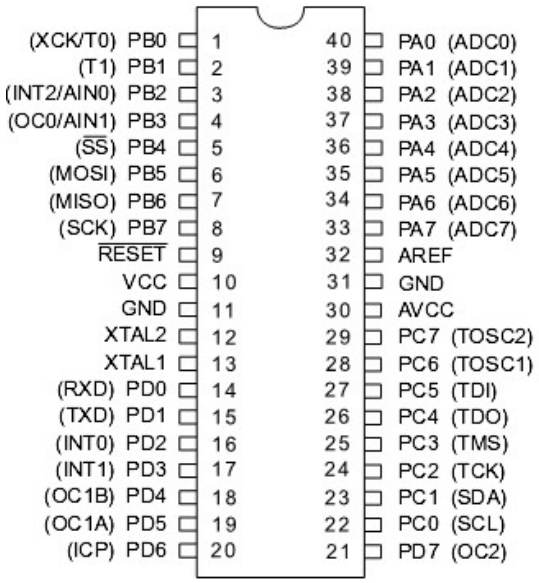

Fig. 2-1 Pinout of ATMEGA16

As it can be observed in the pinout, the microcontroller has 4 ports available for connections: PORTA, PORTB, PORTC and PORTD. Each pin is presented with its designated ranking in the corresponding port (ex. PB1 being line 1 from PORTB) along with its alternated function. A currently used practice in microcontrollers is to multiplex more functions on a pin, thus reducing the number of pins in the capsule. In the case of ATMEGA16 the alternate functions of a pin are written in brackets. For example, pin PD0 is normally a GPIO pin belonging to PORTD but when the serial interface is activated the function of this pin changes to the RXD signal of the serial interface. Same rule is available for all pins. Special attention needs to be taken when using the lines of PORTA. In order for these to work, even in GPIO mode, power needs to be applied to the AVCC pin.

ATMEGA16 may be programmed either by using the ISP interface or by using a dedicated JTAG debugger. When using an ISP programmer the PINS involved in this operation are pins from 5 to 11. Practically, an ISP programmer needs access to the SPI interface of the microcontroller as well as to the RESET pin and, if needed, to the power supply related pins. A possible connection schematic for connecting an ISP programmer to ATMEGA16 through a standard 2x5 connector may be the following:

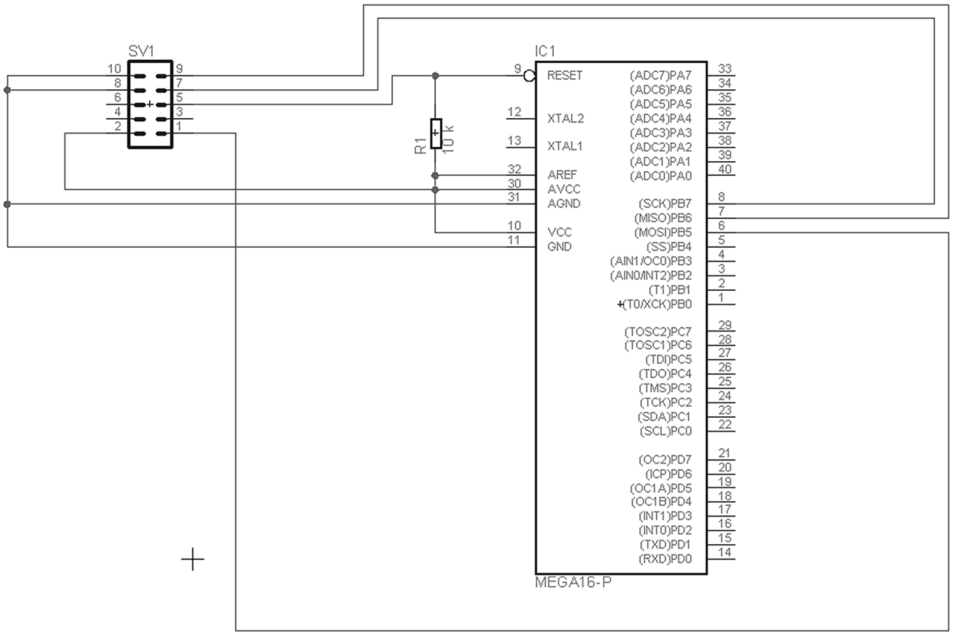

Fig. 2-2 ATMEGA16 ISP connection

The ISP programming of ATMEGA16 is limited only to downloading the executable code from the PC into the microcontroller's flash memory and programming of the Fuse Bits. No real-time debugging can be made using an ISP programming. In order to be able to debug a running code, in real time, on a microcontroller a JTAG debugger is usually

required. The JTAG is connected to the microcontroller through dedicated pins. In the case of ATMEGA16 the dedicated pins for JTAG connections belong to PORTC from PC2 to PC5. It is important to mention that if the JTAG interface is enabled on the ATMEGA16 microcontroller these pins cannot be used by the programmer. These pins remain dedicated to the JTAG interface. The connection between a standard 2x5 pin JTAG connector and ATMEGA16 may be done as shown in the following figure:

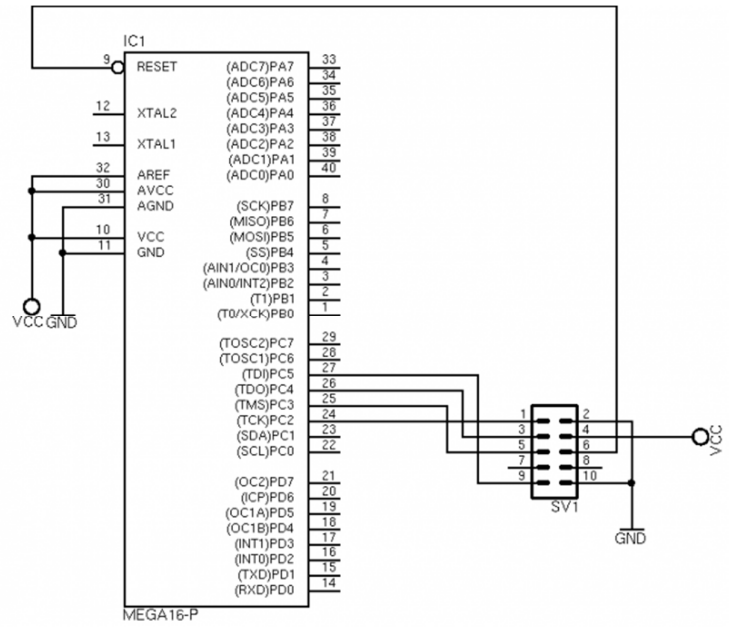

Fig. 2-3 ATMEGA16 JTAG connection

The JTAG that will be used for downloading the code to the ATMEGA16 microcontroller as well as for debugging the running code is the Atmel-ICE JTAG [15], which is supported by the new development tools from Atmel.

The enable/disable of the JTAG interface of ATMEGA16 as well as other critical settings of the microcontroller may be configured by accessing the so called Fuse Bits. These bits are practically represented by registers which may only be accessed by a JTAG or ISP programmer. The Fuse Bits registers cannot be accessed from the running code from the FLASH memory and they are not visible to the programmer.

Using the Fuse Bits the following items may be configured (via JTAG or ISP programmer):

- JTAG interface – it may be enabled or disabled
- ISP interface – it may be enabled or disabled
- Preservation of the contents of the internal EEPROM memory upon programming the FLASH memory
- Brown-out detector

- Clock source – various internal RC oscillator clock frequencies, external quartz oscillator, external clock source

2.1.2.2 ATMEGA16 header board

During this laboratory the ATMEGA16 microcontroller will be used along with a small header board which will not only export all the microcontroller's pins on header but will also contain a JTAG connector a quartz oscillator connected to the microcontroller. Such a board may be the following:

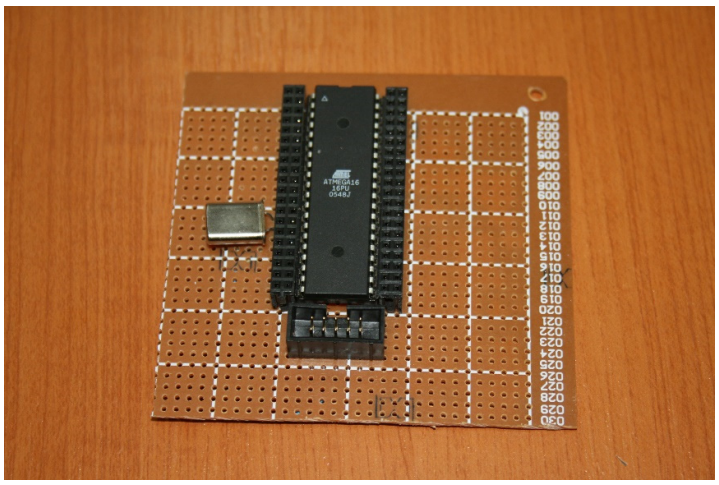

Fig. 2-4 ATMEGA16 header board

As it may be observed in Fig. 2-4, the microcontroller is surrounded by 2 female 2 line headers. Each pin from the microcontroller is directly connected to the corresponding pins near it. Practically all the pins from the microcontroller are accessible using the 2 line female headers. A block schematic of the header board may be found in Fig. 2-5:

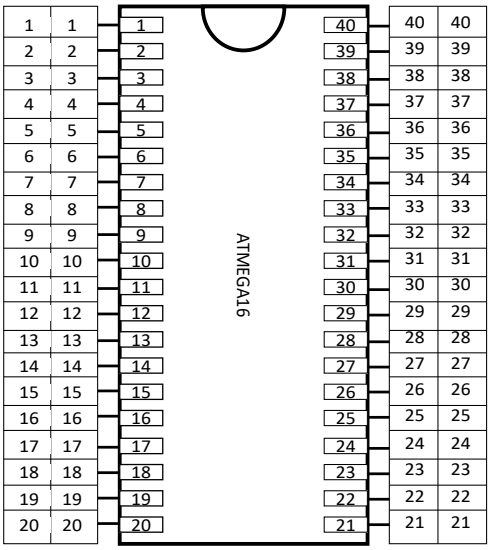

Fig. 2-5 ATMEGA16 header board block schematic

2.1.2.3 Peripheral board

The header board presented above will be interfaced with a peripheral board which will also be provided for de students during these laboratory assignments. The peripheral board, code name PBMCUSLK AXM-0392 [16] was initially designed by Freescale and now it is maintained by NXP. This board practically consists of an isolated breadboard which is only mechanically linked to an electronic board containing various peripherals from LEDs, pushbuttons, serial interface, LCD to various connectors as presented in Fig. 2-6

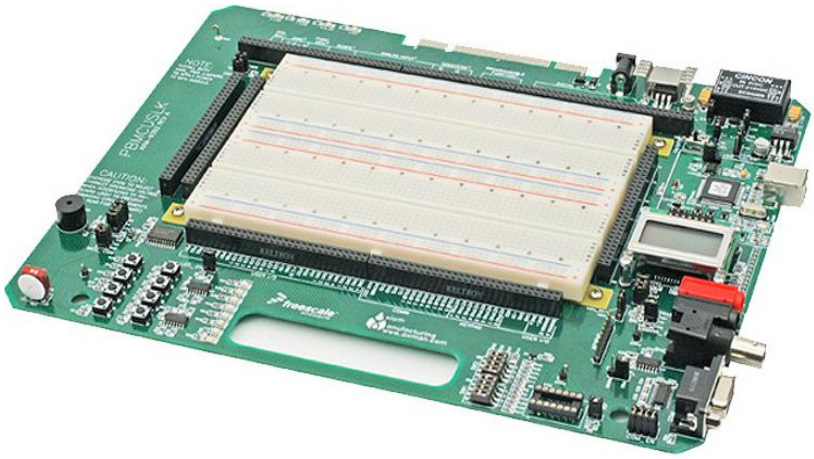

Fig. 2-6 PBMCUSLK peripheral board [16]

As stated before, the breadboard in the middle is only mechanically connected to the rest of the board. No electrical connections are made. In order to connect the peripherals on the PBMCUSLK board to a circuit being designed on the breadboard, the black female header surrounding the breadboard may be used. The significance of each pin in the header is written near the pin itself. No additional documentation is needed in order to use the board in basic applications. If needed, more information about the board may be find in its user manual [16].

2.1.2.4 Relevant documentation

Beside the present laboratory work manual, in order for the attendees to be able to process these laboratory works access to further documentation is needed. A list with some of the needed documents, may be the following:

- Brian W. Kernighan, Dennis M. Ritchie - The C Programming Language [1]
- ATMEGA16 User Manual and Datasheet [14]

2.1.3 Laboratory applications planning

The main goal of this laboratory is to provide the students with the knowledge to acquire and process digital signals using embedded systems. The finality of the laboratory will be an oscilloscope with 2 channels with negative and positive triggering along with frequency measurement and time scale change. In order to accomplish this goal the laboratory works and assignments are structured in such a way that each laboratory work will build on top of the result of the previous laboratory work. Having this approach, each laboratory work will be the project closer to completion.

The first laboratory work will be concentrated into introducing the students into the programming of ATMEGA16 using the provided materials. The developing tools provided by Atmel will be presented along with the structure of the necessary documentation. The practical aspect of this first laboratory work will be to implement a small LED blink application on the microcontroller.

The second laboratory work is oriented into establishing the communication between the microcontroller and the host PC via the RS-232 interface. The communication protocol will be presented in detail along with the serial port terminal emulator running on the host PC which will be used to communicate with the microcontroller through a COM port. Students will have to configure the UART interface of ATMEGA16 and implement a small library containing the necessary operations to work with the UART interface. The mail program will have to be able to continuously send the same character over the serial interface to be viewed on the host PC.

The main subject of laboratory work 3 is represented by the Analog to Digital Converter of the ATMEGA16 microcontroller. Students will have to acquire the signal on one of the converters channels and make the necessary calculations to convert the capture sampled into voltage. The finality of this laboratory work will be a digital voltmeter with the terminal emulation software on the PC as a viewer of the voltage.

In laboratory work 4 the first aspects of the oscilloscopes will be implemented using a dedicated software on the host PC for visualizing the waveforms. In order to reach this goal students will have to use an addiction ADC channel and also implement a small communication protocol overt the RS-232 interface in order to be able to send the correct data to the PC visualization application.

The following laboratory works are mainly concentrated into implementing additional features of the newly developed oscilloscope. In first step triggering will be added. With triggering available the signal frequency values will be calculated for the synchronized channel.

Laboratory work 7 and 8 are responsible for implementing the RS-232 reception for the microcontroller. In this step the microcontroller will have to recognize small commands sent using the visualization software on the PC. The commands will have to implement basic controls that are found on a real oscilloscope.

week	Laboratory work	Observations
1	Establishing laboratory groups	Establish groups of 2
2	Introduction + Laboratory work 1	
3	Laboratory work 2	
4	Laboratory work 3	
5		
6	Laboratory work 4	
7		
8	Laboratory work 5	
9	Laboratory work 6	
10		
11	Laboratory work 7	
12	Laboratory work 8	
13		
14		

Table 4 Laboratory applications planning

2.2 Laboratory work 1 – First project: LED blink

This laboratory work presents to the students the first steps into developing and debugging applications on Atmel ATMEGA16 microcontroller with the aid of Atmel Studio 7 environment. As it is customary, the first application that is to be considered when beginning work on a new microcontroller, or even as first steps in embedded programming, is the LED blink application using software delays.

The presentation of this laboratory work will be divided according to 2 point of views: a hardware point of view and a software point of view. The hardware part will present the necessary connections to be made in order to build the first LED blinking

applications. The software part is responsible for presenting both the developing environment and the coding to be designed in order to build the application.

Before considering into analysis the schematic that should be implemented, the focus needs to go on the basic schematic of a microcontroller with emphasize on the GPIO module. A generic schematic of a microcontroller with a serious customization for the ATMEGA16 microcontroller which will be used for the laboratory works is presented in the following figure:

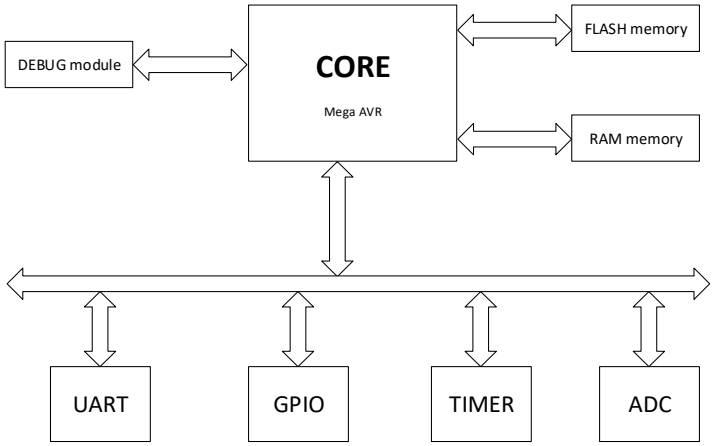

Fig. 2-7 General microcontroller block diagram

In the schematic above, the central piece of the microcontroller is represented by the CORE. This is practically an ALU which has the only task of executing the code. Embedded into the silicon capsule along with the Core are the two usually present memories: the FLASH memory and the RAM memory. The FLASH memory is used to store the code that will be executed by the Core. The RAM memory is practically the Data Memory that will store the variable data of the code. Both of these memories are usually accessed directly by the core event through the FLASH memory, which is sometimes cached. The Core is also connected via various busses to numerous peripheral devices. In our upper schematic we can identify peripherals such as the UART module, TIMER, Analog to Digital Converter (ADC) and the highly used General Purpose Input Output module.

The module that presents great interest to our laboratory work is the GPIO module. This module offers a collection of digital lines, organized in ports that have the advantage that they can be programmed to both be able to establish a logic values on the line but are also able to read the logic value of the line. Our current laboratory work focuses on using this module in order to implement the LED blinking application.

The first aspect to be discussed is the schematic that needs to be implemented to make the LED blink application. The block schematic is presented in Fig. 2-8

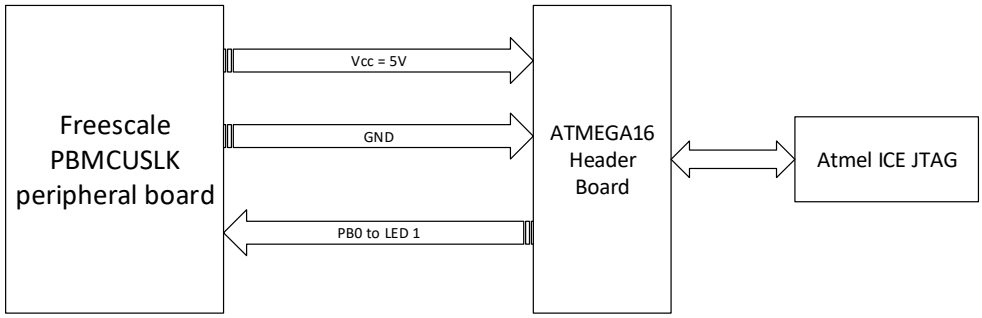

Fig. 2-8 Led blink connection block schematic

Having a more detailed analysis the necessary connections to be made are the following:

- Connect the 5 V power line from the header on the peripheral board to the correct pin of the ATMEGA16 header board (pin 10 on ATMEGA16)
- Connect the GND line from the header on the peripheral board to the correct pin of the ATMEGA16 header board (pin 11 on ATMEGA16)
- Connect one of the LEDs of the peripheral board (using the corresponding pin on one of the headers) to line 0 of PORTB (PB0) of ATMEGA16 from the ATMEGA16 header board
- Connect the Atmel ICE JTAG to the ATMEGA16 header board and to an USB port from the PC

ASSIGNMENT 1: Make the connections described above. Search the correct pins both of the peripheral board and on the ATMEGA16 header board. Have the laboratory teacher verify the connections before powering up the system.

In this first step, making the hardware connections, represents the simplest task from this laboratory applications. The much more complex task is from a software point of view. Firstly, the primary steps into creating and configuring a new project into the developing environment Atmel Studio 7 will be presented. The starting point of the Atmel Studio 7 environment is presented in Fig. 2-9.

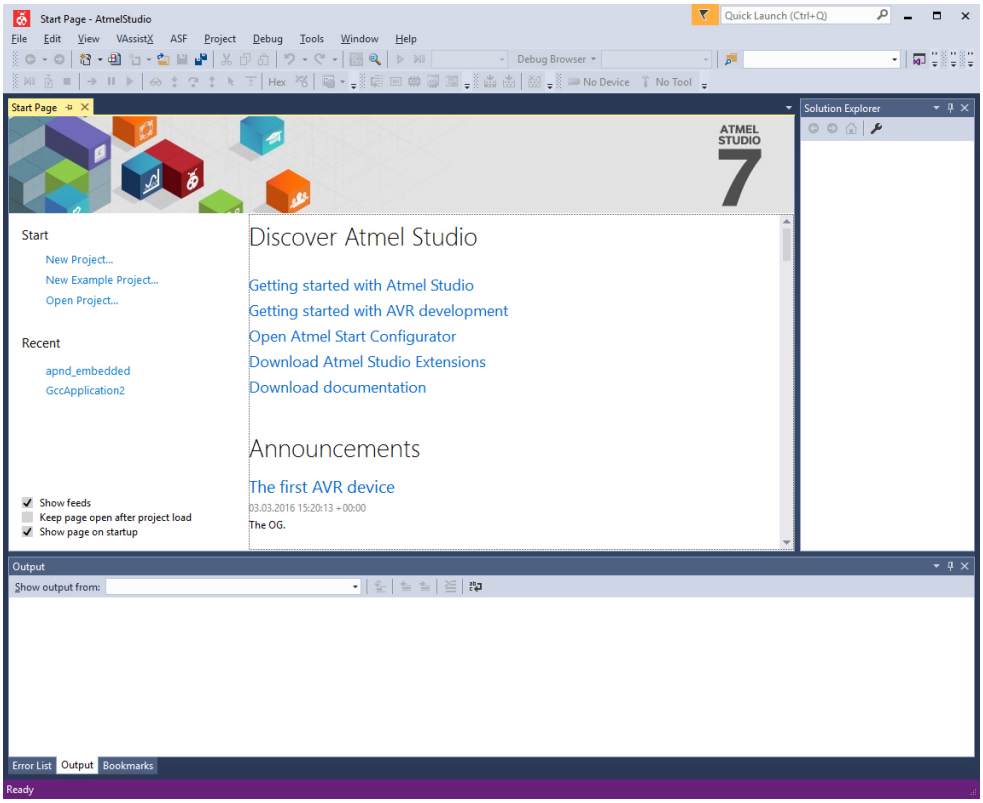

Fig. 2-9 Atmel Studio 7 starting page

To create a new project the “New project…” link found on the left side as presented in Fig. 2-9 needs to be selected. The same behavior is available if using the menu bar: File -> New -> Project. The new project type that will be used is “GCC Executable Project”. Also the location path and project name can be specified as presented in Fig. 2-10:

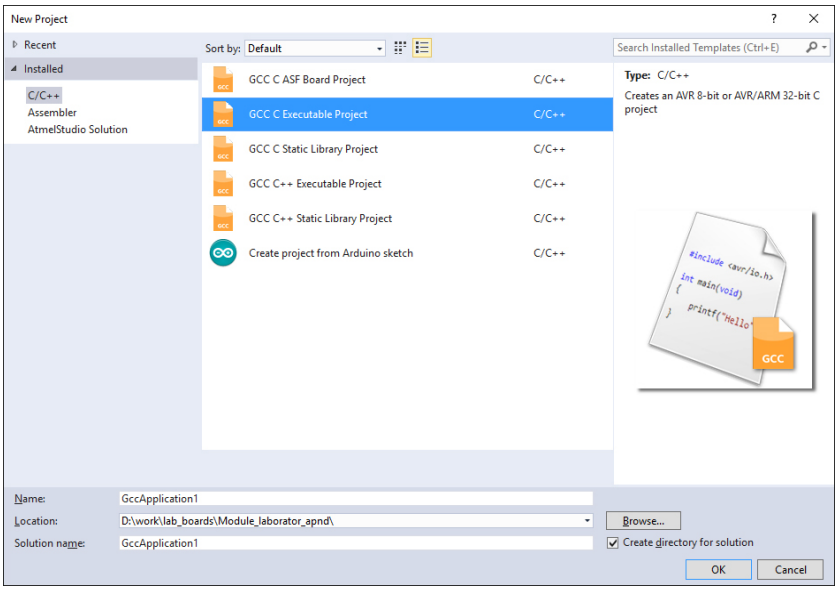

Fig. 2-10 Atmel Studio 7 new project dialog

The next step into creating a new developing project is to specify which microcontroller to be used. Select the Atmega16 device by searching it into the list of microcontrollers supported by Atmel Studio 7. In order to narrow down the search use the Device family combo box to filter the list for Atmega family. Such an example is presented in Fig. 2-11:

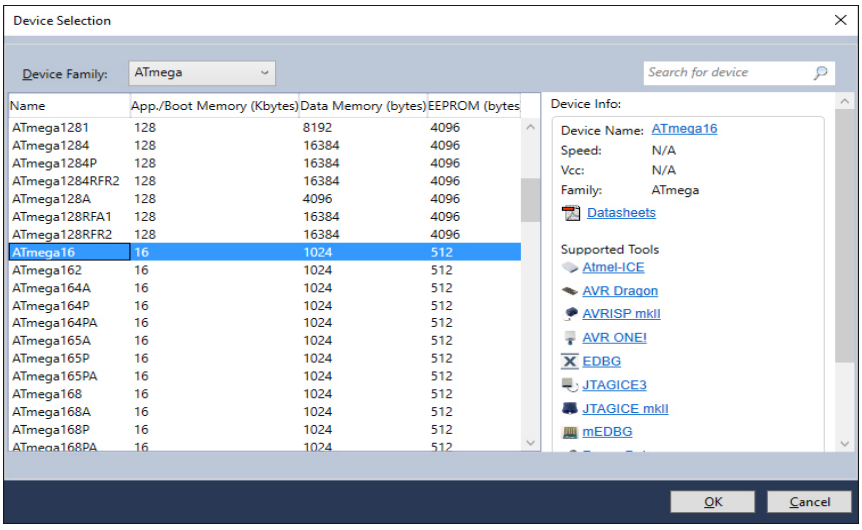

Fig. 2-11 Atmel Studio 7 microcontroller selection

After the project is successfully created, the development studio adds a template code file to the project containing only the main function. In order to view the project structure with the files referred by the project select the "Solution explorer" setting either by finding it on tab in the right part of the application or by using the menu View -> Solution Explorer (hotkey CTRL+AL+L). This usually shows the solution explorer on the right of the applications as shown in figure:

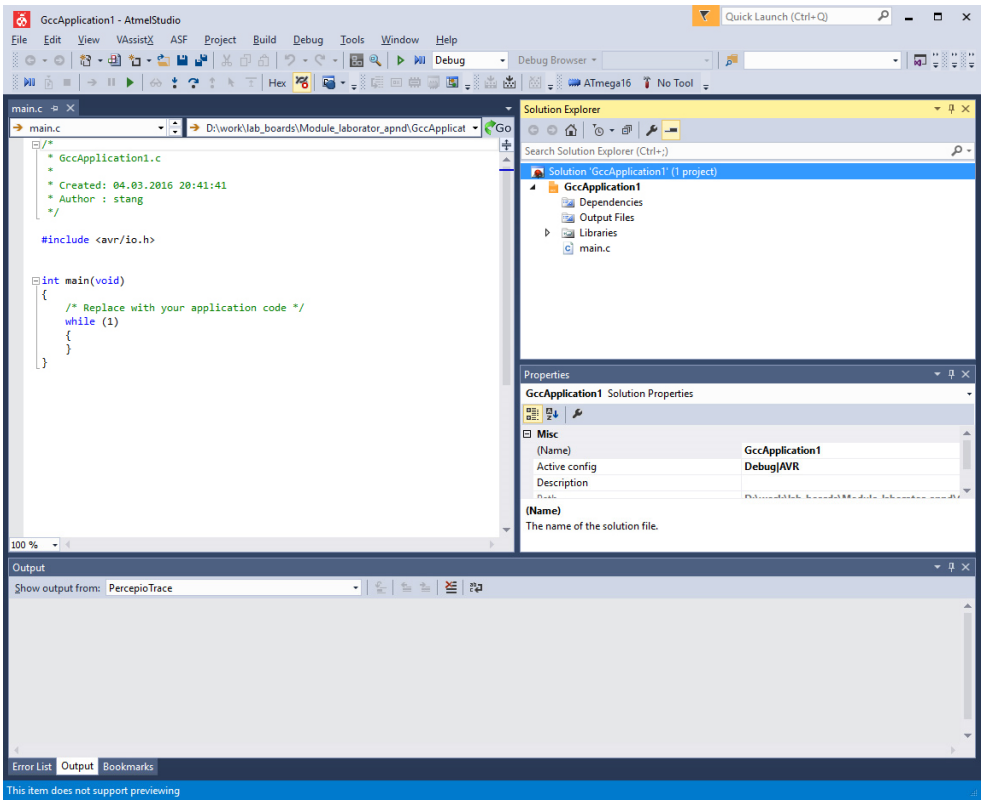

Fig. 2-12 Atmel Studio 7 with project created and solution explorer present

The next step is to configure some options of the newly created project. In order to avoid some issues during programming, the compiler optimizations have to be disabled. There are many positive aspects when using compiler optimizations and in many situation are quite recommended. In our situation it is best to avoid the compiler optimizations mainly because we need to concentrate on the functionality of the applications rather than on performance.

In order to access compiler optimizations a right click on the project is necessary (in our case on GccApplication1 in project explorer) with the selection of Properties in the right click menu. To reach compiler optimizations select Toolchain from the left and under AVR/GNU C Compiler select optimization. From the Optimization Level combo-box

select (None –O0) for optimization level. A preview of the dialog for this issue is presented in Fig. 2-13:

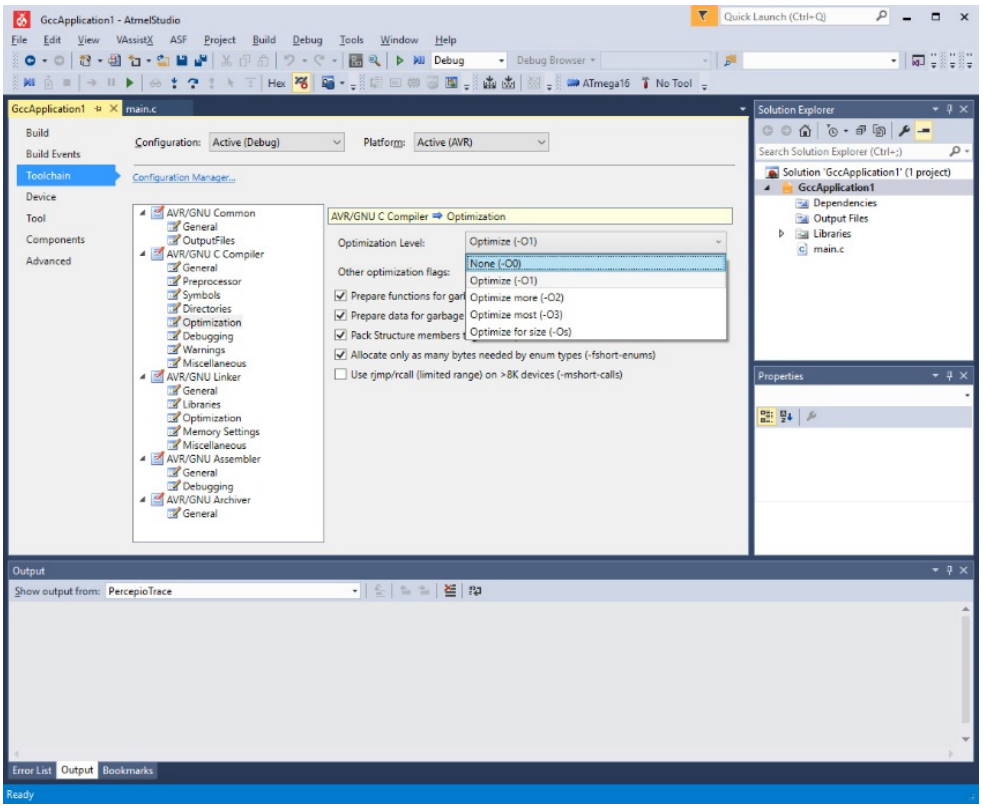

Fig. 2-13 Atmel Studio 7 compiler optimizations

The next important project configuration, which needs to be taken care of, is the selection of the Tool to be used for debugging. Having the previous screen, we used to configure the compiler optimizations select Tool option from the left. The following screen should appear:

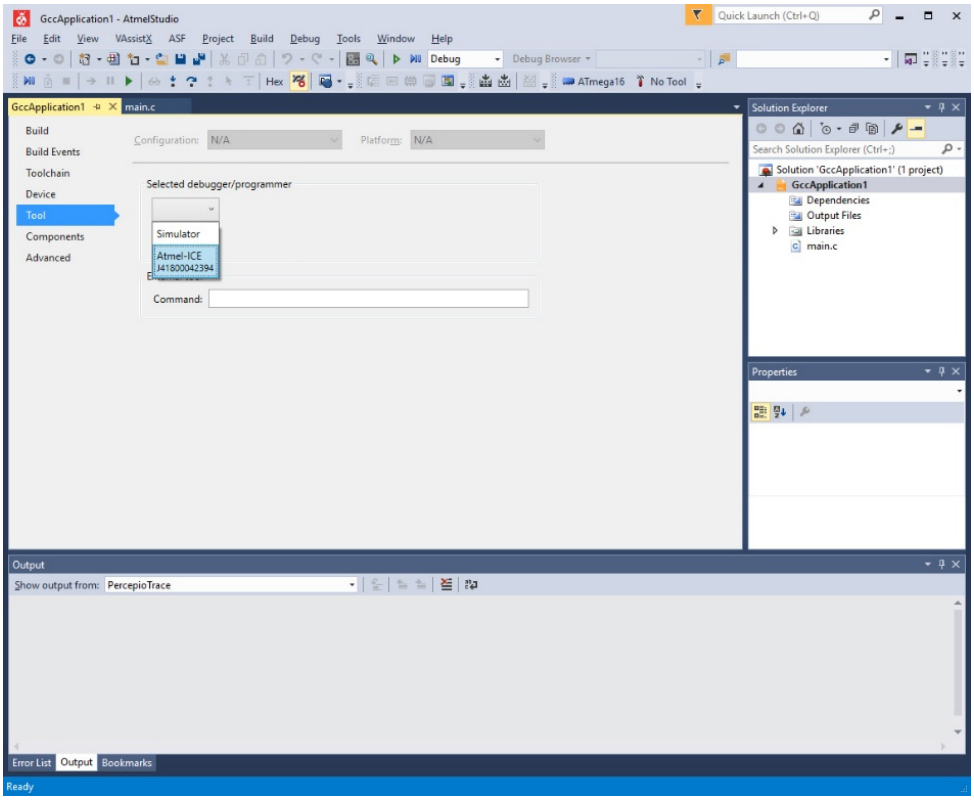

Fig. 2-14 Atmel Studio 7 tool selection

In Fig. 2-10, under the "Selected debugger/programmer" combo-box two options should be available (depending if the Atmel ICE debugger is connected via USB to the PC): the Simulator and the Atmel ICE debugger. The choice of this combo-box should not be permanent. If, in any moment, the student would want to use the simulator instead of the hardware JTAG debugger he can do so by selection the corresponding options. The only observation is that in Simulator mode, the developing environment is disconnected from the target. In order to be able to download the code on the microcontroller and to debug it, the Atmel ICE debugger (in our case) should be selected.

When selecting the Atmel ICE debugger more options regarding this tool will appear on the same dialog. From the interface combo-box the JTAG option needs to be selected. Moreover, special attention needs to be taken on the value of the JTAG clock. A safe value to use would be 200 kHz as the default value should be. An example of settings what should be configured in this dialog is presented in the next figure:

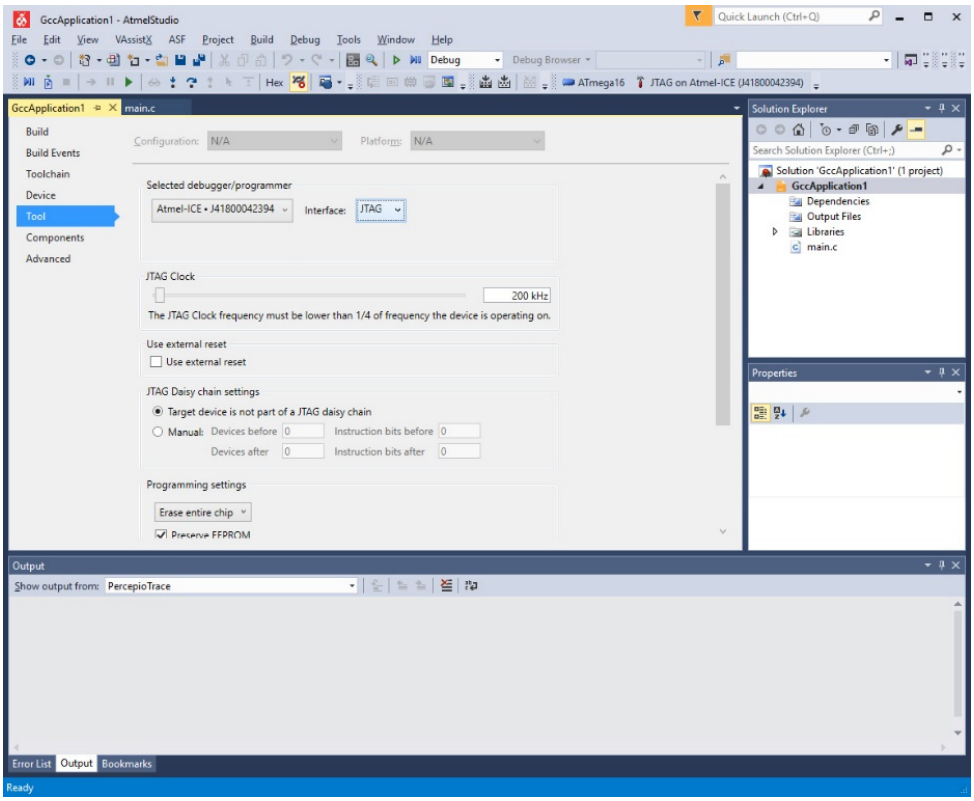

Fig. 2-15 Atmel Studio 7 tool selection and configuration

After these configurations are done the project should be save in order to use the same configured environment next time.

Another important aspect, which needs to be discussed, is how the target and the connection between the target and the JTAG should be tested. This testing method should usually be used before starting working with the target, but usually only once at the beginning of, if malfunctioning is detected. The testing method involved bringing up the Device programming dialog by selecting from main menu Tools -> Device Programming. The dialog that should be brought up is similar to the one presented in the next figure:

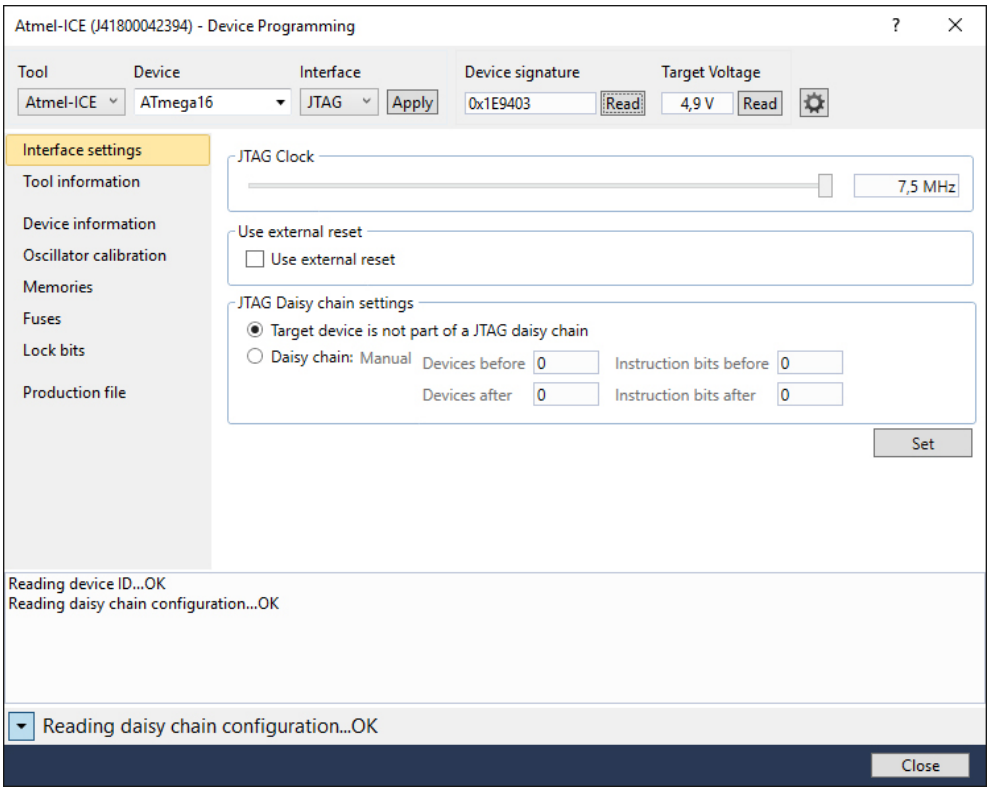

Fig. 2-16 Atmel Studio 7 Device programming dialog

In this dialog, mainly in the upper part, in the first opening, usually only the tool remains selected (as Atmel-ICE in our case). Having Fig. 2-16 as an example, in the "Device" combo-box select ATmega16 and on the interface select JTAG if not already selected. After the selections press Apply and, if the connections are in order, then no error message should be displayed. In order to further verify the JTAG communication press read on the "Device Signature" region in order to read it from your ATMEGA16 microcontroller. A valid serial number should be read, in the case of a good communication. Moreover, in order to assure that the voltages are properly applied, the Target Voltage should be read by using the appropriate button. Having all of this information obtained, one can draw the conclusion that the JTAG communication with the target microcontroller is working properly.

Having all of this configured we can be assured that the project is suited for development. Note that this configuring should only be done once for the same project. Giving the fact that this laboratory tends to use a constructive, building approach, this newly created and configured project should be used for all the coming laboratory works.

Having the first project created in order take it to run on target it first must be compiled and built. This is done by accessing the menu Build -> Build Solution. The very

used shortcut key for this operation is F7. The result of the compilation is presented in the Output tab on the bottom of the Atmel Studio 7 screen as shown in Fig. 2-17:

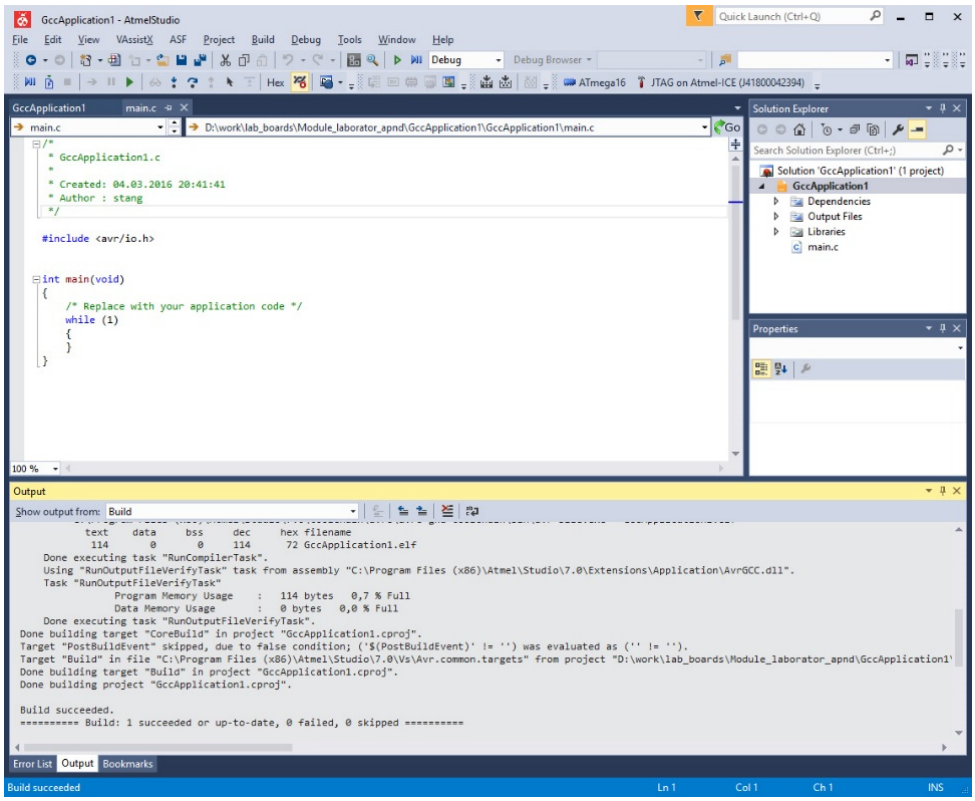

Fig. 2-17 Atmel Studio 7 Compilation result

Having a closer look on the output of the compilation in Fig. 2-17 we can observe not only the results on the last line but also, in case of successful compilation, the amount of code that the executable uses and also the amount of data memory needed with values in both bytes and percentage. The percentage is calculated relatively to the maximum amount of memory available for the currently selected microcontroller.

This information is important to an embedded developer not only to know if the code can fit the flash of the microcontroller or if the data memory is enough but also to calculate the differences if compiler optimizations are used. Currently our only interest is knowing if the program fits the available memories.

The next step into developing our application is to write the necessary code. Taking a closer look of the generated code of the newly created project we can identify a very important include statement:

Code listing 2-1 Register definition header include

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

This line of code includes, into the code file it is written, the header file containing the definitions of register names of ATMEGA16. Beside this include, further includes should always be present into every ATMEGA16 project:

Code listing 2-2 Necessary includes

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
```

All the necessary includes that should be present in almost every file within an ATMEGA16 project are the ones presented above. The first include was detailed before. The second include header file contains the definitions of the interrupt vectors of ATMEGA16. The last include contains the file defining delay functions. In order for this included library to properly work the CPU frequency should be properly defined using:

Code listing 2-3 CPU Frequency definition

```
#define F_CPU 14745600UL
```

This definition “informs” the delay library of the frequency the processor is clocked by. In our situation, as presented above, the clock frequency is 14.7456 MHz or 14754600 Hz.

A full list of the inclusions and definitions that should be present in manly all the files referring to the ATMEGA16 periphery can be the following:

Code listing 2-4 CPU Frequency definition

```
#define F_CPU 14745600UL

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
```

As stated before, the module that will be used in order to blink the LED, is the GPIO module.

The digital lines of the GPIO module are organized in ports. The ATMEGA16 microcontroller has 4, 8 bit lines wide ports (PORTA, PORTB, PORTC, PORTD). The main characteristic of a port line is the direction. For example, if we need to drive a LED connected to a port line, meaning we would like to establish a high or low logical value in order to power on or off the LED, the line is considered to be an output line. In another example, if we want to read the logical value of a line, for instance, when connecting a push button to a port line and wanting to read the state of a push button, the line is considered to be an input line.

Another important aspect is the one related to how a port line can be driven, when it is an output port line, or how it can be read, when it is configured as an input line. This and the configuration of the direction of a port line can be accessed through a collection of registers. All the registers have the same structure, as in Fig. 2-18, but with different meaning. Every port of the microcontroller has the same collection of registers. Each bit of the register controls the "characteristic" of the corresponding digital line of the port.

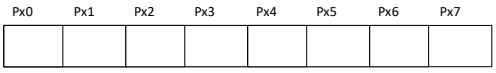

Fig. 2-18 ATMEGA16 General GPIO register structure

The above structure is available for all the registers corresponding to the GPIO module of ATMEGA16. The first aspect to be analyzed, as discussed above, is represented by the direction of a port line. As stated in the documentation of ATMEGA16 the DDRx registers are related to this aspect as that writing a logic 1 to one bit of this register configures the selected line as an output line. Also, writing a logic 0 to one bit of this register configures the corresponding line as an input line. The DDRx registers (where x identifies the port: A, B, C or D) have the same structure as in Fig. 2-18.

Another important set of registers for the GPIO module of ATMEGA is the set of PORTx registers (where x identifies the port: A, B, C or D). These registers are only to be used for the lines which are configured as output. These registers define the logical value that the line has. For example, writing a logic 1 on one of the bits of a PORTx register, the corresponding line of port x is driven to logic 1. Same algorithm applies for writing a logic 0 value.

A register set related to the PORTx register set presented above is the PINx register set. The PINx registers are used when working with input lines. From these registers we can determine the logical value of a line (pin) of a microcontroller. Reading this registers practically offers the logic state of a line or pin configured as input. For example, when reading logic 1 of a bit in a PINx register the corresponding pin has a logic 1 value applied on it. Some goes for 0 logic.

More information about how the GPIO system works may be found in the ATMEGA16 datasheet on chapter named I/O Ports [14]

ASSIGNMENT 2: Read the ATMEGA16 documentation and find the registers that have to be configured in order to drive a LED connection to line PB0 of the microcontroller. Establish and explain the values to be written in the registers.

A pseudocode implementation of a program which blinks the LED can be the following:

Code listing 2-5 Led blink main program flow

```
void main()
{
  init_led(); // initialize the port direction using  DDRx register
  while(1)
  {
     led_on(); // turn on led, logic 1 on coresponding bit from PORTx register
     delay(); // delay loop
     led_off(); // turn off led, logic 0 on coresponding bit from PORTx register
     delay(); // delay loop
  }
}
```

The init_led pseudocode function represents the configuration of the directions of the pin the LED is connected to. The led_on and led_off represent the code lines needed to drive the pin the LED is connected to, in order to turn the LED on or off.

The delay function may be implemented using the provided library functions with the following prototypes:

Code listing 2-6 Delay functions prototypes

```
void _delay_ms(int milliseconds);
void _delay_us(int microseconds);
```

ASSIGNMENT 3: Write a microcontroller program that blinks a LED with a period of 500 ms using delays.

HOME ASSIGNMENT: Read documentation related to the RS-232 communication protocol. Read the documentation about the RS-232 serial interface of ATMEGA16. Make a list with the registers that should be used to configure the serial interface of ATMEGA16 along with the values that should be considered. Take into account the serial communication parameters of BAUD 9600, 8 bits per character, 1 stop bit. Also, make a list of the bits and registers that have to be accessed in order to transmit a byte over the serial UART line.

2.3 Laboratory work 2 – Serial communication - transmission

This second laboratory work is concentrated into developing the first microcontroller application with serial communication. The serial communication, universal asynchronous receiver transmitter (UART) RS-232 protocol, will be presented. From the microcontroller only the transmission will be implemented for now. The serial reception is scheduled to be presented in another laboratory application. During all the laboratory applications we will consider a simple UART communication with no hardware flow

controls. The outcome of this laboratory will be to build an application that transmits a character every second over the serial interface. The character is then displayed on a PC using a terminal software.

The UART protocol is probably one of the oldest communication protocols that are still being used in many applications. Even though it was designed in the 1960's the protocol is highly used even now because of its simplicity. Of course, nowadays it is used at much higher speeds than in the ones in the 60's.

When communicating using a synchronous protocol, a clock signal is present, thus the time synchronization is assured. In an asynchronous communication, the clock signal is not present and the data must carry its own information for time synchronization [17]. In the simple UART communications with no hardware flow control there may be only 2 communicating partners which may switch their role from receiver to transmitter. Each terminal has 2 dedicated lines for communication: a receive line (RX) and a transmit line TX. These lines are connected as displayed in the following figure:

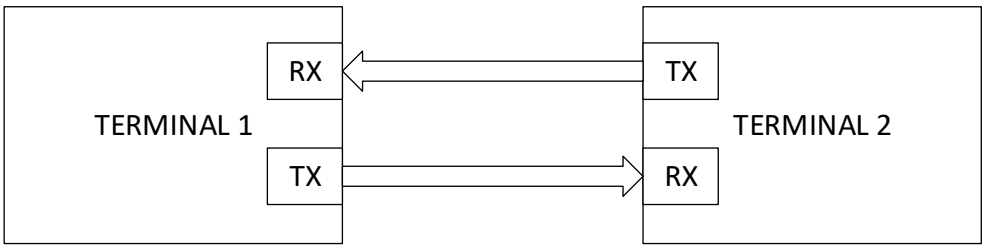

Fig. 2-19 UART communicating terminals

As stated before, giving the fact that no clock signal is present, the synchronization needs to be carried out by the transmitted data. In order for the synchronization to be accomplished, each communicating partner needs to sample the bits on the line with the same sample rate, which in communication, is translated into a symbol rate. The same symbol rate is needed to be configured in each communicating terminal. The symbol rate has the BAUD as a unit of measurement. Moreover, another important measuring unit is the transmission speed which is the number of bits transmitted per second (bps). It is customary to use the term "the BAUD of the serial communication is …", for example, 9600 bps. Having a more practical approach, the main interest is actually on how long a bit is in time. This is usually calculated using:

$$t = \frac{1}{BAUD}$$

Having an example of a BAUD of 9600 bps the length of a bit is:

$$t = \frac{1}{BAUD} = \frac{1}{9600} = 104\ us$$

If having to watch the character 'a' being transmitted using an oscilloscope it should look like in the following figure:

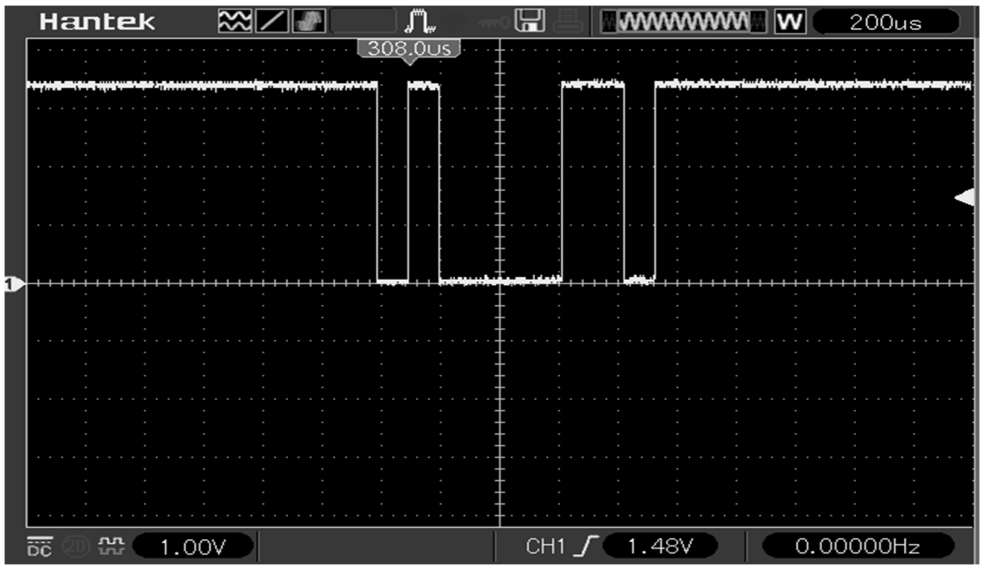

Fig. 2-20 UART character serialization

Using the oscilloscope to measure the time a bit occupies when having a BAUD of 9600 for bit sample rate, the following result may be found as shown in the oscilloscope capture:

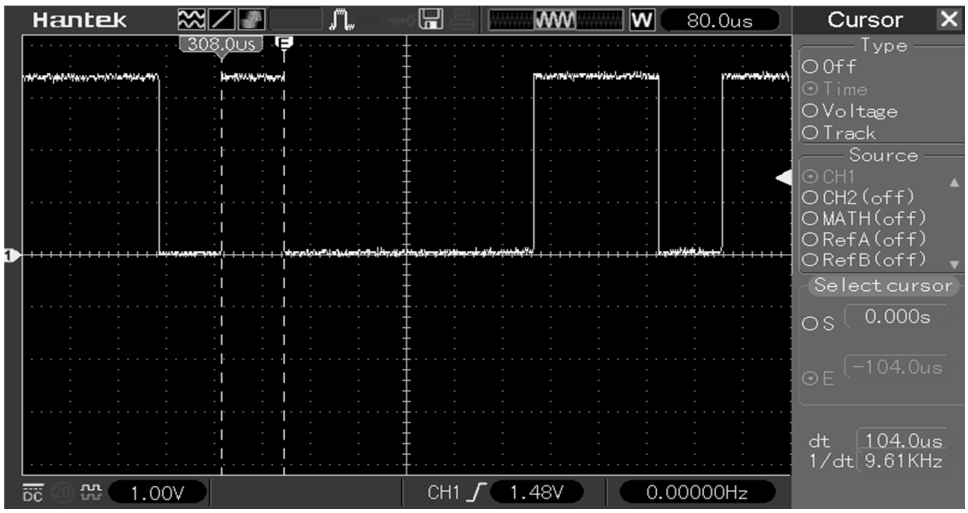

Fig. 2-21 UART bit measurement

In Fig. 2-21 the cursors were set to measure the one bit. The result is displayed in the bottom right corner as being 104.0 us (or 9.61 kHz if converted to frequency). We can notice that the value in kHz is almost equal to the value of the BAUD of 9600 bps.

Both the transmitted and the receiver have to be configured in order to both use the same BAUD rate. Having differences between the sampling rates of the two communicating partners may result in transmission errors.

The next important aspect, which needs to be discussed, is how data is encapsulated by the protocol in order to be transmitted over the line. This aspect is described by the following table:

Length (bits)	1	5-9	1-2
meaning	Start bit	Data bits	Stop bits

Table 5 UART protocol encapsulation

As found in Table 5 the protocol starts with a start bit. This bit announces that a new frame begins. Having an UART line inactive at logic "1" the start bit is usually encoded as logic "0". This is, in many cases, hard coded. The programmer usually cannot modify the number, length or values of the start bit.

Following the start bit, there are 5 to 9 bits of data. This is represented by a configurable parameter and needs to be the same on the receiver and transmitter. The data for which the protocol was designed is represented by characters which may be encoded in 5 to 8 bits according to the ASCII table. In the situation when 9 bits of data are specified, the 9th bit serves as parity which is calculated by both the transmitter and receiver. The receiver also compares the calculated parity with the one transported by the 9th bit in order to detect transmission errors. There may be an even parity or an odd parity. In an even parity the 9th bit is logic "0" when there is are even number of logic "1" bits in the data word. Same algorithm goes for the odd parity. In many situations the parity is not used, thus the number of data bits is set to 8 in order to disable parity. This is also usually configurable.

After the data is sampled, the frame ends with one, one and a half or two stop bits which are usually encoded as logic "1" bits. This option is also configurable.

Having these explain, the following conclusions may be deducted:

- Both the receiver and the transmitter have to function on the same parameters
- Only 2 communication partners may be used in serial UART communication bus
- Both of the communication partners may be receivers or transmitters
- Each communication partner has 2 lines: a reception line and a transmission line
- The start bit is only one with logic value "0" and is not configurable
- The configurable parameters are:
    - Character length: 5,6,7,8 bits
    - Parity
        - Odd parity
        - Even parity
        - No parity
    - Number of stop bits (1 bit, 1+1/2 bits, 2 bits)
    - BAUD rate
- Same configuration needs to be present on both communicating partners

The ATMEGA16 microcontroller has a dedicated peripheral module serving as an UART interface. The full documentation of the USART interface of ATMEGA16 may be found in the ATMEGA16 datasheet [14] at the USART Chapter. The pins that are mapped for the USART interface are found on PORTD and are PD0 serving as RX (RXD) and PD1 serving as TX (TXD).

The next step is to make the necessary connections between the ATMEGA16 header board and the peripheral board on one hand and on the other hand between the peripheral board and the PC. A block schematic of the connections to be made is displayed in the following block diagram:

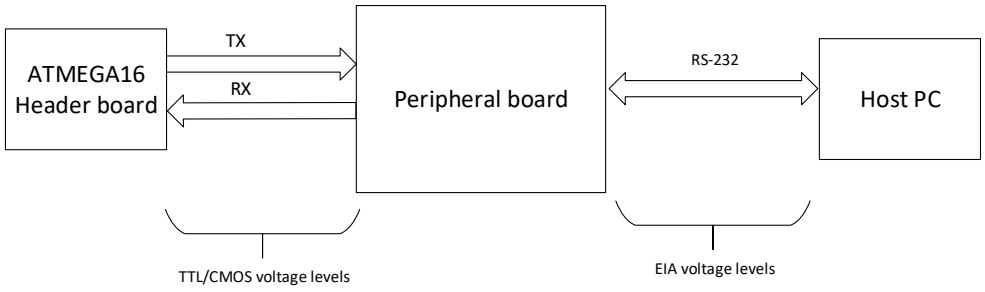

Fig. 2-22 UART connections block diagram

It is important to mention that the signals (RX and TX) between the ATMEGA16 and the peripheral board have CMOS/TTL voltage levels. These levels cannot be used to send data over long lines. A solution to this is to translate these signals into EIA voltage levels, which are more resistant to hazardous environments and can also be used to send data over longer lines. In our case, EIA voltage levels are used to transfer the data from the peripheral board to the host PC using a standard DB9 serial cable. The translation of the signals is done on the peripheral board using a dedicated integrated circuit, like MAX232 [18]

The first set of connections can be made using the provided wires. The TXD pin (PD1) from ATMEGA16 header board needs to be connected to the TXD header pin on the peripheral board and the RXD pin (PD0) from ATMEGA16 header board needs to be connected to the RXD pin on the header of the peripheral board. The signals of the peripheral board are named from a peripheral point of view.

The second set of connection can be made using a standard DB9 serial cable in order to connect the serial interface (through the DB9 connector) of the peripheral board to the serial interface of the host PC.

ASSIGNMENT 1: Make the necessary connections and have the laboratory supervisor verify them.

The next step is to concentrate on the software part of the ATMEGA16. Prior to the configuration of the USART module of ATMEGA16 the direction of the responsible pins needs to be set accordingly using the DDRD register. The PD1 serving as TXD pin, acting as the transmission pin of the USART interface should be configured as output. The PD0 pin serving as RXD pin, acting as the receiver pin, should be configured as input. Same algorithm is applied as in the previous laboratory work. For more information please read the I/O Ports Chapter in the ATMEGA16 documentation.

ASSIGNMENT 2: Read the documentation regarding the USART module of ATMEGA16 concentrating on the registers. Make a list with all the registers that should be used for configuring the USART interface. The interface should be configured with the following parameters: BAUD 9600 bps, 1 stop bit, 8 bits per character, no parity. Pay special attention on the address sharing of registers UBRRH and UCSRC (URSEL bit makes the difference).

The first aspect in the configuration of the UART peripheral module of ATMEGA16 is to calculate the divisor value (UBRR) that the microcontroller will use to generate the BAUD. This can be done by using the formula provided by the producer which can also be found the official documentation:

$$UBRR = \frac{F_{OSC}}{16 \cdot BAUD} - 1 \qquad (2\text{-}1)$$

Where:
F_{OSC} – represents the frequency of the ATMEGA16 internal clock in Hz (in our case 8 MHz)
BAUD – represents the actual baud rate (in our case 9600)
UBRR – represents the calculated value of the divisor which must not exceed 16 bits in size (no more than 0xFFFF)

The calculated baud rate divisor needs to be written into UBRRH:UBRRL registers which separate the most significant byte and the least significant byte of the 2 byte value UBBR. The UBRRH register, containing the most significant byte of UBRR needs to written first. The UBRRH register and the UCSRC register of the UART interface share the same address space. They can be differentiated by the value of bit 7, URSEL, in UCSRC register. According to the documentation when this bit is set to 0 the UBRRH is accessible. When needing to access UCRSC, this bit (URSEL) needs to be set to 1.

Beside the UBRRH and UBRRL registers, here is a collection of configuration and control registers which are used to control the UART interface. The full documentation of these registers has to be read in order to fully understand the functionality. In the following paragraphs only basic aspects will be discussed.

The UCSRA register contains mainly flags that are important when configuring the interface. The only bits that are significant for configuration are the U2X bit and MPCM bit. These bits should be left as logic 0 in our case.

Most of the configuration of the interface is done using the UCSRB register. We should be focused on the bits RXEN, TXEN and UCSZ2. The RXEN and TXEN should be written as logic one in order to enable the UART received (RXEN) and the transmitter (TXEN). Even if, for now, we will only work with the transmitter, we should enable also the receiver, thus it will be used in the coming laboratory works. The UCSZ2 register has meaning only along with UCSZ1 and UCSZ0. The value formed by these three registers define the size of the data word. The corresponding values can be identified in a table in the documentation under the UCSRC register. For this laboratory work, considering that we will use a data word of 8 bits wide, we will consider the bits having the following values: UCSZ2 = 0, UCSZ1 = 1, UCSZ0 = 1.

Pay attention that the UCSZ2 bit is contained in the UCSRB register but UCSZ1 and UCSZ0 are contained in the UCSRC register. Regarding the UCSRC register practically only these 2 pins need to be set to logic 1, the rest should be left as logic 0. The UCRSC register contains bits that configure the number of stop bits, the parity settings, and the synchronous/asynchronous operation of the interface. Letting the rest of the bits 0, beside UCSZ1 and UCSZ0 will let the interface configured as asynchronous, no parity and 1 stop bit.

It is important to mention that the attendees must read the whole documentation of these registers and not rely only on the explanations found in this laboratory work.

The configuring of a register should be implemented inside a function with a proper name. Once the interface is configured, the data transmit algorithm needs to be implemented. The flowchart for the configuration of the UART interface may be the one described in the following figure:

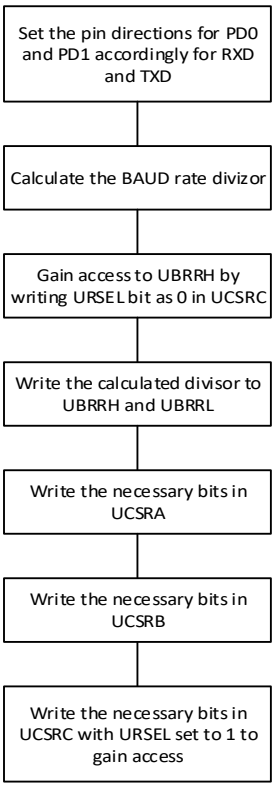

Fig. 2-23 UART initialization flowchart

In order for the interface to transmit a character, the data needs to be written into the transmission register: UDR. The same register is also used to read a newly arrived byte from the serial interface. Writing a byte to the UDR register is not enough when making a transmission over the UART interface. The programmer must also wait for the current byte to be transmitted. This may be done by using the UDRE bit in UCSRA. This bit informs the programmer when the transmit UDR data register is empty. After the UDR register is written for transmission the UDRE bit becomes logic 0. After the interface serializes the byte over the line the UDR data register becomes empty thus signaled to the programmer with UDRE bit becoming logic 0. If the programmer does not wait for the data to be transmitted over the serial line, more exactly for the transmission register to be emptied by the interface, there is a risk for this register to be written when it is not empty. In this situation the currently transmitted byte is corrupted and data overrun error is signaled through the appropriate byte in UCRSA.

A possible flowchart of the function capable in transmitting a byte over the UART may be found in the following figure:

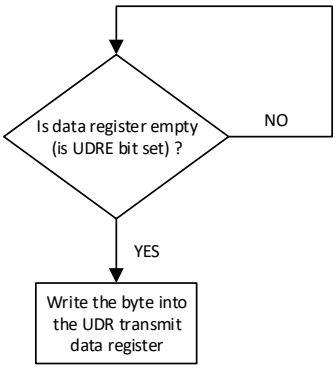

Fig. 2-24 UART transmit flowchart

When a character is transmitted over the serial line, in our case, using the connection to the PC as presented in Fig. 2-22, it can be displayed using a dedicated terminal software. Such a software is Docklight Scripting which can configure a serial COM port from the PC and can also be used for sending and receiving data.

Docklight scripting is an easy to use, but powerful, serial terminal software. The main advantages of *Docklight scripting* are:

- possibility to have access to all the settings of the serial port
- can function as a TCP/UDP client or server
- offers the possibility to define and send macros over the line (serial or network)
- has scripting features in order to simplify protocol interpretation
- offers good representation of unprintable characters
- byte interpretation may be ASCII, hexadecimal, decimal and binary

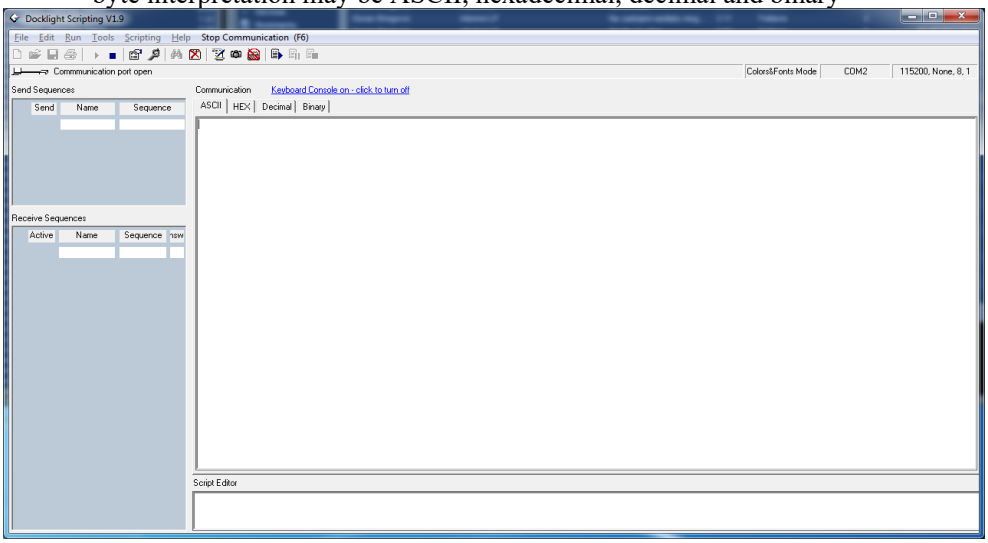

Fig. 2-25 Docklight main window

The main window of Docklight offers quick access to all of the features. The command bar contains practically all the necessary commands to configure, open, close and enable data write to the serial port.

Fig. 2-26 Docklight command bar

The active serial port along with its current configuration is displayed on the right side of the bar. In order to modify the COM port or the configuration a double click on the COM port name (ex COM 2 in Fig. 2-26). The configuration window is displayed in Fig. 2-27.

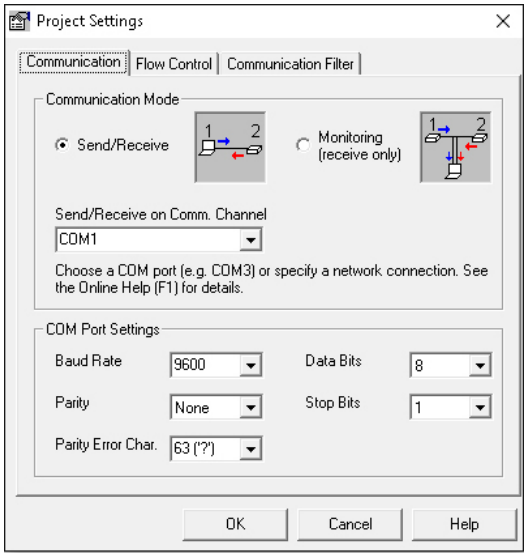

Fig. 2-27 Docklight COM port configuration window

The configuration of the serial port does not imply also the opening of the COM port for reception and transmission. These operations are made using some of the buttons on the command bar (Fig. 2-26). The buttons that present the most interest are: *Start Communication*, *Stop Communication*, *Keyboard Console On* and *Clear Communication Window*. These commands are highlighted in this order, from left to right, in Fig. 2-28.

Fig. 2-28 Docklight most used commands

The most important commands found on the command bar are those responsible for opening and closing the serial COM port. The first two highlighted buttons in the above figure are responsible for these actions. The opening of the port is activated through the *Start Communication* button and the closing of the port through the *End Communication* button. At the moment the COM port has been successfully opened, the state is updated below the button bar and *Docklight scripting* is ready to receive data through the serial port which will be displayed in the main window in the currently selected format. The window may be cleared using the *Clear Communication Window*. It is important to mention that opening the communication window will only activate the receive process. Any typed data in the main window will be discarded. In order to activate the transmission of data using the keyboard the *Keyboard Console On* button must the accessed. The status bar will be updated accordingly.

The main window of *Docklight scripting* displays the received and transmitted data in a strictly defined format. Each operation is preceded by a full timestamp along with a tag that specified whether it is a transmission ([TX]) or a reception ([RX]). Usually the transmitted data are colored in blue and the received data in red. The special characters are also displayed using a simple syntax: the definition of the special character according to the ASCII table between angle brackets. A sample of a short transaction displayed by *Docklight scripting* may be the following snippet.

Code listing 2-7 Docklight scripting communication sample

```
04.01.2016 12:21:51.575 [TX] – data transmitted from to the microcontroller

04.01.2016 12:21:52.455 [RX] – data received from the microcontroller
```

ASSIGNMENT 3: Open *Docklight Scripting*, configure the port for a BAUD rate of 9600 bps, 8 bit per character, 1 STOP bit and no parity. Open the COM port and activate the keyboard transmission feature.

ASSIGNMENT 4: Write a C library (a c file with a header file) which contains an initialization function for the UART interface and a function capable of transmitting one byte over the serial UART interface. The project should then have 3 files, for example:

- Serial.c – the C file containing the implementation of the functions
- Serial.h - the Header file containing the declarations for the function implemented in serial.c file that need to be exported
- Main.c – the C file containing the main program and function

The serial port needs to be configured as following: BAUD 9600 bps, 1 stop bit, 8 bits per character, no parity.

Write a main program that sends the same character over the serial interface once per second using the previous developed library. Test the program using Docklight scripting.

ASSIGNMENT 5: Modify the previous assignment in order to communicate at a BAUD of 115200 bps. Change the settings in Docklight scripting accordingly.

HOME ASSIGNMENT: Read about Analog to Digital Converters in general. Read documentation related to the Analog to Digital Conversion peripheral of ATMEGA16. Make a list with the registers that should be used to configure the ADC to convert a voltage (between 0V and 3.3V) applied to channel 0. Also make a list of the bits and registers

required in order to start a conversion and read the result. Write the formula in order to transform the data from a 10 bit number to voltage value considering the AVCC voltage as reference.

2.4 Laboratory work 3 – Analog to digital conversion. Digital voltmeter

The main peripheral device when dealing with digital signal acquisition is the Analog to Digital Convertor (ADC). During this laboratory work we will concentrate on the ADC peripheral module of ATMEGA16.

The scope of this laboratory work is to familiarize the attendees with analog to digital conversion. Students will have to develop another small library with basic functions to control the ADC, similar to the library in the previous laboratory work. The serial library will also be extended to support the transmission of a string over de serial interface. The actual finality of this laboratory is a digital voltmeter with the display on Docklight serial terminal. Practically the microcontroller program has to print the converted and calculated voltage in mV on the Docklight terminal via the serial interface.

The analog to digital conversion is a set of operations which transforms an analog input voltage into a binary code offered as output. This process in performed in 3 steps: sampling, quantization and binary coding.

Sampling is the first step of analog to digital conversion and consists into acquiring values of the analog input usually at periodic moments in time. The values of the samples are still continuous and belong to an infinite precision interval. The next step, the quantization, is the one responsible in obtaining finite precision values of the samples. The final step of the process is the binary coding which practically represents the values obtained after quantization using numbers represented on a finite number of bits [19].

The number of discrete values on which an ADC can represent the samples is indicated by the resolution of the ADC. This parameter is one of the most important characteristics of an ADC. For example having a resolution of an ADC of 10 bits means it can convert an analog voltage value into 1024 different levels in an interval of discrete values from 0 to 1023. The resolution can also be presented in volts introducing the term called the least significant bit voltage. The LSB represent the minimum change of the input voltage in order for the output binary code to change. The resolution of the ADC can then be defined as:

$$Q = \frac{FSR}{2^n - 1} \qquad (2\text{-}2)$$

Where
FSR – Full Scale Range defines full voltage range of the ADC
n – Represents the number of bits the ADC uses to encode the sample, practically the resolution in bits.

The Full Scale Range can be defined as follows:

$$FSR = V_{REF_{HI}} - V_{REF_{LO}} \quad (2\text{-}3)$$

Most of the conversions made are for voltages that are referenced to ground (0 V) thus in this situation the FSR is usually equal to the reference represented by the highest value of the voltage.

Having a simple example where the high reference of the ADC is 3.3 V, the low reference of the ADC is grounded and the resolution of the ADC is 8 bits then we can calculate the voltage resolution of the ADC:

$$Q = \frac{3.3\ V}{2^8 - 1} = 12.94\ mV \quad (2\text{-}4)$$

Practically every increase of the input voltage by 12.94 mV causes the encoded value of the ADC to change. Practically, in our example, the ADC only "feels" changes of 12.94 mV.

As state before, the actual output of an ADC is represented by a number which is *n* bits wide, with a maximum value of:

$$N = 2^n - 1 \quad (2\text{-}5)$$

So, practically, having the output of the ADC of a converted input voltage, denoted as $ADC_{VALUE,}$ the voltage represented by this discrete value, denoted as $V_{RESULT,}$ can be calculated as follows:

$$V_{RESULT} = ADC_{VALUE} \cdot Q = ADC_{VALUE} \cdot \frac{FSR}{2^n - 1} \quad (2\text{-}6)$$

Considering our previous example, let us continue by supposing that after the conversion the ADC gives a result binary represented as 0b01111001 which in hexadecimal is equal to 0x79 and in decimal as 121. The actual value of the voltage that the ADC converted (V_{RESULT}) may be calculated (in mv) as follows:

$$V_{RESULT} = 121 \cdot \frac{3300\ mV}{2^8 - 1} = \frac{121 \cdot 3300}{255} = 1565\ mV \quad (2\text{-}7)$$

The ATMEGA16 microcontroller has an Analog to Digital converter module which will be used in order to implement a digital voltmeter during this laboratory work. The ADC of ATMEGA16 has 8 possible channels for conversion with different reference voltage selection possibilities. The ADC also supports the possibility for converting differential signals where the low reference is not considered to be grounded. During these laboratory works we will not consider differential conversion thus only using channels

that have the low voltage reference grounded and the high voltage reference equaled to the Vcc power line. The maximum resolution of the ADC is 10 bit.

In order to have a simpler serial communication in the next laboratory sessions the maximum resolution, which will be used, is of 8 bits, thus only the most significant 8 bits will be taken into consideration (the lest significant 2 bits will be discarded). Furthermore, the reference that the ADC will use, should be the AVCC power pin.

The first aspect that is to be discussed is regarded to the connections that have to be made. The starting point will be the connection schematic used in the previous laboratory application for serial communication as presented in Fig. 2-22. Having this as a starting point the resulted block connection schematic for this laboratory application is presented in Fig. 2-29:

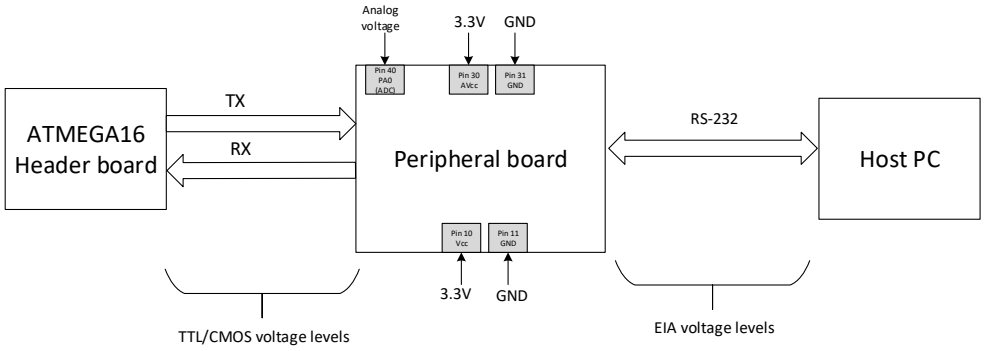

Fig. 2-29 ADC connection block diagram

The new connections added to Fig. 1-22, resulting Fig. 1-23, are the following:

- In order to use the ADC, the AVCC pin (pin 30) MUST be connected to the power supply, in our case to the 3.3V line. Same goes for the ground pin next to it, which, of course, must be connected to the ground line (pin 31 needs to be connected to ground)
- The analog input voltage should be connected to one of the ADC input channels, thus, for simplicity, ADC0 channel 0 was selected in the previous diagram. Practically, the PA0 pin (pin 40) will serve as ADC0 function and will be connected to an analog voltage

The analog voltage, which will be applied to the ADC, can be generated using an adjustable laboratory power supply. The ground of the laboratory power supply MUST be connected to the ground of the whole circuit. Another important aspect is that the analog voltage MUST NOT EXCEED 3.3 V. The interval, in this situation, should be [0,3.3] V. In order to verify the results, it is recommended that the laboratory adjustable power supply is able to display the actual voltage that it is applied. If not, then, a real voltmeter should be used in order to be sure that the 3.3V limit is not exceeded.

ASSIGNMENT 1: Make the necessary connections and have the laboratory supervisor verify them.

The next aspect, which should be discussed, is related to the software part of the application which should imply the configuration of the ADC of ATMEGA16, the starting of the conversion, the collection of the raw data result, the calculation of the actual voltage in mV and the printing of the result on the serial terminal.

ASSIGNMENT 2: Read the documentation regarding the ADC module of ATMEGA16 concentrating on the registers. Make a list with all the registers that should be used for configuring the ADC module. Furthermore, make separate lists of the registers used for starting a conversion, for waiting for the conversion to be finished and for collecting the result. Define the algorithm for both operations (configuration, conversion). The ADC should be used without interrupts, having completion checked by polling and with a resolution of 8 bits.

The ADC of the ATMEGA16 is not hard to use. It has a small number of registers and the operation is almost trivial. The first aspect to be considered is the configuration of the direction of the pins involved in ADC conversion. As stated before, the ADC has 8 channels for conversion and they are all mapped on PORTA of the microcontroller. In this case, the programmer has to be assured that the direction of the pins involved in conversion is set to input.

The configuration of the ADC is mainly done by writing the necessary bits in ADCSRA register which is the ADC Control and Status Registers. The most important bit that needs to be set, prior to any usage of the ADC, is the ADEN bit which enabled the ADC module. Other bits which interest us on this register are the ADPSx bits which form the clock division factor. As a starting point, a divisor of 8 should be used having the bits equaled to the following values: ADPS2 = 0, ADPS1 = 1, ADPS0 = 1. Practically, along with the direction configuration, these writings into the ADCSRA register should suffice. A flowchart of the function that implements the configuration of the ADC module may be found in Fig. 2-30:

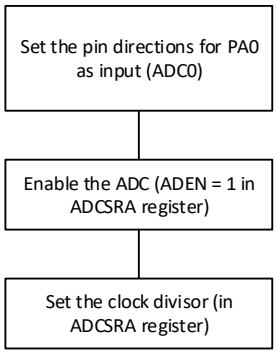

Fig. 2-30 ADC initialization flowchart

The slightly complicated part of the ADC is implementing a method to start the conversion, wait until the conversion is complete and collect the result. The first step is writing the ADMUX register by selecting the channel for conversion and selecting the voltage reference (AVCC in our case). Giving that we will use a resolution of 8 bits, having a result formed out of the most 8 significant bits of the resulted conversion (ignoring the least 2 significant bits) the Left Adjusted Result option for the conversion should be used. This can be configured by writing to logic 1 the ADLAR bit in ADMUX register.

After configuring the ADMUX register, the conversion can be started by writing ONLY the ADSC (ADC Start Conversion) bit in ADCSRA register. This bit should also be used for checking whether the conversion has finished. According to the documentation, after this bit is set to logic 1, the conversion begins. After the conversion is finished this bit is reset to logic 0, by hardware, signaling the completion of the conversion.

When the conversion is finished the only thing remaining is collecting the result. Giving the fact that we used the Left Adjusted Result option (by setting ADLAR = 1) the 8 bit result needed can be obtained by reading only the ADCH register (most significant part of the ADC data register). The flowchart for reading a sample from one of the ADC channels may be found in Fig. 2-30:

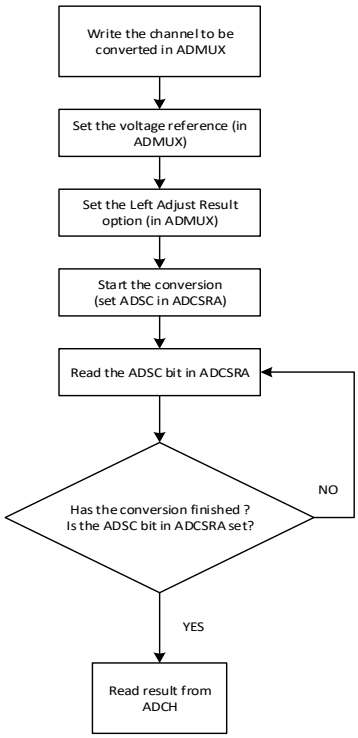

Fig. 2-31 ADC conversion logic

ASSIGNMENT 3: Modify the serial communication library developed in the previous laboratory work by adding an additional function that is able to send a C string (a NULL terminated character array) over the serial interface. Modify the main program in order to send "Hello World!" over the serial interface once per second. The function prototype should be:

Code listing 2-8 Send string function prototype

```
void SendString(char *string);
```

Being a NULL terminated string use the *strlen* function from string.h library to obtain the length of the string.

ASSIGNMENT 4: Write a C library (a c file with a header file) which contains an initialization function for the ADC mode and a function capable of starting the conversion and reading one raw sample value from the ADC on channel 0. Add the library to the project developed in the previous laboratory work. The project should now contain the following:

- Serial.c – the C file containing the implementation of the functions
- Serial.h - the Header file containing the declarations for the function implemented in serial.c file that need to be exported
- Adc.c – the C file containing the implementation of the ADC functions (init function and read data function)
- Adc.h – the header file containing the declarations for the functions implemented in adc.c file that need to be exported to the rest of the program
- Main.c – the C file containing the main program and function

The communication parameters should be the last used in the previous laboratory application (BAUD 115200 bps, 8 bit data, no parity, 1 stop bit)

The main program should mainly read one sample from the ADC, calculate the resulted voltage and print the result on a new line containing VOLTAGE = <value> mV. Make use of the standard *sprintf* function in order to print into a string which should be sent over the serial line using the required function in the serial library. The program should print the voltage once per second. The newline is obtained by inserting the characters '\r' and '\n'.

Do not use float or double when calculating the resulting voltage value. Use only integers. ATMEGA16 does not have hardware support for floating point values.

Use Docklight scripting, as in the previous laboratory application, to view the result.

The flow of the program should be the following:

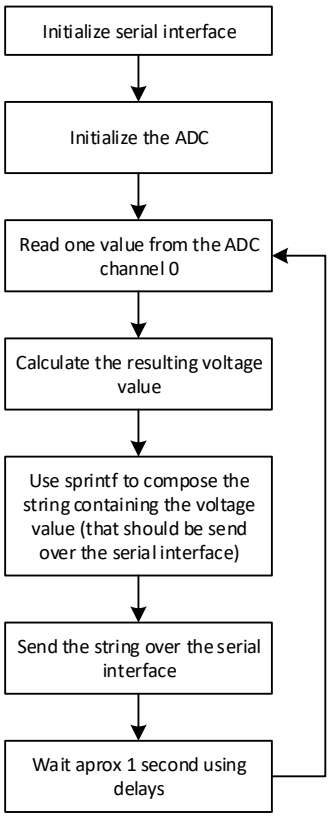

Fig. 2-32 Digital voltmeter program flow

2.5 Laboratory work 4 – Minimal 2 channel oscilloscope

The first product, close to a digital oscilloscope, will be outcome of this laboratory and may be even considered an important milestone. During this laboratory work a second channel will be added for conversion in the ADC library previously developed. Furthermore, the data collected from the ADC will be stored in buffers, one buffer for each channel. The buffers will be then sent over the serial interface to the PC to be processed. This implies the addition of new functions to the serial interface library and also the development of a short protocol to encapsulate the data. The display of the oscilloscope will be represented by a dedicated application on the PC which displays the data received from the serial interface. In general lines, these will be the aspects discussed within this laboratory work.

The first step into implementing the basics for our 2 channel oscilloscope is to first acquire signals on 2 channels (currently we have only one channel of the ADC working)

and then to store the gathered data into buffers. Before storing the data we need to enable the conversion on another channel of the ADC. This can be done simply by writing the channel number that will be converted into the ADMUX register. In the previous laboratory work the channel was hardcoded in the read data function (in ADMUX register). So, practically the only change would be to modify the read data function to receive the channel to be converted as parameter. The conversion on the 2 channels do not have to be simultaneous, thus only the modification of this function should suffice.

ASSIGNMENT 1: Modify the read data function in the ADC library developed in the previous laboratory work to support specifying which channel to be converted with the help of a dedicated parameter. An example prototype of this function should be:

Code listing 2-9 ADC read data function prototype with channel number as argument

```
uint8_t ADC_ReadData(uint8_t channel_number);
```

Also adapt the main program to print both samples on the Docklight Scripting serial terminal. Loose the calculation for the actual voltage and print the 2 acquired samples from the 2 channels as raw numbers, preferable in hexadecimal.

The next step is to store the gathered data from the channels into 2 separate buffers, one buffer for each channel. The 2 buffers should be equal in size. The size of the buffers should be configurable at compile time using a define directive. After the buffers are full, they should be sent over the serial interface to be received by a dedicated application on the PC which will display the signals as an oscilloscope screen would do. This PC application will be presented shortly.

In order for the PC application to be able to correctly use the data from the microcontroller, some transmission protocol needs to be used on top the UART protocol. The data that the microcontroller sends (the 2 buffers containing the samples from the 2 channels) needs to be encapsulated in a certain manner. The frame that encapsulates the data must have the following structure:

Description	synchronization pattern				buffer size		ch1_buffer	ch2_buffer	ch frequency	
size	<1>	<1>	<1>	<1>	<1>	<1>	<CNT>	<CNT>	<1>	<1>
value	0xA0	0xA3	0xB0	0xB3	CNT_HI	CNT_LO	ch1_buffer	ch2_buffer	F_LO	F_HI

Table 6 Oscilloscope to PC communication protocol

Let us have a look on the protocol encapsulation presented in Table 6. The frames described above are the only frames accepted by the PC viewing software. No exceptions are permitted.

The first field in the frame is represented by the four bytes forming the synchronization pattern. This pattern is fixed and it is used by the PC software to identify the beginning of a data frame. These 4 bytes must be used as they are. The following 2 bytes contain the buffer size split into a most significant byte and a least significant byte. The buffer size field defines how long are the buffers containing the acquire channel data. Each buffer has the size defined by the buffer size field. So, practically the amount of data

to be transmitted in total is twice the value of buffer size. After the buffer size, the actual data is transmitted: first the values for channel 1 (the number of the values is defined by the buffer size value) then the values for channel 2 (again the number of the values is defined by the buffer size value. After the data buffers are transmitted an additional 2 bytes are required containing the frequency of one of the channels that will be calculated later in another laboratory session. For now, consider these 2 byes as zero.

ASSIGNMENT 2: Modify the serial interface library developed in the previous laboratory works by adding a function that received as parameter a pointer to the data buffer of channel 1, a pointer to the data buffer of channel 2 and the size of the buffers which is able to encode the data as previously described and send it over the serial interface.

A possible prototype for this function could be:

Code listing 2-10 Data encode and send function prototype

```
void UART_SendOscData(uint8_t* channel_1_buffer, uint8_t* channel_2_buffer,
uint32_t buffer_size);
```

ASSIGNMENT 3: Implement a main program that using an infinite loop it gathers data from the ADC from both channels and stores the data in the corresponding buffer for each channel. The buffers must only contain raw data from the ADC without any transformations. The transformation as used in the voltmeter application should have already been eliminated. Both buffer must have the same size.

After the buffers are full, the program uses the serial interface library to encode and send the gathered data over the serial interface to be viewed on the PC software. The flow of the program should be as presented in Fig. 2-33:

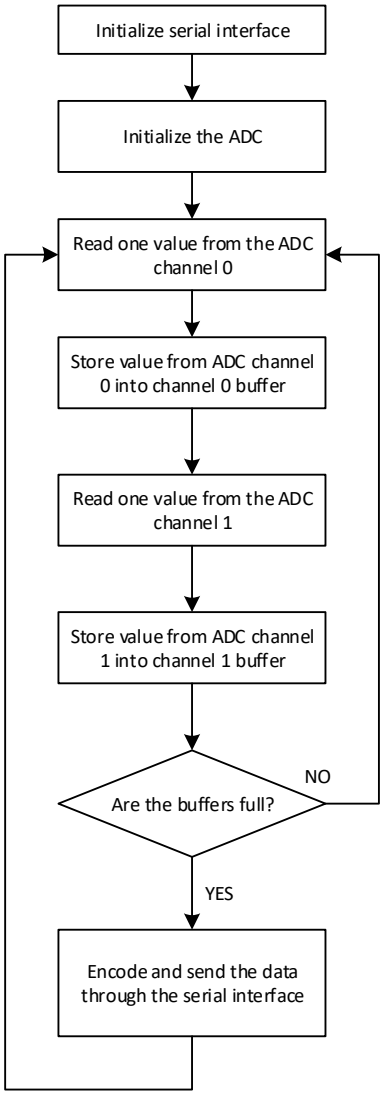

Fig. 2-33 Simple 2 channel oscilloscope program flowchart

In order to test the newly developed oscilloscope the PC software that will be used as a display and control panel for the oscilloscope is needed. The PC software, called APND, is a simple, one window .NET application capable in both displaying the signal wave forms transmitted by the microcontroller over the serial interface, but can also be used to send 2 byte long words over the serial interface to the microcontroller. This latter feature will be used in the following laboratory works.

Our current focus is on using the APND software to display the signal wave forms. In order for the APND software to display the waveforms, the data must be sent in the correct protocol encapsulation as presented above.

The main window of the application is presented in the following figure:

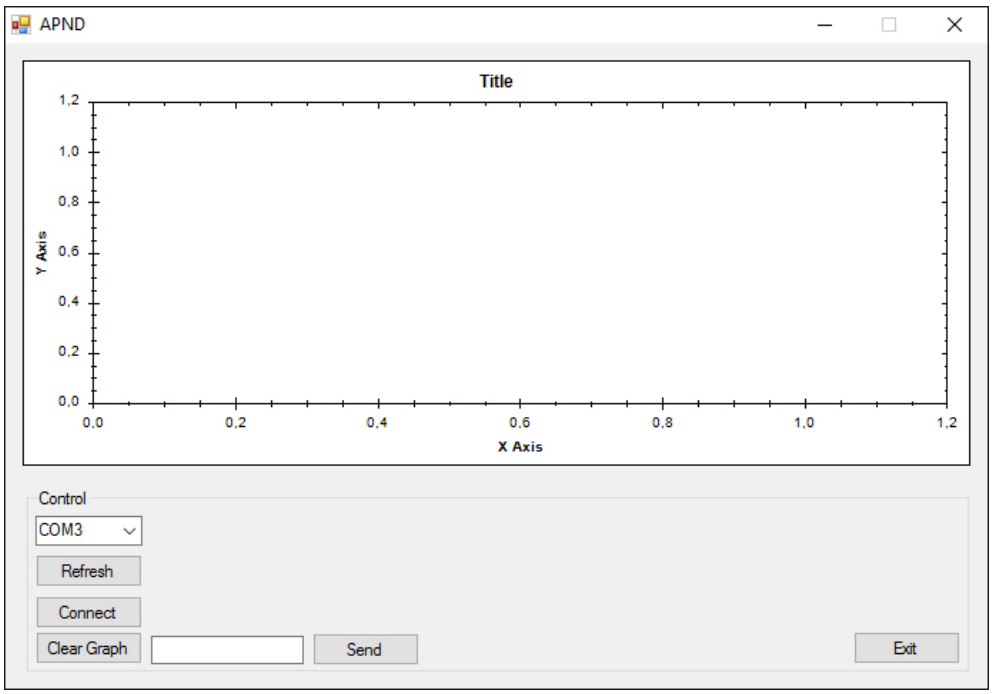

Fig. 2-34 APND software main window

As it can be seen in Fig. 2-34 the APND software is quite simple. The COM which will be connected to the embedded system, our microcontroller, can be selected from the combo-box. The COM port list can be refreshed by using the Refresh button. After the correct COM port has been selected, the Connect button should be pressed in order to start the communication between the microcontroller and the APND software. If the correct frames are received from the microcontroller the waveform will be displayed in the graph as presented in Fig. 2-35

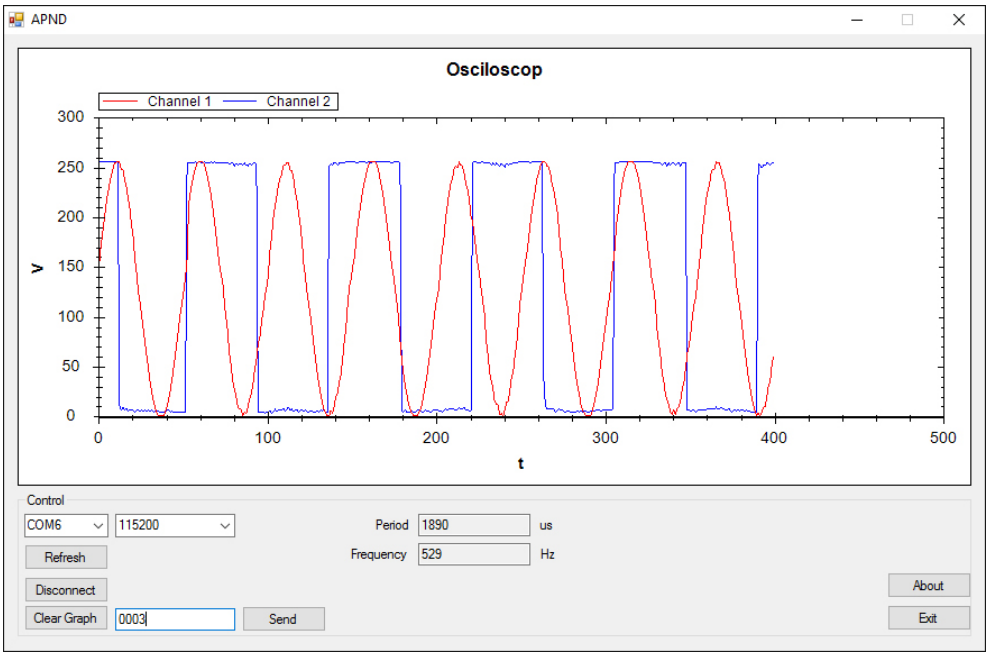

Fig. 2-35 APND software main window with waveforms

2.6 Laboratory work 5 – Oscilloscope trigger

In the previous laboratory work the first basic implementation of the oscilloscope was made. The basic oscilloscope, in this moment, is only able to acquire the signal and display it on the PC applications without any processing. The first feature of the oscilloscope that should be implemented is the trigger sequence. The main role of the trigger is to synchronize a periodical signal on the display of the oscilloscope. Without the trigger, the signal on the oscilloscope has a "moving" behavior. The idea of the trigger is that the oscilloscope always begins to display the signal from the same sample, thus making it stable on the display.

On a real oscilloscope the trigger is available only on one channel at a given moment in time, but it can be configurable into being on any of the channels. Furthermore, the trigger may be set on the positive edge of the signal or on the negative edge. The most important characteristic is that the trigger is able to synchronize only periodical signals.

The actual coding that will be done for this laboratory work is more than simple. No more than 2 or 4 lines of code have to be written. The actual aspect of this laboratory work is to present the methodology and analysis on how the signal synchronization is made, thus how to deduct the code lines to be written and where.

Let us start by explaining the data displayed by the PC application. Such a waveform is presented in Fig. 2-36.

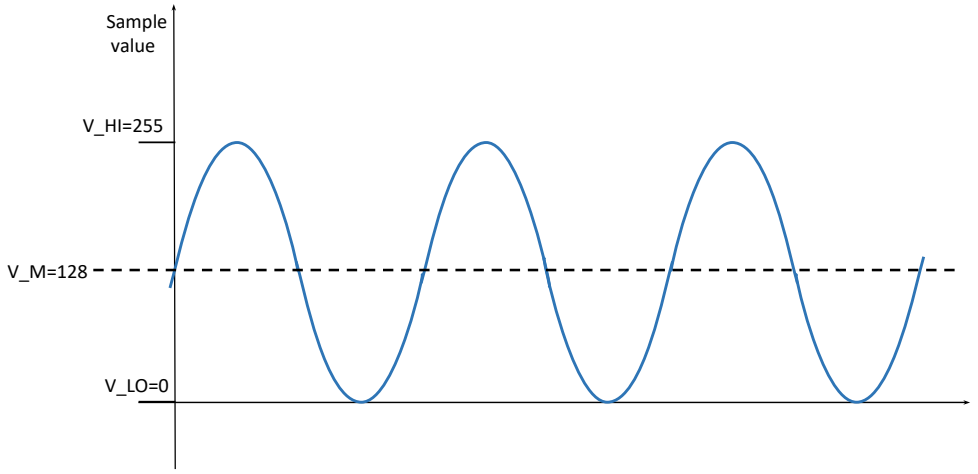

Fig. 2-36 Simple waveform

In Fig. 2-36, we can observe that the y axis is named “Sample value”. On this axis the actual values of the samples are noted, not transformed into volts. This is done in order to simplify all the explanations and to make things easier to observe. The minimum values is denoted at V_LO which is 0 and the maximum value is denoted as V_HI with a value of 255, being the maximum value of a sample represented as an 8 bit unsigned integer. Another notation is made on the x axis which is the middle value of the sample denoted as V_M with the sample value of 128. The x axis is a discrete time axis.

An important observation is that in the moment the microcontroller begins to acquire the first sample of data which is saved into the buffer, the actual position of the signal is unknown. In this moment we have no information from what point the acquisition of the signal begins: the acquisition may begin on the rising edge of the signal or on the falling edge. This is practically the reason why the signal is not synced: because the starting point of the acquisition is different at each startup of the buffer that stores the samples of the signal.

In order to realize the synchronization we have to make sure that the acquisition and storing into the buffer begins from the same position of the waveform, more exactly from the same sample on the same edge. Having this in mind, if, for example, the buffer always starts with sample of value 128 on the rising edge, them every time the buffer is displayed it is perfectly synced on the display of the oscilloscope (the graph of the APND software). Same should go for the falling edge. We can state that the triggering procedure is depended on the edge, falling or rising, as it can be seen on any real oscilloscope available on the market. In the following paragraphs, the signal synchronization will be analyzed considering the 2 situations: falling edge and rising edge.

The first case that will be analyzed is the case where we want to use synchronization on the falling edge. This actually means that the acquisition and storing will begin from the same sample value on the falling edge. This results in the fact that the signal will be displayed starting from a falling edge with the same starting point (value). The explanations are much easier by taking a look on the following figure:

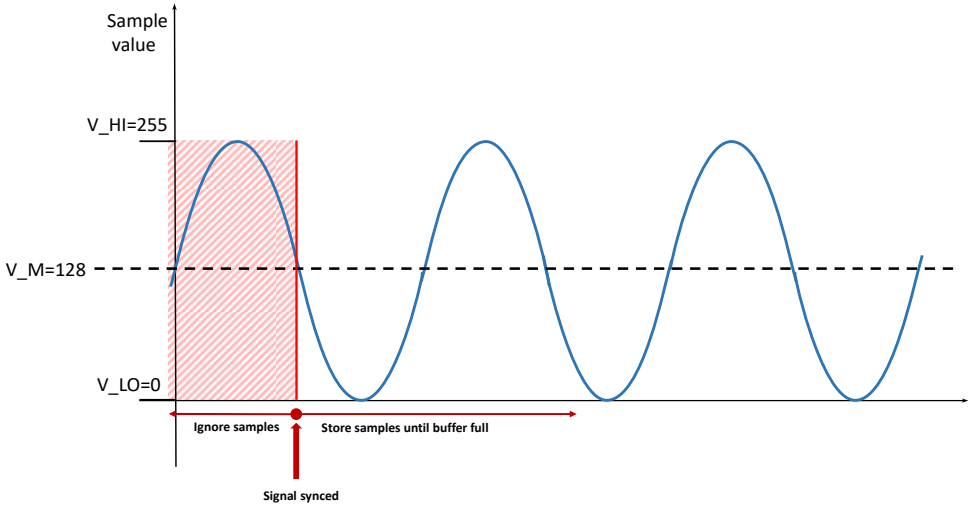

Fig. 2-37 Negative edge trigger using middle value

In each analysis we must consider the fact that in the moment we want to start storing the data in the buffer we do not know where the acquisition starts in respect to the waveform. The acquisition may start anywhere. Having this in mind, we must practically ignore all the samples until we obtain the sample presented in Fig. 2-37 and denoted with the red dot as "Signal synced". The ignored samples are the samples contained in the red hash of the signal. Starting the acquisition at this point and more important starting to store the data into the buffer from this point, after the buffers are full and the data is sent to the APND software, the signal will always be displayed from the same point and thus it will be synchronized on falling edge.

The question to arise is how to practically ignore those samples. First, giving the fact that we don't know the starting point of the signal in the moment of acquisition we must first reach our point 0, the origin of our axis. The value that we will use for comparison is the middle value, V_M. So, in the first step, we will ignore all values of the samples that are lower than V_M. Having these values ignored, we will find ourselves on the situation where we may be practically in the origin prior to the start of the rising edge. Our primary objective is the synchronization of the signal on the falling edge. In this case the next group of samples to be ignored are the samples which are higher than the value of V_M. After all these samples are ignored, we can begin the acquisition and storing the data into the buffer and we can observe that we have a synchronized signal on the display. A pseudo-code representation of the algorithm can be found in the following code snippet:

Code listing 2-11 Pseudo-code for negative trigger

```
while (current sample is below V_M)
{
  ignore current sample;
  get next sample;
}
while (current sample is above V_M)
{
  ignore current sample;
  get next sample;
}

while (buffers_are_not_full)
{
  get samples;
  store sample;
}

SendBuffersOverSerialInterface;
```

The same algorithm goes for the trigger on the rising edge of the signal. Having the same approach, we will use the next figure:

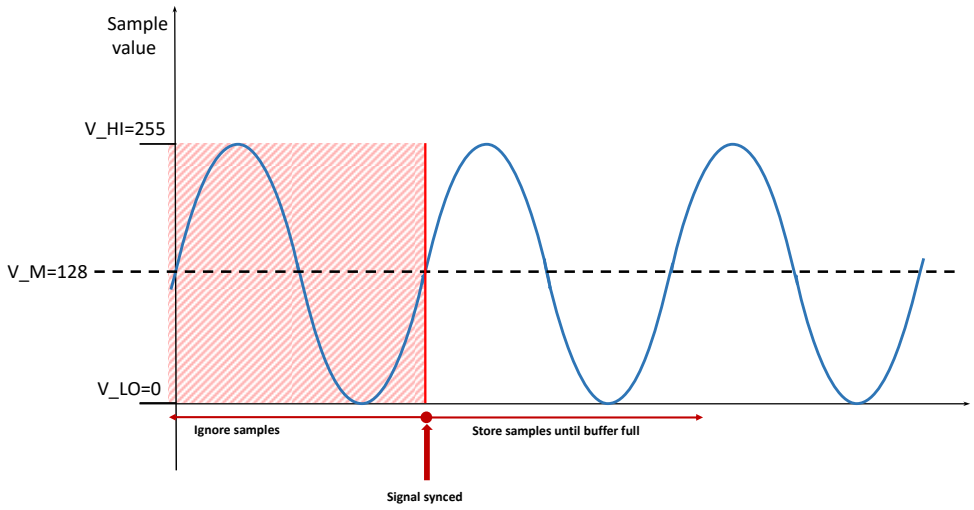

Fig. 2-38 Positive edge trigger using middle value

Practically in order to achieve positive triggering we can use the same algorithm. The approach though is slightly different. First we must assure that we have reached the negative synchronization then ignore half a period of samples in order to achieve the positive sync. More exactly, we first have to ignore all the samples which are above the middle V_M sample value. Then, the next step is to ignore the sample which are below the middle V_M sample value. After this, we can be assured that we have an exact starting point, which in the figure above is represented by the red dot and arrow pointing to it, denoted as "Signal Synced". The next and final step is to start the actual acquisition and storing, the same way we did in the previous situation. The pseudo-code algorithm for the positive triggering is almost identical to the prior situation:

Code listing 2-12 Pseudo-code for positive trigger

```
while (current sample is above V_M)
{
 ignore current sample;
 get next sample;
}
while (current sample is below V_M)
{
 ignore current sample;
 get next sample;
}
while (buffers_are_not_full)
{
 get samples; store sample;
}
SendBuffersOverSerialInterface;
```

Having an analysis over Code listing 2-11 and Code listing 2-12 we can identify that the only difference is that the first 2 while statements are switched between themselves. Practically, behind all the explanations this is the only noticeable difference.

In theory, the algorithm presented above is applicable and it should be enough. In practice though certain adapting is needed. The main reason is the noise which influences the ADC and also the errors the ADC conversion itself. The result of this errors and influences is that sometimes the middle value is not stable and because of this the triggering may change without reason. Furthermore, it is possible that one implemented a positive trigger but on the display a negative triggered signal is displayed. All of this is because the instability of the middle value. This can be corrected by using a middle interval instead of a middle value, an interval that, of course, contains the middle value. The middle interval can be considered an instability interval and may be considered similar to the transition interval from TTL circuits (the interval in volts where the circuit is not sure about the logical value). This situation is presented in the following figure:

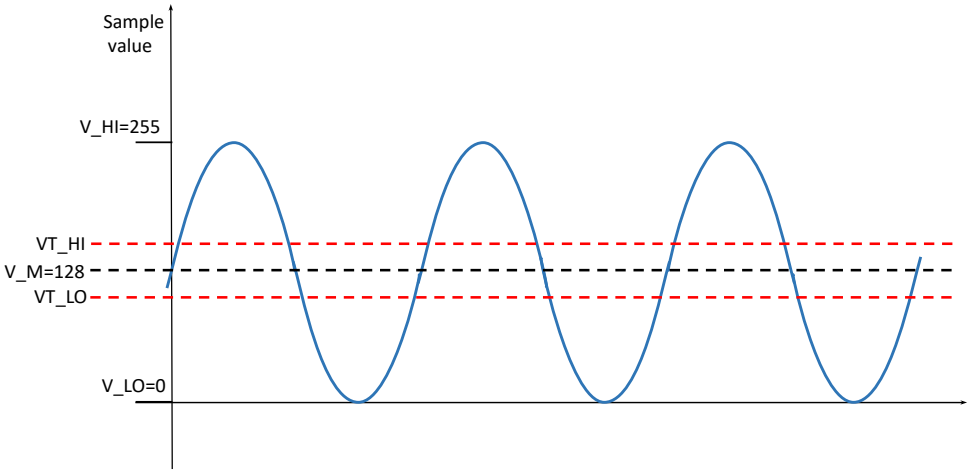

Fig. 2-39 Simple waveform with instability interval

As presented Fig. 2-39 the instability interval is defined by [VT_LO; VT_HI]. The V_M middle value has to be contained by the interval. The actual values of the interval margins cannot be defined theoretically. They are usually hardware and noise depended so practically they can only be found empirically. In order to obtain good result in practice we need to apply the instability interval to the situations described in previous paragraphs, more exactly to adapt the situations in Fig. 2-37 and Fig. 2-38.

Applying the interval on the first situation, the negative triggering, the resulted figure will be:

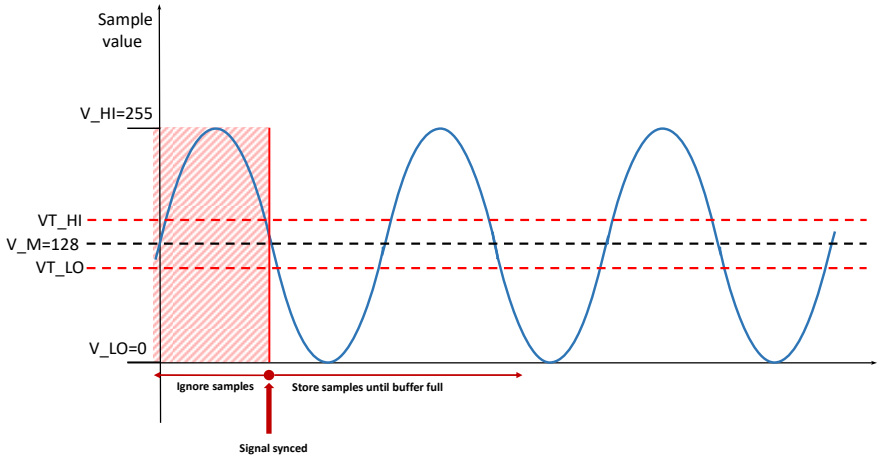

Fig. 2-40 Negative edge trigger using instability interval

The algorithm also needs to be adapted. The adaptation is made by practically replacing the V_M middle value with the instability interval. In our first approach, the first step was to ignore all the values that are lower than the V_M middle value. This will

be adapted into ignoring all the values that are lower than VT_HI value. The next step that was to ignore all the values that are higher than the value of V_M will be translated into ignore all the values that are higher than VT_LO. Using this replacements the algorithm may be adapted as in the following paragraph.

First, giving the fact that we don't know the exact starting point of the signal at the moment of the acquisition, we must first reach our point, the origin of our axis in this case. The values that will be used for comparison are the margins of the instability interval, VT_LO and VT_HI. In the first step, we will ignore all the values of the samples that are lower than VT_HI. Having these values ignored we will find ourselves on the situation where we may be practically in the origin prior to the start of the rising edge. Our primary objective is the synchronization of the signal on the falling edge. In this case the next group of samples to be ignored are the ones which are higher than the VT_LO values. After all these samples are ignored the negative trigger is accomplished.

The adapted pseudo-code for negative triggering can be found in the following code snippet:

Code listing 2-13 Pseudo-code for negative trigger with instability interval

```
while (current sample is below VT_HI)
{
  ignore current sample;
  get next sample;
}
while (current sample is above VT_LO)
{
  ignore current sample;
  get next sample;
}
while (buffers_are_not_full)
{
  get samples; store sample;
}
SendBuffersOverSerialInterface;
```

Same idea goes for applying the instability interval for the positive edge triggering. All the comparisons with the fixed V_M middle values will be replaced with the instability interval as presented in the following diagram:

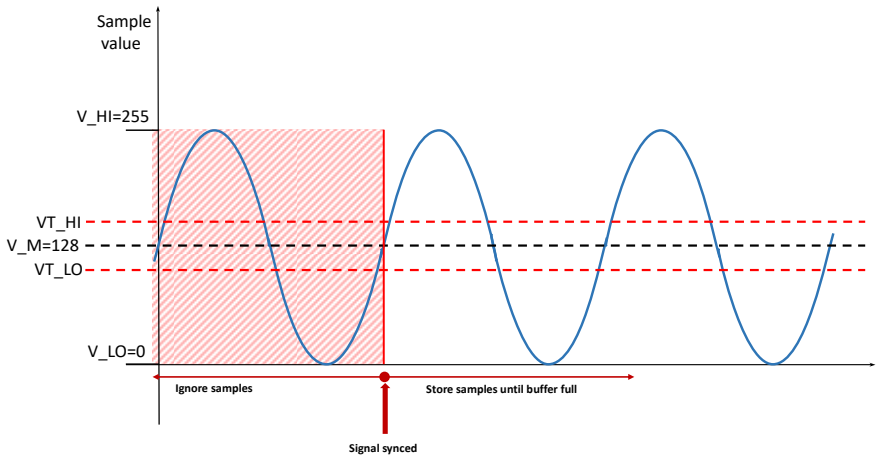

Fig. 2-41 Positive edge trigger using instability interval

The approach is almost the same. Firstly, we must assure we have reached the negative synchronization, then ignore half a period of samples in order to achieve the positive sync. More exactly, we first have to ignore all the samples which are above the value represented by VT_LO. Then, the next step is to ignore the sample which are below the value represented by VT_HI. After this, we can be assured that we have an exact starting point which in the figure above is represented by the red dot and arrow pointing to it, denoted as "Signal Synced". The next and final step is to start the actual acquisition and storing, the same way we did in the previous situation. The adapted pseudo-code could look similar to the following code snippet:

Code listing 2-14 Pseudo-code for positive trigger with instability interval

```
while (current sample is above VT_LO)
{
  ignore current sample;
  get next sample;
}
while (current sample is below VT_HI)
{
  ignore current sample;
  get next sample;
}
while (buffers_are_not_full)
{
  get samples;
  store sample;
}
SendBuffersOverSerialInterface;
```

An important observation is that, the triggering on any real oscilloscope is only available on one channel at the same time, from obvious reasons, but may be changed from one channel to another. The same approach will followed here. The trigger can only

be applied on one of the channels but features for changing the channel for triggering should be considered. The algorithm is the same for any of the channels just the proper samples should be used (for channel 1: triggering only samples from channel 1 should be used for comparison; for channel 2: triggering only samples from channel2 should be used for comparison).

The adapted version of the oscilloscope program presented in Fig. 2-33 with the triggering on negative edge could be the following:

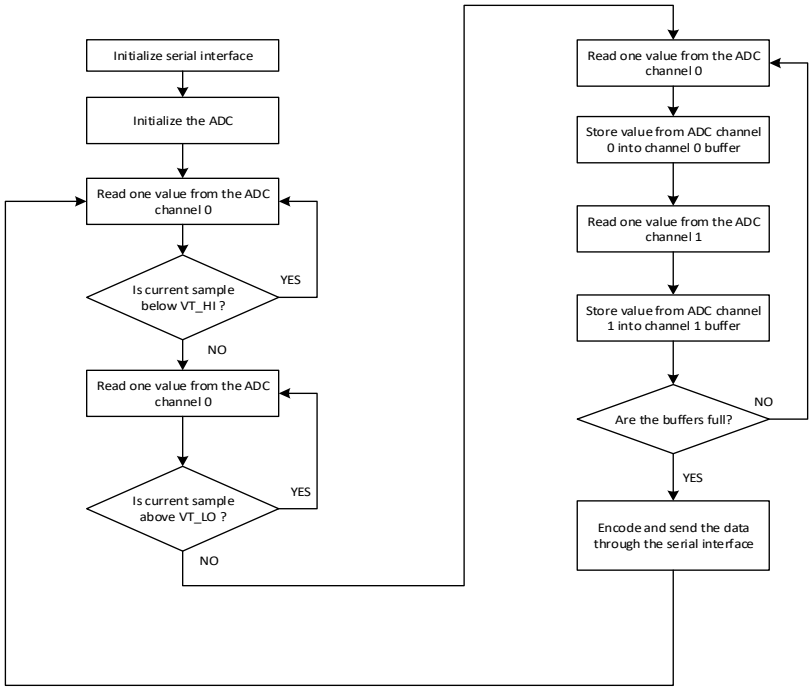

Fig. 2-42 A 2 channel oscilloscope program flowchart with negative edge trigger using instability interval

The same algorithm and approach used for positive edge triggering could have as result the adapted version of the program flow presented in Fig. 2-33 as presented in the following diagram:

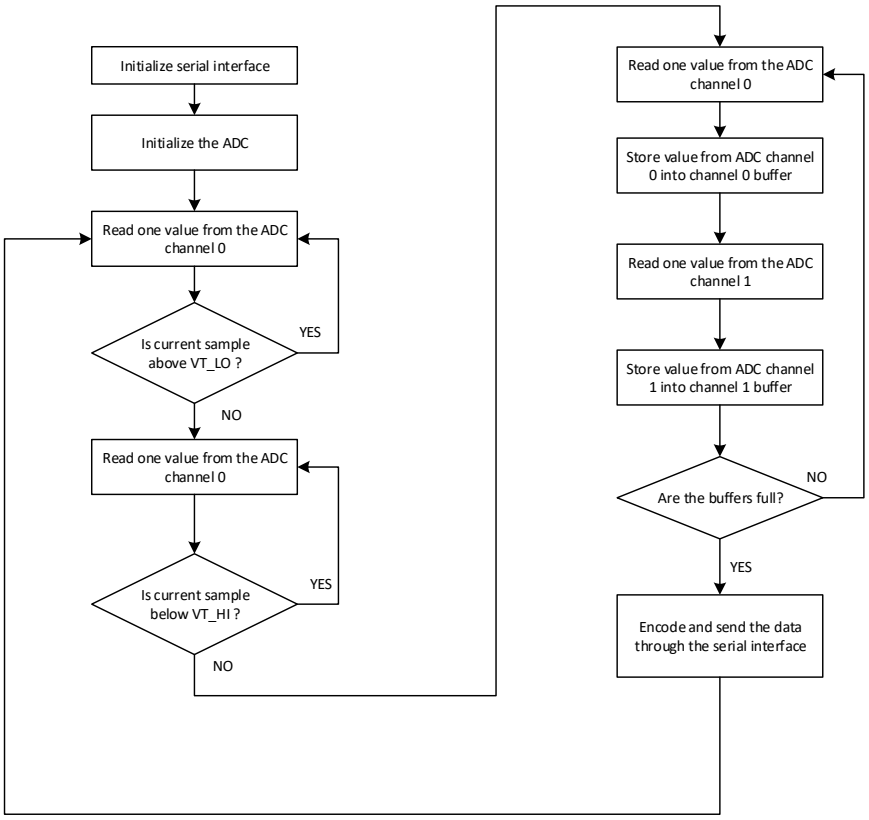

Fig. 2-43 A 2 channel oscilloscope program flowchart with positive edge trigger using instability interval

The only aspects left to be discussed is the choosing of the margins of the instability interval. As stated before these values have to contain the V_M middle value and are chosen empirically. A starting point could be to use the following values:

Table 7 Starting point values for instability interval margins

VT_HI	140
VT_LO	110

ASSIGNMENT 1: Implement triggering on the already existing code for the simple 2 channel oscilloscope. Implement both positive and negative triggering. The positive/negative triggering should be chosen at compile time based on a define statement. Use only a define statement! Do not use code exclusion statements (based on ifdef, ifndef, endif statements)! Furthermore, using defines statements, give the possibility of choosing the channel on which the synchronization should be made.

OPTIONAL ASSIGNMENT: Push the limits of your oscilloscope! Find the limits! Use the oscilloscope triggered on one of the channels and find the maximum frequency that it can display without deforming the waveform. Try to the change the input frequency of the ADC by changing the divisor and find out if better results may be obtained

HOME ASSIGNMENT: Read about the TIMER module of ATMEGA16 focusing on 16 bit TIMER1. Concentrate only on the free running capability of the timer. Make a short list with the registers and bits that should be used.

2.7 Laboratory work 6 – One channel frequency calculation

The old style oscilloscopes were basically able only to display 2 signals and make the trigger on one of them, of course, along with the changing of the time base. The new generation of oscilloscopes have measuring capabilities. One of the most important measuring capability is the measurement of the frequency of the signal. This aspect will be treated in this laboratory work.

Our current state of the oscilloscope is that it is capable of 2 channels waveform display with triggering on either of the channels and either on positive and negative edge. The next connected application should be to extend the features of the oscilloscope into having the measurement of the frequency embedded into the project. The frequency measurement should only be considered for the channel that is used for triggering. A different approach is much more difficult. Measuring the frequency of the signal is translated into measuring the period of the signal. The period T of the signal is depicted in the following figure:

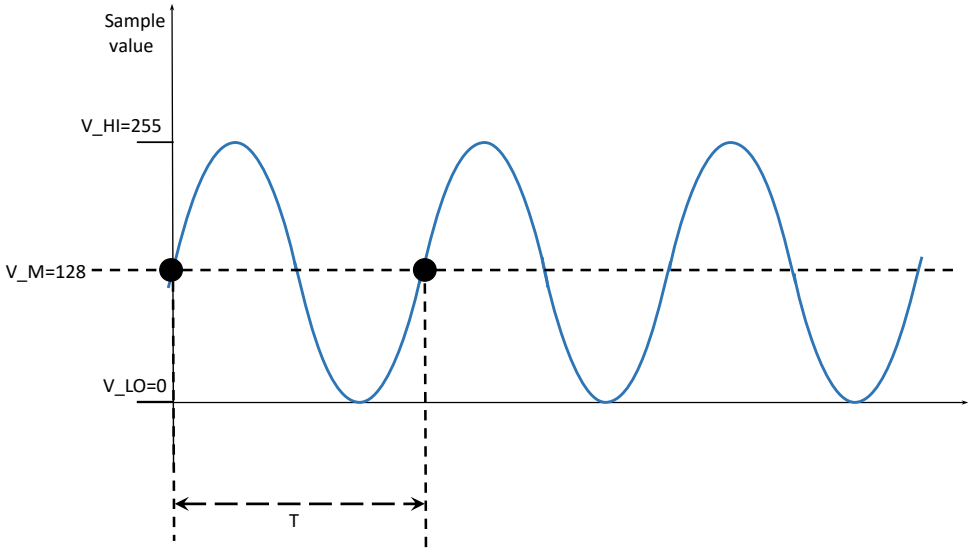

Fig. 2-44 Period definition

In order to accomplish this task another peripheral device of ATMEGA16 must be used: the TIMER module. In order to have the highest precision the TIMER1 peripheral module of ATMEGA16 will be used. The TIMER1 module has the special feature that it is 16 bit wide.

The timers of ATMEGA16 have a lot of operating modes: compare match, capture, PWM and of course free run. Giving the fact that we only want to measure software events the free running operating mode will be used. This is the simplest mode of using the timer, thus it is used only at the level of a counter. In order to start the TIMER module of ATMEGA16 the clock source needs to be specified. A TIMER of ATMEGA16 once it has its clock source defined it starts counting. The counting is stopped only when the clock source it is disabled.

In order to have the best precision, the TIMER1 module will be used, which is a 16 bit wide TIMER. The rest of the timers of ATMEGA16 are only 8 bit which are not enough for our measurements. Giving the fact that all the peripheral modules of ATMEGA16 are designed to be working 8 bit wide, in order to have 16 bit peripherals, the word is divided, usually, into two 8 bits registers, with a most significant byte and a least significant byte. Same goes for the TIMER1 module. So, the timer counter register, which contains the counting value, is also divided into two 8 bit wide registers.
A two read operation will be required in order to obtain the full value.

Practically, in order to obtain the current value of the timer counter the TCNT1H and TCNT1L should be used. The reset of the timer counter implies to set the TCNT1H and TCNT1L to zero. There is restriction that should be taken into account: when reading the timer counter registers the TCNT1L must be read first and TCNT1H last. This observation may be found in the documentation of ATMEGA16.

ASSIGNMENT 1: Implement a timer library that provides an initialization function for the timer in free running mode which resets and starts the timer, a function that stops the timer and a function that returns the current value of the timer counter. The possible function prototypes could be the following:

Code listing 2-15 Timer library function prototypes

```
void start_timer(void);
void stop_timer(void);
uint16_t get_timer_value(void);
```

The library should have the following files (as the previous developed libraries have):

- Timer.h – a header file containing only the function prototypes
- Timer.c – a C code file containing the implementation of the functions defined in the header file.

The measurement of the period/frequency of the signal is dependent on what kind of triggering is used: positive or negative. Moreover, the measurement will only work on the signal that is triggered.

A real oscilloscope is capable in measuring the frequency of a signal even if it is not the one the triggering is made on. The algorithms for this kind of measurement are complicated thus this situation will not be considered.

The triggering edge is necessary to be known when measuring the frequency of the signal because the method that will be used is based on the signal crossing the edges of the instability interval defined in the previous laboratory work. We will analyze both situations.

The first situation is represented by the measuring of the frequency of the signal triggered on its negative edge. The discussion will be based on Fig. 2-45:

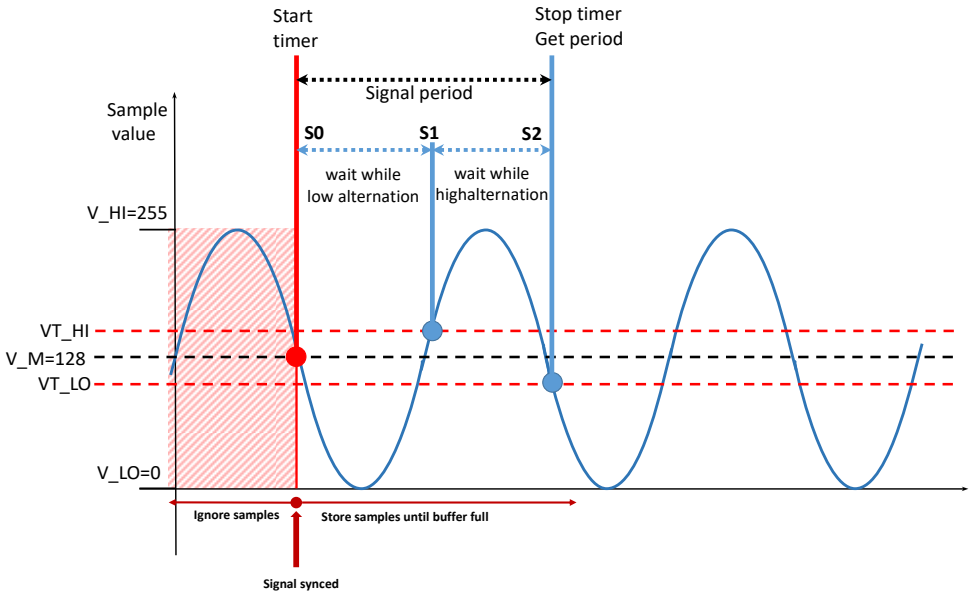

Fig. 2-45 Frequency measurement on negative edge triggering

These easiest method for measuring the frequency of the signal is add a simple finite state machine into the oscilloscope program developed so far. In Fig. 2-45 three states can be distinguished. State S0 is practically in the moment the signal has obtained the trigger, more exact in the moment the actual acquisition and storing of the signal begins. The idea, from this point on, is to let the timer run until a complete period of the signal has passed, this being the moment the timer should be then stopped and the counter value read in order to make the necessary calculations.

The obvious question that arises is how to determine that a complete period the signal has passed? Now that we have the starting point of the period of the signal we can obtain the whole period of the signal first by detecting the negative alternation and then determining the high alternation. The low alternation is determined by waiting while the signal is below VT_HI. The determination of the low alternation of the signal is the moment when switching from state S0 to state S1. In finite state machine will remain in state S1 until the high alternation of the signal has passed. This is found by waiting until the signal is below VT_LO value. When this condition is satisfied the finite state machine will switch to state S2. This state is responsible for the stopping of the TIMER and also for the reading of the actual counting value of the timer. The last operation that should be

made is to calculate the period of the signal by using the newly obtained counter value divided by the frequency of the timer (which is the same as the microcontroller's if no pre-scaler is applied).

Same algorithm, with minimal changes, may be applied in order to calculate the frequency of the triggered signal synchronized on positive edge as presented in the figure:

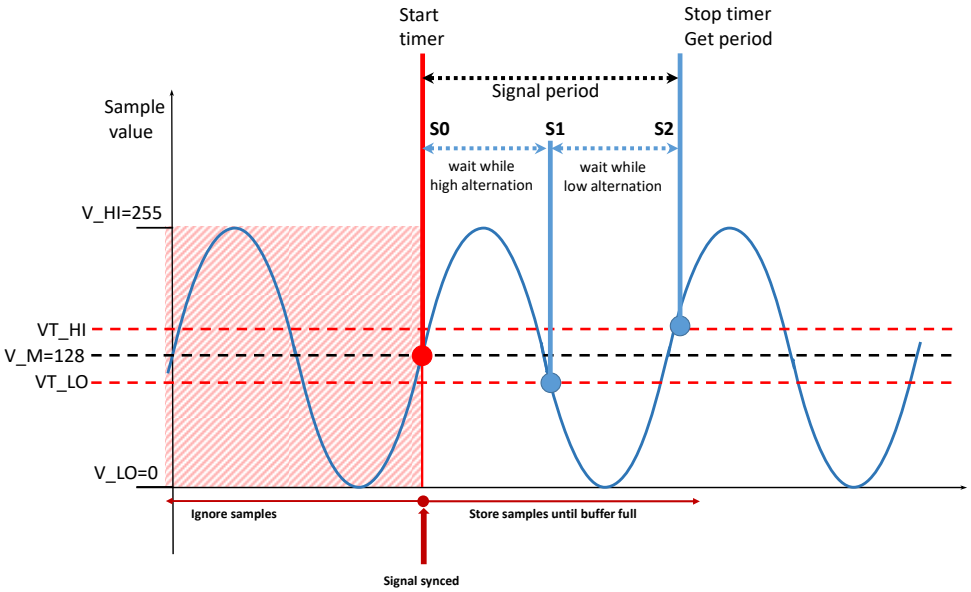

Fig. 2-46 Frequency measurement on positive edge triggering

As in the previous case, the starting point, state S0, is represented by the moment we have the signal synchronized, more exact, the moment the acquisition and storing of the signal begins. Same as in the previous situation state S0 marks the start of the timer. The following period of time, before switching to state S1, is represented by the fact that we must wait for the positive alternation of the signal. Practically we must wait until the signal is above VT_LO. The moment the signal goes below VT_LO the finite state machine can switch to S1 state. The next period of time covers the waiting for the negative alternation of the signal which is represented by waiting until the signal is below VT_HI. Immediately after the signal goes above VT_HI the state machine may switch to state S2. This state is again the final state which is responsible for stopping the timer, retrieving the value of the counter and the calculation of the actual period by dividing it by the frequency of the timer.

Giving the fact that ATMEGA16 does not have floating point support, it is best to make the necessary calculations as integers. In order to do this, the value of the timer counter can be multiplied by 1000 and the value of the frequency in hertz may be divided by 1000 thus obtaining the value of the period of the signal in microseconds as presented in the following formula.

$$SignalPeriod[\mu s] = \frac{(TimerCounterValue \cdot 1000)}{F\_CPU/1000} \quad (2\text{-}8)$$

The final step is to encode the calculated period into the protocol used to send the date to the PC application. As presented in Laboratory work 4 – Minimal 2 channel oscilloscope, the protocol has 2 bytes for the channel period at the end of the frame. The expected format should be a 2 byte values encoded as most significant byte first. Therefore, a limitation is worth mentioning: the calculated signal period must not exceed value 65535, the maximum value represented by a 2 byte integer. Therefore, the maximum period of a signal would be 65535 us – 65.5 ms.

ASSIGNMENT 2: Implement signal measurement for both channels and on both positive/negative trigger situations. Differentiate the cases with preprocessor define statement. The situation should be chosen at compile time. Add the calculated value of the signal period to the communication protocol and visualize the value on the APND PC application.

HOME ASSIGNMENT: Read again the documentation referring to the UART serial communication focusing on the reception. Write a function capable on waiting for a byte on the serial interface and returning it after it has been received. Furthermore, find out on how the reception can be used as interrupt source. The final goal is to implement the UART serial reception using interrupts.

2.8 Laboratory work 7 – Serial communication - reception

This laboratory work is the second laboratory work which concentrates on UART serial communication. This laboratory work is also practically isolated from the previous assignments. The application that will be implemented is a serial echo, an application that waits for a byte to arrive on the serial line and sends it back afterwards. There will be 2 approaches: one to implement a blocking receiver function that waits for a byte and returns it after arrival and another approach to use interrupts to be notified on the arrival of one byte over the serial interface. The latter method will be than integrated into the oscilloscope application during the next laboratory work.

The idea of this lab work is to extend the serial library with receiving possibilities first by using a blocking polling method and then using the interrupt system.

In Laboratory work 2 – Serial communication - transmission we had configured the UART module of ATMEGA16 at a BAUD of 115200 (as the final version), 8 bit wide data word, no parity and 1 stop bit. Furthermore, the library supports the transmission of both raw bytes and strings. The library also contains the necessary functions for encoding and sending the data frame recognized by the APND PC software. This library will be used as a starting point and it will be completed with reception. The initialization of the UART module in this library was designed to also enable the receiver. This can be verified

by checking that the RXEN bit in UCSRB register is set to logic 1. So, practically all the initializations have been made. The only operation left to be implemented is the actual reception of one byte over the serial line.

The function that is responsible for the serial reception should be blocking, thus waiting for a byte to arrive over the serial line into the receive buffer. This information can be extracted from the RXC (Receive Complete) bit in the UCSRA register. This bit is set to value logic 1 when unread data is available into the receive buffer. For the rest of the time this bit is set to logic 0. Using this information we can state that this is the most important status bit when implementing the reception. Having this into consideration, the reception function flow chart diagram may be the following:

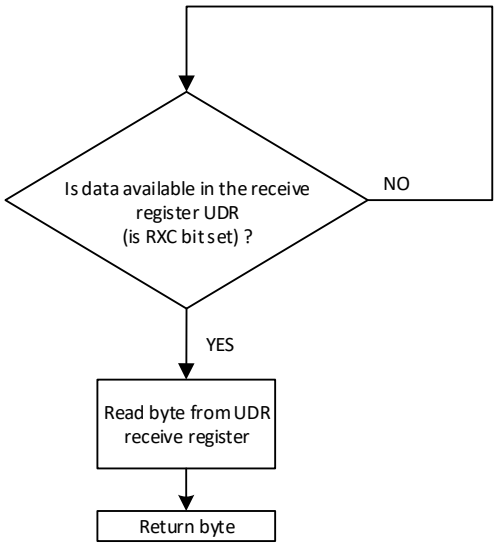

Fig. 2-47 UART receive flowchart

It is important to mention, in this case, that the register named UDR is used for transmitting data over the serial line (transmit register) but it is also used for receiving the data that has arrived over the serial line (receive register). It is important for the attendees to distinguish these aspects. Even though it has the same name, there are practically 2 registers which are accessed by the same name. The difference is the access method. When reading the register named UDR, the *receive* register is actually accessed. When writing the register named UDR, the *transmit* register is actually accessed. This is normally handled by hardware.

The actual function that should implement the flow chart described in Fig. 2-47 should have the following prototype:

Code listing 2-16 UART receive byte function prototype

```
uint8_t UART_ReceiveByte(void);
```

It is again important to mention that the receive function, as presented in Fig. 2-47 and in Code listing 2-16, is blocking until a character is received, which means that the execution of the microcontroller is halted until a character (byte) is received over the serial line.

ASSIGNMENT 1: Implement the function responsible for receiving a byte over the UART interface according to the explanations above. Add to function into the already developed serial library.

The easiest method to test this function is to implement a trivial serial echo program. This program practically waits for a character to be received over the serial interface and after it has arrived it is sent back over the serial interface, thus the name "echo" is justified. The actual main loop of the microcontroller should implemented according to the diagram:

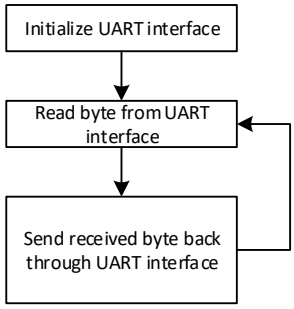

Fig. 2-48 UART echo main program loop

ASSIGNMENT 2: Implement the UART echo main program used to test the receiver function. In order to test this echo program open the Docklight Scripting software, connect to the serial port the microcontroller is connected to and send character over the serial line. In order for the transmission to be enabled in Docklight Scripting, the "Keyboard Console On" button needs to be pressed or the Tools -> Keyboard Console On menu needs to be accessed. The shortcut key for this operation is CTRL+F5. After the transmission is enabled you can write the characters into the main window using your keyboard. A successful test of the program and receive function is when a transmitted character is returned back to the Docklight Scripting software.

The next aspect to be discussed in this laboratory work is how to transform the echo application to work using interrupts. The transmission will be left the way it is. The

reception is the actual operation that is most blocking and must be transformed in order to be interrupt based.

An interrupt is a procedure which interrupts the normal program execution of the microcontroller in order to serve an external event of the microcontroller's core. The serving is done by "calling" an interrupt service routine where the necessary code must be found in order to treat the event. The call of the interrupt service routine is NOT done by the programmer, but it is done by the core itself. The interrupt service routine is practically a normal function that is called by the core when an external event occurs. After the interrupt service routine finishes its execution the program returns to its code that was executed before the interrupt service routine was called.

It is important to mention that the normal code being executed by the microcontroller's core has absolutely no knowledge that it was interrupted. Most microcontrollers have the possibility to use more than one interrupt service routine. Usually each interrupt source may have its own interrupt service routine. An interrupt source may be a timer module, an SPI interface or, in our case, the UART interface.

The programmer may define the interrupt service routine as a normal function but the actual declaration is compiler dependent. This approach where there are more interrupt service routines, one for each interrupt source, and where the programmer may define these routines separately in code for each interrupt source is called vectored interrupts.

In the case of ATMEGA16 and the compiler we are going to use (avr-gcc) an interrupt service routine may be declared in any code file (not in a header file) using the following syntax:

Code listing 2-17 ATMEGA16 interrupt service routine declaration syntax in Atmel Studio 7

```
ISR(INTERRUPT_VECTOR_NAME_vect)
{
  // some code
}
```

The "INTERRUPT_VECTOR_NAME_vect" parameter must be the actual interrupt vector name defined in the register definition header file (io.h in general or, in our case, iom16.h).

OPTIONAL ASSIGNMENT: Open the iom16.h header file and search the appropriate interrupt vector name to be used for the UART reception interrupt.

The interrupt vector that we will use in this laboratory work is the one related to the reception of a character over the serial UART interface. Searching the iom16.h could lead to vector responsible for the UART reception interrupt: USART_RXC_vect. Having the name of the interrupt vector, the actual definition of the interrupt service routine may be the following:

Code listing 2-18 ATMEGA16 interrupt service routine for UART reception interrupt

```
ISR(USART_RXC_vect)
{
  // some code
}
```

The above syntax is newly introduces by Atmel Studio replacing the old declaration using the *SIGNAL* keyword. The *SIGNAL* mode of declaring an interrupt is currently deprecated. The above syntax should be used instead. If one wants to use the deprecated declaration style in Atmel Studio 7 a special define needs to be inserted at the beginning of the code file containing the deprecated declaration. Such an example is presented in the following code snippet:

Code listing 2-19 ISR deprecated style definition define statement

```
#define __AVR_LIBC_DEPRECATED_ENABLE__ 1
```

The reasonable question that could arise is what code should be written into the interrupt servicing routine function? Giving the fact that this function is called by the core when a byte is received over the serial interface, the obvious operation here should be to read that character from the UDR register, store it in a global variable and announce the main loop, using another global variable as a flag that a new character has arrived. Keep in mind that the global variables that are accessed from the interrupt service routine should be declared as volatile.

There are two main configuring operations that need to be done in order for this interrupt service routine to be activated and taken into consideration of the core. First, the serial UART interface needs to be configured in order to send an interrupt signal upon reception of a byte over the serial line. The responsible bit for this is RXCIE in UCSRB register. Having this bit set as logic 1 instructs the serial UART interface to send an interrupt signal to the ATMEGA16 core when data is received over the line. The second configuration, which needs to be done, is to enable the global interrupt system of the core. This is done by calling the following function before the serial interface is configured:

Code listing 2-20 Global interrupt enable function call

```
int main(void)
{
  // some code
  sei();  // global core interrupt enable function
  //some code
}
```

Having all of this written, the final discussion is on the main loop program, which most of the time must verify if new data has arrived by checking the flag written by the interrupt service routine. If the flag has the correct value then the program should read the newly arrived byte by accessing the global variable containing it (written by the interrupt service routine). Having the new data it should be transmitted back over the serial line using the transmit function routine already presented in the serial library developed earlier. The only operation left doing is to reset the flag that announces the arrival of a

new character, actually, the flag variable written by the interrupt service routine function. If this flag is not reset then the code will be stuck infinitely sending the last received character over the serial line. A flow chart diagram of how the main loop program of the interrupt based echo should look like is presented in the following figure:

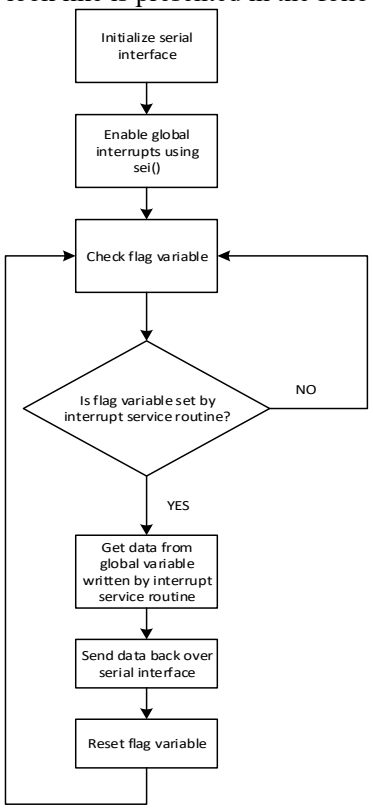

Fig. 2-49 UART echo main program loop with interrupts

ASSIGNMENT 3: Modify the serial interface library previously developed in order to support UART reception using interrupts. Add the interrupt service routine into the library and export the necessary variables to the main program through the serial library header file (using the *export* statement). Modify the main program loop in order to have the serial echo work with reception, using interrupts.

2.9 Laboratory work 8 – Oscilloscope control

This laboratory work can be seen as an integration laboratory work which will combine all the features of the oscilloscope previously developed into one applications. The idea is to also add a basic communication protocol which should serve as a control

panel of the oscilloscope. The APND PC software application, beside the display part, also offers the possibility to send a 2 bytes word over the serial interface. This word can be used in order to control the features of the oscilloscope, thus serving as a control panel.

The control panel of the oscilloscope should have the following functions:

1. Switch any of the channels on or off
2. Change the triggering mode: positive or negative
3. Change the triggering source channel: channel 1 or channel 2
4. Change the time base of the oscilloscope

The features 2 and 3 are already implemented in the previous laboratory works and should also have the possibility to change the settings. As requested in the assignments in Laboratory work 5 – Oscilloscope trigger the code should offer the possibility to change the triggering edge on compile time using define statements. Moreover, also on compile time using define statements, the code should offer the possibility to change the trigger channel source.

The idea in this laboratory work is to offer the possibility to change some of the oscilloscope settings at runtime. Currently can only be changed at compile time. In order to do so, the serial interface could be used to send commands to the embedded side of the oscilloscope from the PC software. Considering the fact that from the PC software only a 2 byte word may be sent to the embedded software, the commands should be bitwise encoded into this word.

In the next paragraphs we will discuss a solution on how the commands may be bitwise encoded in order to be transmitted to the embedded part of the oscilloscope. The attendees may use any other solution to transport the commands and any working solution will be accepted.

We will proceed with one solution where we will bitwise encode the commands as presented in the following figure:

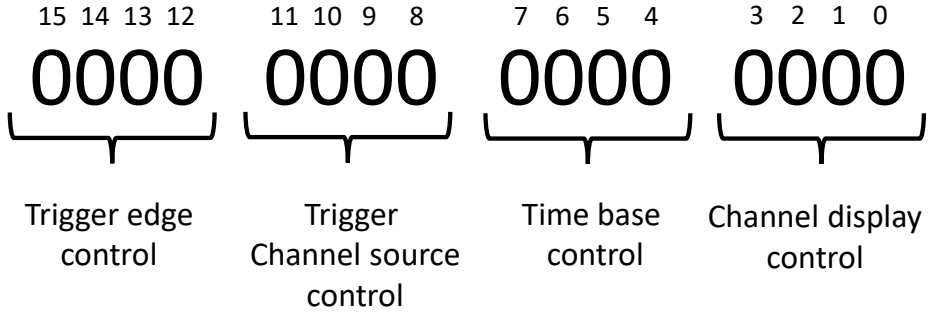

Fig. 2-50 Bitwise command encoding protocol

In Fig. 2-50 above, the 2 byte word that can be sent from the APND PC software application is bitwise presented and the bits a split in groups of 4 as 4 bits is the minimum number of bits for hexadecimal representation. Having the grouping in the above figure,

we can distribute the commands to the grouping of these bits and so we can obtain the following rules:

- Group represented by bits 15-12 is responsible for the control of the triggering edge. Using this group the oscilloscope should be controlled whether to use positive edge triggering of negative edge triggering
- Group represented by bits 11-8 is responsible for the control of the channel source the triggering is made on. Using this group the oscilloscope should be controlled whether to use channel 1 or channel 2 for triggering source
- Group represented by bits 7-4 is responsible for the control of the oscilloscope's time base. The number formed by these bits should divide the maximum time base. This aspect will discussed further in this laboratory work
- Group represented by bits 3-0 is responsible for the control of which channel should be displayed on the oscilloscope. The possible options should be: no channel, channel 1, channel 2 or channel 1 and 2 (a feature available on any real oscilloscope).

In order to use the 2 byte word we must first implement a small finite state machine that is able to receive the 2 bytes and form one 16 bit word. First off all, it is important to know how the PC software sends the 2 byte word over the serial line, giving the fact that only one byte may be sent over the serial line at a time, in a frame.

The APND PC software accepts a hexadecimal 2 byte unsigned integer to be written into the textbox and sends this number with the least significant byte first followed by the most significant byte (ex. Byte formed by bits 0-7 is sent first followed by the byte formed by bits 8-15).

In order for the microcontroller program to receive this 16 bit word it must implement in the serial UART interrupt service routine a small finite state machine that receives the 2 bytes and combines them in the end before passing the newly formed 16 bit number to the main loop application.

The finite state machine should contain only 2 states and it should be called by the interrupt service routine. In this case, each time finite state machine is called it would be in a moment a new byte is available in the receive buffer. In the first state the finite state machine should get the byte that represents the least significant part of the 16 bit word, store it in a variable and switch to the next state. In the second state the finite state machine should receive the most significant part of the 16 bit word. In this case it should combine it with the previously received byte, form the 16 bit word and pass it to the application main loop. This state should also reset the whole finite state machine. Even if this finite state machine is quite simple the following figure summarizes the operations:

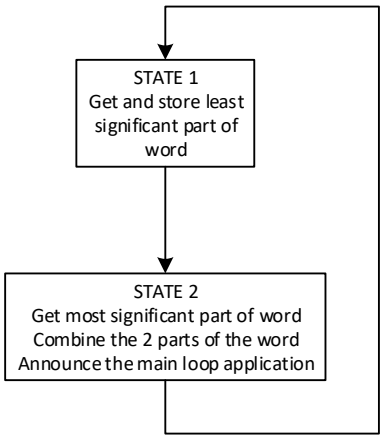

Fig. 2-51 UART 16 bit word reception finite state machine

Each time the finite state machine finishes receiving a new 16 bit word, it should announce the main loop program about the availability of new data. This can be done by using a volatile variable with the functionality of a flag. The interrupt service routine, once it has a new 16 bit word, can set the flag. The main loop program must then test the flag and when it finds it on the correct value it should read the newly arrived 16 bit word and then reset the flag. The newly arrived 16 bit value should be stored in a volatile variable. Also the flag variable should be a global variable.

After having the 16 bit word available we can use some of the bits to implement the first control methods. The first controlling should be the edge of the triggering and the source channel used for triggering. This can be easily done by using the code developed so far and replacing the testing of the define statements with the testing of the required bits.

ASSIGNMENT 1: Implement a finite state machine algorithm inside the interrupt service routine function that is able 2 receive and combine the 16 bit command word from the APND PC application. Use 2 global variables to communicate between the interrupt service routine and the main program loop: one variable to announce the arrival of a new command word and one variable to contain the actual command word. Both variables should be made volatile. The main program loop should check for the first variable (flag variable – which is set by the interrupt service routine when a new command has arrived) and if the variable flag is found of value 1 than the program should read the second variable containing the actual data and make the necessary operations. After reading the actual command variable the main loop program should reset to 0 the flag variable, in order to avoid being stuck executing same command over and over again.

Regarding the controlling of the edge of the triggering, upon the arrival of a new command 16 bit word, if the bits 12-15 have, for example, value 0, then negative edge should be used and if the value is 1 then it should be switched to positive edge triggering.

The embedded part of the oscilloscope should only sense the command at the beginning of the acquisition and storing of a buffer, practically at the beginning when the buffer is considered empty. During the time the buffer is being filled, the microcontroller shouldn't be expected to process new commands.

Regarding the controlling of the source channel that should be used for triggering, upon the arrival of a new command 16 bit word, if the bits 8-11 have, for example, value 0, and then channel 1 should be used as source channel for triggering, and if the value is 1, then channel 2 should be used as source channel for triggering. Again, a command, should only be executed at the beginning of a new loop execution.

These two aspects regarding trigger should not imply too much code change thus these have already been implemented but the configuration was made only at compile time using defines. The only adaptation here is to use the 16 bit command word to change the settings at runtime instead of the define statements at compile time.

The only code change, which will be a little difficult, is the part related to the frequency calculation of the triggered signal. According to the explanations in Laboratory work 6 – One channel frequency calculation, the frequency calculation is depended on the triggering. Changing the triggering will affect frequency calculation. The calculation of the frequency needs to be adapted if the edge of triggering is changed and also if the source channel of the triggering is changed.

ASSIGNMENT 2: Implement the 2 commands related to triggering: the command for changing the edge of triggering and the command for changing the source channel used for triggering. Use the APDN PC software in order to validate the functionality.

ASSIGNMENT 3: Adapt the signal frequency calculation module in order to correctly react to the changes of the triggering coming from the 16 bit word command from the PC application.

Two of the control panel features defined above have been taken care of (more exactly feature 2 and 3). Regarding feature 1 which says that the oscilloscope should be able to enable and disable the display on any of the channels. This command, according to Fig. 2-50, is encoded into bits 0-3 from of the 16 bits available in the command word. There should be 4 states of this feature:

- Display off for both channels
- Display only Channel 1
- Display only Channel 2
- Display both Channel 1 and Channel 2

The implementation of these 4 states of the command are trivial to implement on the embedded side. The only operation that needs to be done is, according to the selected setting (command), to store the actual acquired data into the buffer or to store value 0 instead of the acquired data. For example if only Channel 1 is selected to be displayed, when writing the data into the 2 buffers, the buffer for channel 1 should be written with the actual data from the ADC and the buffer for channel 2 should be written with 0. No

other modification is necessary thus the PC application will display, in this case, the actual waveform for channel 1 and a line on value 0 for channel 2. Same algorithm should be applied for all situations.

ASSIGNMENT 4: Implement the channel on/off functionality. Only the actual data that goes into the buffers should be affected and not the ADC conversion itself. The ADC should convert data for both channels even if it would be discarded. Having this approach will not modify the sample rate. Validate the functionality using the PC software application by sending the correct command word to the embedded system to process.

The only control command, which remained untouched, is the command responsible for the changing of the time base of the oscilloscope. This can be done in 2 ways: one way is to change the acquisition period by modifying the clock of the ADC and another way is the change the actual size of the buffers.

In this laboratory work the second method will be used. In order to do so, the buffers will still be statically declared and will have an absolute maximum size, but the actual size will vary depending on a divisor value coming from the PC application using the 16 bit command word. From this command word, bits 8-11 will be used for this control. These bits will serve as a number that will divide the maximum buffer size forcing the acquisition to be made not for the whole size of the buffers, but for the resulted size of the division. Pay special attention what is the number of samples changes; this number needs to be update into the protocol encoding as described in Table 6 from Laboratory work 4 – Minimal 2 channel oscilloscope.

ASSIGNMENT 5: Implement changing of the time base of the oscilloscope. Modify the size of the buffers accordingly in the protocol encoding. Make sure, that in the protocol, encoding the real size of the acquisition is encoded and not the maximum allocated size of the buffers.

3 Digital Signal Processing

3.1 Introduction

The digital signal processing discipline aims at introducing the student into minimal aspects regarding signal processing. As a subdomain of digital signal processing we will address audio processing during this laboratory work.

In order to attend to this laboratory the students must have the following mandatory prerequisites:

1. Strong C programming skills [1]
2. Basic knowledge of embedded systems and embedded programming and debugging
3. Knowledge data structures and algorithms: ring buffer data structure management

3.2 Laboratory work – The GPIO System of Blackfin BF537

This laboratory work is used as a general introduction and has, as main purpose, to prepare the student for the following laboratory assignments.

This laboratory makes the first steps in writing, compiling and executing a program on a Blackfin BF537 processor. Also this laboratory presents the General Purpose Input Output module of BF537. As a usage example of the GPIO module, a LED blink application will be implemented using a software delay procedure.

The main components of the BF537 periphery are: Ethernet controller, CAN controller, I2C controller, PPI interface, SPI interface, 2 synchronous serial ports (SPORT), 2 asynchronous serial ports UART, timer module, 48 GPIO signals, a DMA controller, etc. The pins of this processor are organized under 16 bit ports: PORTF, PORTG, PORTH, and PORTJ. All the ports have 16 signals with the exception of PORTJ which has only 12 signals. After a system reset all the pins are configured as GPIO. The only exception is PORTJ which does not offer GPIO functionality.

The GPIO module is controlled by the following registers:

- PORTxIO_DIR
- PORTxIO_INEN,
- PORTxIO_IO,
- PORTxIO_SET,
- PORTxIO_CLEAR,
- PORTxIO_TOGGLE,
- PORTxIO_POLAR,
- PORTxIO_EDGE,

- PORTxIO_BOTH,
- PORTxIO_MASKA,
- PORTxIO_MASKB,
- PORTxIO_,
- PORTxIO_MASKA_SET,
- PORTxIO_MASKB_SET,
- PORTxIO_MASKA_CLEAR,
- PORTxIO_MASKB_CLEAR,
- PORTxIO_,
- PORTxIO_MASKA_TOGGLE,
- PORTxIO_MASKB_TOGGLE

In the register names presented above the x can be either 'F', 'G' or 'H'. Only the most important registers regarding the GPIO will be presented in this laboratory. The full description of the registers presented here may be found in the Hardware Reference Manual of Blackfin [20], chapter 14.

One of these registers that present interest to our laboratory work is the PORTxIO_DIR register collection. Each bit of this register controls the direction (INPUT or OUTPUT) of the pin for the selected port. In order to configure a pin as an output pin, the corresponding bit of PORTxIO_DIR has to be set as "1". After reset, all the pins are configured as inputs. Example: PORTFIO_DIR = 0xFF00 has the following effect: lines (pins) PF8 to PF15 are outputs and lines (pins) PF0 to PF7 are inputs. The description of PORTxIO_DIR can be found in the following figure [20]:

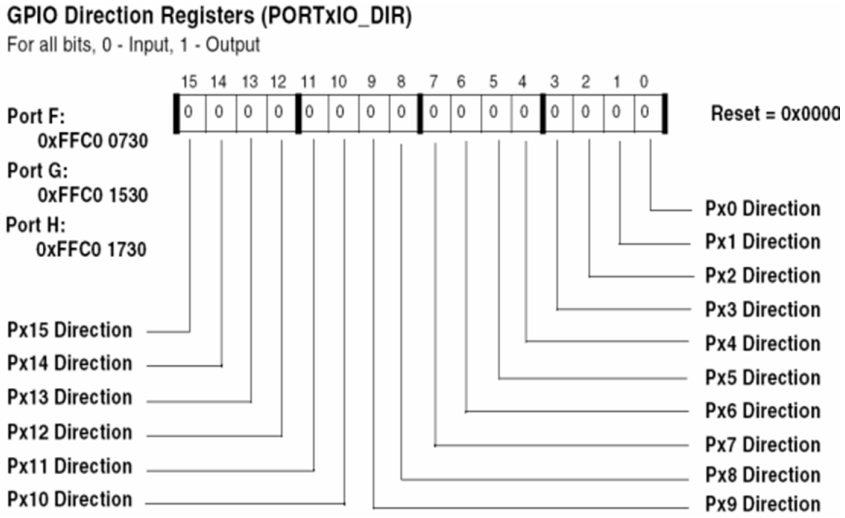

Fig. 3-1 GPIO Direction registers description [20]

The register that actually controls the stat of the port working in a GPIO manner is PORTxIO. This register has a double meaning: it can modify the logical levels of the pins

that are configured as outputs and can read the logical levels of the pins that are configured as inputs. In addition, in order for the processor to detect the logical level of an input, an extra configuration has to be made. In order for a pin to function as an input pin not only the "0" logic value has to be written in the corresponding bit of PORTxIO, but also a logic "1" has to be written in the corresponding bit of the register PORTxIO_INEN. These registers are described by Fig. 3-2 and Fig. 3-3.

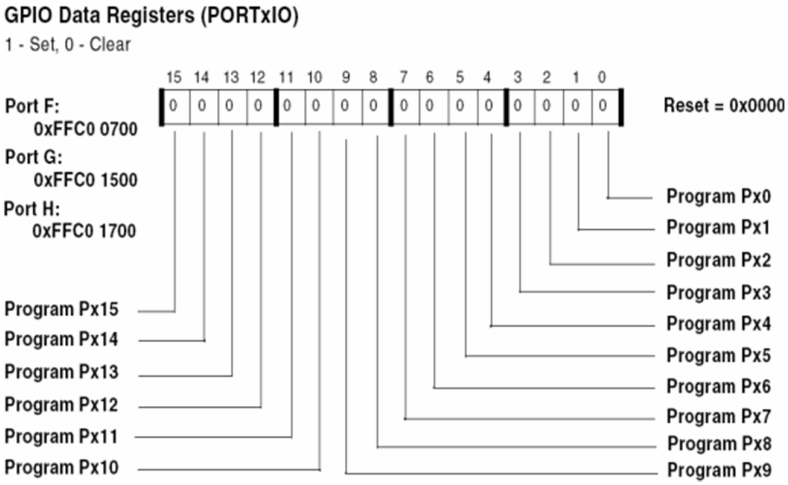

Fig. 3-2 GPIO Input / Output registers description [20]

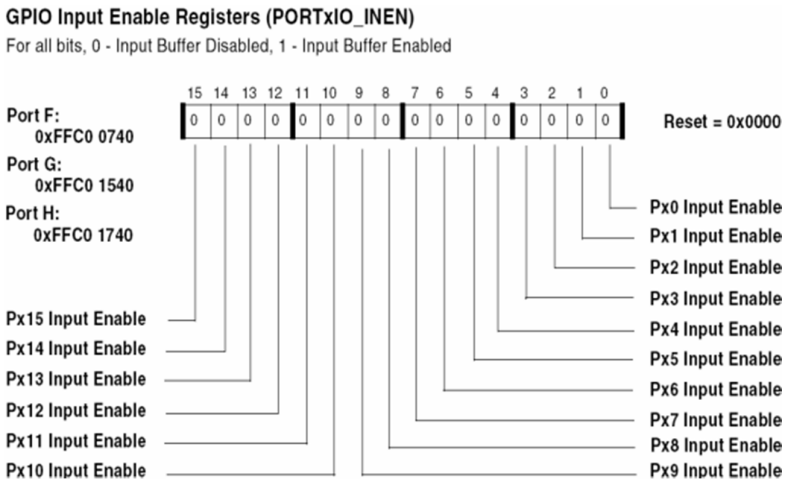

Fig. 3-3 GPIO Input Enable registers description [20]

ASSIGNMENT 1: Create a new Visual DSP++ Project with the following settings: "Standard Application", ADSP-BF537 as processor. Create a new debug session (Session

-> New Session -> ADSP BF537 -> EZ-KIT Lite -> ADSP-BF537 EZ-KIT Lite via Debug Agent). In this moment the Visual DSP++ Environment is connected via a debug channel to the evaluation laboratory board. In order to compile the project: press F7 (or Project -> Build project). After a successful compilation an automatic upload of the executable to the processor is performed. The executable code can also uploaded after pressing CTRL-R. In function main add an int variable and increment it in an infinite loop (an infinite while loop). Visualize this variable as following: View -> Debug -> Windows -> Expressions. Run the resulting program step by step (F11 or Debug -> Step Into) and observe the way the defined variable modifies).

ASSIGNMENT 2: Using the project created at laboratory application 1:

- Open window "Manage Custom Register Windows" from Register -> Custom;
- add to this new window the registers PORTFIO_DIR, PORTFIO, PORTFIO_INEN from the Port I/O tree;
- activate this newly created window (Register->Custom -> <Window name>);
- in the PORTFIO_DIR register configure as outputs the 6 signals that are connected to the LEDs on the board, LED1 to LED 6 leaving the other pins of port F as inputs (check the board schematic in the manual);
- in PORTFIO_INEN configure as inputs the 4 signals that are connected to the buttons on the board
- Try to turn on the LEDs by writing logical "1" at the corresponding bits of the PORTFIO register. The status of the LEDs will change immediately after writing the register
- In order to view the changing of the values of the inputs, an extra step into the execution is required. Hold one of the buttons pressed while executing a new step into and observe the modifications in the port registers.

ASSIGNMENT 3: Write a program that reflects the state of the buttons using the LEDs.

- Include the cdefBF537.h header file in your main C file

```
#include <cdefBF537.h>
```

- access to the processor register are made as the example below:

```
*pUART1_GCTL = 0x0001;
```

ASSIGNMENT 4: Write a simple delay routine using the "busy-loop" mechanism in order to blink the LEDs. A busy-loop can be implemented as following:

Code listing 3-1 Busy-loop implementation example

```
int main(void)
{
  // some code
   volatile int i;
   for (i = 0; i < xxxxxx; i++);
  //some code
}
```

3.3 Laboratory work – The Timer module of Blackfin BF537

This laboratory assignment can also be considered as a general introduction for the upcoming laboratory assignments. The main purpose of this laboratory work is to instruct the student into the clock system of the processor. The student will also learn how to use the timer module inside BF537.

In most cases, microcontrollers, DSPs and general processors are synchronous devices, which implies the usage of a clock signal. This signal can be internally generated, using an RC oscillator, or can be externally provided by a dedicated clock generator or by using a quartz oscillator. The Blackfin BF537 may use a quartz oscillator in order to generate its internal clock signals. The quartz oscillator is the main input of a PLL (Phase Locked Loop) circuit which is meant to multiply and stabilize the main clock source. Additional dividers and multiplexers are also used in order to generate all the clocks needed internally by the processor. The clock generation scheme is presented in Fig. 3-4 [20].

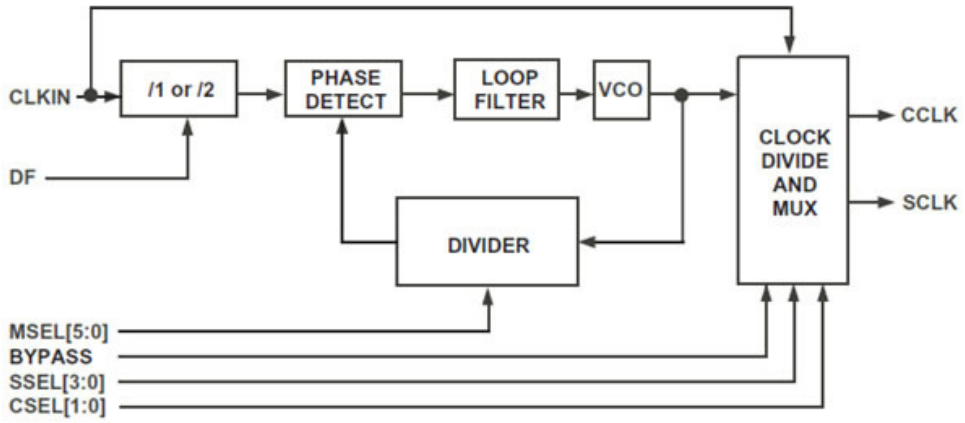

Fig. 3-4 GPIO Blackfin BF537 clock generation diagram [20]

As described in the figure above, after dividing the multiplying the CLKIN signal (provided by the quartz oscillator) the VCO signal is obtained. This signal is the main input for obtaining the most important clock signals used by the core and the periphery of the BF537 DSP: CCLK (Core Clock) and SCLK (System Clock). The SCLK signal is the clock input for the DSP's periphery. More information regarding the clock system of BF537 can be found in the Black BF537 Hardware Reference Manual, chapter 20 [20]. Based on Fig. 3-4, the clock signals can be calculated using the following formulas:

$$VCO = CLKIN \cdot VCO, \quad DF = 0 \tag{3-1}$$

$$VCO = \frac{1}{2} \cdot CLKIN \cdot MSEL, \quad DF = 1 \tag{3-2}$$

$$CCLK = \frac{VCO}{CSEL}, \quad CSEL \in \{1,2,4,8\} \tag{3-3}$$

$$SCLK = \frac{VCO}{SSEL}, \quad SSEL \in \{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\} \tag{3-4}$$

Based on the formulas above an important observation can be made: the peripheral clock, SCLK, may be divided using higher divisors that the ones used to divide CCLK, thus SCLK may be significantly lower than CCLK. This is usually needed mainly because the periphery of a microcontroller, or a DSP, is usually clocked with lower frequency clock signals than the core of the processor.

After system reset the variables above have the following values: MSEL = 10, CSEL = 1, SSEL = 5.

The BF537 on the laboratory board is clocked using a 25 MHz quartz. Using the default values presented above, immediately after system reset, the processor starts at a 250 MHz core clock and a 50 MHz peripheral clock.

Regarding the timers of BF537, there are 8 general, identical, 32 bit wide timers with interrupt generation capabilities. The general structure of the 8 timers is presented in Fig. 3-5 [20]:

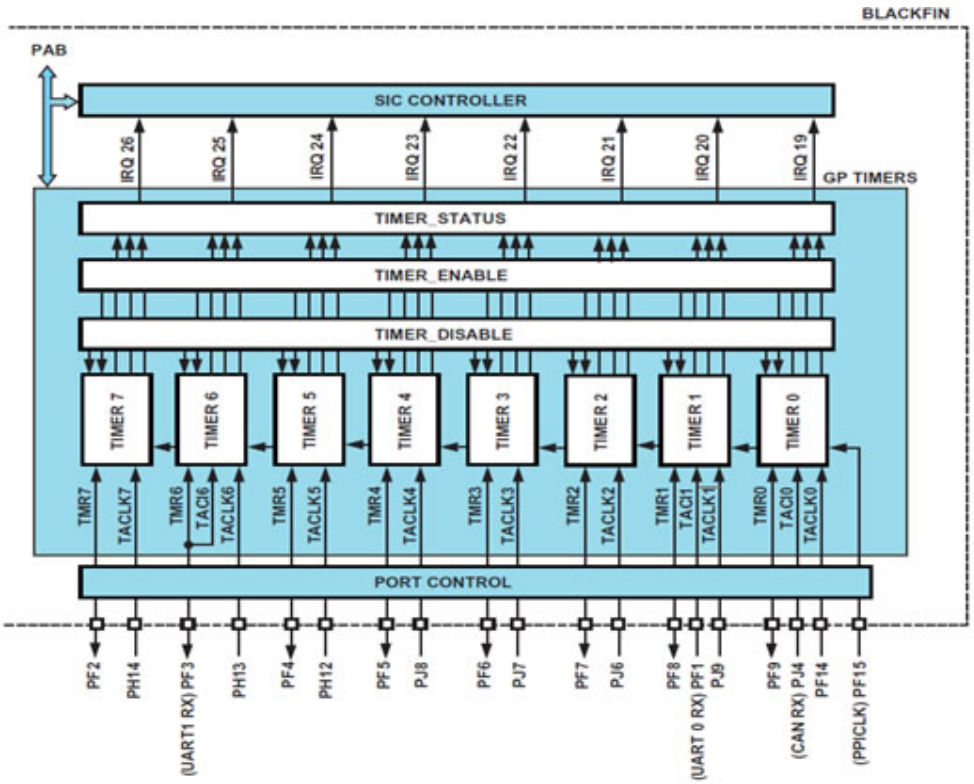

Fig. 3-5 Blackfin BF537 timer block diagram [20]

All the timers of this DSP are controlled and monitored by 3 general registers. TIMER_STATUS, TIMER_ENABLE and TIMER_DISABLE. Each register controls all the 8 timers. The general internal structure of a timer is described by the following figure [20]:

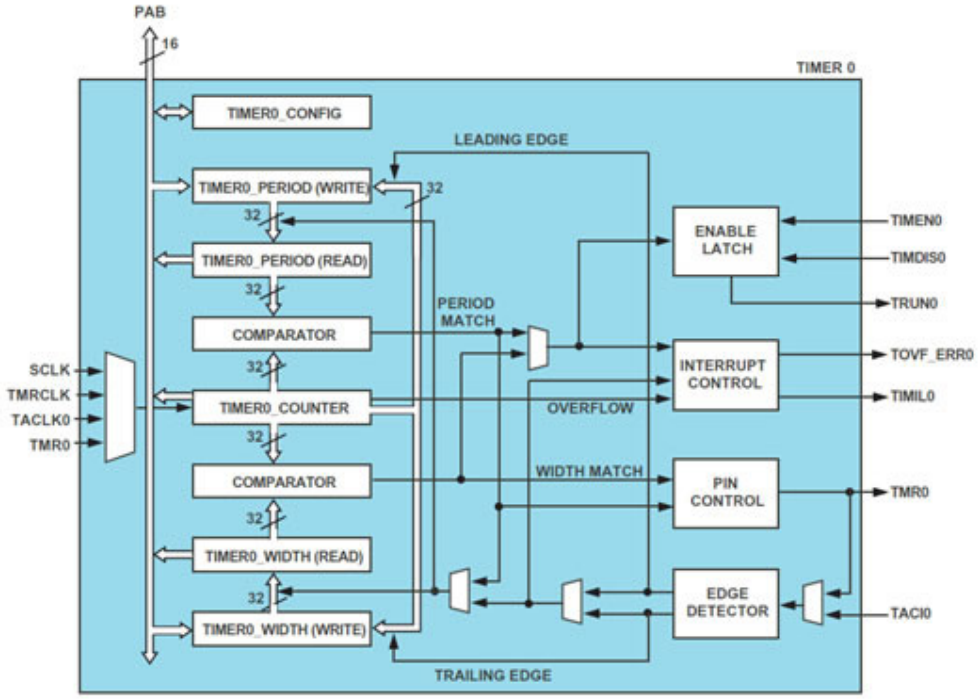

Fig. 3-6 Blackfin BF537 timer internal structure [20]

All the timers have several operating modes. In this laboratory work, only the PWM (Pulse Wide Modulation) operating mode will be used and detailed. This operating mode can be used for waveform generation, for pulse generation or for generating periodic events. In order to generate periodic events, the timer interrupts have to be properly configured. In order to verify if an interrupt is pending, the programmer may read the status register.

A more detailed documentation for the BF537 timers can be found in the Blackfin BF537 Hardware Reference Manual [20], Chapter 15.

The timers of BF537 are controlled by the following registers:

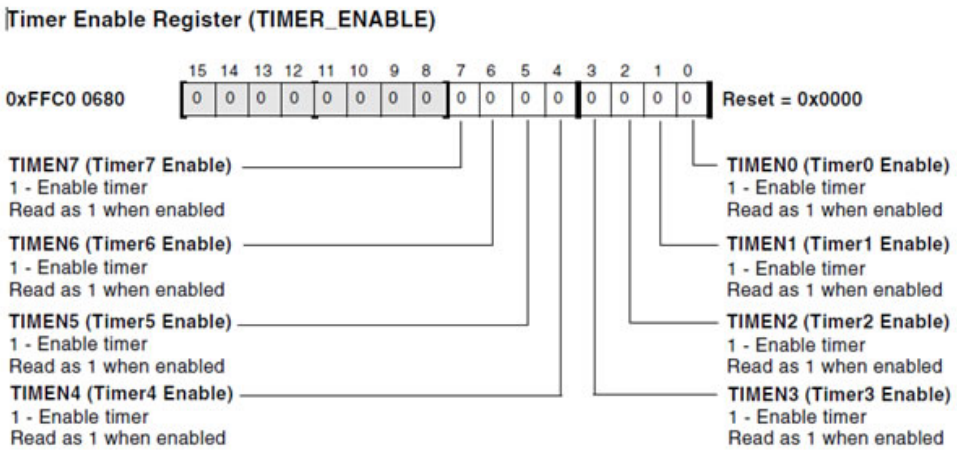

Fig. 3-7 Blackfin BF537 timer enable register structure [20]

TIMER_ENABLE is a register that can be used to enable the timers. The structure is presented in Fig. 3-7. TIMER_DISABLE is the register that can be used to disabled the timers. The structure is identical to the structure of TIMER_ENABLE register.

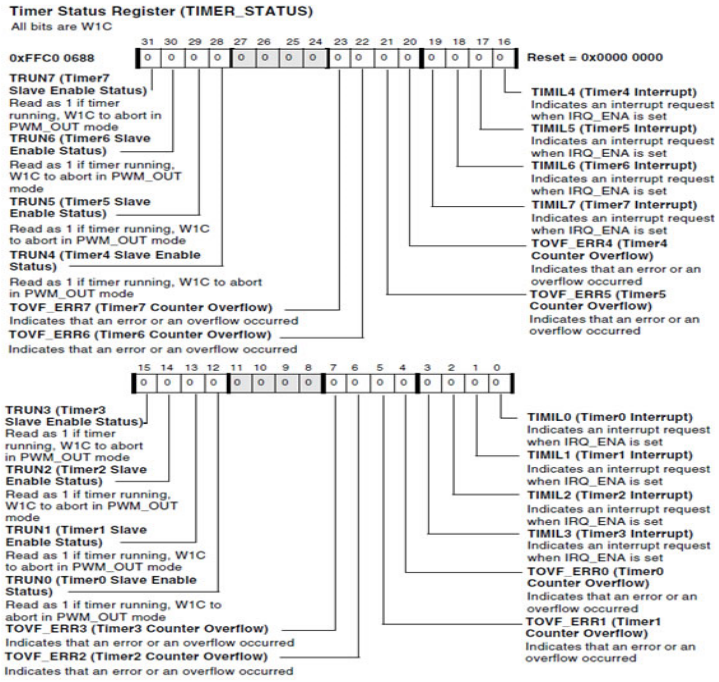

Fig. 3-8 Blackfin BF537 timer status register structure [20]

TIMER_STATUS is a register which offers the status of each timer and a way to reset this status by the programmer. For instance, the programmer may check if a timer generated an interrupt by reading the corresponding bit in this status register. If the value is logical "1" then the timer has a pending interrupt. In order to reset the status of the checked interrupt, the programmer has to write a logical "1" to the corresponding bit of that timer. The structure of the timer status register is presented in Fig. 3-8

All of the registers presented above are shared by all of the timers. Each timer has its dedicated bits in these registers. Also, each timer has dedicated registers like TIMERx_CONFIG, TIMERx_COUNTER, TIMERx_PERIOD, where x can be 0 - 7 to identify the timer.

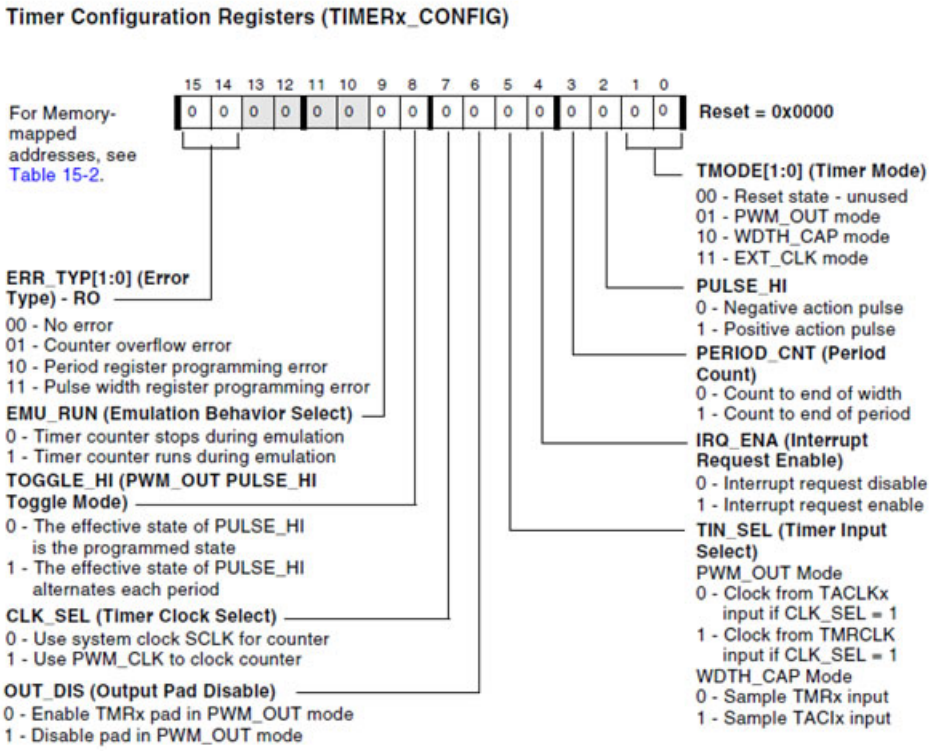

Fig. 3-9 Blackfin BF537 timer configuration register structure [20]

TIMERx_CONFIG is the register responsable for the configuration of the designated timer. Each timer has a configuration register associated with it. The description of this register is found in Fig. 3-9.

TIMERx_COUNTER is the 32 bit counter register. Each timer has its own counter register.

TIMERx_PERIOD is a 32 bit wide register that identifies the counting period. The value of this register is compared during counting with the timer counter register in order to generate an interrupt on match.

The maximum counting period of a timer can be calculated using the following formula:

$$T = \frac{2^{32} - 1}{F_{count}} \quad (3\text{-}5)$$

where F_{count} is the input frequency of the counter (usually equal to SCLK).

The value of the counter value for a designated period may be obtained using the following equation:

$$N = t \cdot F \quad (3\text{-}6)$$

where t is the timer period in seconds, F the input frequency of the timer, N - the timer period tick value (the number of ticks).

According to the manual, it is mandatory that the timer enable operation (through TIMER_ENABLE register) to be made after the timer configuration. The enabling operation is the last operating to be made to the timer. After the timer is enabled it cannot be reconfigured until it is disabled again.

ASSIGNMENT 1: Calculate the PLL coefficients (MSEL, CSEL, SSEL) in order to make the core function at maximum frequency (600 MHz) and the periphery at 300 MHz. Consider the case when the BF537 chip is clock using a 25 MHz quartz.

ASSIGNMENT 2: Calculate the TIMERx_PERIOD register value (in hexadecimal format), using the clock condition from application 1 in order to obtain the period 100 us interrupt.

ASSIGNMENT 3: Write a small program that blinks a LED on the board at 1Hz (0.5 seconds on, 0.5 seconds off) using timer 7 and its status register to identify a pending interrupt. Hints:

- write a modular code (using functions for timer initialization, LED initialization, LED blink, etc...)
- use only the timer status register to check whether the interrupt event occurs. Do not use and activate the global processor interrupts
- check laboratory work 1 for GPIO usage and use some of the code written the previous laboratory.
- read about PORTx_TOGGLE in the hardware reference manual

ASSIGNMENT 4: Change the code written at application 3 in order to use the global processor interrupts. Blink the LED in the interrupt service routine. Hints:

- initialize the global interrupts to generate an interrupt for Timer7 using the following function

```
void initInterrupts(void)
{
     *pSIC_IAR3 = 0xFFFFF5FF;
     *pSIC_MASK = 0x04000000;
     register_handler(ik_ivg12, ISR_Timer7);
}
```

- use the following function as the timer 7 interrupt service routine:

```
EX_INTERRUPT_HANDLER(ISR_Timer7)
{
      //user code
}
```

- the necessary include statements are:

```
#include <cdefbf537.h> // register definitions for BF537
#include <sys/exception.h> // definitions for the interrupt system
```

- the first instruction of the interrupt service routing has to reset the current interrupt state (check TIMER_STATUS register or processor user manual)

3.4 Laboratory work – Audio signals

The main purpose of this laboratory work is to introduce the student into audio processing. The theoretical aspects presented in this laboratory work are related to the acquisition and digital processing of the audio signal. The laboratory assignments are meant to show how the Blackfin DSP captures the audio signal and also to show how the programmer has access to each of the captured sample. This laboratory is not oriented on configuring the communication interfaces between the BF537 and the rest on the chips involved in the audio processing. A sample code will be provided in order to configure the chips on the board to capture the audio signal.

A digital signal processor is a special type of processing unit which is oriented on signal processing using specialized functional units. This instruction set of a DSP has special instructions that are used to implement the basic operations in digital signal processing; for example: convolution can be implemented using instructions like MAC (multiply and accumulate), which in most cases can execute this operation in a single clock cycle. Another important characteristic of a DSP is the architecture. Most DSP are organized using a Harvard architecture or even a modified Harvard architecture.

A Harvard architecture is characterized by the existence of 2 distinct memories: a code memory and a data memory. In addition, in a modified Harvard architecture the data memory is divided into 2 distinct memories. This architecture speeds up the fetch process of an instruction and its operands. Also in many cases DSPs have additional modules that are meant to accelerate the execution of the code like pipelines and cache memories.

The signals that are present in nature are physical signals continuous in time which are transformed into electrical signals (analog signals), continuous in time, using sensors and transducers. Mathematically these signals are modeled using a continuous, single

variable (time) function. In order to study, analyze and process these analog signals by a digital system they have to be converted into digital signals. The operation that converts an analog signal to a digital signal is called analog to digital conversion. The main steps involved in an analog to digital conversion are: sampling, quantization and binary representation. The opposite operation is digital to analog conversion. For more information regarding these operations check the Digital Signal Processing Course.

The audio signal is an analog signal with frequencies between 20 Hz and 20 KHz, the ideal perception of the human ear. Using the sampling theorem, in order to properly convert the audio signal into a discrete signal the minimal sampling frequency has to be greater than 40 KHz. The most common sampling frequencies used to convert the audio signal to a digital signal are 44.1 KHz, 48 KHz and 96 KHz. When using a low resource system to process the audio signal, lower sampling frequencies are used: 22050 Hz, 11025 Hz, and 8000 Hz. The disadvantage in using such sampling frequencies is that the quality of the converted signal is significantly reduced. There are some type of applications for which this disadvantage is not important; for example in telephony a sampling frequency of 8 KHz is used.

Another important aspect in audio processing is the intensity of the audio signal. The intensity of the audio signal is represented on a logarithmic scale mainly because of the human ear perception. The human ear can easily distinguish the variations of low intensity sounds than the variations of very high intensity sounds. The human ear get saturated when receiving high intensity sounds. The intensity of the audio signal is represented using a logarithmic measuring unit, the decibel (dB). The conversion between the logarithmic and linear scale and vice versa is modelled using the following expressions:

$$L[dB] = 20 \cdot \log_{10} \frac{V}{V_0} \qquad (3\text{-}7)$$

$$V = V_0 \cdot 10^{\frac{L[dB]}{20}} \qquad (3\text{-}8)$$

In the formulas above, L designated the intensity in dB of the audio signal. In an audio processing system, in order to modify the output intensity of the sound the processor has to modify the L parameter on a linear scale. A value of L = 0 specified that the intensity of the audio signal is not modified; a positive value (L > 0) specified the amplifying of the signal; a negative value of L (L < 0) specified the attenuation of the audio signal.

The general form of an audio processing system is presented into the following block diagram:

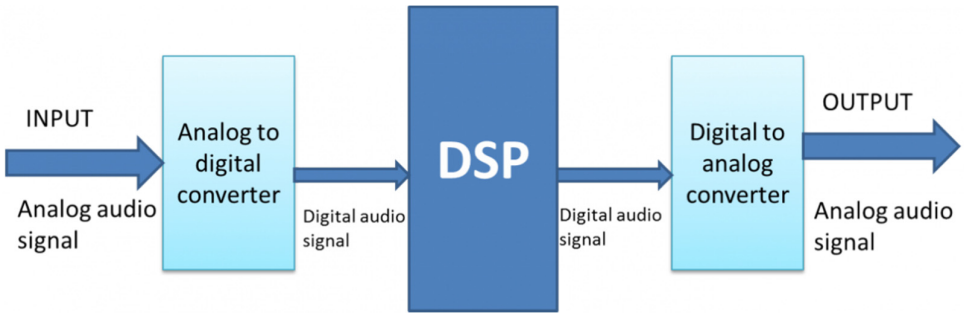

Fig. 3-10 Block diagram of a general audio processing system

The digital algorithms and processing applied on the digital signals are implemented on the DSP which operates on the captured samples of the audio signal. The DSP has the audio signal as input, in digital form provided by the analog to digital converter. The result of the DSP's processing is fed to the digital to analog converter which reconstructs the audio signal in order to be perceptible by the human ear again.

The ADSP-BF537 EZ-KIT Lite laboratory boards has the necessary elements to implement the blocks presented in Fig. 1. The DSP block is represented here by the Blackfin BF537 processor. The analog to digital converter is implemented using an AD1871 [21] chip and the digital to analog converter is represented by an AD1853 [22] chip.

An important observation has to be made: The laboratory board only has a digital to analog converter to output the resulting audio signal and no amplifying circuit after it. In this situation the DSP can only lower the volume (attenuate) the signal and cannot amplify it. Amplifying of the signal may result to distortion.

In order to ease this laboratory work, a sample code is provided. This code is meant to initialize the audio processor as stated in the introduction of this laboratory work. The configuring operations of the ADC and DAC chips as well as the communication interfaces between them and the DSP are not the scope of this laboratory work. This code is organized under a proper configured Visual DSP++ project. This code implements a simple audio loopback (the input samples are transferred to the output of the board unprocessed). The project contain the following code files:

- main.c - contains the main program file
- ISR.c - contents the interrupt service routines
- initialize.c - contains the initialization of the hardware components used to capture, process and output the audio signal
- process_data.c - contains a callback function that offers the instant captured values of the audio samples (there are 2 audio samples captured at a time - stereo left and right)
- talkthrough.h - header file

The main function is used to call the initialization functions defined in initilize.c which are meant to configure the ADC and DAC chips as well as the communication interfaces. After this, the main function blocks the main thread into a while loop. The audio samples are transferred from the ADC to the DSP and from the DSP to the DAC using 2 DMA channels. An interrupt is generated after the reception of a double sample (left sample and right sample of a stereo audio signal). This interrupt calls the function in process_data.c which offers the captured double sample to the programmer. The programmer has to take into consideration the fact that the processing function in process_data.c executes in interrupt context.

ASSIGNMENT 1: Download and study the execution of the sample code:

- Connect an audio signal source to the line in connector of the board. Using a double jack cable connect the audio output of the computer (the green connector) to the line in connector of the board.
- Connect the headphones to the line out connector.
- execute the program

ASSIGNMENT 2: Explain what these lines in process_data.c do. Explain the difference between them. Correct the situation.

```
iChannel0LeftOut  = ((iChannel0LeftIn<<8)>>3)>>8;
iChannel0RightOut  = ((iChannel0RightIn<<8)>>0)>>8;
```

ASSIGNMENT 3: Implement a logarithmic volume using two of the buttons on the board, one as volume up and another as volume down. The press of each button should modify the volume by -/+1dB.

- Use expression presented above where V_0 is the input sample and V is the output sample. The input sample should be multiplied by a coefficient calculated based on L parameter that will be modified by the actions on the two buttons.
- Take into consideration that the processing function is execute in interrupt context; try not to make time consuming calculations in this function. Also try not to make unnecessary calculations.
- observe and explain what happens when the L parameter is above 0

ASSIGNMENT 4: Write a short Matlab function that modifies the volume of a wave file. The function should have the following parameters: the input wavefile path, the signed value in dB which amplifies/attenuates the wavefile, the output wavefile. Use the same logic as in application 3. Make use of the following functions: wavread, wavwrite, plot, figure, and subplot. Use also "help <function>" to see the meaning and parameters of these functions. Parse the wave samples using a “for loop”. Plot the input wave and the output wave on a same figure to see the difference.

3.5 Laboratory work – Echo effect

The purpose of this laboratory work is to teach how to implement the echo effect on an audio signal using a delay line. The circular buffer data structure will be used in implementation.

The echo effect was first made by Mike Battle in 1959 using a tape recorder. The main idea was to apply a delayed (using recording) version of the audio signal on the original audio signal. There are certain types of audio echo:

- Echoplex - the first version of the echo produced by Mike Battle in 1959
- Doubling echo - is the echo obtained by adding a short range delay over a recorder audio
- Slapback echo - same as doubling echo but the delay is significantly longer.
- Flanger, chorus, reverb - these are echo based audio effects where the delay time is modulated

The generation of these audio effects was a little complicated when using devices working directly with analog signal. When using DSPs the generation is much simpler. In this case the idea is that a delay sample is added to the current sample. The newly obtained sample is sent to the output and saved in memory in order to be used over the next sample. Before the delayed sample is added to the current sample it needs to be attenuated at half of its value. The same operation has to be performed on the newly obtained sample before storing it for later use. This attenuation has to be performed in order to avoid saturation. The delay system is presented in the following diagram:

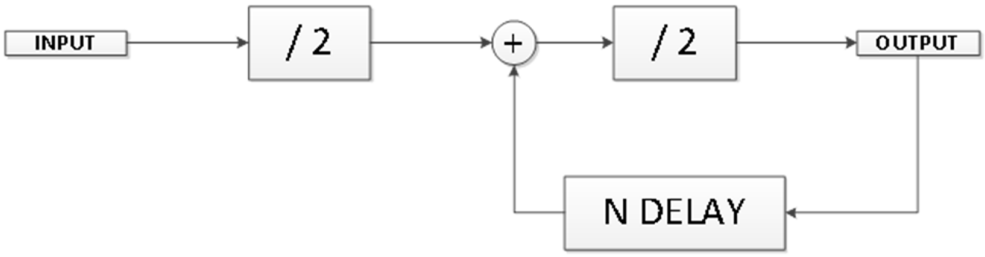

Fig. 3-11 Echo generation system block diagram

This delay system can be easily implemented using a circular buffer. The delay time is proportional to the size of the buffer. The upper schematic can be explained as following: each received sample is reduced at half of its intensity, added to sample stored in the circular buffer at a current index, reduced to half of its intensity again and then stored at the current index of the circular buffer. A circular buffer is accessed using 2 pointers: a read pointer and a write pointer. The most important rule of a circular buffer is that the read pointer is always behind the write pointer. The circular buffer is

implemented using a normal linear buffer with the property that it is accessed in a circular manner: the entry immediately after the last entry of the buffer is actually the first entry of the buffer. In order to achieve this rule the pointers are incremented using modulo based arithmetic. For instance, if N is the buffer physical dimension, buff is the actual physical buffer, *rd_ptr* is the read pointer, *wr_ptr* is the write pointer and value is the actual value stored in the buffer the main operations (load/store) can be implemented as following:

```
//Circullar buffer store:

buff[wr_ptr] = value; wr_ptr = (wr_ptr +  1) % N;
//Circullar buffer load:

value = buff[rd_ptr]; rd_ptr = (rd_ptr +  1) % N;
```

In order to achieve an echo effect, a circular buffer has to be used in order to implement the diagram presented in Fig. 3-11, thus in this case is sufficient to use a single pointer for reading and writing operations over the circular buffer (*rd_ptr* = *wr_ptr*).

ASSIGNMENT 1: Using the loopback framework presented in the previous laboratory work, implement the echo effect. Use a circular buffer. The size of the circular buffer should be between 2048 and 4096 in order to achieve best echo quality.

ASSIGNMENT 2: Combine the logarithmic volume implemented in the previous laboratory work with the echo effect.

ASSIGNEMNT 3: Write a Matlab function that adds echo to a wave file using a circular buffer implemented in Matlab using a 2 dimensional array. The array of the circular buffer can be declared using function zeros (length, 2). In Matlab the array of an index begins at 1. The echo function should take the following parameters: input wave file path, output wave file path, and size of the circular buffer. The index of an array should be implemented using a uint32 variable (var = uint32 (1) - a uint32 variable initialized with 1). Parse the wave samples using “a for” loop.

4 Embedded Systems Design and Development

4.1 Introduction

The Embedded Systems Design and Development laboratory presents the first steps into designing and developing embedded applications. The laboratory is oriented both on the features of a microcontroller but also on working with peripheral devices such as 7 segment displays, alphanumerical LCDs, sensors, keyboards, etc.

The laboratory works are mainly decoupled and each application presents a certain subject. The finality of the laboratory is represented by a syncretic project which practically combines all the previous lessons so that students build a fully functional application.

The project will consist of an alarm clock with temperature sensing capabilities, able to display result both on the LCD and on a 7-segment display, having the configuration done from a control panel represented by a keyboard. The alarm clock application will also have serial communication capabilities.

In order to attend to this laboratory the students must have the following mandatory prerequisites:

- Strong C programming skills [1]
- Basic knowledge in the fundamentals of Electronics
- Capacity to interpret an electronic schematic

4.1.1 Provided materials

This laboratory will be oriented on introducing the basics of embedded development using various peripherals. The main component that the students will use is the Atmel ATMEGA16 microcontroller [14]. The microcontroller will be encapsulated on a header board which the students can easily use to connect to other external components. Another important component provided to the students is a project board-based student learning kit formally designed by Freescale, now currently maintained by NXP. The main advantage of this board is that it consists of a breadboard that can be used to build prototype circuits and a peripheral board which contains an important number of peripherals.

Furthermore, students will also have access to various peripheral devices such as: alphanumerical LCD, 7 segment display modules, serial communication interfaces, push buttons, 4x4 keyboard, LEDs and many more.

In this chapter, the following subsections will be reserved for the description of the modules that the students will be provided with.

4.1.1.1 ATMEGA16 Microcontroller, header board and debugger

ATMEGA16 is an 8bit MEGA-AVR microcontroller designed around a RISC architecture core surrounded by peripheral devices. The microcontroller has 16 KB of Flash memory available for code along with 1 KB of SRAM and 512 bytes of EEPROM memory. The main peripheral devices available in the ATMEGA16 microcontroller are:

- 2 x 8-bit timers
- 1 x 16-bit timer
- Real time counter
- 4 channel of PWM
- SPI interface
- UART interface
- Analog to Digital Converter
- 4 x 8-bit General Purpose Input Output Ports

Although this microcontroller has very low performance comparing to the existing microcontrollers currently present on the market it maintains its high didactical value thus being one of the most suitable microcontrollers for teaching. A strong argument to sustain this statement is that it only requires a power supply in order to run and it is available in 40 pin DIP capsule thus making it perfect for building small circuits on a breadboard. Another important advantage is that it can be clocked using an internal RC oscillator with a maximum frequency of 8MHz.

The pinout of ATMEGA16 is also very simple and well organized as presented in Fig. 4-1 [14]:

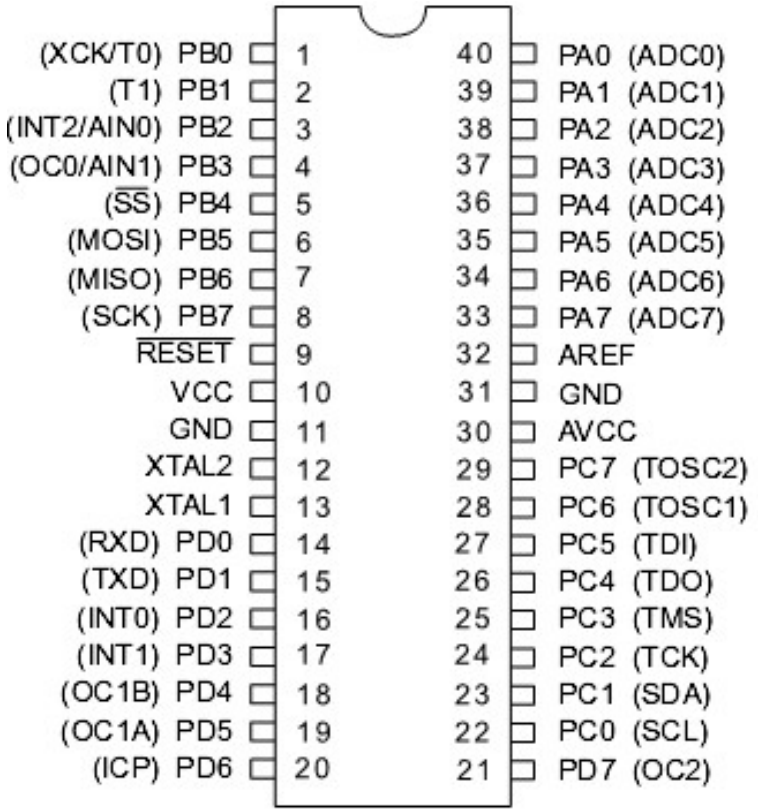

Fig. 4-1 Pinout of ATMEGA16

As it can be observed in the pinout, the microcontroller has 4 ports available for connections: PORTA, PORTB, PORTC and PORTD. Each pin is presented with its designated ranking in the corresponding port (ex. PB1 being line 1 from PORTB) along with its alternated function. A currently used practice in microcontrollers is to multiplex more functions on a pin thus reducing the number of pins in the capsule. In the case of ATMEGA16 the alternate functions of a pin are written in brackets. For example, pin PD0 is normally a GPIO pin belonging to PORTD but when the serial interface is activated the function of this pin changes to the RXD signal of the serial interface. Same rule is available for all pins. Special attention needs to be taken when using the lines of PORTA. In order for these to work, even in GPIO mode, power needs to be applied to the AVCC pin.

ATMEGA16 may be programmed either by using the ISP interface or by using a dedicated JTAG debugger. When using an ISP programmer the PINS involved in this operation are pins from 5 to 11. Practically, an ISP programmer needs access to the SPI interface of the microcontroller as well as to the RESET pin and, if needed, to the power supply related pins. A possible connection schematic for connecting an ISP programmer to ATMEGA16 through a standard 2x5 connector may be the following:

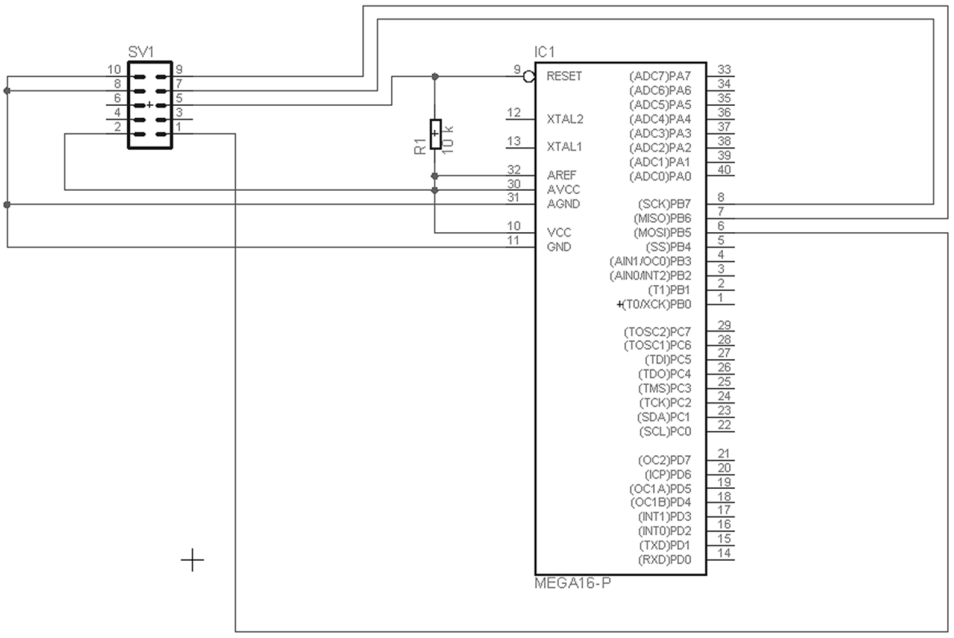

Fig. 4-2 ATMEGA16 ISP connection

The ISP programming of ATMEGA16 is limited only to downloading the executable code from the PC into the microcontroller's flash memory and programming of the Fuse Bits. No real-time debugging can be made using an ISP programming. In order to be able to debug a running code, in real time, on a microcontroller a JTAG debugger is usually required. The JTAG is connected to the microcontroller through dedicated pins. In the case of ATMEGA16 the dedicated pins for JTAG connections belong to PORTC from PC2 to PC5. It is important to mention that if the JTAG interface is enabled on the ATMEGA16 microcontroller these pins cannot be used by the programmer. These pins remain dedicated to the JTAG interface. The connection between a standard 2x5 pin JTAG connector and ATMEGA16 may be done as shown in the following figure:

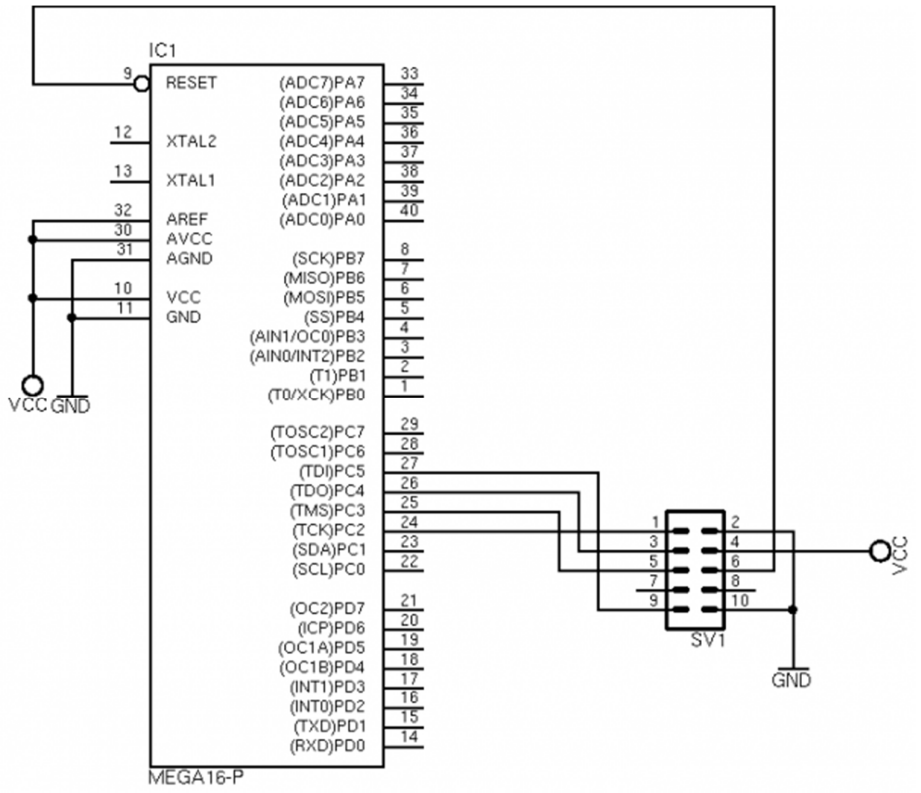

Fig. 4-3 ATMEGA16 JTAG connection

The JTAG that will be used for downloading the code to the ATMEGA16 microcontroller as well as for debugging the running code is the Atmel-ICE JTAG [15] which is supported by the new development tools from Atmel.

The enable/disable of the JTAG interface of ATMEGA16 as well as other critical settings of the microcontroller may be configured by accessing the so called Fuse Bits. These bits are practically represented by to registers which may only be accessed by a JTAG or ISP programmer. The Fuse Bits registers cannot be accessed from the running code from the FLASH memory and they are not visible to the programmer.

Using the Fuse Bits the following items may be configured:

- JTAG interface – it may be enabled or disabled
- ISP interface – it may be enabled or disabled
- Preservation of the contents of the internal EEPROM memory upon programming the FLASH memory
- Brown-out detector
- Clock source – various internal RC oscillator clock frequencies, external quartz oscillator, external clock source

4.1.1.2 ATMEGA16 header board

During this laboratory the ATMEGA16 microcontroller will be used along with a small header board which will not only export all the microcontroller's pins on header but will also contain a JTAG connector a quartz oscillator connected to the microcontroller. Such a board may be the following:

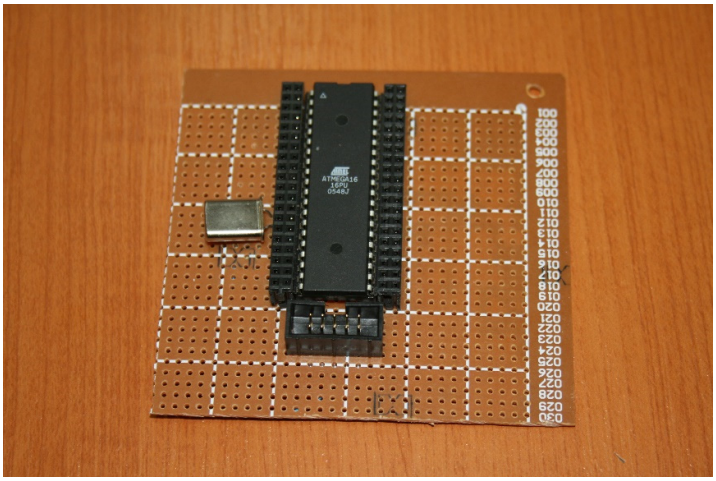

Fig. 4-4 ATMEGA16 header board

As it may be observed in Fig. 4-4, the microcontroller is surrounded by 2 female 2 line headers. Each pin from the microcontroller is directly connected to the corresponding pins near it. Practically all the pins from the microcontroller are accessible using the 2 line female headers. A block schematic of the header board may be found in Fig. 4-5:

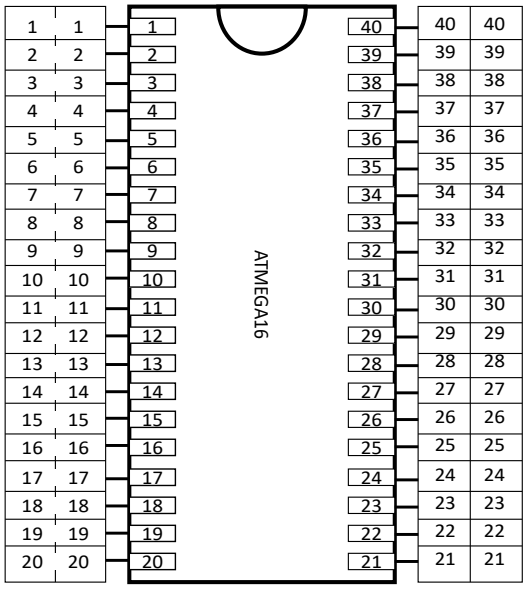

Fig. 4-5 ATMEGA16 header board block schematic

4.1.1.3 Peripheral board

The header board presented above will be interfaced with a peripheral board which will also be provided for de students during these laboratory assignments. The peripheral board, code name PBMCUSLK AXM-0392 [16] was initially designed by Freescale and now it is maintained by NXP. This board practically consists of an isolated breadboard which is only mechanically linked to an electronic board containing various peripherals from LEDs, pushbuttons, serial interface, LCD to various connectors as presented in Fig. 4-6

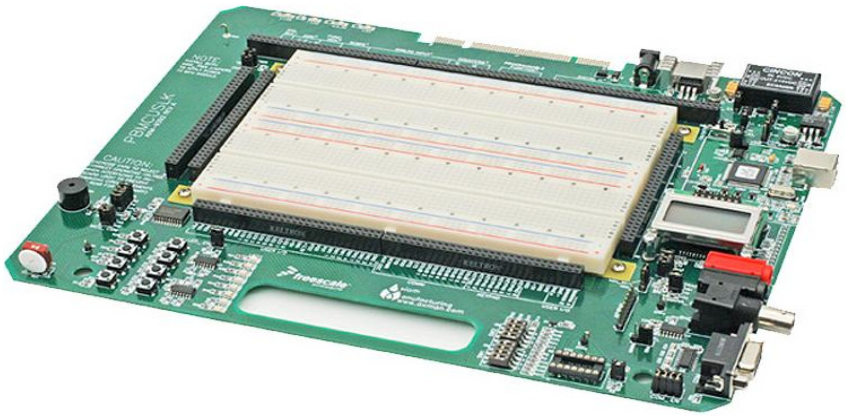

Fig. 4-6 PBMCUSLK peripheral board [16]

As stated before, the breadboard in the middle is only mechanically connected to the rest of the board. No electrical connections are made. In order to connect the peripherals on the PBMCUSLK board to a circuit being designed on the breadboard, the black female header surrounding the breadboard may be used. The significance of each pin in the header is written near the pin itself. No additional documentation is needed in order to use the board in basic applications. If needed, more information about the board may be find in its user manual [16].

4.1.1.4 Relevant documentation

Beside the present laboratory work manual, in order for the attendees to be able to process these laboratory works access to further documentation is needed. A list with some of the needed documents needed may be the following:

- Brian W. Kernighan, Dennis M. Ritchie - The C Programming Language [1]
- ATMEGA16 User Manual and Datasheet [14]

4.1.2 Laboratory applications planning

The main goal of these laboratory lessons is to introduce the students into basic low level embedded system design and development. The idea is to present to the students low level aspects without having libraries destined to control various peripherals internal or external to the microcontroller. These aspects construct the main reason why the lessons make use of a microcontroller which is small, easy to understand but with a great didactical value. The microcontroller is thus represented by Atmel's ATMEGA16. The lab sessions are somehow decoupled, each presenting a different peripheral device. The finality of this laboratory will be a project in which the attendees must construct a fully functional alarm clock using the provided peripheral devices.

The first laboratory work will be concentrated into introducing the students into the programming of ATMEGA16 using the provided materials. The developing tools provided by Atmel will be presented along with the structure of the necessary documentation. The practical aspect of this first laboratory work will be to implement a small LED blink application on the microcontroller. This laboratory work concentrates into presenting the output function of the GPIO system of the microcontroller.

The second laboratory work presents the input function of the GPIO system of the microcontroller as well as simple method for connecting a push button to the microcontroller and thus determining its state.

One of the most important peripherals of a microcontroller, the TIMER, will be the main aspect presented in the third laboratory work. Furthermore, the students will be introduced into basic aspects regarding the interrupt system.

The forth laboratory application will present another external peripheral device which is the 7 segment display digit. Most of this work will be concentrated on the design of the software part for controlling such a peripheral device.

In order to have more complex applications, other input devices are required in order to facilitate the user interaction with a system. Such a device, the 4 by 4 matrix keyboard, will be introduces during this laboratory work. The most important aspects which it will address are related to the methodology, both hardware and software, which will be used in order to work with the keyboard.

Another important peripheral device internal to the microcontroller which is frequently used is represented by the UART interface and this will be the content presented into the 6th laboratory work. Both transmission and reception will be address along with the combination with the interrupt system.

The 7th laboratory work is oriented into presenting a very popular alphanumerical LCD display. This laboratory work will be divided into 2 section. In the first section the students will have access to a simulator which provides means to manipulate the LCD's signals. This way, the students can easily understand the parallel communication protocol between the LCD and a host microcontroller. The simulator is web based and easy to use. The second part of this laboratory work will be to connect and control the LCD with the provided microcontroller. For this second part an already developed library for the LCD will be provided to the students.

The final laboratory work introduces the Analog to Digital Converter of the ATMEGA16. The first application that the students will have to implement will be a digital voltmeter using both the Analog to Digital Converter and the UART interface. The second application will be to connect an analogue temperature sensor the ADC of the microcontroller and the students will have to display the converted ambient temperature.

All these laboratory applications will be then concentrated into a project where the students will have to implement a fully functional devices which will be represented by a digital alarm clock with temperature sensing.

week	Laboratory work	Observations
1	Establishing laboratory groups	Establish groups of 2
2	Introduction + Laboratory work 1	
3	Laboratory work 2	
4	Laboratory work 3	
5	Laboratory work 4	
6	Laboratory work 5	
7		
8	Laboratory work 6	
9	Laboratory work 7	
10	Laboratory work 8	
11	Project	
12		
13		
14		

Table 8 Laboratory applications planning

4.2 Laboratory work 1 - First project: LED blink

This laboratory work presents to the students the first steps into developing and debugging applications on Atmel ATMEGA16 microcontroller with the aid of Atmel Studio 7 environment. As it is customary, the first application that is to be considered when beginning work on a new microcontroller, or even as first steps in embedded programming, is the LED blink application using software delays.

The presentation of this laboratory work will be divided from 2 points of view: a hardware point of view and a software point of view. The hardware part will present the necessary connections to be made in order to build the first LED blinking applications. The software part is responsible for presenting both the developing environment and the coding to be designed in order to build the application.

Before considering into analysis the schematic that should be implemented, the focus needs to go on the basic schematic of a microcontroller with emphasize on the GPIO module. A generic schematic of a microcontroller with a serious customization for the ATMEGA16 microcontroller which will be used for the laboratory works is presented in the following figure:

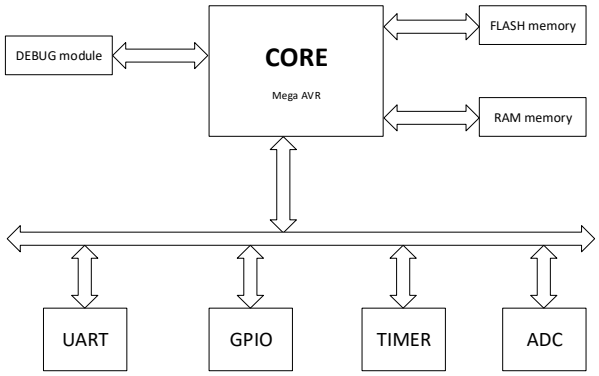

Fig. 4-7 General microcontroller block diagram

In the schematic above, the central piece of the microcontroller is represented by the CORE. This is practically an ALU, which has the only task of executing the code. Embedded into the silicon capsule along with the Core are the 2 usually present memories: the FLASH memory and the RAM memory. The FLASH memory is used to store the code that will be executed by the Core. The RAM memory is practically the Data Memory that will store the variable data of the code. Both of these memories are usually accessed directly by the Core. The Core is also connected via various busses to numerous peripheral devices. In our upper schematic we can identify peripherals such as the UART module, TIMER, Analog to Digital Converter (ADC) and the highly used General Purpose Input Output module.

The module, which presents great interest to our laboratory work, is the GPIO module. This module offers a collection of digital lines, organized in ports that have the advantage that they can be programmed to be able to establish a logic values on the line,

but also to be able to read the logic value of that line. Our current laboratory work focuses on using this module in order to implement the LED blinking application.

The first aspect to be discussed is the schematic that needs to be implemented to make the LED blink application. The block schematic is presented in Fig. 4-7

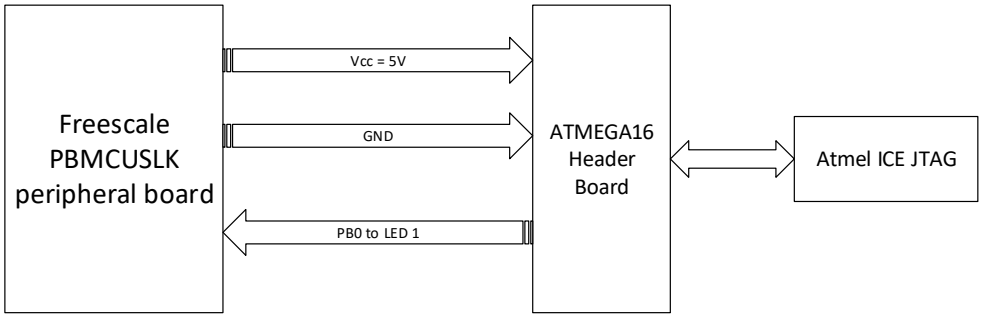

Fig. 4-8 LED blink connection block schematic

Having a more detailed analysis the necessary connections to be made are the following:

- Connect the 5 V power line from the header on the peripheral board to the correct pin of the ATMEGA16 header board (pin 10 on ATMEGA16)
- Connect the GND line from the header on the peripheral board to the correct pin of the ATMEGA16 header board (pin 11 on ATMEGA16)
- Connect one of the LEDs of the peripheral board (using the corresponding pin on one of the headers) to line 0 of PORTB (PB0) of ATMEGA16 from the ATMEGA16 header board
- Connect the Atmel ICE JTAG to the ATMEGA16 header board and to an USB port from the PC

ASSIGNMENT 1: Make the connections described above. Search the correct pins both of the peripheral board and on the ATMEGA16 header board. Have the laboratory teacher verify the connections before powering up the system.

In this first step, the hardware connections, represents the simplest task from this laboratory applications. The much more complex task is from a software point of view. Firstly, the primary steps into creating and configuring a new project into the developing environment Atmel Studio 7 will be presented. The starting point of the Atmel Studio 7 environment is presented in Fig. 4-9.

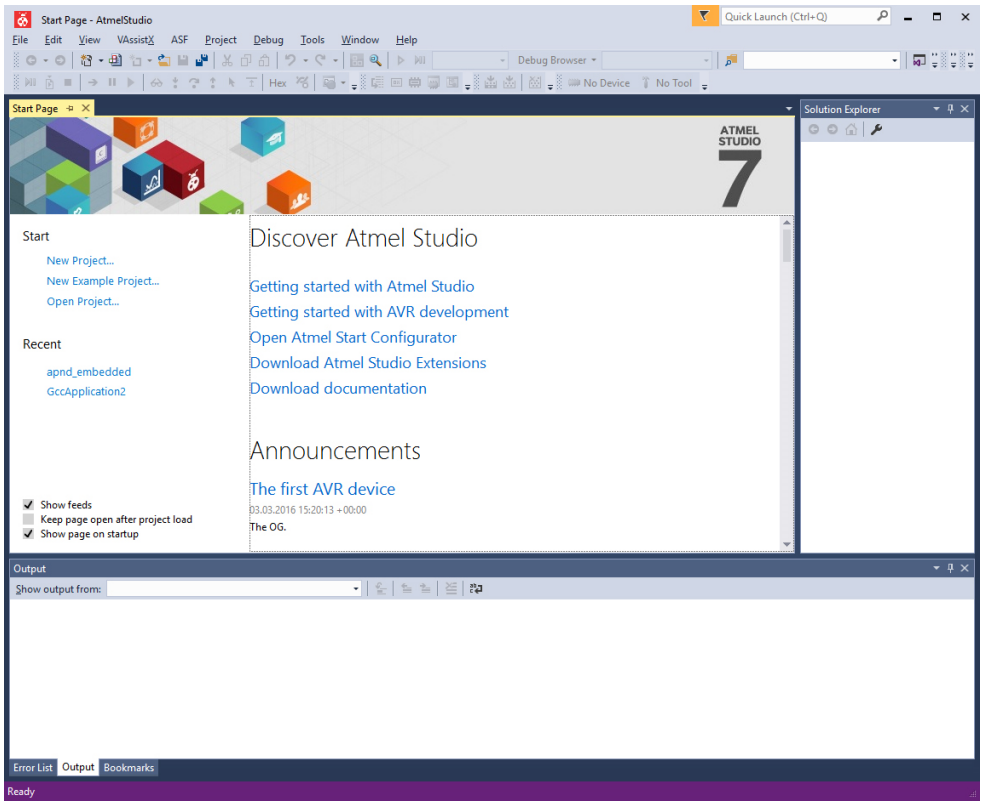

Fig. 4-9 Atmel Studio 7 starting page

To create a new project the “New project…” link found on the left side, as presented in Fig. 2-9, needs to be selected. The same behavior is available if using the menu bar: File -> New -> Project. The new project type that will be used is “GCC Executable Project”. Also, the location path and project name can be specified as presented in Fig. 4-10:

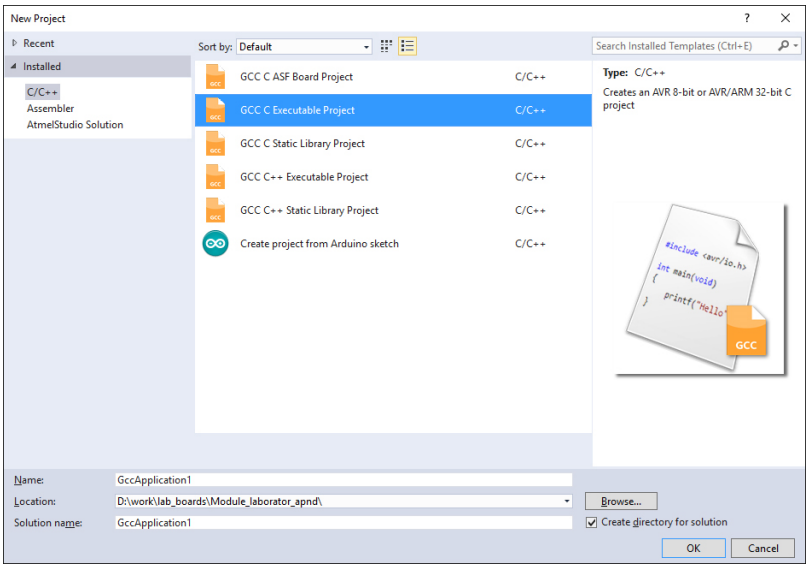

Fig. 4-10 Atmel Studio 7 new project dialog

The next step into creating a new developing project is to specify the microcontroller to be used. Select the Atmega16 device by searching it into the list of microcontrollers supported by Atmel Studio 7. In order to narrow down the search use the Device family combo box to filter the list for ATmega family. Such an example is presented in Fig. 4-11:

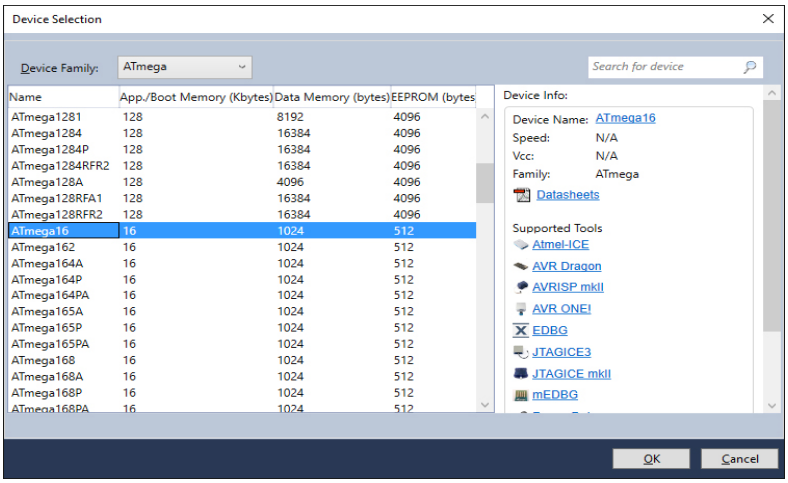

Fig. 4-11 Atmel Studio 7 microcontroller selection

After the project is successfully created, the development studio adds a template code file to the project containing only the main function. In order to view the project structure with the files referred by the project select the “Solution explorer” setting either by finding

it on tab in the right part of the application or by using the menu View -> Solution Explorer (hotkey CTRL+AL+L). This usually shows the solution explorer on the right of the applications as shown in figure:

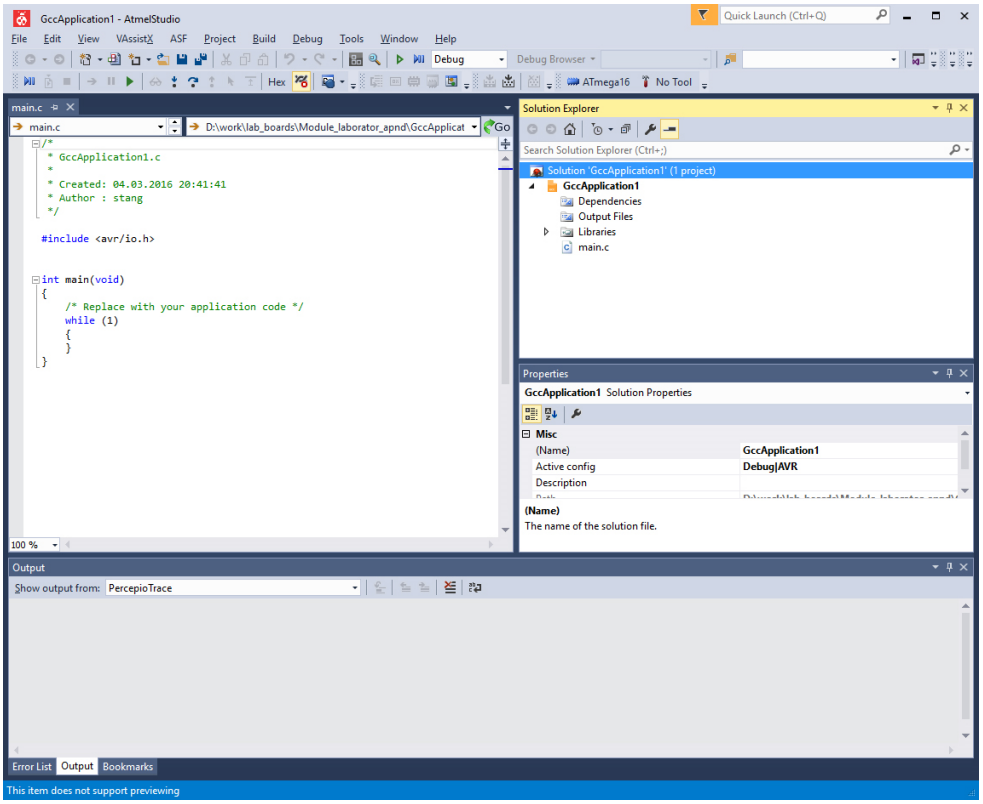

Fig. 4-12 Atmel Studio 7 with project created and solution explorer present

The next step is to configure some options of the newly created project. In order to avoid some issues during programming the compiler optimizations have to be disabled. There are many positive aspects when using compiler optimizations and in many situation are quite recommended. In our situation it is best to avoid the compiler optimizations mainly because we need to be concentrated on the functionality of the applications rather than on performance.

In order to access compiler optimizations a right click on the project is necessary (in our case on GccApplication1 in project explorer) with the selection of Properties in the right click menu. To reach compiler optimizations select Toolchain from the left and under AVR/GNU C Compiler select optimization. From the Optimization Level combo-box select (None –O0) for optimization level. A preview of the dialog for this issue is presented in Fig. 4-13:

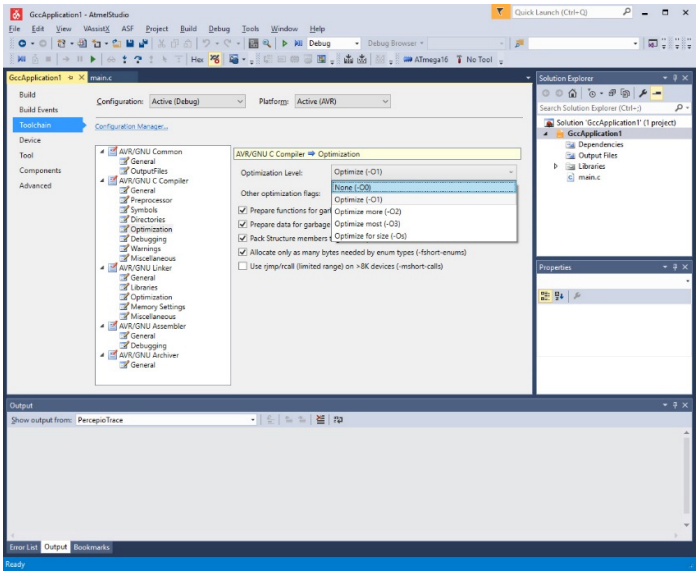

Fig. 4-13 Atmel Studio 7 compiler optimizations

The next important project configuration that needs to be taken care of is the selection of the Tool to be used for debugging. Having the previous screen we used to configure the compiler optimizations select Tool option from the left. The following screen should appear:

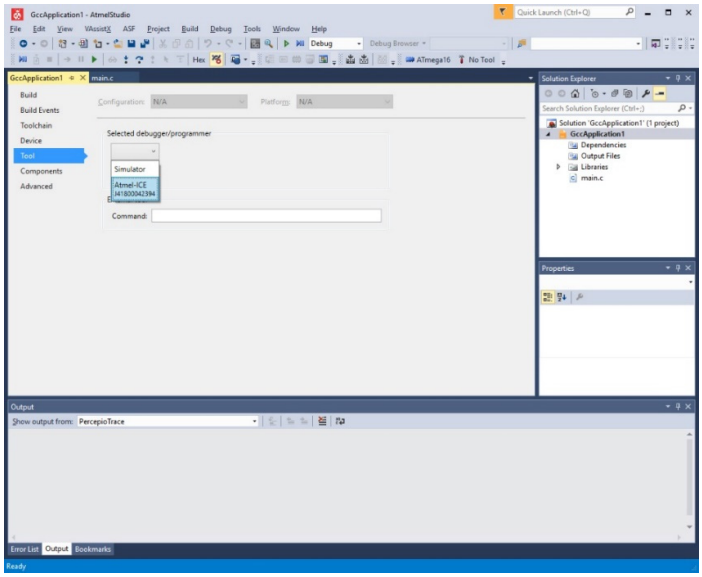

Fig. 4-14 Atmel Studio 7 tool selection

In Fig. 4-14, under the "Selected debugger/programmer" combo-box two options should be available (depending if the Atmel ICE debugger is connected via USB to the PC): the Simulator and the Atmel ICE debugger. The choice of this combo-box should not be permanent. If in any moment the student should want to use the simulator instead of the hardware JTAG debugger, he can do so by selection the corresponding options. The only observation is that in Simulator mode, the developing environment is disconnected from the target. In order to be able to download the code on the microcontroller and to debug it, the Atmel ICE debugger (in our case) should be selected.

When selecting the Atmel ICE debugger more options regarding this tool will appear on the same dialog. From the interface combo-box the JTAG option needs to be selected. Moreover, special attention needs to be taken on the value of the JTAG clock. A safe value to use would be 200 kHz, as the default value should be. An example of the settings that should be configured in this dialog is presented in the next figure:

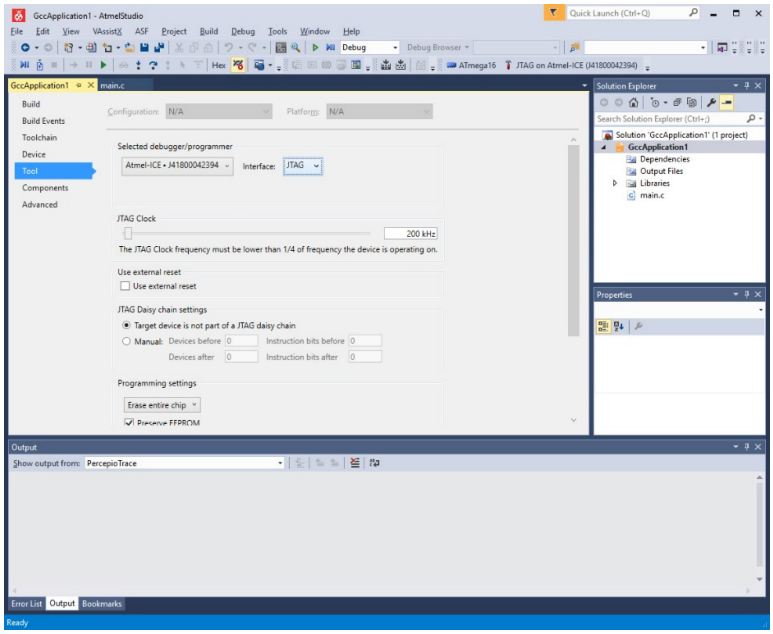

Fig. 4-15 Atmel Studio 7 tool selection and configuration

After these configurations are done, the project should be saved in order to use the same configured environment next time.

Another important aspect that needs to be discussed is how the target and the connection between the target and the JTAG should be tested. This testing method should usually be used before starting working with the target but usually only once at the beginning of it, if malfunctioning is detected. The testing method involved bringing up the Device programming dialog by selecting from main menu Tools -> Device Programming. The dialog that should be brought up is similar to the one presented in the next figure:

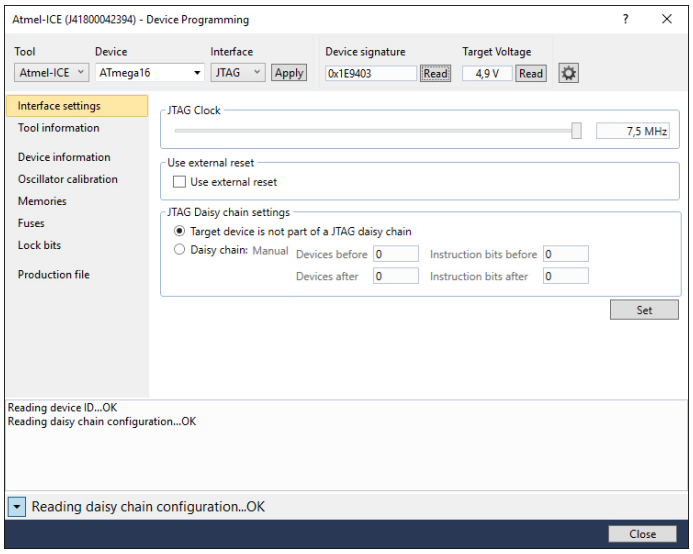

Fig. 4-16 Atmel Studio 7 Device programming dialog

In this dialog, mainly in the upper part, in the first opening, usually only the tool remains selected (as Atmel-ICE in our case). Having Fig. 4-16 as an example, in the "Device" combo-box select ATmega16 and on the interface, select JTAG if not already selected. After the selections press Apply and if the connections are in order than no error message should be displayed. In order to further verify the JTAG communication press read on the "Device Signature" region in order to read it from your ATMEGA16 microcontroller. A valid serial number should be read in case of a good communication. Moreover, in order to assure that the voltages are properly applied, the Target Voltage should be read by using the appropriate button. Having all of this information obtained, one can draw the conclusion that the JTAG communication with the target microcontroller is working properly.

Having all of this configured we can be assured that the project is suited for development. Note that this configurations should only be made once for the same project. Giving the fact that this laboratory tends to use a constructive, building, approach this newly created and configured project should be used for all the coming laboratory works.

Having the first project created in order to take it to running on target, it first must be compiled and built. This is done by accessing the menu Build -> Build Solution. The very used shortcut key for this operation is F7. The result of the compilation is presented in the Output tab on the bottom of the Atmel Studio 7 screen as shown in Fig. 4-17:

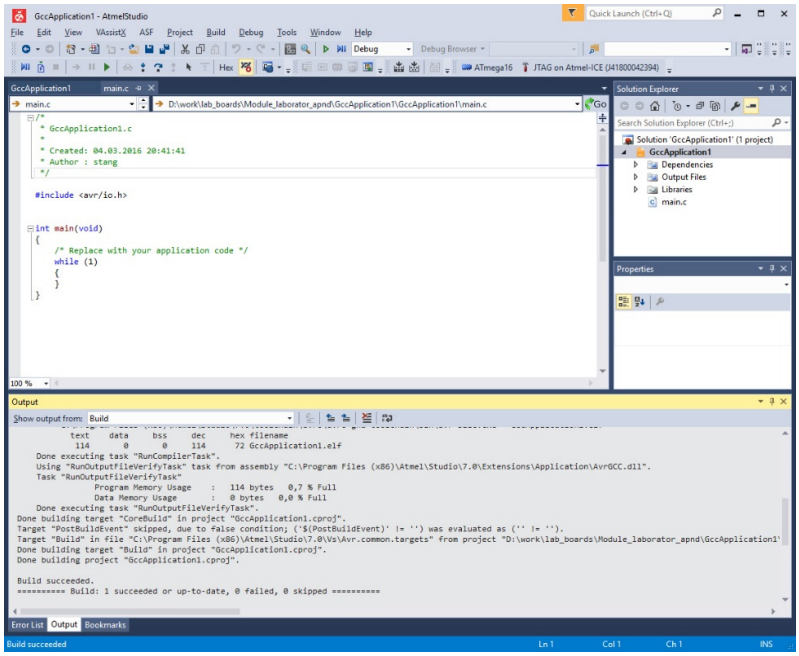

Fig. 4-17 Atmel Studio 7 Compilation result

Having a closer look on the output of the compilation in Fig. 4-17 we can observe not only the results on the last line but also, in case of successful compilation, the amount of code that the executable uses and the amount of data memory needed, with values in both bytes and percentage. The percentage is calculated relatively to the maximum amount of memory available for the currently selected microcontroller.

This information is important to an embedded developer not only to know if the code can fit the flash of the microcontroller or if the data memory is enough, but also to calculate the differences if compiler optimizations are used. Currently our only interest is knowing if the program fits the available memories.

The next step into developing our application is to write the necessary code. Taking a closer look at the generated code of the newly created project we can identify a very important include statement:

Code listing 4-1 Register definition header include

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

This line of code includes, into the code file it is written, the header file containing the definitions of register names of ATMEGA16. Beside this include, further includes should always be present into every ATMEGA16 project:

Code listing 4-2 Necessary includes

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
```

All the necessary includes that should be present in almost every file within an ATMEGA16 project are the ones presented above. The first include was detailed before. The second include header file contains the definitions of the interrupt vectors of ATMEGA16. The last include contains the file defining delay functions. In order for this included library to properly work the CPU frequency should be properly defined using:

Code listing 4-3 CPU Frequency definition

```
#define F_CPU 14745600UL
```

This definition "informs" the delay library of the frequency the processor is clocked by. In our situation, as presented above the clock frequency is 14.7456 MHz or 14754600 Hz.

A full list of the inclusions and definitions that should be present in manly all the files referring to the ATMEGA16 periphery can be the following:

Code listing 4-4 CPU Frequency definition

```
#define F_CPU 14745600UL

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
```

As stated before, the module that will be used in order to blink the LED, is the GPIO module.

The digital lines of the GPIO module are organized in ports. The ATMEGA16 microcontroller has 4, 8-bit lines wide ports (PORTA, PORTB, PORTC, PORTD). The main characteristic of a port line is the direction. For example if we need to drive a LED connected to a port line, meaning we would like to establish a high or low logical value in order to power on or off the LED, the line is considered to be an output line. In another example, if we want to read the logical value of a line, for instance when connecting a push button to a port line and wanting to read the state of a push button, the line is considered to be an input line.

Another important aspect is the one related to how a port line can be driven, when it is an output port line, or how it can be read, when it is configured as an input line. This, and the configuration of the direction of a port line can be accessed through a collection of registers. All the registers have the same structure, as in Fig. 4-18, but with different meaning. Every port of the microcontroller has the same collection of registers. Each bit of the register controls the "characteristic" of the corresponding digital line of the port.

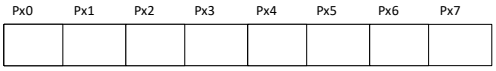

Fig. 4-18 ATMEGA16 General GPIO register structure

The above structure is available for all the registers corresponding to the GPIO module of ATMEGA16. The first aspect to be analyzed, as discussed above, is represented by the direction of a port line. As stated in the documentation of ATMEGA16 the DDRx registers are related to this aspect as that writing a logic 1 to one bit of this register configures the selected line as an output line. Also, writing a logic 0 to one bit of this register configures the corresponding line as an input line. The DDRx registers (where x identifies the port: A, B, C or D) have the same structure as in Fig. 4-18.

Another important set of registers for the GPIO module of ATMEGA is the set of PORTx registers (where x identifies the port: A, B, C or D). These registers are only to be used for the lines which are configured as output. These registers define the logical value that the line has. For example, writing a logic 1 on one of the bits of a PORTx register, the corresponding line of port x is driven to logic 1. Same algorithm applies for writing a logic 0 value.

A register set related to the PORTx register set presented above is the PINx register set. The PINx registers are used when working with input lines. From these registers we can determine the logical value of a line (pin) of a microcontroller. Reading this registers practically offers the logic state of a line or pin configured as input. For example, when reading logic 1 of a bit in a PINx register the corresponding pin has a logic 1 value applied on it. Some goes for 0 logic.

More information about how the GPIO system works may be found in the ATMEGA16 datasheet on chapter named I/O Ports [14]

ASSIGNMENT 2: Read the ATMEGA16 documentation and find the registers that have to be configured in order to drive a LED connection to line PB0 of the microcontroller. Establish and explain the values to be written in the registers.

A pseudocode implementation of a program that blinks the LED can be the following:

Code listing 4-5 LED blink main program flow

```
void main()
{
  init_led(); // initialize the port direction using  DDRx register
  while(1)
  {
     led_on(); // turn on led, logic 1 on coresponding bit from PORTx register
     delay(); // delay loop
     led_off(); // turn off led, logic 0 on coresponding bit from PORTx register
     delay(); // delay loop
  }
}
```

The init_led pseudocode function represents the configuration of the directions of the pin the LED is connected to. The led_on and led_off represent the code lines needed to drive the pin the LED is connected to in order to turn the LED on or off.

The delay function may be implemented using the provided library functions with the following prototypes:

Code listing 4-6 Delay functions prototypes

```
void _delay_ms(int milliseconds);
void _delay_us(int microseconds);
```

ASSIGNMENT 3: Write a microcontroller program that blinks a LED with a period of 500 ms using delays.

4.3 Laboratory work 2 - Push buttons

The previous laboratory work was aimed at making the first steps into embedded programming. As an application a simple LED blink example was used. In order to do this the GPIO system was presented, but only for output. This laboratory work also concentrates on the usage of the GPIO system but for input. The best application for the demonstration of the input function of the GPIO is the read of a push button.

The schematic that needs to be implemented in order to connect a push button to a GPIO port of a microcontroller is relatively simple. Besides a push button and a microcontroller, only a resistor is needed to complete the schematic as presented in the following figure:

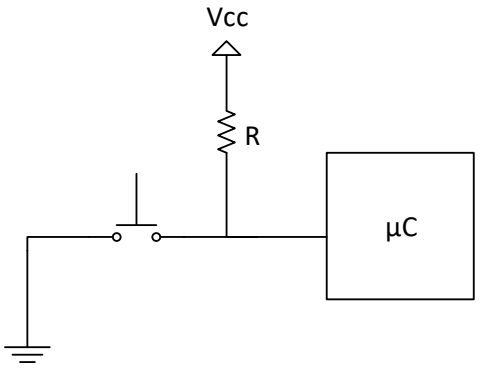

Fig. 4-19 Push-button connection to microcontroller GPIO port

As it can be observed in Fig. 4-19 a pull-up resistor is used to pull the line between the microcontroller and the push button to logic 1 value. Analyzing the schematic we can state that when the push button is not pressed the logic level of the input GPIO line of the microcontroller is logic 1, mainly because the voltage is Vcc. The resistor is needed in

order to limit the amount of current flowing through the microcontroller. This resistor is mandatory and without it the microcontroller may be damaged.

When the push buttons is pressed the current flows through the resistor into ground thus the other terminal of the push button is connected directly to ground. This flow will establish a near to 0 V voltage value on the line which will be seen as logic 0 by the microcontroller.

In conclusion, when the microcontroller sees a logic 1 value on the input line the push button is not pressed and when it sees a logic 0 value the push button is pressed.

The schematic presented in Fig. 4-19 is already implemented on the peripheral board, except the connection with the microcontroller. There are 8 buttons available on the peripheral board and they are exported to one of the headers surrounding the breadboard in the middle. The buttons are denotes as PB1, PB2… PB8. The pull-up resistor is embedded on the peripheral board so the only connection that needs to be made is the connection between the buttons on peripheral board and the ATMEGA16 header board.

So, in the next step let's choose, for example, button PB1 on the peripheral board and connect it to the GPIO line PA6 on pin 20 of the ATMEGA16 header board. This implies that we need to search where the PB1 button is mapped on the peripheral's board headers. Additionally, we should maintain the connection previously made in where a LED on the peripheral board was connected to the ATMEGA16 header board on line PB0, pin 1 of ATMEGA16. The LED will be used in order to test the functionality of the button. The result schematic of this current laboratory work can be summarized as following:

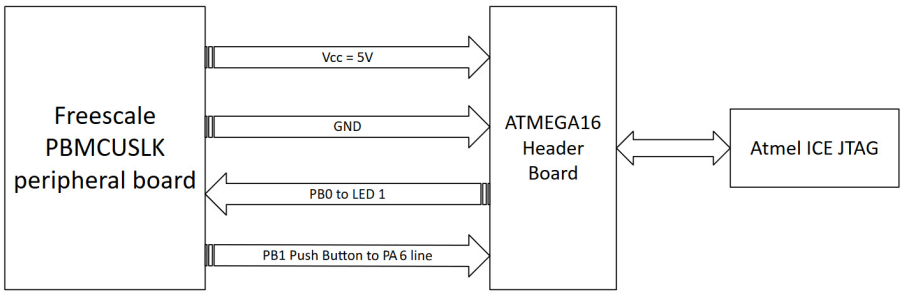

Fig. 4-20 Push button connection block schematic

ASSIGNMENT 1: Make the connections described above. Search the correct pins both of the peripheral board and on the ATMEGA16 header board. Have the laboratory teacher verify the connections before powering up the system.

As discussed in the previous laboratory work each port line may be configured to be an input line or an output line. The output configuration was used in the previous laboratory work in order to control the LED. In the output mode, the programmer may control the logic value of the specified line.

The input mode of the GPIO line may be used not to control the logic value of the line but to read the logic value applied to that line from external devices. This way the programmer may use to interface in a basic way with the external environment.

In order for a GPIO line to be used as an input line the direction needs to be set first. As presented in the previous laboratory work the DDRx (where x may be the port identifier A, B, C or D) registers are responsible for the pin direction. Each bit of this register control the direction of the corresponding line of that port. A logic 1 value written for a bit of the register sets the direction as output for the corresponding GPIO line. A logic 0 value written for a bit of the register sets the direction as input for the corresponding GPIO line.

After the direction of the line has been set the programmer may easily read the status of the desired input pin using the PINx registers. The structure of the PINx registers is identical to the structure of DDRx or PORTx. The role of the PINx registers is that it reflects the input logic value that was read from the GPIO pin corresponding to the bit of the register. If the pin has a 0 logic value applied on it, then the corresponding bit in the PINx register will be read as 0. If the pin has a 1 logic value applied on it then the corresponding bit in the PINx register will be read as 1.

In our case, in the end, when the corresponding bit in the PINx register we will use has the value equal to 1 then the push button is not pressed and when the bit is read as 0 then the push button is pressed. This is practically how the embedded programmer may detect a push/release of a button.

So, practically, a simple pseudo code application example on how to read the state of a push button could be the following:

Code listing 4-7 Simple Push button application pseudo-code

```
int main(void)
{
  …
  init_buttons();
  while(1)
  {
     …
     if ((PINx & (1 << b)) == 0) // if button_pressed
     {
            // some code to do when button is pressed
     }
     …
  }
  …
}
```

The code above could work very well but only in an ideal environment using ideal push buttons and other parts. In the real life such a code may give false pressings of the push button. The only reason why this may happen is because the push button is a mechanical part which is extremely unpredictable and unstable. The transient mechanical processes when pushing/releasing the button may give many false contacts. When

pressing a push button, the actual signal that reaches the microcontroller's pin could like the following:

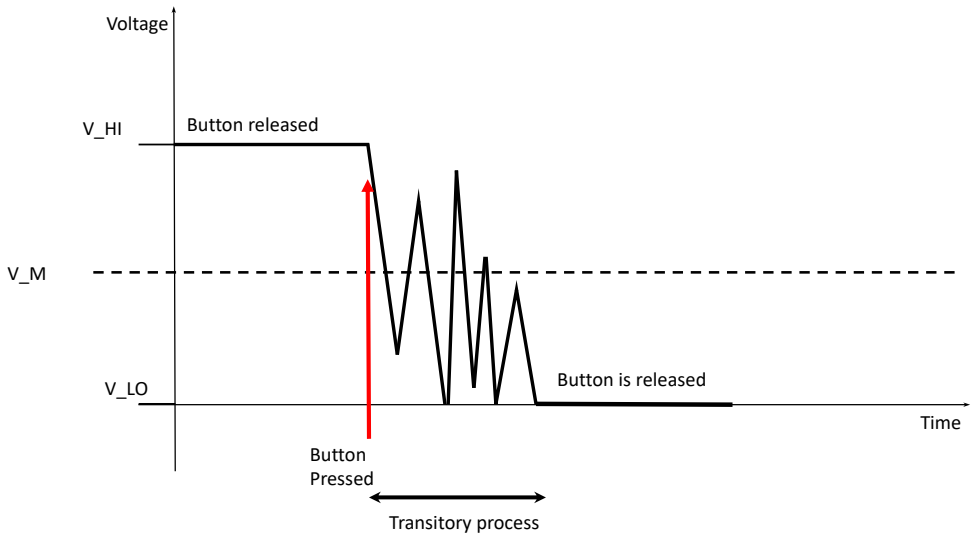

Fig. 4-21 Push-button electrical transient process

In Fig. 4-21 before the red arrow the button is not pressed thus the line of the microcontroller is set at the high voltage value as logic 1. In the moment represented by the red arrow the user presses the button. In that moment the value of the voltage should drop in order to reach 0 logic value. This process is not instantaneous and lots of variations occur because of the mechanical parts of the push buttons. These parts induce a transient process into the line. The problem with this situation is that the signal crosses the middle value of the voltage that separates the logical values detected by the GPIO line of the microcontroller. This leads to the fact that the microcontroller may detect not one but more transitions from logic 1 to logic 0 and back to logic 1. Using the pseudo-code example in Code listing 4-7 the program running on the microcontroller may sense multiple pressings of the push button. This happens mainly because the signal on the pin bounces between the high and low logical values. In order to eliminate this inconvenience the bouncing between the values have to be ignored until the signal is stabilized. The transient process cannot be fully eliminated electrically but it can be compensated by software in order to ignore the signal bounce.

The technique used to compensate the signal bounce is called de-bounce and consists of a trivial algorithm. The idea is to wait an amount of time until the signal is stabilized immediately after the microcontroller senses the first time the signal drops from one logic value to another, in our case from logic 1 to logic 0. The amount of time to wait for the transient process to be over is usually established empirically. It may vary from hundreds of microseconds to tens of milliseconds. In most cases a wait time of maximum 1 millisecond should suffice.

Code listing 4-8 Push button application pseudo-code with de-bounce

```
int main(void)
{
  …
  init_buttons();
  while(1)
  {
     …
     if (button_pressed) // if button_pressed
     {
             delay(1 millisecond max);
             if (button_pressed)
             {
                // some code to do when button is pressed
             }
     }
     …
  }
  …
}
```

The idea in Code listing 4-8 is to wait for the transient process to be finished when sensing a button is pressed. After the wait is over, the programmer should check again if the button is pressed in order to be sure that there was a real pressing of the button by the user and not a parasitical press.

The first application that we will implement is a simple LED switch. The push button will be used as a switch. One press of the push-button will power on the LED and the next press will power off the LED. Practically each time the button is pressed the LED is toggled: if the LED is power on then it should be power off and if the LED is power off then the LED should be powered on. The LED state should be kept the same between button pressings.

A preliminary pseudo-code of such an application that also integrated de-bouncing could be the following:

Code listing 4-9 Preliminary pseudo-code LED switch application

```
int main(void)
{
  …
  init_buttons();
  init_leds();
  while(1)
  {
     …
     if (button_pressed) // if button_pressed
     {
             delay(1 millisecond max);
             if (button_pressed)
             {
                // switch on or off the led depending of previous state
                led_toggle();
             }
     }
  }
}
```

Our current application states that the LED should be switched on or off depending on its previous state when a user presses the button. The code presented above works but not clearly as intended. If we analyze the code above we can first find that while the button is pressed by the user, the LED is toggled several times and after the user releases the button, the state of the LED is unclear, it is unpredictable. This happens because the user (human) is much more slower than the microcontroller running at 14.7456 MHz which leads to a lot of executions of the code inside the last if statement several times, not only one time, as intended. A very simple solution to this issue is to introduce another line of code that waits until the user releases the button before toggling the LED. An adapted version of the code snippet above may be the following:

Code listing 4-10 Preliminary pseudo-code LED switch application

```
int main(void)
{
  …
  init_buttons();
  init_leds();
  while(1)
  {
     …
     if (button_pressed) // if button_pressed
     {
             delay(1 millisecond max);
             if (button_pressed)
             {
                while(button_pressed); // wait for user to release button
                // switch on or off the led depending of previous state
                led_toggle();
             }
     }
     …
  }
  …
}
```

ASSIGNMENT 2: Based on the explanations presented above implement the LED switch application on the microcontroller. The applications role is to switch on/off the LED when pressing the push-button.

ASSIGNMENT 3: Connect all the LEDs of the peripheral board to the ATMEGA16 header board along with 2 push buttons. Implement a small application which will move the LED light from one LED to another by pressing one the buttons. One button should move the LED up, the other one should move the LED down. Only one LED should be on at a time. When the LED turned on is in one of the edges rounding should be applied.

Example situations:

- When LED1 is on and the user pressed the button down, LED1 should be turned off and LED2 should be turned on
- When LED2 is on and the user pressed the button down, LED2 should be turned off and LED3 should be turned on

- When LED3 is on and the user pressed the button up, LED3 should be turned off and LED2 should be turned on
- Regarding rounding:
    - When LED8 is turned on and the button down is pressed, LED8 should be turned off and LED1 should be turned on
    - When LED1 is turned on and the button up is pressed, LED1 should be turned off and LED8 should be turned on

The state of the LEDs should be stable between the button presses by the user.

4.4 Laboratory work 3 - Timer, compare match, interrupts

The TIMER is one of the most important peripheral module of a microcontroller because of its high applicability. The main role of a timer module is to generate high accuracy, periodical or non-periodical events. Another important role is to measure time intervals of external events and also to count the occurrences of the events.

Because of its importance the timer module will have a dedicated laboratory work. Furthermore, this laboratory work will also be concentrated on introducing interrupts. The finality of this laboratory work will be to blink the LED with a desired accurate frequency.

In Table 8 Laboratory applications planning
Laboratory work 1 - First project: LED blink, the same kind of application was implemented but using software delays. A simple time analysis experiment may reveal the fact that in the first laboratory work the LED blink was not exactly accurate. Using an oscilloscope, one can identify that the LED blink interval is not always the same due to software unpredictability. This drawback may be overcome by using the TIMER module.

Before presenting the TIMER module of ATMEGA16 a short introduction into the general idea of a TIMER is needed. The base component of a TIMER is a counter. It is important to mention that the timer and counter are not the same. The counter is the basic component of a timer but nothing more. The counter is a sequential circuit which is practically a register with an additional logic that allows its content to be incremented and decremented [23] upon the arrival of a clock edge. A block schematic of a counter can be found in the following figure:

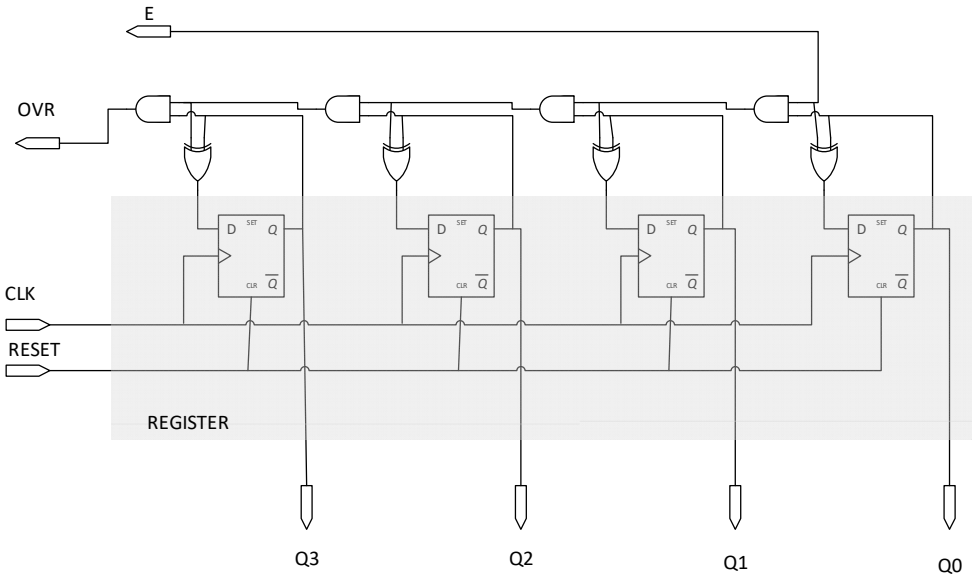

Fig. 4-22 Counter block schematic

As it can be seen in the previous figure the main component of a counter is a register (which is made out of D type flip-flops). Above the register, additional logic is added which changes it into a counter. On each edge (rising or falling) of the clock signal, the content of the register is incremented. The initial value is usually zero which is loaded into the register upon reset signal is applied. The value of the counter can be read via Q0-Q3 terminals [24]. Practically in Fig. 4-22 a 4-bit counter is presented.

One of the most important signals presented in the above figure is the OVR signal which is set to logic 1 once the counter has overflowed. An overflow of a counter appears when the maximum value of the register is reached and it is increment by one.

The maximum value that can be stored in a register which is *n* bit wide is:

$$N = 2^n - 1 \tag{4-1}$$

Having the maximum value of the data stored in an *n* bit wide register and the frequency *F* that is applied to the counter, the time needed for the overflow to occur (the time needed by the counter to count from value 0 to the maximum value) may be:

$$t_{OVR} = \frac{N}{F} = \frac{2^n - 1}{F} \tag{4-2}$$

The OVR signal may be used to generate periodic interrupt events but it is not configurable. The t_{OVR} parameter is dependent on the counting frequency *F* and the width of the register. Only the counting frequency may be modified in order to modify the overflow period but it is not enough, thus it cannot be used to obtain a user defined time period.

In order to be able to obtain a user defined time period with high precision, additional components and logic needs to be added to the classic counter module, thus obtaining the basic schematic of the timer having compare match features.

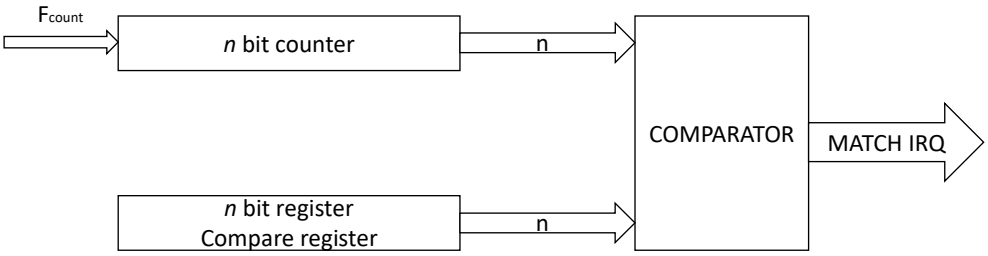

Fig. 4-23 Timer with compare match block schematic

In Fig. 4-23 a simple version of a timer is presented emphasizing the compare match feature. As it can be observed, the main component is a counter register which has the F_{count} frequency as input. The width of the register is of *n* bits. In the presented schematic another register is connected, thus the one designated as *Compare register*. This register contains a value preprogrammed by the developer which is compared using the Comparator at each clock edge (positive or negative). The moment the counter register contains a value equal to the value in the compare register, an event is generated by the comparator module. This event can be used as an interrupt event in order to generate user defined time period with very high precision. The only unknown term in the above schematic is the value, which we will designate as *N*, which should be written into the Compare register in order for our timer to generate the desired time period. This value may be calculated as follows:

$$N = t \cdot F \qquad (4\text{-}3)$$

Where *t* is the desired time period and *F* is the input count frequency applied to the timer.

It is important to mention that the above schematic is highly simplified in order to be easier to explain the basic functionality of the timer module. Other features could be added into this context:

- The counter could be reset when a compare match interrupt has occurred
- The counter could be restarted from zero when a compare match interrupt has occurred
- An additional frequency divider could be added in order to divide the input frequency of the time

These features are usually present in a timer module inside a microcontroller.

The ATMEGA16 microcontroller has two 8 bit wide timer/counter modules and one 16 bit wide timer/counter module. For the purpose of this laboratory work TIMER1 of ATMEGA16 will be used. This is the 16 bit wide timer of ATMEGA16 and the compare match function described above is implemented and denoted as CTC (Clear Timer on

Compare match). Beside the CTC operating mode, TIMER1 also supports other operating modes like PWM, free run, input capture.

The input frequency of the timer is the peripheral clock which is practically the global clock of ATMEGA16, in our case it will be the frequency generated of the quartz oscillator.

The first application that we will discuss is an application where we will want to blink the LED with a chosen, fixed and accurate period. We will consider a period of 100ms and the fact that ATMEGA16 is clocked using a quartz crystal with on oscillation frequency of 14.7456MHz. As stated above, for such an application, TIMER1 of ATMEGA16 will be chosen thus it has the CTC operating mode and is 16 bit wide. Moreover, the CTC operating mode will be used along with the Compare Match Unit A where the compare match register is represented by OCR1A (the top counting value of the counter will be OCR1A). The OCR1A register is 16 bit wide and is accessible by using 2 registers that split the most significant 8 bits from the least significant 8 bits (practically splitting it in half). In this case the OCR1A value is formed by using register OCR1AH for the most significant byte and OCR1AL for the least significant byte. Having this in mind, the value of the compare match must not exceed 65535 (it must fit within 16 bits).

The compare match value that needs to be written into the OCR1A register pair may be calculated using (4-3) where N is the OCR1A value, t represents the desired period of time (in seconds) and F is the input frequency of the timer (in Hz).

The configuring of the timer must first start with the operation of writing the correct value of the OCR1A register pair.

The next steps in configuring the timer are related to the registers TCCR1A and TCCR1B. The first aspect that can be configured using these registers is represented by the Waveform Generation Mode which may be specified through bits WGM13, WGM12, WGM11 and WGM10 which are split between the TCCR1A and TCCR1B registers. The available Waveform Generate Modes for TIMER1 are presented in Table 46 of the ATMEGA16 documentation, page 112 [14]. Regarding our application, the suitable Waveform Generation Mode should be mode 4 which implies the above presented bits to have the value 0100.

Another important aspect is the configuration of the timer's clock source. It is important to mention that the moment the timer has its clock source configured it immediately starts counting, thus this should be the last configuration operation to be made. Information on what clock sources are available for TIMER1 can be found in Table 48 of the ATMEGA16 documentation, page 113 [14]. In our case the internal clock source will be used which is represented by the global frequency of the microcontroller, more exact, the frequency of the oscillator, 14.7456MHz in our case. Before being applied to the counter submodule of the timer, the clock source may be divided. A number of 5 prescaler values are available: 1, 8, 64, 256 and 1024.

The actual prescaler value has to be chosen in concordance with the calculated value of the OCR1A compare match value. If the calculated value does not fit into a 16 bit number then a higher prescaler value needs to be used and the value recalculated.

Let's use, for example, the first prescaler, which is 1. Practically in this case there is no division applied to the input clock signal, thus the frequency applied to the timer remains at 14.7456MHz. Considering this and applying formula (4-3) the resulting value for the OCR1A compare match register pair could be:

$$OCR1A = 0{,}1 \cdot 14.745.600 = 1474560 = (16\ 8000)_H \quad (4\text{-}4)$$

As it can be observed, the calculated value (represented in both decimal and hexadecimal format) does not fit into the 16 bit register pair. A solution to this issue would be to lower the clock frequency in order to lower the compare match value by applying a prescaler. Let's continue the algorithm and choose the third prescaler which is 64. In this case we would obtain the following clock frequency:

$$F = 14.745.600\ Hz\ \div 64 = 230.400\ Hz \quad (4\text{-}5)$$

Using the value of the frequency calculated before in (4-5) and reapplying formula in (4-3), the resulted value for the compare match OCR1A could be:

$$OCCR1A = 0{,}1 \cdot 230.400 = 23.040 = (5A00)_H \quad (4\text{-}6)$$

The value we have obtained in (4-6) can easily fit into the 16 bit wide OCR1A register pair. As stated before the value of OCR1A is divided into 2 registers OCR1AH (for the most significant part of OCR1A) and OCR1AL (for the least significant part of OCR1A). A short pseudo-code on how such a value may be written correctly into the 2 registers may be the following:

Code listing 4-11 Dividing match value into most and least significant parts

```
void timer_init()
{
  // some code
  unsigned int value = 0x5A00;
  // some code
  OCR1AH = (value >> 8) & 0xFF;
  OCR1AL = (value & 0xFF);
  // some code
}
```

The idea behind the algorithm is to obtain the most significant part of the number by using bitwise operations like shifting and applying masks. For example the most significant part of a number may be obtained by right shifting it with the necessary number of bits (8 in our case) and applying an isolation mask.

In conclusion the correct value for the OCR1A compare match register pair has been calculated and the according prescaler value has been chosen. These operations should suffice in configuring the TIMER1 module of ATMEGA16.

The only aspect remaining to be discussed is how the programmer can find out when a compare match event has occurred. This can be obtained either by using the interrupt

system and configuring the timer accordingly or by using the flag register TIFR of the timer module. Both of these solution will be presented during this laboratory work.

The first solution that will be used is the one based on polling the correct flag in TIFR register. The flag responsible for the compare match event on OCR1A is bit 4 denoted as OCF1A. When this bit is read as 1, the compare match event has occurred. After the programmer detected such an event he must then reset this flag by writing logic one on it as stated in the documentation of ATMEGA 16 on TIFR register at pages 115 and 116.

An example code of the polling method may be the following:

Code listing 4-12 Timer compare match polling method

```
int main(void)
{
  // some code
  // initialization code
  while(1)
  {
     if (((TIFR & (1 << 4)) != 0)
     {
            // compare match on channel 1 has occurred
            // code to be done on this event
            TIFR |= 1 << 4; // reset flag
     }
  }
  // some code
}
```

ASSIGNMENT 1: Write a microcontroller software that blinks a LED with a period of 500ms with high accuracy using the TIMER1 module. The timer related functions should be written inside a library (which, for example, is made out of a timer.c code file a timer.h header file). Information on how a C library has to be written check the The C Programming Language manual by B. Kernighan and D. Ritchie [1]. The LED should be blinked within the infinite while loop in the main function using a polling method of the capture event. Test your program using the hardware connections in Table 8 Laboratory applications planning
Laboratory work 1 - First project: LED blink. Validate your program by measuring the LED blink period using an oscilloscope.

The second part of this laboratory work is dedicated to presenting the general concept of an interrupt system mainly focused on the interrupt system of ATMEGA16.

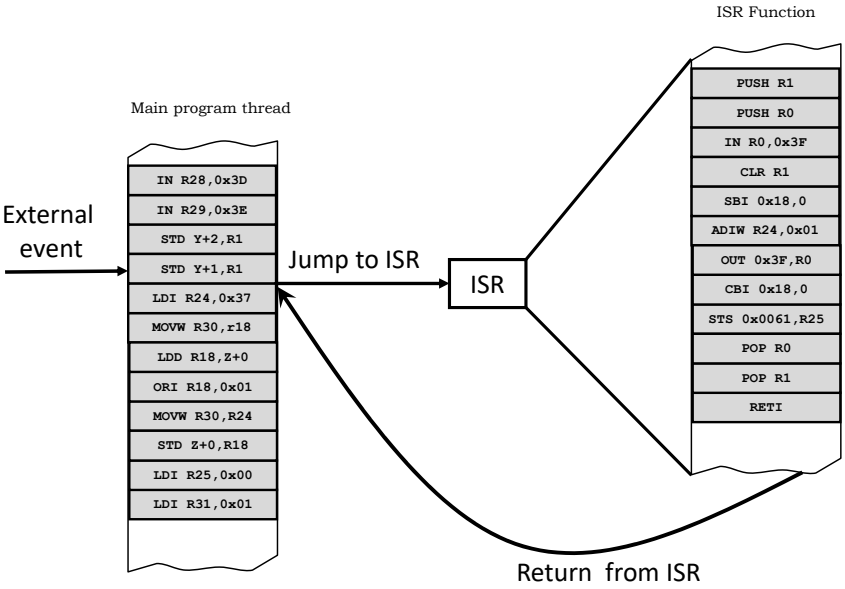

Fig. 4-24 Interrupt diagram

In Fig. 4-24 a diagram on how an interrupt process works is presented. On the left side, the main program thread is presented. This code can be seen as the main program that is running on the microcontroller. It is presented in ATMEGA16 assembly language in order to simplify the explanations. As it can be observed, the microcontroller's core receives an external event during the execution of instruction 4. Usually the instructions have an atomic execution which means that the execution of an instruction cannot be interrupted.

Immediately after the current instruction finishes its execution the core senses the external event and an interrupt is generated. This interrupt implies the fact that the core freezes its execution and jumps to an interrupt service routine function which has the necessary code to treat the external event. The core then starts the execution of the interrupt service routine's code as it can be observed on the right side of Fig. 4-24. The interrupt service routine function has a RETI (Return from Interrupt) instruction at the end of the code. This marks the ending of the interrupt service routine function and the core will return from the call back to the point where it was interrupted. After the return from the ISR function the core resumes its execution.

The interrupt service routine is merely a function, like any other function written by the programmer, which contains the necessary code for treating the external event.

This whole process may be seen as a classic function call with the difference that the core is the caller of the function, not the developer. An interrupt service routine thus is only called by the microprocessor's core when an external interrupt is generated. The developer does not need to call this function. The developer only needs to preconfigure this function into the interrupt controller of the microcontroller using dedicated API functions and/or registers.

The above method has some clear disadvantages. One important drawback is that only one interrupt service routine may be defined, thus even if there are more than one interrupt sources only one ISR function may be defined. In this case the programmer will have to differentiate the interrupt source by writing appropriate code. This can be easily resolved by using the concept of vectored interrupts.

In case of vectored interrupts, upon the arrival of an external event, the core doesn't jump to a fixed ISR routine. The core jumps to a lookup table placed in memory where it chooses the ISR routine address based on the source it was interrupted by. Using this approach for different interrupt sources different interrupt service routine functions may be defined. This concept is presented in the following diagram:

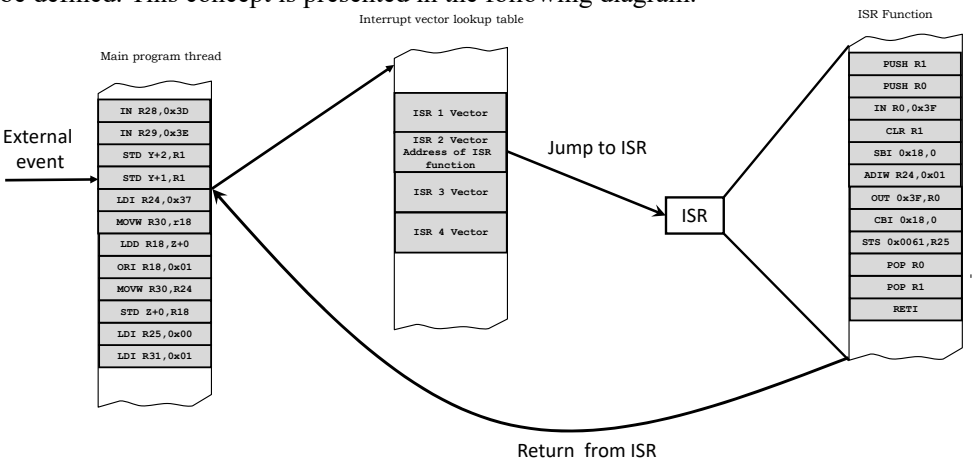

Fig. 4-25 Vectored interrupt diagram

As it can be observed in Fig. 4-25 when the microcontroller received an interrupt, it first searches the correct interrupt service routine function address after identifying the interrupt source. Immediately after the function address is found in the interrupt vector lookup table, the core jumps to its address thus executing the function. This solution offers more flexibility for the developer.

A very simple use case for interrupts could be an application where a microcontroller, running on batteries, has to blink a LED once per second. Giving the fact that most of the time the microcontroller has nothing to do, and only once per second it has to blink the LED, for the idle time the microcontroller could switch into a low power mode, thus reducing battery consumption. In this case the microcontroller could switch back to active mode, once per second, with the help of an interrupt, in order to switch the LED.

Another important advantage when using interrupts could be the fact that polling methods are eliminated. In many cases polling an event results in a failure (the event did not occur… yet) thus the processor could do something else not just polling for a slow occurring event. Using interrupts can eliminate unnecessary polling of events.

In a microcontroller the peripheral modules, such as timers, may or may not generate interrupts to the core. The core may or may not consider these interrupts. All of these aspects are related to how the programmer configures the microcontroller through code.

The ATMEGA16 microcontroller has a general interrupt flag which can configure the core to enable/disable the possibility for it to consider any interrupts. This implies that in order for the core to consider interrupts this flag must be set by the programmer. There are dedicated intrinsic function that control this aspect as presented in the following code snippet.

Code listing 4-13 ATMEGA Global Interrupt Control Macros

```
sei(); // enable ATMEGA global interrupts
cli(); // disable ATMEGA global interrupts
```

The above macros control the state of the global interrupts. It is important to mention that after reset the interrupt system is disabled. In order for any interrupt source to be taken into consideration by the core, the interrupt system must first be enabled using one of the "defines" above.

In order to declare an interrupt servicing routine for ATMEGA16 a certain syntax has to be used. In the prior versions of Atmel's developing environment, versions like AVR Studio 4, the syntax for declaring an ISR function is the following:

Code listing 4-14 AVR Studio 4 ISR declaration syntax

```
SIGNAL(SIG_<vector_name>)
{
  // your ISR code here
}
```

As it can be observed the syntax is similar to the syntax used for declaring a C function with the difference that no return type is specified (not even void) and no parameter name or type is specified. The interrupt vector is presented similar to a function parameter. An example of such a function which declares an ISR routine for TIMER1 output compare match for channel A can be found in the following code snippet.

Code listing 4-15 AVR Studio 4 ISR declaration syntax example

```
SIGNAL(SIG_OUTPUT_COMPARE1A)
{
  // your ISR code here
}
```

The declarations for the names of the interrupt vectors for ATMEGA16 can be found in the header file *iom16.h* (at the last quarter of the file), which is usually included through the inclusion of header file *io.h*. In the *iom16.h* header file a number of 20 interrupt vectors are defined, which the only ones are permitted to be declared using the above syntax.

The above syntaxes have become deprecated since Atmel has released the new version of the developing environment, Atmel Studio 7. A new syntax has been issued, which can be used for declaring interrupt servicing routines. The new syntax allowed can be found in the following code snippet.

Code listing 4-16 Atmel Studio 7 ISR declaration syntax

```
ISR(<vector_name>_vect)
{
  // your ISR code here
}
```

The difference is first given by the reserved macro name *ISR* (instead of SIGNAL) and by the fact that the vector is slightly different denominated. An example should be clearer:

Code listing 4-17 Atmel Studio 7 ISR declaration syntax example

```
ISR(TIMER1_COMPA_vect)
{
  // your ISR code here
}
```

Both the deprecated interrupt vector names and the new interrupt vector names are present in the header file *iom16.h*.

The last and important operation that needs to be done when dealing with interrupts is to also enable the interrupt signaling for the specific peripheral device that is needed to generate the interrupt. In this case we will continue to use TIMER1 and thus use the output compare match channel A as an interrupt source. For TIMER1, the interrupt generation mask can be configured via TIMSK register through the OCIE1A flag. If this flag is set to logic value 1 then, when a compare match event occurs, an interrupt for vector TIMER1_COMPA_vect is generated.

ASSIGNMENT 2: Modify the timer library and the application developed earlier in order to blink the LED only using interrupts. The following steps should be considered:

- In the initialization part of the main program call the global interrupt enable function
- Search the appropriate interrupt vector name in the *iom16.h* header file
- Declare the interrupt servicing routine in the C file of the timer library
- Reduce the main program loop to an infinite empty loop
- Add to the timer initialization function in the timer library C file the enabling of the timer interrupt using the TIMSK register
- Write the LED blink code inside the interrupt servicing routine

Test your program using the same setup and validate the signal using an oscilloscope.

ASSIGNMENT 3: Modify the previous assignment in order to blink the LED inside the main program loop but only when an interrupt occurs. Use a flag that is set as 1 in the interrupt flag (when an interrupt occurs). The main program should test this flag and when it is found 1 the main program loop should turn on or off the LED using the same logic. After the LED has been blinked the main program should reset the flag at value 0. The flag should be defined as volatile in the timer.c library file and it should be declared with

the *extern* keywork in the timer.h file. This way the main program can have access to this flag variable.

4.5 Laboratory work 4 - Control 2 digit 7 segment display

This laboratory work will introduce another peripheral device external to the microcontroller. The role of this laboratory work is to present the 7 segment digit display but using a very simple method for implementation.

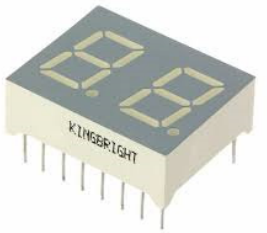

Fig. 4-26 Kingbright 2 digit 7 segment display module [25]

The 7 segment digit display [25], as presented in Fig. 4-26, is used a variety of applications and is practically implemented using LEDs and is frequently used for implementing simple digital alarm clocks. The module that we will use is a 2 digit 7 segment display but only one digit will be used.

For this module, the 2 digits are completely separated among the module's pins. Each segment of the digit is practically a LED. All the segments may be connect with a common anode or a common cathode. In our case all the segments (LEDs) are connected with a common anode mapped on one of the pins and the cathode terminals are all mapped to 7 pins (being 7 segments). The actual internal schematic of a digit for the module in Fig. 4-26 is presented in Fig. 4-27:

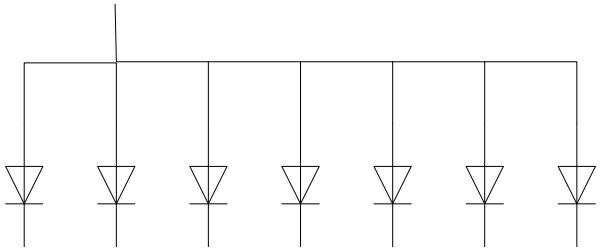

Fig. 4-27 Common anode segment connection

Each LED in the schematic practically represent a segment of the digit. The segments are usually referenced using letter from '*a*' to '*g*' or more depending on how many

segments are used to display a digit. In our case, 7 segments are used, and they the represented as following:

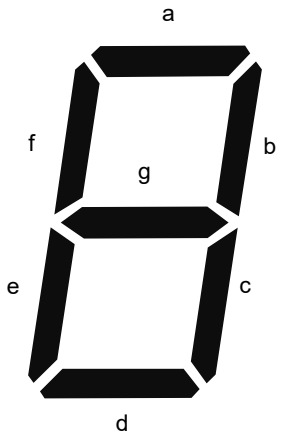

Fig. 4-28 Digit segment notation

In the datasheet [25] of the 7 segment display module that we will use, the pinout, the schematic in Fig. 4-27 and the notation of the segments in Fig. 4-28 are combined into a detailed schematic. Such a schematic is presented in Fig. 4-29 but only for one digit (first digit).

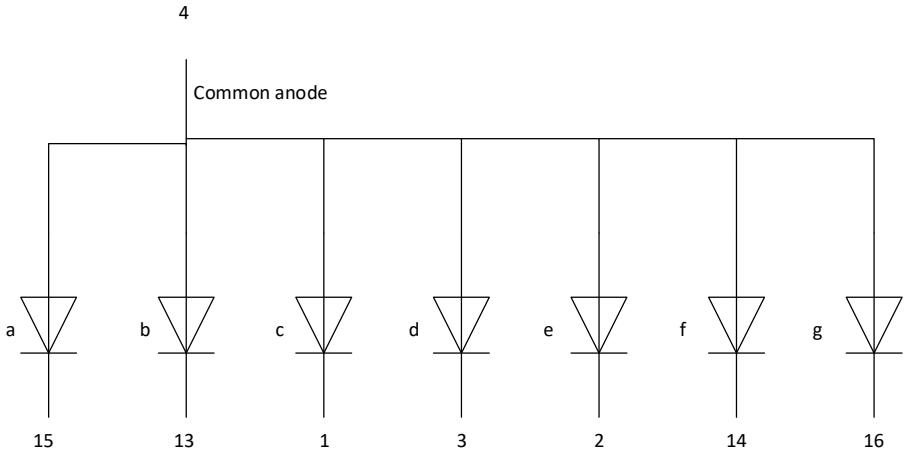

Fig. 4-29 Detailed digit schematic

The pinout of the actual part (Kingbright DA04-11EWA) that we are going to used is the following:

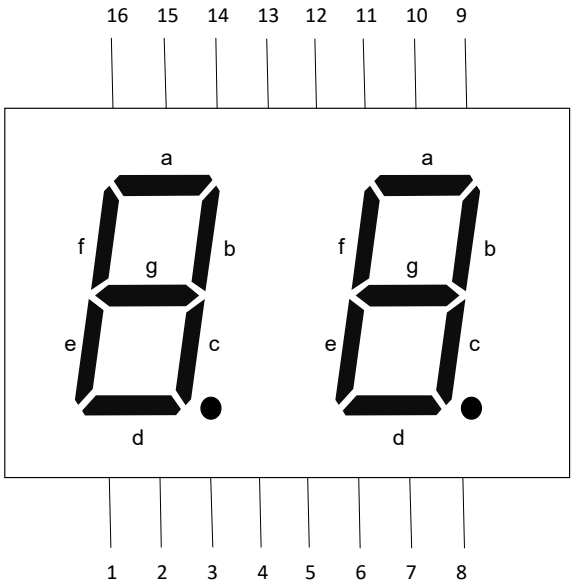

Fig. 4-30 Kingbright DA04-11EWA two digit 7 segment display module pinout

In the above pinout, pins 1, 2, 3, 4, 13, 14, 15 and 16 are related to the first digit and the rest are for the second digit. We will concentrate on the first digit which has the common anode connected to pin 4 where the rest of the pins are the cathodes of the LED segments.

The above information and schematics may be used to make the necessary connection between the first digit of the module and the ATMEGA16 microcontroller. Pin 4 (the common anode of the first digit) will be connected to the power line of the circuit (Vcc) in our case to 3.3V. The cathode lines of the digit should be connected to the same port of the microcontroller and in order for the bit mapping to be easy they should be connected in alphabetical order. A connection pin mapping could be the following:

ATMEGA16		**Kingbright DA04-11EWA**	
Port Line	**Pin number**	**Digit segment**	**Digit pin**
PA0	40	a	15
PA1	39	b	13
PA2	38	c	1
PA3	37	d	3
PA4	36	e	2
PA5	35	f	14
PA6	34	g	16
PA7	33	NC	NC

Table 9 Connection of 7 segment display to ATMEGA16 pin map

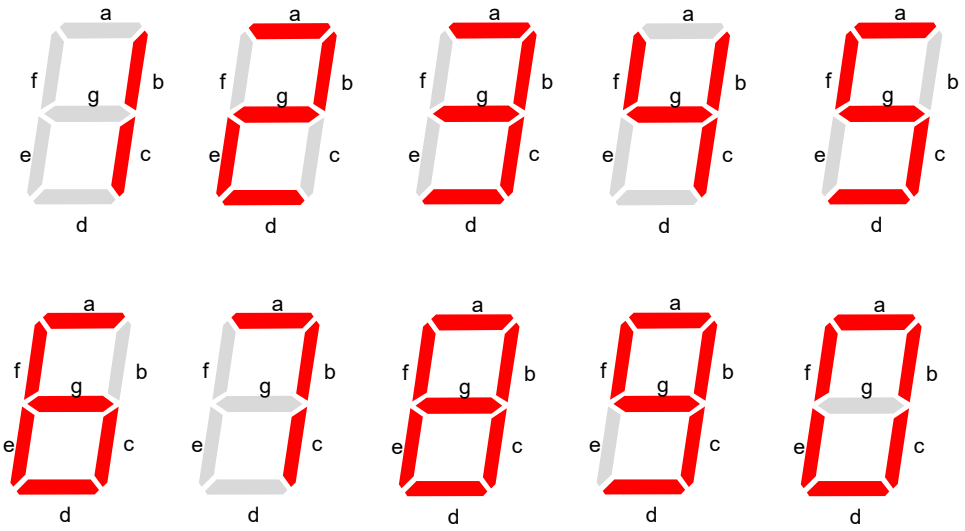

Fig. 4-31 Digit mapping on 7 segment display

Handling a 7 segment digit display is similar to handling 7 LEDs connected to a port of the microcontroller. In our case, giving the fact that all the LEDs have common anode, a LED will be on, when on the corresponding cathode, 0 logic will be applied and off, when on the same cathode, logic 1 will be applied.

The only thing that needs to be taken into consideration is what should be the pattern for turning on or off the LEDs in order to form a digit. The pattern to use is presented in Fig. 4-31. If we take as an example digit 3, we can state that segments denoted as *a*, *b*, *c*, *d*, *g* need to be on and the rest of the segments, denoted as *f* and *e,* need to be turned off. Another example could be digit 7 where segments *a*, *b*, *c* should be turned on and segments *d*, *e*, *f* and *g* need to be turned off.

Considering these examples, a mapping table needs to be constructed in order to determine what value should be written into the output register of the microcontroller's port in order to switch on or off the required segments. Such a table should look like Table 10 where only the example for digits 3 and 7 are completed.

	g	f	e	d	c	b	a	Bit mask	HEX
	7	*6*	*5*	*4*	*3*	*2*	*1*	*0*		
0										
1										
2										
3	1	0	1	1	0	0	0	0	10110000	B0
4										
5										
6										
7	1	1	1	1	1	0	0	0	11111000	F8
8										
9										

Table 10 Segment mapping table

ASSIGNMENT 1: Complete the above table for all the digits based on the explanations in this laboratory work.

After the encoding table has been established the implementation of a library handling the 7 segment digit display is trivial. The table above should be implemented as a byte array containing the numerical representation of the bit masks from the above table. The hexadecimal numerical representation can be found in the table in the last column. For example on position 3 of the table numerical value 0xB0 should be contained. Such an example of an array may be found the following code snippet:

Code listing 4-18 Example declaration of digit mapping array

```
#include <stdint.h>

static const uint8_t digitmap[10] = {
                                        TBD_0,
                                        TBD_1,
                                        TBD_2,
                                        0xB0,
                                        TBD_4,
                                        TBD_5,
                                        TBD_6,
                                        0xF8,
                                        TBD_8,
                                        TBD_9
                                    };
```

In the example only the numerical value for digit 3 and 7 have been written, the rest should be added by the attendees based on the mapping table. Using such a structure, one can easily find a digit mapping value by addressing the mapping table. For example, if the mapping value for digit '3' is required then a simple addressing like *digitmap[3]* should suffice.

Beside the declaration of the structure above the digit library should also contain a function for initializing the port line that are connected to the segments and a function which displays a certain digit on the module. An additional function could be considered which clears the digit (turns off all the segments).

Code listing 4-19 Function prototypes for 7 segment digit display library

```
void init_digit(void); // initialize the port lines
void display_digit(uint8_t digit); // display the digit given as parameter
void clear_digit(void); // clear the digit on the module
```

The initialization function should mainly set the correct value in the port direction register in order to configure as output GPIO pins the lines connected to the 7 segment display module. After the initialization of the direction, the programmer should consider the fact that the segments should be turned off in order to offer a clean start.

ASSIGNMENT 2: Based on all the explanations above implement a library (containing a .h header file and C code implementation .c file) which handles a 7 segment display module. Use the above example for what functions should be exported.

ASSIGNMENT 3: Using the program implemented for the previous laboratory work where the LED is blinked with a period of 1 second, and using the digit display library previously developed, implement a small program that counts from 0 to 9 using the 7 segment display to show the current value of the counter. The counter should change the value once per second. Practically the application would be a 10 second counter using the 7 segment display module for viewing the value of the counter. From the previous laboratory work, the last exercise should be used as a starting point (where the LED is blinked via the interrupt handler). The seconds counter should be incremented and displayed using the digit library in the interrupt servicing routine function of the timer.

ASSIGNMENT 4: Modify the previous assignment in order to change the digit displayed on the 7 segment display, not in the interrupt servicing routine, but in the main program loop. This should be done using a flag to communicate between the interrupt servicing routine and the main program. Such a situation has been implemented in Laboratory work 3 - Timer, compare match, interrupts.

4.6 Laboratory work 5 - Read 4x4 keyboard 16 keys

Another important and yet trivial peripheral device is the subject of this laboratory work. The simple matrix keyboard is presented by this laboratory work along with methods on how to use it in an embedded project.

The matrix keyboard is practically a matrix of interconnected push buttons with no additional logic or other circuits. The matrix keyboard is available in many shapes and sizes but the most popular for embedded usage are the 4x4 or 3x3 matrix keyboard. For this laboratory work we will use a 4x4 matrix keyboard such as the one in Fig. 4-32.

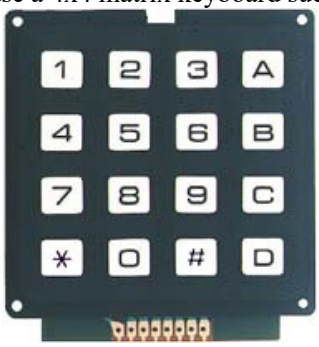

Fig. 4-32 4x4 Matrix Keyboard

The schematic of the matrix keyboard as well as the additional connections that need to be made are presented in Fig. 4-33. The grayed part of the schematic in Fig. 4-33 is the actual internal schematic of the keyboard in Fig. 4-32. The keyboard has in this case 8 pins exported outside the chassis. Having the keyboard orientation as in Fig. 4-32 the pinout is the following:

1	Column 0
2	Column 1
3	Column 2
4	Column 3
5	Row 0
6	Row 1
7	Row 2
8	Row 3

Table 11 Keyboard pinout

It is important to make the observation that giving the fact that the keyboard is a passive component, no Vcc or ground line is needed.

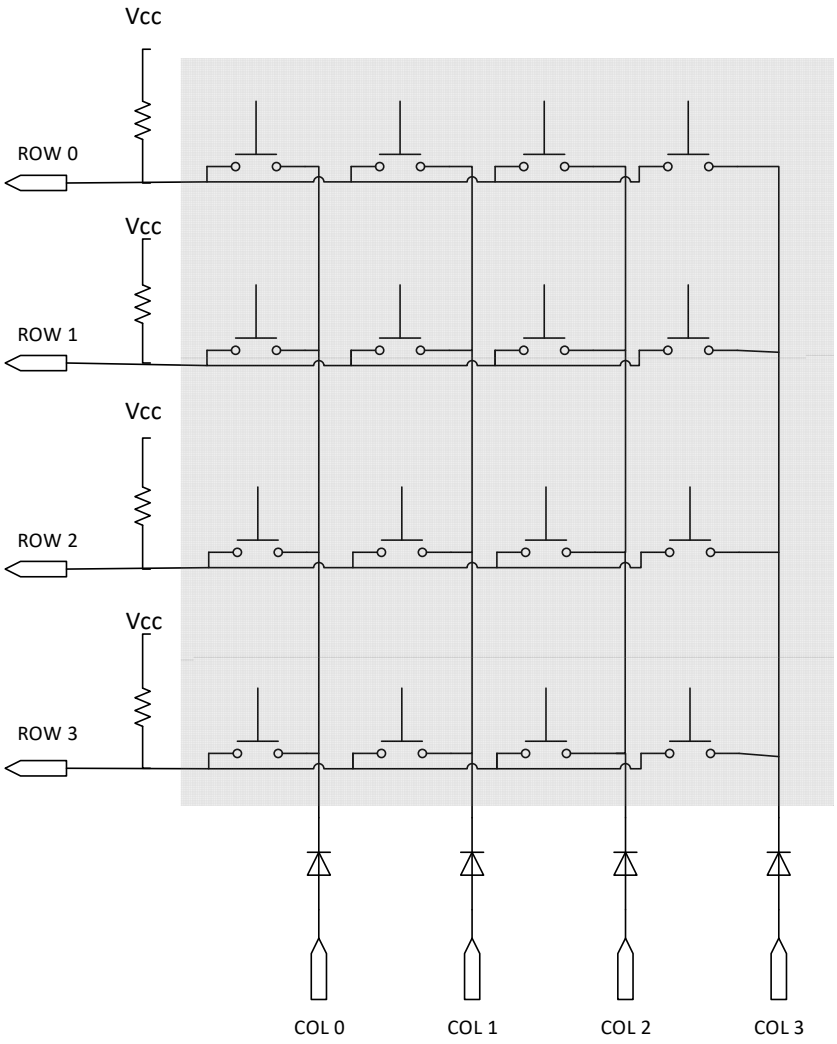

Fig. 4-33 4x4 Matrix Keyboard Schematic

There are 8 terminals exported from the keyboard's chassis: 4 lines for the columns and 4 lines for the rows as presented in the schematic. In order to drive the keyboard by the microcontroller, the columns need to be connected to output GPIO lines of the microcontroller and the rows have to be connected to input GPIO lines of the microcontroller. It is important to mention that the connection could be made vice versa (columns to input lines and rows to output lines). We will use the first solution as suggested by the previous schematic. The connection between the microcontroller and the matrix keyboard could look like the following:

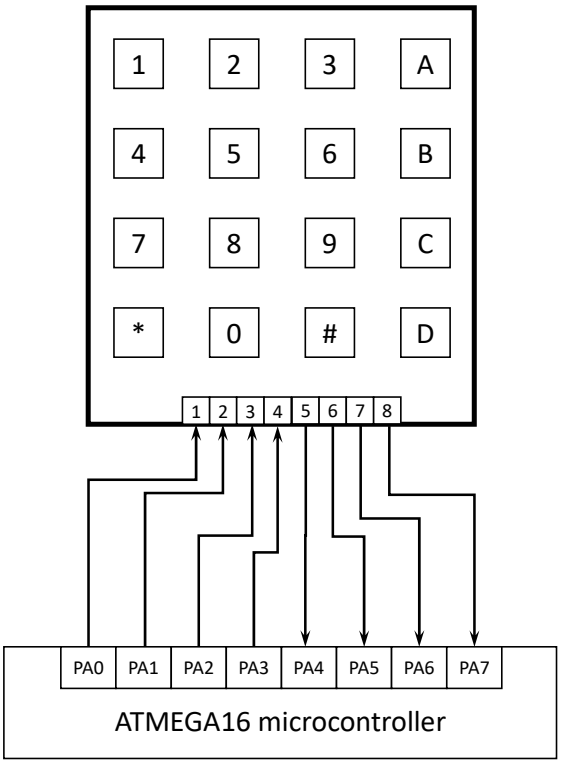

Fig. 4-34 4x4 Microcontroller and keyboard interconnection schematic

As it can be observed the entire PORTA of the microcontroller will be used. The first part of the port (PA0-PA3) will be used as output while the second part (PA4-PA7) will be used as input.

Taking a look over Fig. 4-33 and Fig. 4-34 a short analysis over some aspects needs to be considered. The first aspect is represented by the pull-up resistors present on the row lines (corresponding to the input line of the microcontroller). A pull-up resistor is used when a certain line is desired to be maintained at logic "1" (the resistor is only used to limit the flow of the current through the microcontroller). In Fig. 4-34 these resistors are not present because the microcontroller has internal pull-up resistors that can be activated when the port lines are configured as input.

The other important aspect is represented by the diodes present on in column lines of the keyboard (corresponding to the output lines of the microcontroller). These diodes are used to offer additional protection when multiple keys are pressed and the output lines that are connected by the key may have different logic levels. This situation could lead to an electrical short inside the microcontroller which may damage the port lines. Again, these diodes are not present in the final schematic but certain precautions will be taken in software in order to compensate the diodes and to avoid hazardous situations.

Connecting the matrix keyboard to a microcontroller is trivial task. The complicated task when handling a matrix keyboard by a microcontroller is generated by the software. The microcontroller will have to constantly scan the keyboard in order to detect if a key was pressed. The next paragraphs will explain and exemplify how such a scan could be implemented.

In the idle state, when no key is pressed all the input lines of the microcontroller (row lines) are stable at logic "1" with the help of the internal pull-up resistors of ATMEGA16. So, in the idle state, when no key is pressed, the microcontroller will read logic "1" on each row. The scan of the microcontroller consists in a procedure where the microcontroller pulls down to logic "0" one of the column at a time (using the output lines of the port) and reads the rows via the input lines. When a logic "0" is read then at the intersection of the pulled down column line and the row read as "0" a button has been pressed. A row is read by reading the logic value of the corresponding input line of the microcontroller.

In order to avoid hazardous situations the actual output lines will only be configured at output, only one at a time and only when it is needed to be pulled down to logic "0". Furthermore, the internal pullup resistor for the input lines needs to be active before reading the line. Practically the internal pullup resistors need to be kept activated all the time.

The scan is made periodically and continuously. One scan session is terminated either when a push button event has been detected or when all the columns have been scanned. The algorithm is also presented in the following flow chart:

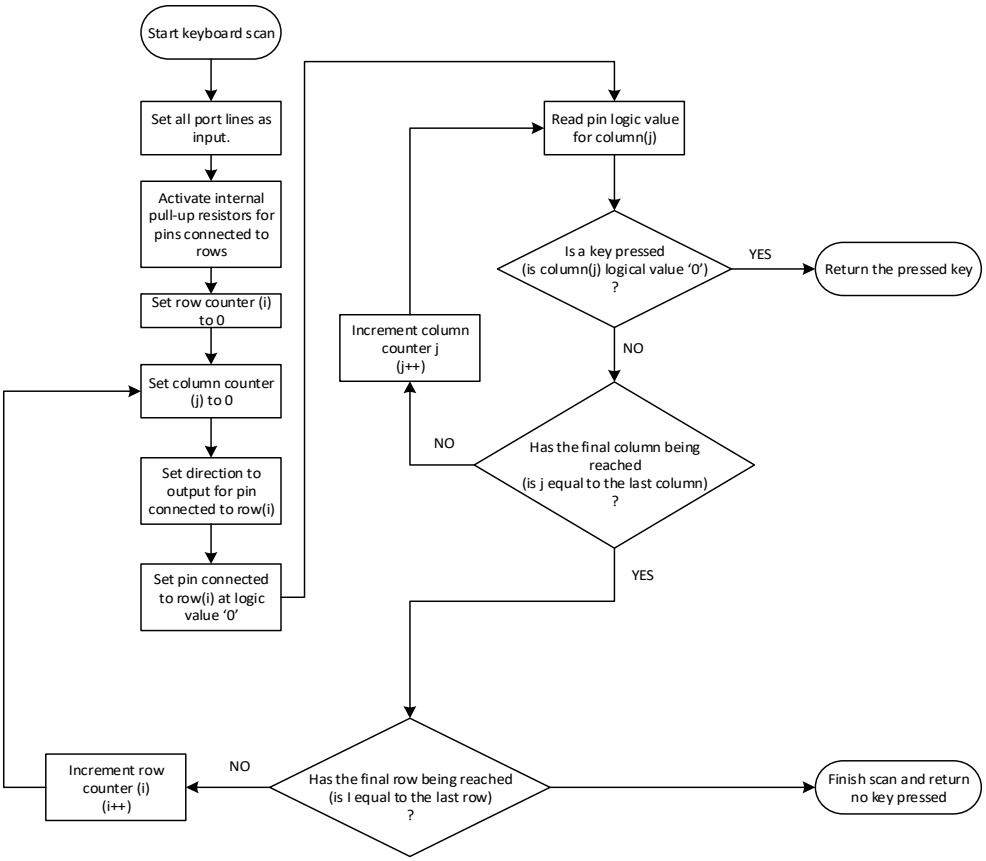

Fig. 4-35 Matrix keyboard scan algorithm

The implementation of the flow diagram presented in Fig. 4-35 is not very complicated. It only involves working with GPIO lines similar on how it was presented in 0Table 8 Laboratory applications planning

Laboratory work 1 - First project: LED blink and in 4.3 Laboratory work 2 - Push buttons.

As a reminder only:

- GPIO line direction can be configured using DDRx registers
- GPIO lines configured as output can be set to logic level 0 or logic level 1 using the PORTx registers
- The logic value applied on a GPIO line configured as input may be read using the PINx registers.

Only one aspect was not presented yet, regarding how the microcontroller's internal pull-up resistors can be controlled. The internal pull-up resistors may only be activated for the GPIO lines that are configured as input (makes no sense otherwise). This feature may be controlled by using the PORTx registers. If a GPIO line is configured as input and the corresponding bit from the PORTx register is set to logic 0, (as default) then the

corresponding input line will have the internal pull-up resistor disabled. If a GPIO line is configured as input and the corresponding bit from the PORTx register is set to logic 1 then the corresponding input line will have the internal pull-up resistor enabled. For example, if we would like to set the PA5 line as input with internal pull-up resistor enabled we will have to write bit 5 of DDRA register as 0 and then write bit 5 from PORTA register as logic 1.

It is important to mention that, giving the fact that the keyboard is a matrix of interconnected push buttons, debouncing needs to be applied as explained in 4.3 Laboratory work 2 - Push buttons.

ASSIGNMENT: Based on all the explanations above implement a library (containing a .h header file and C code implementation .c file) which handles the keyboard. Also integrate the previous library handling the 7 segment display and implement an application that displays the key pressed of the keyboard on the 7 segment display digit.

Hints:

- Implement a function that makes a single scan of the keyboard based on the flow chart diagram in Fig. 4-35. This function should not be exported into the header file
- Implement a function, which uses the previously developed function, which implements the debouncing algorithm. It practically initiates a scan of the keyboard and if a keypress is sensed it waits for debouncing and then it scans the keyboard again. If a keypress was detected again then the function should return the pressed key, otherwise a “no key pressed” value should be returned. This function should be exported in the header file and this one should be used by the main program to scan the keyboard.
- The main program should periodically scan the keyboard and if a key pressed is detected it should display the corresponding key on the 7 segment display
- Add the characters A,B,C,D,E and F to the mapping table of the 7 segment display library in order to display them on the module.

4.7 Laboratory work 6 - UART interface

This laboratory work is concentrated into developing the first microcontroller application with serial communication. The serial communication, universal asynchronous receiver transmitter (UART) RS-232 protocol, will be presented. The outcome of this laboratory work will be an application where the microcontroller transmits text data with a fixed period. Also, the microcontroller will received text data over the serial interface and respond to some simple commands. Therefore both transmission and reception will be considered which will function in blocking and non-blocking modes, thus interrupts will be required.

The UART protocol is probably one of the oldest communication protocols that are still being used in many applications. Even though it was designed in the 1960's the protocol is highly used even now because of its simplicity, of course, nowadays used at much higher speeds than in the ones in the 60's.

When communicating using a synchronous protocol, a clock signal is present, thus the time synchronization is assured. In an asynchronous communication, the clock signal is not present and the data must carry its own information for time synchronization [17]. In the simple UART communications with no hardware flow control there may be only 2 communicating partners which may switch their role from receiver to transmitter. Each terminal has 2 dedicated lines for communication: a receive line (RX) and a transmit line TX. These lines are connected as displayed in the following figure:

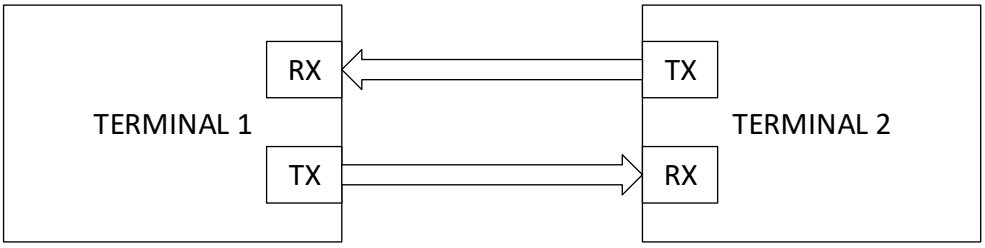

Fig. 4-36 UART communicating terminals

As stated before, giving the fact that no clock signal is present, the synchronization needs to be carried out by the transmitted data. Not only is a time synchronization necessary. Each communicating partner needs to sample the bits on the line with the same sample rate which in communication is translated into a symbol rate. The same symbol rate is needed to be configured in each communicating terminal. The symbol rate has the BAUD as a unit of measurement. Moreover, another important measuring unit is the transmission speed which is the number of bits transmitted per second (bps). It is customary to use the term that "the BAUD of the serial communication is.." for example 9600 bps. Having a more practical approach, the main interest is actually on how long a bit is in time. This is usually calculated using:

$$t = \frac{1}{BAUD}$$

Having an example of a BAUD of 9600 bps the length of a bit is:

$$t = \frac{1}{BAUD} = \frac{1}{9600} = 104\ us$$

If having to watch the character ‘a’ being transmitted using an oscilloscope it should look like in the following figure:

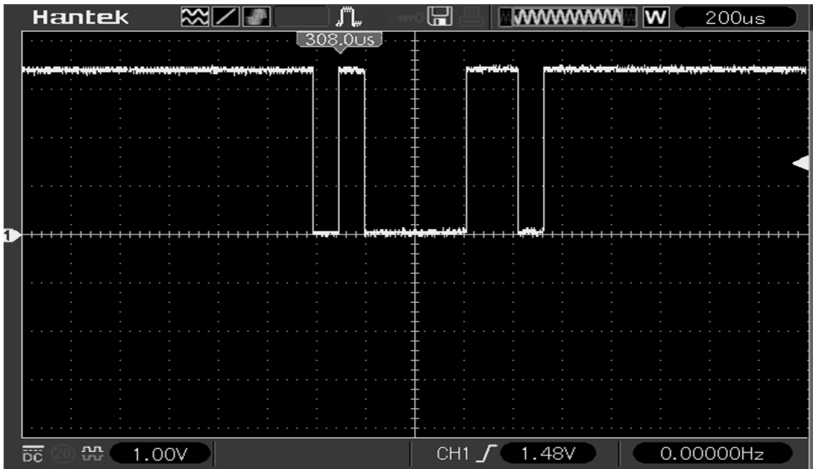

Fig. 4-37 UART character serialization

Using the oscilloscope to measure the time a bit occupies when having a BAUD of 9600 for bit sample rate the following result may be found as shown in the oscilloscope capture:

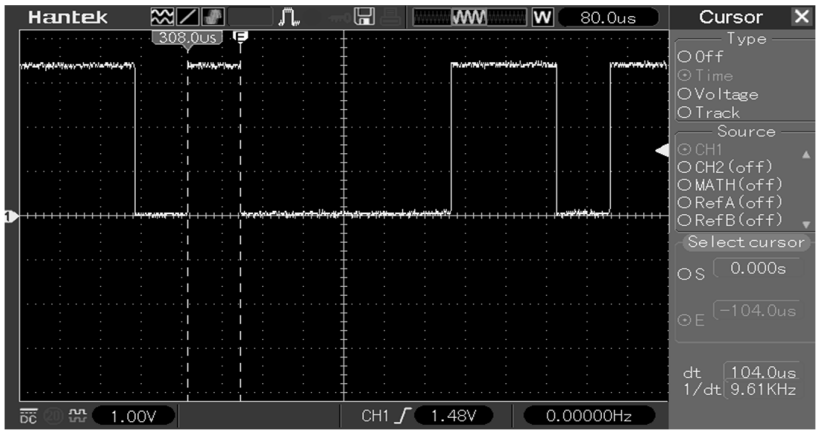

Fig. 4-38 UART bit measurement

In Fig. 4-38 the cursors were set to measure the one bit. The result is displayed in the bottom right corner as being 104.0 us (or 9.61 kHz if converted to frequency). We can notice that the value in kHz is almost equal to the value of the BAUD of 9600 bps.

Both the transmitted and the receiver have be configured in order to both use the same BAUD rate. Having differences between the sampling rates of the 2 communicating partners may result in transmission errors.

The next important aspect that needs to be discussed is how data is encapsulated by the protocol in order to be transmitted over the line. This aspect is described by the following table:

Length (bits)	1	5-9	1-2
meaning	Start bit	Data bits	Stop bits

Table 12 UART protocol encapsulation

As found in Table 12 the protocol starts with a start bit. This bit announces that a new frame begins. Having an UART line inactive at logic "1" the start bit is usually encoded as logic "0". This is, in many cases, hard coded. The programmer usually cannot modify the number, length or values of the start bit.

Following the start bit are 5 to 9 bits of data. This is a configurable parameter and needs to be the same on the receiver and transmitter. The data for which the protocol was designed is represented by characters which may be encoded in 5 to 8 bits according to the ASCII table. In the situation when 9 bits of data are specified, the 9^{th} bit serves as parity, which is calculated by both the transmitter and receiver. The receiver also compares the calculated parity with the one transported by the 9^{th} bit in order to detect transmission errors. There may be an even parity or an odd parity. In an even parity the 9^{th} bit is logic "0" when there is an even number of logic "1" bits in the data word. Same algorithm goes for the odd parity. In many situations the parity is not used, thus the number of data bits is set to 8 in order to disable parity. This is also usually configurable.

After the data is sampled, the frame ends with one, one and a half or two stop bits which are usually encoded as logic "1" bits. This option is also configurable.

Having these explained, the following conclusions may be deducted:

- Both the receiver and the transmitter have to function on the same parameters
- Only 2 communication partners may be used in serial UART communication bus
- Both of the communication partners may be receivers or transmitters
- Each communication partner has 2 line: a reception line and a transmission line
- The start bit is only one with logic value "0" and is not configurable
- The configurable parameters are:
    - Character length: 5,6,7,8 bits
    - Parity
        - Odd parity
        - Even parity
        - No parity
    - Number of stop bits (1 bit, 1+1/2 bits, 2 bits)
    - BAUD rate

- Same configuration needs to be present on both communicating partners

The ATMEGA16 microcontroller has a dedicated peripheral module serving as an UART interface. The full documentation of the USART interface of ATMEGA16 may be found in the ATMEGA16 datasheet [14] at the USART Chapter. The pins that are mapped for the USART interface are found on PORTD and are PD0 serving as RX (RXD) and PD1 serving as TX (TXD).

The next step is to make the necessary connections between the ATMEGA16 header board and the peripheral board on one hand, and on the other hand, between the peripheral board and the PC. A block schematic of the connections to be made is displayed in the following block diagram:

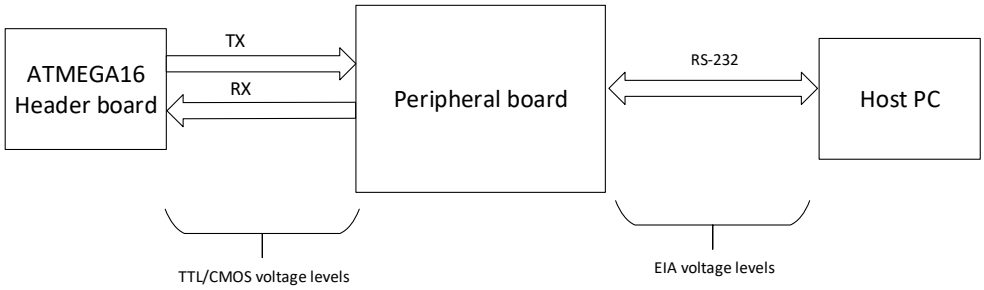

Fig. 4-39 UART connections block diagram

It is important to mention that the signals (RX and TX) between the ATMEGA16 and the peripheral board have CMOS/TTL voltage levels. These levels cannot be used to send data over long lines. A solution to this is to translate these signals into EIA voltage levels where are more resistant to hazardous environments and can also be used to send data over longer lines. In our case, EIA voltage levels are used to transfer the data from the peripheral board to the host PC using a standard DB9 serial cable. The translation of the signals is done on the peripheral board using a dedicated integrated circuit, like MAX232 [18].

The first set of connections can be made using the provided wires. The TXD pin (PD1) from ATMEGA16 header board needs to be connected to the TXD header pin on the peripheral board and the RXD pin (PD0) from ATMEGA16 header board needs to be connected to the RXD pin on the header of the peripheral board. The signals of the peripheral board are named from a peripheral point of view.

The second set of connection can be mode using a standard DB9 serial cable in order to connect the serial interface (through the DB9 connector) of the peripheral board to the serial interface of the host PC.

ASSIGNMENT 1: Make the necessary connections and have the laboratory supervisor verify them.

The next step is to concentrate on the software part of the ATMEGA16. Prior to the configuration of the USART module of ATMEGA16, the direction of the responsible pins

needs to set accordingly using the DDRD register. The PD1 serving as TXD pin, acting as the transmission pin of the USART interface should be configured as output. The PD0 pin serving as RXD pin, acting as the receiver pin, should be configured as input. For more information please read the I/O Ports Chapter in the ATMEGA16 documentation.

ASSIGNMENT 2: Read the documentation regarding the USART module of ATMEGA16 concentrating on the registers. Make a list with all the registers that should be used for configuring the USART interface. The interface should be configured with the following parameters: BAUD 9600 bps, 1 stop bit, 8 bits per character, no parity. Pay special attention on the address sharing of registers UBRRH and UCSRC (URSEL bit makes the difference).

The first aspect in the configuration of the UART peripheral module of ATMEGA16 is to calculate the divisor value (UBRR) that the microcontroller will use to general the BAUD. This can be done by using the formula provided by the producer which can also be found the official documentation:

$$UBRR = \frac{F_{OSC}}{16 \cdot BAUD} - 1 \qquad (4\text{-}7)$$

Where:
F_{OSC} – represents the frequency of the ATMEGA16 internal clock in Hz (in our case 8 MHz)
BAUD – represents the actual baud rate (in our case 9600)
UBRR – represents the calculated value of the divisor which must not exceed 16 bit in size (no more than 0xFFFF)

The calculated baud rate divisor needs to be written into UBRRH:UBRRL registers which separate the most significant byte and the least significant byte of the 2 byte value UBBR. The UBRRH register, containing the most significant byte of UBRR and it needs to be written first. The UBRRH register and the UCSRC register of the UART interface share the same address space, they can be differentiated by the value of bit 7, URSEL, in UCSRC register. According to the documentation when this bit is set to 0 the UBRRH is accessible. When needing to access UCRSC this bit (URSEL) needs to be written to 1.

Beside the UBRRH and UBRRL registers, here is a collection of configuration and control registers which are used to control the UART interface. The full documentation of these registers has to be read in order to fully understand the functionality. In the following paragraphs only basic aspects will be discussed.

The UCSRA register contains mainly flags that are important when configuring the interface. The only bits that are significant for configuration are the U2X bit and MPCM bit. These bits should be left as logic 0 in our case.

Most of the configuration of the interface is done using the UCSRB register. We should be focused on the bits RXEN, TXEN and UCSZ2. The RXEN and TXEN should be written as logic one in order to enable the UART received (RXEN) and the transmitter (TXEN). Even if, for now, we will only work with the transmitter, we should enable also the receiver, thus it will be used in the coming laboratory works. The UCSZ2 register has meaning only together with UCSZ1 and UCSZ0. The value formed by these three registers define the size of the data word. The corresponding values can be identified in a table in the documentation under the UCSRC register. For this laboratory work, considering that we will use a data word of 8 bits wide, we will consider the bits having the following values: UCSZ2 = 0, UCSZ1 = 1, UCSZ0 = 1.

Pay attention that the UCSZ2 bit is contained in the UCSRB register but UCSZ1 and UCSZ0 are contained in the UCSRC register. Regarding the UCSRC register practically only these 2 pins need to be set to logic 1, the rest should be left as logic 0. The UCRSC register contains bits that configure the number of stop bits, the parity settings, and the synchronous/asynchronous operation of the interface. Letting the rest of the bits 0, beside UCSZ1 and UCSZ0 will let the interface configured as asynchronous, no parity and 1 stop bit.

It is important to mention that the attendees must read the whole documentation of these registers and not rely only on the explanations found in this laboratory work.

The configuring of a register should be implemented inside a function with a proper name. Once the interface is configured, the data transmit algorithm needs to be implemented. The flowchart for the configuration of the UART interface may be the one described in the following figure:

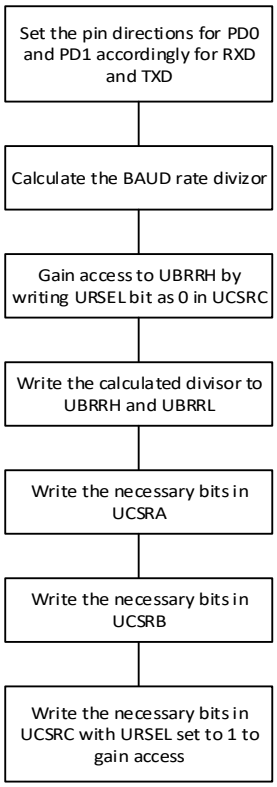

Fig. 4-40 UART initialization flowchart

In order for the interface to transmit a character it needs to be written into the transmission register: UDR. The same register is also used to read a newly arrived byte from the serial interface. Writing a byte to the UDR register is not enough when making a transmission over the UART interface. The programmer must also wait for the current byte to be transmitted. This may be done by using the UDRE bit in UCSRA. This bit informs the programmer when the transmit UDR data register is empty. After the UDR register is written for transmission the UDRE bit becomes logic 0. After the interface serializes the byte over the line the UDR data register becomes empty thus signaled to the programmer with UDRE bit becoming logic 0. If the programmer does not wait for the data to be transmitted over the serial line, more exactly for the transmission register to be emptied by the interface, there is a risk for this register to be written when it is not empty. In this situation the currently transmitted byte is corrupted and data overrun error is signaled through the appropriate byte in UCRSA.

A possible flowchart of the function capable in transmitting a byte over the UART may be found in the following figure:

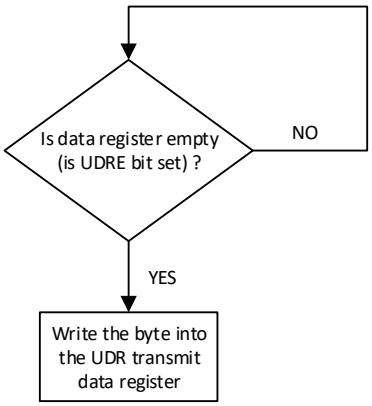

Fig. 4-41 UART transmit flowchart

When a character is transmitted over the serial line, in our case, using the connection to the PC as presented in Fig. 4-39 , it can be displayed using a dedicated terminal software. Such a software is Docklight Scripting which can configure a serial COM port from the PC and can also be used for sending and receiving data.

Docklight scripting is an easy to use but powerful serial terminal software. The main advantages of *Docklight scripting* are:

- possibility to have access to all the settings of the serial port
- can function as a TCP/UDP client or server
- offers the possibility to define and send macros over the line (serial or network)
- has scripting features in order to simplify protocol interpretation
- offers good representation of unprintable characters
- byte interpretation may be ASCII, hexadecimal, decimal and binary

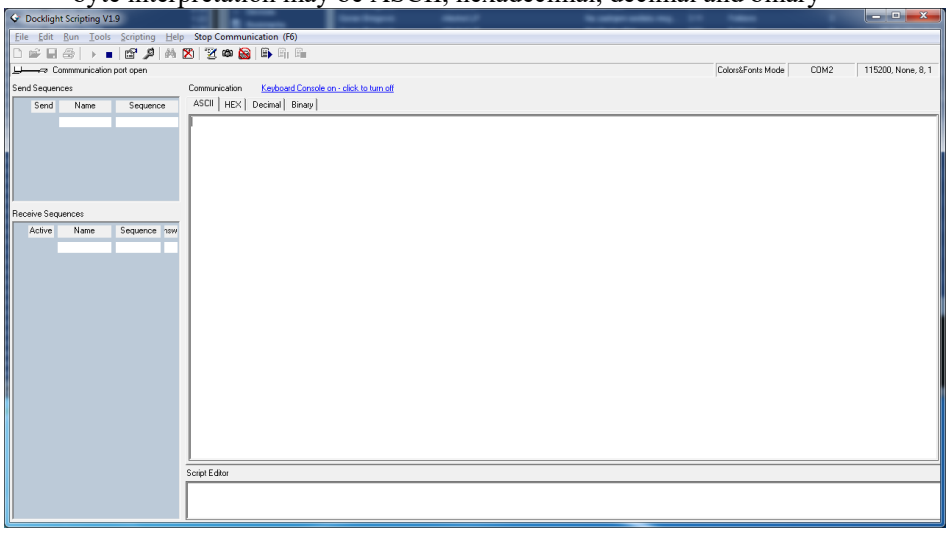

Fig. 4-42 Docklight main window

The main window of Docklight offers quick access to all of the features. The command bar contains practically all the necessary commands to configure, open, close and enable data write to the serial port.

Fig. 4-43 Docklight command bar

The active serial port along with its current configuration is displayed on the right side of the bar. In order to modify the COM port or the configuration, a double click on the COM port name (ex COM 2 in Fig. 4-43). The configuration window is displayed in Fig. 4-44.

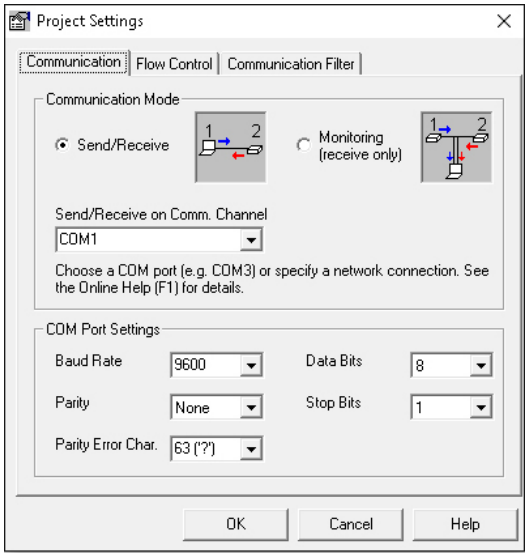

Fig. 4-44 Docklight COM port configuration window

The configuration of the serial port does not imply also the opening of the COM port for receiving and transmission. These operations are made using some of the buttons on the command bar in Fig. 4-45. The buttons that present the most interest are: *Start Communication*, *Stop Communication*, *Keyboard Console On* and *Clear Communication Window*. These commands are highlighted in this order in Fig. 4-45.

Fig. 4-45 Docklight most used commands

The most important commands found on the command bar are those responsible for opening and closing the serial COM port. The first two highlighted buttons in the above figure are responsible for these actions. The opening of the port is activated through the *Start Communication* button and the closing the port through the *End Communication* button. In the moment the COM port has been successfully opened the state is updated below the button bar and *Docklight scripting* is ready to receive data through the serial port which will be displayed in the main window in the currently selected format. The window may be cleared using the *Clear Communication Window*. It is important to mention that opening the communication window will only activate the receive process. Any typed data in the main window will be discarded. In order to activate the transmission of data using the keyboard the *Keyboard Console On* button must the accessed. The status bar will be updated accordingly.

The main window of *Docklight scripting* displays the received and transmitted data in a strictly defined format. Each operating is preceded by a full timestamp along with a tag that specified whether it is a transmission ([TX]) or a reception ([RX]). Usually the transmitted data are colored in blue and the received data in red. The special characters are also displayed using a simple syntax: the definition of the special character according to the ASCII table between angle brackets. A sample of a short transaction displayed by *Docklight scripting* may be the following snippet.

Code listing 4-20 Docklight scripting communication sample

```
04.01.2016 12:21:51.575 [TX] – data transmitted from to the microcontroller

04.01.2016 12:21:52.455 [RX] – data received from the microcontroller
```

ASSIGNMENT 3: Open *Docklight Scripting*, configure the port for a BAUD rate of 9600 bps, 8 bit per character, 1 STOP bit and no parity. Open the COM port and activate the keyboard transmission feature.

ASSIGNMENT 4: Write a C library (a c file with a header file) which contains an initialization function for the UART interface and a function capable of transmitting one byte over the serial UART interface. The project should than have 3 files for example:

- Serial.c – the C file containing the implementation of the functions
- Serial.h - the Header file containing the declarations for the function implemented in serial.c file that need to be exported
- Main.c – the C file containing the main program and function

The serial port needs to be configured as following: BAUD 9600 bps, 1 stop bit, 8 bits per character, no parity.

Write a main program that sends the same character over the serial interface once per second using the previous developed library. Test the program using Docklight scripting.

ASSIGNMENT 5: Modify the previous assignment in order to communicate at a BAUD of 115200 bps. Change the settings in Docklight scripting accordingly.

ASSIGNMENT 6: Implement and add to library a function which sends a standard POSIX C string over the serial interface. A string is considered a character array terminated with the 0x00 byte (or '\0' character). The usage of the *strlen* function recommended. The function prototype should look like the one in the following code line:

Code listing 4-21 Send string function prototype

```
void SendString(char *string);
```

Test the function implementing an application that sends a string over the serial line once per second. The string should contain a counter variable which increments after it has been sent. Use *sprintf* to format a string a send it in a character array which should then be sent over the serial interface. Suggestion code snippet:

Code listing 4-22 Send string assignment suggestion code

```
void main()
{
  char text[100]
  uint32_t counter = 0;
  // some code

  while(1)
  {
     // some code
     sprintf(text, "some text %d", counter);
     counter++;
     // delay
     //some code
  }
  // some code
}
```

The only operation left to be implemented is the reception of one byte over the serial line. We will consider a blocking and a non-blocking approach. The easiest way is to use the blocking approach.

Practically a new function has to be developed which is responsible for the serial reception and it should be blocking, thus waiting for a byte to arrive over the serial line into the receive buffer. This information can be extracted from the RXC (Receive Complete) bit in the UCSRA register. This bit is set to value logic 1 when unread data is available into the receive buffer. In the rest of the time this bit is set to logic 0. Using this information we can state that this is the most important status bit when implementing the reception. Having this into consideration, the reception function flow chart diagram may be the following:

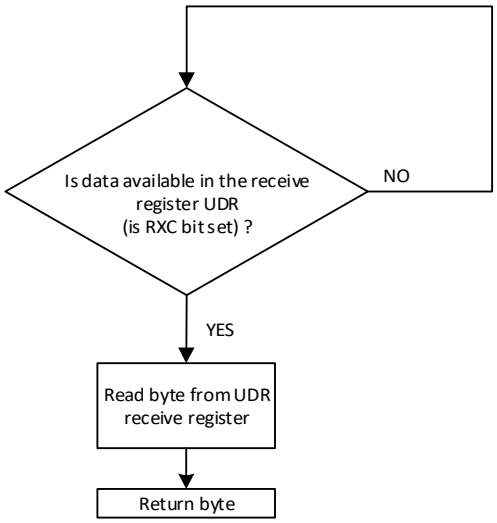

Fig. 4-46 UART receive flowchart

It is important to mention, in this case, that the register named UDR is used for transmitting data over the serial line (transmit register) but it is also used for receiving the data that has arrived over the serial line (receive register). It is important for the attendees to distinguish these aspects. Even though it has the same name, there are practically 2 registers which are accessed by the same name. The difference is the access method. When reading the register named UDR, the receive register is actually accessed. When writing the register named UDR, the transmit register is actually accessed. This is normally handled by hardware.

The actual function that should implement the flow chart described in Fig. 4-46 should have the following prototype:

Code listing 4-23 UART receive byte function prototype

```
uint8_t UART_ReceiveByte(void);
```

It is again important to mention that the receive function, as presented in Fig. 4-46 and in Code listing 4-23, is blocking until a character is received, which means that the execution of the microcontroller is halted until a character (byte) is received over the serial line.

ASSIGNMENT 7: Implement the function responsible for receiving a byte over the UART interface according to the explanations above. Add to function into the already developed serial library.

The easiest method to test this function is to implement a trivial serial echo program. This program practically waits for a character to be received over the serial interface and after it has arrived it is sent back over the serial interface, thus the name “echo” is justified.

The actual main loop of the microcontroller should implemented according to the diagram:

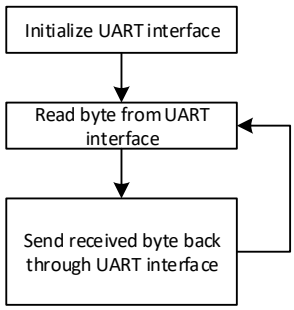

Fig. 4-47 UART echo main program loop

ASSIGNMENT 8: Implement the UART echo main program used to test the receiver function. In order to test this echo program open the Docklight Scripting software, connect to the serial port the microcontroller is connected to and send character over the serial line. In order for the transmission to be enabled in Docklight Scripting, the "Keyboard Console On" button needs to be pressed or the Tools -> Keyboard Console On menu needs to be accessed. The shortcut key for this operation is CTRL+F5. After the transmission is enabled you can write the characters into the main window using your keyboard. A successful test of the program and receive function is when a transmitted character is returned back to the Docklight Scripting software.

The next aspect to be discussed in this laboratory work is how to transform the echo application to work using interrupts. The transmission will be left the way it is. The reception is the actual operation that is most blocking and must be transformed in order to be interrupt based. The first step is to find the appropriate interrupt vector responsible for the UART reception interrupt and to declare the interrupt servicing routine similar to how it was presented in Laboratory work 3 - Timer, compare match, interrupts. The only difference is the interrupt vector.

OPTIONAL ASSIGNMENT: Open the iom16.h header file and search the appropriate interrupt vector name to be used for the UART reception interrupt.

The interrupt vector that we will be use in this laboratory work is the one related to the reception of a character over the serial UART interface. Searching the iom16.h could lead to vector responsible for the UART reception interrupt: USART_RXC_vect. Having the name of the interrupt vector, the actual definition of the interrupt service routine may be the following:

Code listing 4-24 ATMEGA16 interrupt service routine for UART reception interrupt

```
ISR(USART_RXC_vect)
{
  // some code
}
```

The reasonable question that could arise is what code should be written into the interrupt servicing routine function? Giving the fact that this function is called by the core when a byte is received over the serial interface, the obvious operation here should be to read that character from the UDR register, store it in a global variable and announce the main loop, using another global variable as a flag, when a new character has arrived. Keep in mind that the global variables that are accessed from the interrupt service routine should be declared as volatile.

There are two main configuring operations that need to be done in order for this interrupt service routine to be activated and taken into consideration of the core. Firstly, the serial UART interface needs to be configured in order to send an interrupt signal upon reception of a byte over the serial line. The responsible bit for this is RXCIE in UCSRB register. Having this bit set as logic 1 instructs the serial UART interface to send an interrupt signal to the ATMEGA16 core when data is received over the line. The second configuring that needs to be done is to enable the global interrupt system of the core. This is done by calling the following function before the serial interface is configured:

Code listing 4-25 Global interrupt enable function call

```
int main(void)
{
  // some code
  sei();  // global core interrupt enable function
  //some code
}
```

Having all of this written, the final discussion is on the main loop program which most of the time must verify if new data has arrived by checking the flag written by the interrupt service routine. If the flag has the correct value then the program should read the newly arrived byte by accessing the global variable containing it (written by the interrupt service routine). Having the new data, it should be transmitted back over the serial line using the transmit function routine already presented in the serial library developed earlier. The only operation left doing, is to reset the flag that announces the arrival of a new character, actually, the flag variable written by the interrupt service routine function. If this flag is not reset then the code will be stuck infinitely sending the last received character over the serial line. A flow chart diagram of how the main loop program of the interrupt based echo should look like is presented in the following figure:

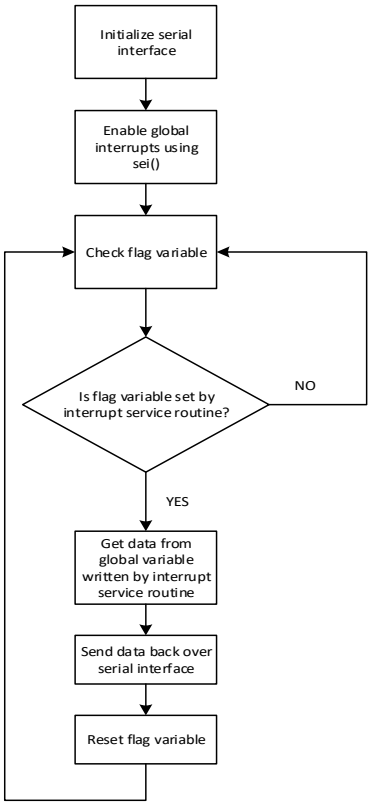

Fig. 4-48 UART echo main program loop with interrupts

ASSIGNMENT 9: Modify the serial interface library previously developed in order to support UART reception using interrupts. Add the interrupt service routine into the library and export the necessary variables to the main program through the serial library header file (using the *export* statement). Modify the main program loop in order to have the serial echo work with reception using interrupts. Use a volatile flag to communicate the arrival of a new character from the interrupt to the main program loop and a volatile variable that contains the actual character. The communication flag should be reset in the main program loop after the new character has been read from the variable containing it.

ASSIGNMENT 10: In order to make a better use of the UART communication let us make the connections we had for the laboratory works handling the 7 segment display which was connected at PORTA of ATMEGA16. Modify the interrupt based echo application with the 7 segment display digit library integrated into the project and implement an application which not only sends back over the serial interface the received character but also displays the character on the 7 segment display digit module if the character is a hexadecimal digit. Practically this application replaces the keyboard in Laboratory work 5 - Read 4x4 keyboard 16 keys with the serial interface. Change the digit

on the display inside the main program loop. As used in the previous laboratory works, have a flag variable responsible with the communication between the serial interface interrupt servicing routine and the main program loop. An additional variable is also needed where the interrupt servicing routine saves the newly arrived character which is read by the main program loop. Use the same approach as in the previous laboratory works.

4.8 Laboratory work 7 - Working with alphanumerical LCD display

Until now, the display methods we have investigated were the 7 segment digit display module and the serial terminal based on the UART interface. The first method is very limited (it can only display 2 digits and has high current demands) and the second method not only that it is uncomfortable but it also implies the connection to a host computer, where a serial terminal software is running, which is also not user friendly. A much more accessible method for a display interface will be presented during this laboratory work: an alphanumerical LCD.

Many embedded systems have an LCD as an interface display. LCDs may be graphical or alphanumerical. The graphical LCD is a little harder to use than an alphanumerical one which is why the latter will be presented in this laboratory work. The LCD which will be presented is a 2 line alphanumerical LCD with an integrated controller which is handled via an 8 bit parallel interface along with a couple of control signals. It is important to mention that not all available LCDs are controlled the way this laboratory work presents it. The communication method is usually dependent on the LCD controller. The LCD controller that we will use is the popular Hitachi HD44780U [26] which will be used on LCD similar to the one produced by Shenzhen Eone Electronics, the 1602A-1 LCD module [27, 28]. Such LCDs have a parallel bus exported on the pins for communication along with some control signals. The pinout of this LCD family may be the following:

Number	Symbol	Description	Comments
1	GND	Ground	Usually a 5V power supply is needed. Check module datasheet for details
2	V_{dd}	Power Supply	
3	V_0	Contrast	Connected to 5V through an adjustable resistor
4	RS	Register Select	High – Data, Low – Instruction
5	RW	Read/Write	High – Read, Low – Write (from/to LCD)
6	E	Enable/Strobe signal	Usead to enable data transfer/strobe
7	DB0	Data bus lines	Not used in 4 bit data transfer
8	DB1		
9	DB2		
10	DB3		
11	DB4		Data bus lines may be used for 4/8 bit data transfer. Only these lines may be used in 4 bit data transfer mode
12	DB5		
13	DB6		
14	DB7		
15	BLA	Backlight GND	Backlight power supply. Not necessarily 5V. Sometimes 4.6V
16	BLK	Backlight Power Supply	

Table 13 LCD pinout

Based on their function, there 3 major groups of pins: data bus line pins, communication control pins and power pins. Regarding the power pins, there are separate pins for general LCD and backlight power supply. The LCD general power is assured by pins 1 and 2 and usually according to the documentation only a 5V power supply may be used. This is not a general rule, in some implementations a wide interval of power supply values is permitted. This information is present in the datasheet. Same applies for the power supply of the backlight of the LCD but this power supply is optional. The LCD may be used without any problems without a backlight. One important pin that needs to be taken into consideration is the V_0 pin (pin 3) which is responsible for the contrast adjustment. This pin should be connected to a power supply through an adjustable resistor.

Another important group of pins is represented by the data line bus. The bus consists of 8 data pins, thus being an 8 bit data bus. The special aspect about this data bus is that it can function as a 4 bit data bus. Having this approach is of course slower than using an 8 bit data bus but fewer pins are required. The 4 bit data bus consists of the lines DB4-DB7 (pins 11-14). The DB0-DB3 lines are not used in 4 bit data bus transfers.

The last group of pins is represented by 3 control pins. One of the control lines is the RS line which specifies whether the data on the bus lines is instruction or data. This line should also be settled and stable during the transition of the E signal. A similar line is the RW line which specifies whether a read or write operation is performed. When RW is logic 1, then a read operation is performed and when the RW line is logic 0, a write operation is performed. In most of the cases write operations are needed. Only 2 situations are related to the read operation: the read of the internal status flag and current address counter and the read of the character in the memory at the current address counter.

The most important pin in communication is the enable E pin. This pin functions as a strobe pin. According to the documentation [27], the time diagrams on page 5, the in inactive state of line E is logic 0. After the data has been established and stabilized on the data bus (4 or 8 bits), the LCD will sample the data when a transition of line E from 0 to 1 followed by a transition from 1 back to logic 0. As it was stated before this behavior is similar to the behavior of a strobe line. The E line is a strobe/enable line not only for the data lines but also for the RW and RS lines. A general and much more simplified example of how the E lines needs to be driven is presented in the following diagram:

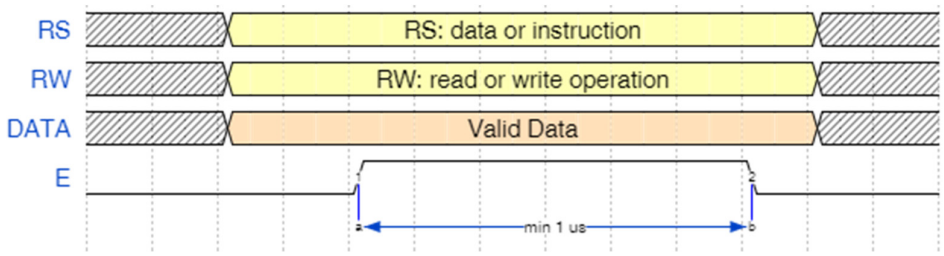

Fig. 4-49 LCD general bus communication example

As it can be observed in Fig. 4-49 the RS, RW and data bus lines are settled to the correct and valid value during the inactive state (logic 0) of line E. After the lines are considered stable the E line is used to enable/strobe the value through the 0 to 1 and back

to 0 transition. During the time the E line is at logic 1 the LCD controller reads the data present on the lines. Any changes of the lines during the E line is at logic 1 is forbidden. A violation of this rule may result in unpredictable and unknown behavior of the LCD.

When an 8 bit bus transfer is used then the whole byte will be sampled by the LCD when the correct transitions are applied to the E line. On the other hand, when 4 bit bus transfer is used, 2 operations will have to be performed in order to transmit the whole byte. First, the most significant 4 bits of the byte will be applied on the DB4-DB7 data lines followed by the correct transitions of the E line, then the least significant 4 bits of the byte will be settled on DB4-DB7 data lines followed by the correct transitions of control line E.

Before working on the hardware, in order to better understand how the LCD works, we will use a simulator to implement the communication protocol. The simulator, as presented in Fig. 4-50, is web based and implemented by Dincer Aydin who made it publicly available for everyone who is interested in teaching or learning to work with these kind of LCDs [29].

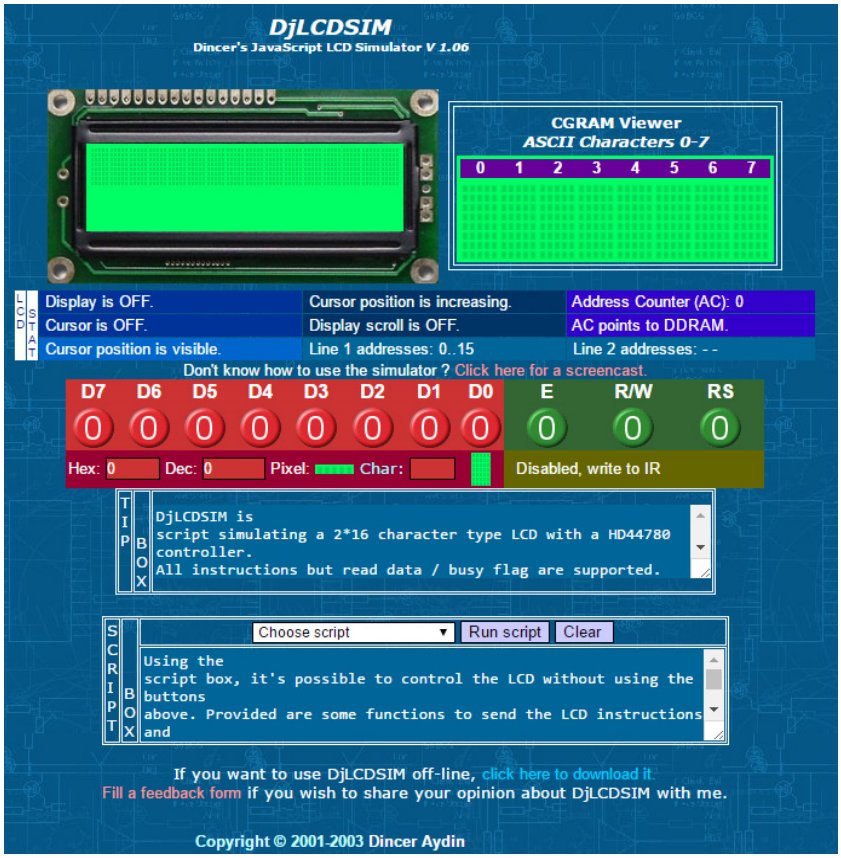

Fig. 4-50 LCD simulator screenshot [29]

As it can be seen in the screenshot above, the simulator offers the display of the LCD as a result, as well as the CGRAM memory viewer and the current settings in the LCD STAT section. In order to control the LCD the data and control lines are available for the user. The data lines are the ones colored in red and the green lines are the control lines. The correct transitions of the lines as requested by the protocol may be applied here and the LCD will respond accordingly. Furthermore, important tips will be available in the text boxes below the LCD. With the help of the documentation one could easily learn how to control the LCD using this simulator.

ASSIGNMENT 1: Using the documentation of the LCD and controller configure the LCD on the web based simulator to work with both 2 lines, with a blinking cursor and write “Hello World” on the first line and your name on the second line. Consider an 8 bit wide data bus.

ASSIGNMENT 2: Have the same requirements as in the previous assignment but consider a 4 bit wide data bus.

After a successful configuration and usage of the LCD on the simulator was made, the next step is to connect the LCD to the microcontroller and use it in a real embedded system environment. Again, as in the previous laboratory works, there are 2 main issues to be addressed: the hardware connections that need to be made and the software components that need to be written or integrated into the project. Giving the fact that the communication protocol is slightly complicated and also that there are many commands to be implemented for the LCD a specialized library will be used to handle the LCD.

The starting point in making the schematic for connecting the LCD to the microcontroller will be the combination between the schematic from Laboratory work 5 - Read 4x4 keyboard 16 keys where the keyboard is connected to the microcontroller and the schematic from Laboratory work 6 - UART interface where the microcontroller is connected to the serial interface of the computer through the peripheral board. The resulting block schematic for this laboratory work should look like the following:

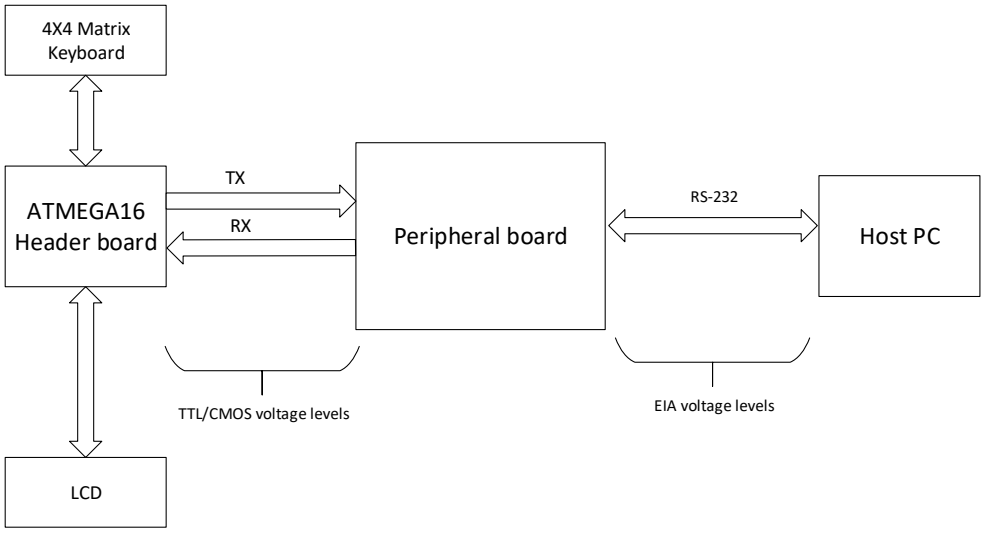

Fig. 4-51 Block schematic

With the help of the pinout of the LCD, presented in Table 13, the following connection have to be made in order to connect the LCD to the microcontroller using a 4 bit bus:

- Pin 1 of the LCD connected to the common ground
- Pin 2 of the LCD, the V_{dd} power pin, has to be connected to a 5V power line (same as the microcontroller)
- Pin 3 of the LCD, the V_0 contrast pin, has to be connected to the 5V power line through an adjustable resistor. In our case we will connect this line to the POT header pin of the peripheral board which connects it to the onboard potentiometer
- For the data and control pins lines from PORTD and PORTC of the microcontroller will be used as following

Microcontroller		LCD	
Pin number	**Pin name**	**Pin number**	**Pin name**
18	PD4	11	DB4
19	PD5	12	DB5
20	PD6	13	DB6
21	PD7	14	DB7
16	PD2	6	E
17	PD3	4	RS
22	PC0	5	RW

Table 14 Signal interconnection between LCD and microcontroller

- The backlight power pin are not going to be connected
- The DB0-DB3 signals of the LCD are also going to be left not connected

Regarding the software aspect of this laboratory work for controlling the LCD, as stated before, an already developed C library will be given to the students. The LCD library is structured as a standard C library with a code file and a header file. The library is highly configurable and may control the LCD with any given connections. The configuration of the library is made through a collection of preprocessor defines in the header file. The most important defines related to the port lines that are connected to the LCD. Each port line must be correctly specified in the header file. Another important parameter in the library is the clock frequency of the microcontroller (the XTAL define – in our case it should be set as 14745600UL, where the UL specifier denotes the fact that the constant is to be considered as a long unsigned).

As stated before, all the necessary defines that need to be configured are present in the header file of the library.

ASSIGNMENT 2: Add the provided library to the project. Open the header file of the library (lcd.h). Make a list with the "defines" in the library that need to be configured and have them verified by the laboratory instructor. Consider Table 14 as a reference interconnection. Test the integration with the new library by writing a main function code that configures the LCD with a blinking pointer. Write "Hello World" on the first row of the LCD and your name on the second row of the LCD.

ASSIGNMENT 3: Write an application which combines Laboratory work 5 - Read 4x4 keyboard 16 keys, Laboratory work 6 - UART interface and this current laboratory work. The application should read the keyboard and display the pressed key on the LCD and also send it over the serial interface as was implemented in the previous laboratory work. The LCD should be configured to scroll the text over the lines. When the first line is full, the software should switch displaying the next data on the second line. When the second line is full, then the newly pressed key should be displayed on the last available position after the whole text has been scrolled to the left losing the first character that was previously displayed on the LCD.

4.9 Laboratory work 8 - Analog to Digital Converter

This laboratory work is aimed at both introducing a new peripheral module of the microcontroller to the students and at presenting an analog temperature sensor. There will be 2 finalities for this laboratory work. One finality is represented by a digital voltmeter application and the other will be a digital thermometer using the LCD for displaying the temperature.

The analog temperature sensor and the analog to digital converted have been combined into a single laboratory work mainly because the analog to digital converted is needed in order to read the temperature from the analog sensor. An analog temperature sensor outputs an analog voltage level that is proportional with the outside temperature.

Before working with the analog temperature sensor the analog to digital converter (ADC) peripheral module needs to be presented first and the digital voltmeter application is the best way for testing the functionality of the ADC.

The analog to digital conversion is a set of operations which transforms an analog input voltage into a binary code offered as output. This process in performed in 3 steps: sampling, quantization and binary coding.

Sampling is the first step of analog to digital conversion and consists into acquiring values of the analog input usually at periodic moments in time. The values of the samples are still continuous and belong to an infinite precision interval. The next step, the quantization, is the one responsible in obtaining finite precision values of the samples. The final step of the process is the binary coding which practically represents the values obtained after quantization using numbers represented on a finite number of bits [19].

The number of discrete values on which an ADC can represent the samples is indicated by the resolution of the ADC. This parameter is one of the most important characteristics of an ADC. For example having a resolution of an ADC of 10 bits means it can convert an analog voltage value into 1024 different levels in an interval of discrete values from 0 to 1023. The resolution can also be presented in volts, introducing the term called the least significant bit voltage. The LSB represent the minimum change of the input voltage in order for the output binary code to change. The resolution of the ADC can then be defined as:

$$Q = \frac{FSR}{2^n - 1} \quad (4\text{-}8)$$

Where
FSR – Full Scale Range defines full voltage range of the ADC
n – Represents the number of bits the ADC uses to encode the sample, practically the resolution in bits.

The Full Scale Range can be defined as follows:

$$FSR = V_{REF_{HI}} - V_{REF_{LO}} \quad (4\text{-}9)$$

Most of the conversions made are for voltages that are referenced to ground (0 V), thus in this situation the FSR is usually equal to the reference represented by the highest value of the voltage.

Having a simple example where the high reference of the ADC is 3.3 V, the low reference of the ADC is grounded and the resolution of the ADC is 8 bits, then we can calculate the voltage resolution of the ADC:

$$Q = \frac{3.3\ V}{2^8 - 1} = 12.94\ mV \quad (4\text{-}10)$$

Practically every increase of the input voltage by 12.94 mV causes the encoded value of the ADC to change. Practically, in our example, the ADC only "feels" changes of 12.94 mV.

As state before the actual output of an ADC is a number which is *n* bits wide, with a maximum value of:

$$N = 2^n - 1 \quad (4\text{-}11)$$

So, practically, having the output of the ADC of a converted input voltage, denoted as ADC_{VALUE} the voltage represented by this discrete value, denoted as V_{RESULT} can be calculated as follows:

$$V_{RESULT} = ADC_{VALUE} \cdot Q = ADC_{VALUE} \cdot \frac{FSR}{2^n - 1} \quad (4\text{-}12)$$

Considering our previous example, let us continue by supposing that after the conversion the ADC gives a result binary represented as 0b01111001 which in hexadecimal is equal to 0x79 and in decimal as 121. The actual value of the voltage that the ADC converted (V_{RESULT}) may be calculated (in mv) as follows:

$$V_{RESULT} = 121 \cdot \frac{3300\ mV}{2^8 - 1} = \frac{121 \cdot 3300}{255} = 1565\ mV \quad (4\text{-}13)$$

The ATMEGA16 microcontroller has an Analog to Digital converter module which will be used in order to implement a digital voltmeter during this laboratory work. The ADC of ATMEGA16 has 8 possible channels for conversion with different reference voltage selection possibilities. The ADC also supports the possibility for converting differential signals where the low reference is not considered to be grounded. During these laboratory works we will not consider differential conversion thus only using channels that have the low voltage reference grounded and the high voltage reference equaled to the Vcc power line. The maximum resolution of the ADC is 10 bit.

For this laboratory work we will use the maximum resolution of the ADC. The reference of the ADC will be the AVCC power pin.

Using the maximum resolution of the ADC means that 10 bits of data will be received from the ADC module. The 10 bit unsigned integer will be given by the ADC module within two 8 bit registers: ADCH containing the most significant part of the result (most significant 2 bits) and ADCL containing the least significant part of the result. The result needs to be stored into a 16 bit unsigned integer variable which will combine the 2 registers using bitwise operations.

The next aspect that is to be discussed is related to the connections that have to be made. The starting point will be the connection schematic used in Laboratory work 6 - UART interface for serial communication as presented in Fig. 4-39. Having this as a starting point the resulted block connection schematic for this laboratory application is presented in Fig. 4-52:

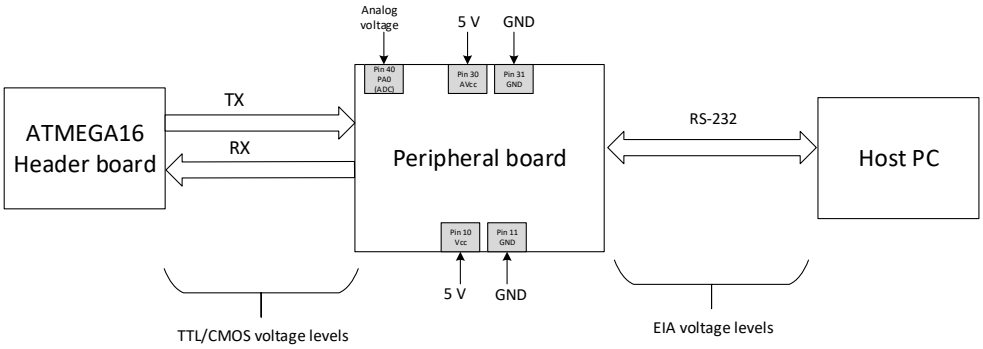

Fig. 4-52 ADC connection block diagram

The new connections added to Fig. 4-39, resulting Fig. 4-52, are the following:

- The power supply voltage for the entire circuit will be 5V
- In order to use the ADC the AVCC pin (pin 30) MUST be connected to the power supply, in our case to the 5V line. Same goes for the ground pin next to it which, of course, must be connected to the ground line (pin 31 needs to be connected to ground)
- The analog input voltage should be connected to one of the ADC input channels, thus, for simplicity, ADC0 channel 0 was selected in the previous diagram. Practically, the PA0 pin (pin 40) will serve as ADC0 function and will be connected to an analog voltage

The analog voltage that will be applied to the ADC can be generated using an adjustable laboratory power supply. The ground of the laboratory power supply MUST be connected to the ground of the whole circuit. Another important aspect is that the analog voltage MUST NOT EXCEED 5V. The interval in this situation should be [0, 5] V. In order to verify the results, it is recommended that the laboratory adjustable power supply has the possibility to display the actual voltage that it is applied. If not, then, a real voltmeter should be used in order to be assured that the upper limit of the voltage interval is not exceeded.

ASSIGNMENT 1: Make the necessary connections and have the laboratory supervisor verify them.

The next aspect, which should be discussed, is related to the software part of the application which should imply the configuration of the ADC of ATMEGA16, the starting of the conversion, the collection of the raw data result, the calculation of the actual voltage in mV and the printing of the result on the serial terminal.

ASSIGNMENT 2: Read the documentation regarding the ADC module of ATMEGA16 concentrating on the registers. Make a list with all the registers that should be used for configuring the ADC module. Furthermore, make separate list of the registers used for starting a conversion, for waiting for the conversion to be finished and for collecting the result. Define the algorithm for both operations (configuration, conversion). The ADC should be used without interrupts, having completion checked by polling and with a resolution of 10 bits.

The ADC of the ATMEGA16 is not hard to use. It has a small number of registers and the operation is almost trivial. The first aspect to be considered is the configuration of the direction of the pins involved in ADC conversion. As stated before, the ADC has 8 channels for conversion and are all mapped on PORTA of the microcontroller. In this case, the programmer has to be assured that the direction of the pins involved in conversion is set to input.

The configuration of the ADC is mainly done by writing the necessary bits in ADCSRA register which is the ADC Control and Status Registers. The most important bit that needs to be set prior to any usage of the ADC is the ADEN bit which enabled the ADC module. Another interest from this register are the ADPSx bits which form the clock division factor. Having a starting point, a divisor of 8 should be used having the bits equaled to the following values: ADPS2 = 0, ADPS1 = 1, ADPS0 = 1. Practically, along with the direction configuration, these writings into the ADCSRA register should suffice. A flowchart of the function that implements the configuration of the ADC module may be found in Fig. 4-53:

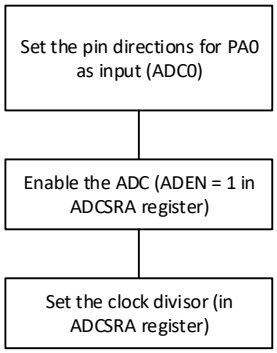

Fig. 4-53 ADC initialization flowchart

The slightly complicated part of the ADC is implementing a method to start the conversion, wait until the conversion is complete and collect the result. The first step is writing the ADMUX register by selecting the channel for conversion and the voltage reference (AVCC in our case). The ADMUX register also contains a bit called ADLAR which specifies whether the result is left or right adjusted. In this situation we will use the result right adjusted by keeping the ADLAR bit logic 0.

After configuring the ADMUX register, the conversion can be started by writing ONLY the ADSC (ADC Start Conversion) bit in ADCSRA register. This bit should also be used for checking whether the conversion has finished. According to the

documentation, after this bit is set to logic 1, the conversion begins. After the conversion is finished this bit is reset to logic 0, by hardware, signaling the completion of the conversion.

When the conversion is finished the only thing remaining is collecting the result. We have considered the maximum resolution of the ADC which translates into the fact that the storage variable should be a 16 bit unsigned integer capable in holding the 10 bit result. As stated before, the result is split into 2 registers ADCH and ADCL containing the most significant part of the result and the least significant part of the result. The final result needs to be obtained by combining the ADCH and ADCL registers using bitwise operations. The only limitation of the microcontroller, which is presented into the documentation, is that the ADCL register must be read first and the ADCH last. The flowchart for reading a sample from one of the ADC channels may be found in Fig. 4-54.

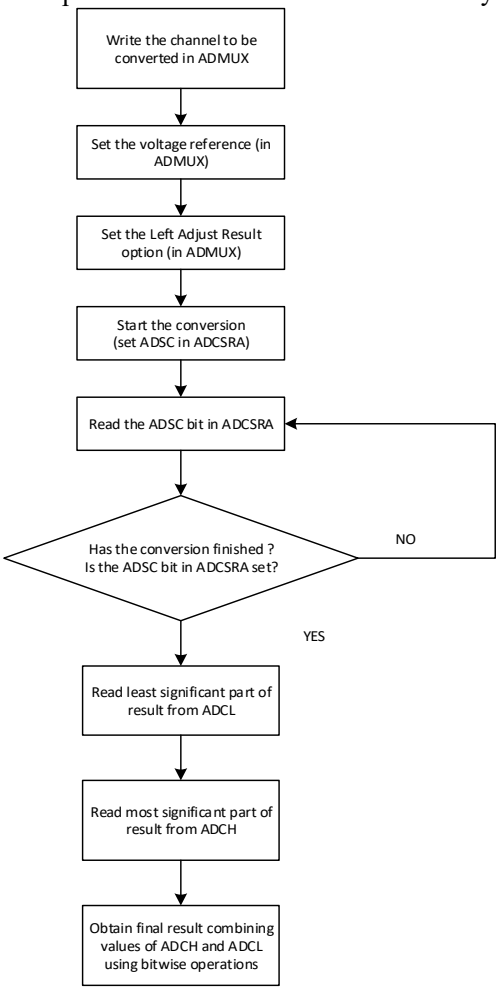

Fig. 4-54 ADC conversion logic

ASSIGNMENT 3: Write a C library (a c file with a header file) which contains an initialization function for the ADC mode and a function capable of starting the conversion and reading one raw sample value from the ADC on channel 0. Add the library to the project previously developed. The new C library should have the following files which should be added to the existing project:

- Adc.c – the C file containing the implementation of the ADC functions (init function and read data function)
- Adc.h – the header file containing the declarations for the functions implemented in adc.c file that need to be exported to the rest of the program

Test the newly developed library responsible for interfacing with the ADC by implementing a digital voltmeter which outputs the value in mV to the serial terminal and on the alphanumerical LCD. As stated before, the test voltage should be brought from an adjustable laboratory power supply.

The main program should mainly read one sample from the ADC, calculate the resulted voltage and print the result on a new line containing VOLTAGE = <value> mV. Make use of the standard *sprintf* function in order to print into a string which should be sent over the serial line using the required function in the serial library. The program should print the voltage once per second. The newline is obtained by inserting the characters '\r' and '\n'.

Do not use float or double when calculating the resulting voltage value. Use only integers. ATMEGA16 does not have hardware support for floating point values.

Use Docklight scripting, as in the previous laboratory application, to view the result.

Calculate the value in mV similar to the examples in (4-12) and (4-13) taking into account that the resolution of the ADC is 10 bit. Implement a function and add it to the ADC library which converts the ADC data to voltage depending on the resolution. The function should return a value in mV and take 2 parameters: a parameter containing the ADC data read from the data registers and the resolution (the number of bits).

The flow of the program should be the following:

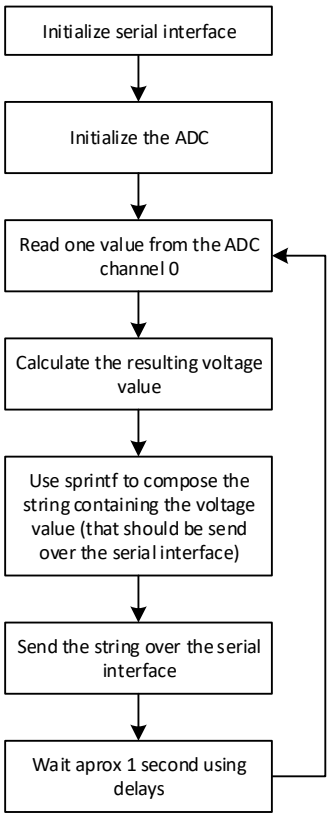

Fig. 4-55 Digital voltmeter program flow

The second and much more practical aspect of this laboratory work is to obtain the current temperature of the environment using an analog temperature sensor. This is the main reason that the first part of this laboratory work was aimed at presenting the analog to digital converter. The temperature sensor that we will use is LM35 which outputs an analog voltage level proportional to the outside temperature [30]. According to the documentation the output voltage of the sensor has a scale factor of 10 mV per 1 degree Celsius. Thus, the conversion formula from voltage to degrees Celsius could be the following:

$$Temp\ [°C] = \frac{OutputVoltage[mV]}{10} \qquad (4\text{-}14)$$

4.10 Laboratory project – Digital alarm clock

This final section is dedicate to a lab project which is represented, in general, by an alarm clock. This project aims at combining the lab works previously presented into a fully functional device. The idea of this project is to show the students the utility of the lab works presented above by constructing an alarm clock with multiple functions and features.

The main components that are available to the students are:

- 1 ATMEGA16 DIP40 microcontroller
- 1 LCD similar to 1602A-1 produces by Shenzhen Eone Electronics
- 1 LM35 temperature sensor
- 2 7 segment display modul
- 1 4x4 matrix keyboard
- LEDs
- Push buttons
- buzzer
- Serial interface MAX232 level translator
- Laboratory solder test board
- Tools
- Sockets
- Header pins
- Lab equipment
- JTAG debugger
- Software: Atmel Studio 7, Docklight scripting
- C library for LCD

The project must have the following mandatory features:

1. Display current date and time on the first line of the LCD
2. Display the current temperature of the environment on the second line of the LCD
3. Blink a LED once per second (when the current date and time is incremented by 1 second)
4. Send the date, time and current temperature over the serial interface once per second
5. Set the current date and time by using 5 push buttons – 2 push buttons to select the field (hour, minute, second, day, month, year) – 2 push buttons to increment/decrement the field, 1 push button to enter and exit editing mode. After the new date time has been set validation is mandatory

6. After the press of one push button display the alarm time for 3 seconds then switch back to the usual display (as in requirement 1)
7. When the alarm time has been reached use a LED to simulate the alarm. Use a push button to stop the alarm
8. The calendar must be implemented correctly including leap years.

All the mandatory features presented above must be implemented in order to pass the project and receive a minimum grade. Also, for a minimal grade, students may implement the schematic on a breadboard.

The project should be extended by adding some of the following additional features:

1. Instead of simple push buttons, connect a 4x4 matrix keyboard and use it to implement all the controls. Use the numbers on the keyboard instead of push buttons to increment/decrement to set a certain field (hour, minute, second, day, month, year). Also select the fields by used dedicated buttons on the keyboard if possible. Any solution is accepted.
2. Instead of the LED to simulate the alarm, use a buzzer instead
3. Implement a serial protocol in order to set the current date time and the alarm time over the serial interface
4. Connect 2 module of 7 segment display to the microcontroller using multiplexing technique and display the current second. Additional components are provided if needed, along with support.

Student may implement the schematic on a solder laboratory test board and solder components and make the wiring soldered. The components are NOT to be solder directly on the board thus sockets will be provided. Students are encouraged to implement their board on soldering test boards and not on breadboards and also to add additional features to the project.

Bibliography

[1] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*: Prentice Hall Professional Technical Reference, 1988, ISBN: 0131103709.

[2] Olimex Ltd, "MOD-LCD4.3 development board User's Manual, Revision E," Koninklijke Philips Electronics N.VSeptember 2012.

[3] NXP Semiconductors, "LPC178x/7x Preliminary Datasheet," Koninklijke Philips Electronics N.VMay 2012.

[4] CooCox - Free/open ARM Cortex-M Development Tool-chain. (2015). *CoLinkEx User Guide*. Available: http://www.coocox.org/wiki/coocox/CoLinkEx/CoLinkEx-User-Guide

[5] Digi International, "XBeeTM/XBee-PROTM OEM RF Modules. Product manual v1.xAx - 802.15.4 protocol," Digi International Inc. 2007.

[6] Digi International, "XBeeTM Series 2 OEM RF Modules Product manual v1.x.2x - ZigBee Protocol," Digi International Inc. 2007.

[7] WizNet Co. Ltd, "WIZ820io User Manual," September 2011.

[8] SimCom, "SIM908-C Hardware Design," September 2011.

[9] NXP Semiconductors, "LPC178x/7x User Manual," Koninklijke Philips Electronics N.VSeptember 2012.

[10] Electronic Industries Association. Engineering Department, "Interface between data terminal equipment and data communication equipment employing serial binary data interchange," ed. Washington: Electronic Industries Association, Engineering Dept.,, 1969.

[11] SimCom, "SIM908 AT Command Manual V1.01," July 2011.

[12] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*: Addison-Wesley Longman Publishing Co., Inc., 1986, ISBN: 0-201-10088-6.

[13] Wikipedia. (2016). *Mobile phone signal*. Available: https://en.wikipedia.org/wiki/Mobile_phone_signal

[14] Atmel, "ATmega16 Datasheet - 8 bit AVR Microcontroller with 16K Bytes In-System programmable Flash," Atmel 2010.

[15] Atmel Corporation, "Atmel ICE User Guide," July 2014.

[16] Freescale Semiconductors, "MCU PROJECT BOARD STUDENT LEARNING KIT (PBMCUSLK) - Prototyping Board with Microcontroller Interface," Freescale SemiconductorJuly 2007.

[17] Adam Osborne, *An Introduction To Microprocessors* vol. 1: Osborne-McGraw Hill Berkeley California, 1980, ISBN: 0-931988-34-9.

[18] Texas Instruments, "MAX232x Dual EIA-232 Drivers/Receivers," Texas InstrumentsFebruary 1989.
[19] M. V. Micea, "Proiectarea si implementarea sistemelor timp-real pentru aplicatii critice de achizitie si prelucrare numerica de semnal," PhD, Politehnica Timisoara, 2004.
[20] Analog Devices, "ADSP-BF537 Blackfin Processor Hardware Reference Manual," Analog Devices Inc.March 2009.
[21] Analog Devices, "AD1871, Stereo Audio, 24-bit, 96 kHz, Multibit Sigma-Delta ADC," Analog Devices Inc. 2002.
[22] Analog Devices, "AD1853, Stereo Audio, 24-bit, 192 kHz, Multibit Sigma-Delta DAC," Analog Devices Inc. 1999.
[23] D. D. Gajski, *Principles of digital design*: Prentice-Hall, Inc., 1996, ISBN: 0-13-301144-5.
[24] Oana Boncalo and Alexandru Amaricai, *Proiectarea circuitelor digitale folosind Verilog HDL – Analiza si Sinteza*: Editura Politehnica, 2011, ISBN: 978-606-554-331-7.
[25] Kingbright, "Part Number: DA04-11EWA - High Efficiency Red," January 2011.
[26] Hitachi Semiconductor & Integrated Circuits, "HD44780U - Dot Matrix Liquid Crystal Display Controller/Driver," Hitachi Ltd 1998.
[27] Shenzhen Eone Electronics CO. Ltd, "Specification for LCD Module 1602A-1," Shenzhen Eone Electronics CO. Ltd, 2014.
[28] AKIHABARA INC, "SC162a," 2011.
[29] Dincer Aydin. (2006). *DjLCDSIM - Dincer's JavaScript LCD Simulator V 1.06*. Available: http://www.dinceraydin.com/djlcdsim/djlcdsim.html
[30] Texas Instruments, "LM35 Precision Centigrade Temperature Sensors," Texas Instruments 2015.