Hochschule Esslingen
University of Applied Sciences

Project Work

ARM9 interrupt latency measurement on an AT91RD9200-EK development board

Christian Wörz
Student ID: 747833

Hochschule Esslingen
Faculty: Graduate School
Course of Study: Software-Based Automotive Systems

Supervisor: M.Sc. Vikas Agrawal

Processing time: 17.03.2014 - 27.06.2014

Revision History

Revision	Date	Author(s)	Description
1.0	27.06.14	Christian WÃűrz	created
1.1	18.07.15	Vikas Agrawal	Added static ip configuration, cygwin, options for makefile, tftpserver
1.1	30.07.15	Vikas Agrawal	Added functioning TFTP server with BDI2000

Contents

List of Abbreviations IV

1 Task description **1**

2 Environment **2**

2.1 The chip ATMEL AT91RM9200 2

2.2 WinARM 3

2.3 Programmers Notepad 2 4

2.4 Cygwin 4

2.5 TFTP-Server 4

2.5.1 BDI-Configuration-File 6

2.6 Static IP 6

2.7 TFTP Configuration 7

2.8 Telnet 7

2.8.1 BDI2000 Telnet command's 8

2.9 System overview 9

3 Hardware abstraction **11**

3.1 LED 12

3.2 UART 14

3.3 Digital Input 16

3.3.1 Sharp sensor 17

3.4 RTC 18

4 Interrupt Latency Measurement Service **20**

4.1 Advanced Interrupt Controller (AIC) 21

4.2 ILMS initialization 21

4.3 Interrupt concept 23

4.3.1 Time correction 24
4.3.2 Timestamp and result conversion 25
4.3.3 Measurement transmission 27
4.3.3.1 RS232 frame format 27
4.3.4 Measurement scenarios 28
4.3.5 Demo interrupt function 31

5 Conclusion **32**
5.1 Further improvements 32

List of Figures **34**

List of Tables **35**

List of Listings **36**

Bibliography **37**

List of Abbreviations

RTC	Real Time Clock
AIC	Advanced Interrupt Controller
RISC	Reduced Instruction Set Computer
PN2	Programmers Notepad 2
TFTP	Trivial File Transfer Protocol
UDP	User Datagram Protocol
RAM	Random Access Memory
CPU	Central Processing Unit
ILMS	Interrupt Latency Measurement Service
MIPS	Million Instructions Per Second
IRQ	Interrupt Request
FIQ	Fast Interrupt Request
PC	Program Counter
CPSR	Current Program Status Register
ISR	Interrupt Service Routine
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
DMA	Direct Memory Access
BCD	Binary Coded Decimal
JTAG	Joint Test Action Group
DIO	Digital Input Output
HAL	Hardware Abstraction Layer
API	Application Programming Interface
PIO	Parallel Input/Output
PMC	Power Management Controller

Chapter 1

Task description

The overall task is to investigate and develop a service that allows to measure the latency an interrupt generates. This has to be achieved without external hardware and as accurate as possible with an AT91RD9200-EK development board. The service should be generic and independent in terms of the main functionality. One other point is, that the service ought to create less overhead and not extent the measured interrupt more than necessary. Further more the the measured value has to be converted into a time unit (e.g. μs) and be compensated to represent the real period the interrupt needed without the may influencing measure service. For the system output the values hast to be transmitted by any terms of connection to a host computer, running a database and stores the received values. For better readability it is recommended that the transmitted values are plain text. This allows easier debugging and also a stand alone operation without the running database. To trace back the recorded latencies timestamps should be provided to identify when and maybe under what conditions, excess length in interrupt routine occurred.

Chapter 2

Environment

This chapter is a short introduction into the microcontroller and environment and software to program the chip. It provides information about the basic setup to get started with the controller plus the required software with the corresponding settings.

2.1 The chip ATMEL AT91RM9200

The chip AT91RM9200 is an ARM9-based microcontroller. The on Reduced Instruction Set Computer (RISC) based architecture features more than 1Million Instructions Per Second (MIPS) per MHz. Several other features like low power consumption and two switchable instruction set are also present. Furthermore the chip hosts an AIC which allows the system designer or programmer, a flexible and prioritized interrupt configuration. The 8-level priority allows nested interrupts. In general the processor has 16 registers. Table 2.1 illustrates the register names and their functionality.

Register	**Description**	**Notes**
R0-R7	general-purpose Register	Registers are unbanked
R8-R12	general-purpose Register	Registers are banked in case of the FIQ
R13	Stack pointer (software convention)	Banked register
R14	return address in an subroutine call	Banked register
R15	Program Counter (PC)	Banked register
CPSR	Current Program Status Register (CPSR) containing: • ALU flags (Negative, Zero, Carry, and Overflow) • Interrupt disable bits • One bit that indicates whether ARM or Thumb execution is used • The current processor mode (*usr,fiq,irq,svc,abt,und,sys* see table 2.2)	

Table 2.1: AT91RM9200 Registers

The banked registers are in hardware separate registers accessed with the same hardware address. This means, that each register depends on the current processor mode. For the Fast Interrupt Request (FIQ) there are more such banked registers. This allows a faster context switch with no need to save this registers. The FIQ-Interrupt Service Routine (ISR) should use the registers R8-R12 for the calculation or to process data. The interrupt entry and exit therefor could reduce the interrupt processor cycle count.
The chip also includes also a power management as well as a powerful debugging unit. Besides this there are several communication ports such as Universal Asynchronous Receiver Transmitter (UART), Universal Serial Bus (USB) or Ethernet located on the development board and supported by the core hardware.
Table 2.2 gives an overview about the different processor modes. Each mode has different privileges and priorities. The FIQ for example can't be interrupted by the "normal" Interrupt Request (IRQ) and serves a fast response and low latency. To transmit data it can use the Direct Memory Access (DMA) to send data with low Central Processing Unit (CPU) load. In general most of the applications will run in the user mode. The other modes are used to access protected resources or serve exception handling.[ATM09a][ATM09b][Stu]

Processor Mode	**Mnemonic**	**Description / Usage**
User	*usr*	Standard (user) execution state
FIQ	*fiq*	Fast data / interrupt processing mode
IRQ	*irq*	Used for General-purpose interrupts
Supervisor	*svc*	Protected mode for the operating system
Abort mode	*abt*	Implements virtual memory and/or memory protection
System	*sys*	Privileged mode for the operating system
Undefined	*und*	Mode for undefined instructions

Table 2.2: Processor Mode [ATM09a][ATM09b][Stu]

2.2 WinARM

The WinARM-package is collection of GNU and other ARM developing tools. It is supposed to run on Windows as host computer and includes with the GCC (compiler,linker and assembler) the way to generate target code. This one is needed, because the host system (in this project Windows 7) differs from the ARM-architecture and so the GCC takes over the part, as a cross compiler, of creating this ARM-specific target code.
To get started with the WinARM package download an actual version (used one: WinARM-20060606) of the package, extract and install it. For best practice locate the files in system root (C:\WinARM).
Then the following lines needs to be added to the system variable "Path":

- C:\WinARM\utils\bin; This inclusion would help to run make itself. There are also other executables such as sh, rm and cp.
- C:\WinARM\bin; The below inclusion includes the binary files for the cross compiling of the arm microcontroller.

2.3 Programmers Notepad 2

The Programmers Notepad is nice to have to be able to run the make utility. However, it is advised to use the shell to invoke the make command. This can be done using Cygwin in the Windows environment. To understand how it can be run, please look at the description below. To be able to compile a program the Programmers Notepad 2 (PN2) need to be **run as Administrator**, otherwise the software does not have the right privileges to pass parameters to the compiler. Therefor right click on the PN2.exe and select "Run as administrator". For the actual build the make-file hast to be opened inside the PN2 and then select "Tools" from the menu bar and click "Make all". This will generate all, by the make-file specified, files. The actual file-name for the output files is declared inside the "config.mk"-file as TARGET (in this project: TARGET= at91_bdi2000).

2.4 Cygwin

Cygwin is the linux emulation. The arm-elf files are still present in the WinARM directory. As soon as the Windows Environment variable **PATH** is updated with the location of the binary files, the cygwin shell recognises it and all the arm commands can be run from inside there. In the project x86 has been used although the underlying chipset is x86_64. The mirrors that has been used is FTP-HS-ESSLINGEN.

2.5 TFTP-Server

The Trivial File Transfer Protocol (TFTP) is a file transfer protocol and is used in the project to transfer the core configuration file to the BDI2000. One characteristic compared to a FTP-Server is, that the User Datagram Protocol (UDP) based TFTP does not provide mechanism for authentication.[Wik]
The TFTP-Server is also used to transfer the compiled runnable to the Microcontroller. Therefore the TFTP-Server should contain the flash and also the configuration file. Cygwin has a built in TFTP-Server and that has been used. The default port of TFTP Server

is 69.
To test if the TFTP server is running on your computer, you need to install the tftp client packet under Cygwin. It is simply known as tftp. If you do man tftp, then a man page should open with all the options and in the man page you will see that its a client. There is also a tftp-server which runs as a daemon (tftpd). The configuration file can be found under inet.d/tftp or xinet.d/tftp. The one under inet.d is for the user and the other is for the superuser i.e parameters under inet.d can be overridden.
However we will use another server. This can be installed by clicking on the exe file TFTPGui. This is written in python and has a very nice interface.
Once TFTPGui is started, start the server at 10.0.0.100 and you will see in the log windows that the server has started listening on the port 69. From Cygwin read any data. Type tftp and then connect 10.0.0.100 and then get readme.txt. You should now see a readme.txt in the same folder.
This worked great because the transfer is done internally i.e the files doesnt have to leave the computer. However, if we are using an ethernet port, then we need to configure the firewall as well so that the TFTPGui.exe although listening on the port 69, receives a command to get the file from its tftproot folder.
There are various ways to check if the port is being blocked by the firewall. First of all check if the port is up and running i.e there must be any program which have started to listen on the port. To test this run TFTPGui.exe and run netstat -an. You will see an entry with the port 69 under the UDP section i.e 0.0.0.0:69 as local address and *.* as foreign address. This means that this port is open and can invite any foreign address. If you stop TFTPGui.exe this line should disappear.
The next is start the logging option. The can be done by going to the Windows Firewall Advanced Security and then click on Firewall Properties. Over there, under the option Logging, you have to enable Dropped packets. The log file can be found under System32->LogFiles->Firewall->plogifle.log. The BDI2000 stores the configuration file internally. Hence to test the behaviour of a unsuccessful connection after a connection has been established once, you have to power off the BDI2000.
Also it is important to note, that if you are using a USB Ethernet or any other device, the profile of that device is not the same as the Ethernet port. It means, you might be connected to the companys network and the profile is either Domain or Private. Changing the firewall settings of the program for the private network wont work as the request for a connection is being received on the USB Ethernet and that is configured to be public.

tftp
connect 10.0.0.100

get readme.txt

2.5.1 BDI-Configuration-File

Besides the initialization of flash, clocks and Random Access Memory (RAM), the configuration file contains information about the target e.g. the CPU-type.
The [HOST]-section (Listening 2.1) comprises the Host-IP. The TFTP-Server has to use this IP-address to enable the BDI to connect and load the configuration file. The FILE parameter provides the default file name which loaded via the Telnet (see section 2.8) if no other file name is specified. The PROMPT can be also customized and is a good way to check if the configuration is loaded correctly. The START option is the address where the CPU starts if no other address is specified in the run command.

```
1 [HOST]
2
3 IP           10.0.0.100
4 FILE         at91_bdi2000.bin
5 FORMAT       BIN 0x20000000
6 LOAD         MANUAL          ;<AGENT> load VxWorks code MANUAL
7 PROMPT       Chris@BDI>      ;new Telnet prompt
8 DEBUGPORT    2001
9 START        0x20000000
```

Listing 2.1: Extract of the rm9200dk.cfg-file

2.6 Static IP

Since the development environment is in the PC, the PC and the BDI2000 should be inside the same subnet to be able to transfer the files. Therefore both of them are provided an IP address that is in the same subnet i.e only the last digit varies. The PC has the subnet of 10.0.0.100. The gateway is 10.0.0.1 and the subnet is 255.255.255.0. These values have already been transferred to the BDI2000 using the serial port. In the projects case, the BDI2000 expects a file rm92000dk.cfg in the root directory of the TFPT server.

2.7 TFTP Configuration

The TFTP server or tftpd (TFTP Daemon) can be configured also using the cygwin tool. Following file needs to be edited with the following command

Sourcecode

2.8 Telnet

Telnet is a widely common protocol to connect clients to their servers. It works across different platforms like Windows, Unix or Mac OS. How ever the TCP-based connection is rather insecure, because all data is transmitted as plain text including passwords for example.
The protocol is separated into two parts. The server is located at the BDI2000. The notebook needs to connect to the server via a Telnet-client. For best practice use PuTTY to connect to the BDI. The configuration setting is IP:*10.0.0.101*, Port: *23* and *Telnet* as connection type.
The most useful work and debug commands are listed in Table 2.4.

2.8.1 BDI2000 Telnet command's

After a successful make of the source code, the target binary is located with in the same directory where the TFTP-server has access. There is no need for a user or password login. After the first power up the BDI requests the "rm9200dk.cfg"-file. Therefor the TFTP-server has to be started and the configuration file need to be provided. Otherwise the BDI replies inside Telnet with "# CONFIG: cannot open rm9200dk.cfg". The transfer of this file could be seen inside the TFTP-window. A second file, the "reg9200.def" is also transmitted, and is used to access hardware addresses by their mnemonics. If all cables are connected correctly (see system overview 2.9 for details) the target messages a successful startup.
To get started and run the target binary, these three simple commands has to be entered in the terminal:

1. ***halt*** // Forces target into debug mode. Displays also the Contend of PC and CPSR
2. ***load*** // Loads the (default) program to the target memory
3. ***go*** // Sets the (default) PC-address and starts the target

The reg9200.def-file hosts information about the address and size in memory. Therefor the "RD"-command can be used with the the corresponding mnemonic. For all not described register addresses the more general "MD"-command has to be used. For example the value of a variable could be debugged with the "RD/RM"-command (variable address is known in the "project_win.map"-file) The following table 2.3 gives an example of two possible ways of reading and modifying commands for the RTC-register.

Command	**Access**	**Description**
RD RTC_TIMR	Read	Reads the RTC-time register
MD 0xFFFFFE08 1	Read	Same as RD RTC_TIMR, 1 $\hat{=}$ 32bit word
RM RTC_TIMR 0x00134530	Write	Sets the time to 13:45:30
MM 0xFFFFFE08 0x00134530	Write	Same as RM RTC_TIMR 0x00134530

Table 2.3: Example of r/w access of the RTC time register[1]

[1] To set a time, the RTC-control register needs to be modified first (see 3.4 for details)

Command	Description
MD [<address>] [<count>]	display target memory as word (32bit)
MM <addr> <value> [<cnt>]	modify word(s) (32bit) in target memory
RD [<name>]	display CPU or user defined register
RM <name> <value>	modify CPU or user defined register
RESET	reset the target system
BREAK [SOFT | HARD]	display or set current breakpoint mode
GO [<pc>]	set PC and start target system
TI [<pc>]	single step an instruction
HALT	force target to enter debug mode
BI <addr>	set instruction hardware breakpoint
CI [<id>]	clear instruction hardware breakpoint(s)
INFO	display information about the current state
LOAD [<offset>] [<file> [<format>]]	load program file to target memory
CONFIG	display or update BDI configuration
HELP	display command list
QUIT	terminate the Telnet session

Table 2.4: Extract BDI2000 command list[AG05]

2.9 System overview

Figure 2.1 visualize an overview of the system. The figure only indicates the data connection. Each device is powered individual via battery or cable.
The host running the TFTP and Telnet-servers is connected via Ethernet to the BDI2000. The IPs have to be assigned in a static way. The computer uses Telnet and the corresponding commands, specified in the section 2.8, to connect to the BDI2000. The BDI itself connects to the AT91RM9200 via a Joint Test Action Group (JTAG)-interface. The Development board with all its interfaces is connected to a computer via a bidirectional UART. The interface itself and its settings, are described in section 3.2. The user RS232 connection is used in this project to transmit the measurement values in a formatted way. The demo interrupt input is connected via a Digital Input Output (DIO) of the development board. The board also provides the power for the infrared based distance sensor, through the USB 5V power supply. This ensures also a common ground, need to read the input voltages correctly.
In the case of programming the data flow is only between the computer and the BDI, as well as between the BDI and the development board.
For the demo application and the Interrupt Latency Measurement Service (ILMS) (descried in chapter 4) the data flow is the following. The input sensor detects an obstacle in range. The output of the sensor turns to high level. The microcontroller process this

interrupt. The ILMS is started and as soon as there is no object, in reach of the sensor, stopped again. The result then is transmitted via the user serial communication to an computer running the database software. This could be the host computer, running the development environment or a separate one.

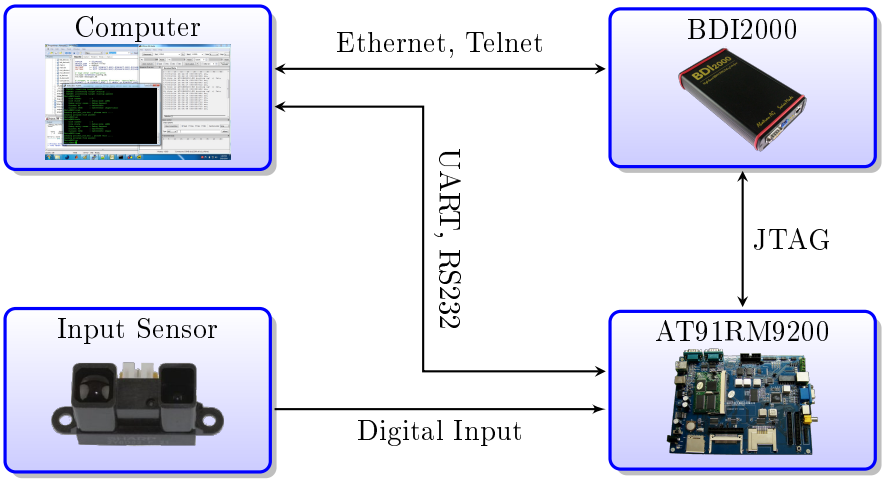


Figure 2.1: System Overview

Chapter 3

Hardware abstraction

In order to achieve an independent application a Hardware Abstraction Layer (HAL) is introduced in the project. The basic idea is to have a strong decoupling between the different layers in the project. Figure 3.1 displays the implemented structure. The benefit of this layout is, that with the same HAL and therefor the same Application Programming Interface (API), a switch to an other target hardware could be realized with no or little changes to the already running application. The so generic designed functionality allows scalability, extendability and hardware independence. Debugging or even the simulation of the application on the host computer is possible.

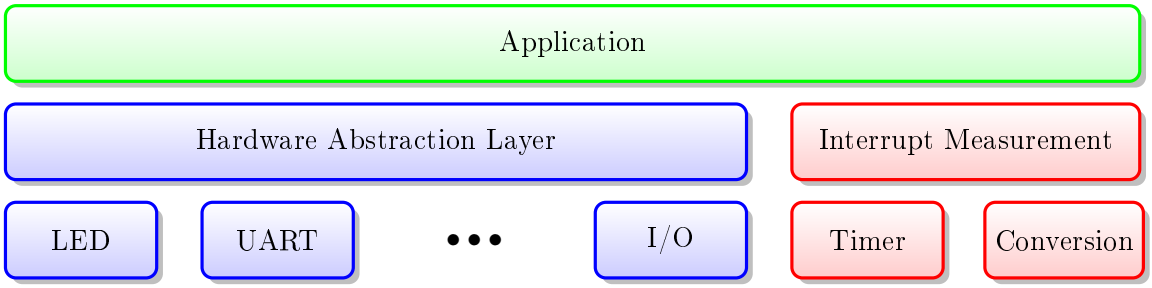


Figure 3.1: Projekt HAL layout

The layering in this project focus most on an abstraction of the hardware interfaces. There are for example different LED-commands allows the higher layers to turn on, off or toggle the states of one or more LED's with out knowing the hardware address or the circuit design. Such high level commands are also available for the UART interface. Serving simple send and receive functions for the application or other function. A similar hardware abstraction is also achieved for the I/O's. The application doesn't to take care of configuring, masking or filtering the I/O. For example digital input function delivers only

true or false as the input state. This HAL serves strict information hiding.This means in terms of the application that it has no direct hardware access. This is only realizable for limited functionality. Some real time critical services needs to accomplish fast reaction times and reach hard deadlines, via a direct hardware access. These services can not afford to go through the complete stack or layers. The in this project work realized ILMS is an example for such a service. Due to short latency to serve and measure an interrupt, to wait until a timer is available is not possible for this kind of service. Therefor like shown in Figure 3.1, one timer unit belong to the service as well as conversion function which is needed to convert the results into a human readable format.

3.1 LED

The development hosts 3 different LEDs. The LEDs are controllable via different bits inside the memory mapped IO. Listing 3.1 is the pin definition according to the circuit schematic (Figure 3.2). This allows for the LED-functions to use symbolic names and also be concatenated inside the function calls (E.g. "resetLed(GREEN | RED | YELLOW);").

```
1 #define GREEN  ((unsigned char) (1<<0))
2 #define YELLOW ((unsigned char) (1<<1))
3 #define RED    ((unsigned char) (1<<2))
```

Listing 3.1: Color definition for the LED-functions

The LED itself are low active. This means to be turned on a low voltage needs to be supplied at the corresponding output pin [ATM06].
The existed LED driver was changed and reworked in a way that the state of the LED is controlled via the output voltage. The old output enable based concept had the flaw that some unknown high frequency dimed the YELLOW-Led. The output signal is now also for extendability available at the output pins.

```
1 void Led_init()
2 {
3   //Enable Register!
4   AT91C_BASE_PIOB->PIO_PER  = AT91C_PIO_PB0|AT91C_PIO_PB1|AT91C_PIO_PB2;
5   //Enable output!
6   AT91C_BASE_PIOB->PIO_OER  = AT91C_PIO_PB0|AT91C_PIO_PB1|AT91C_PIO_PB2;
7   //Assign PB.0-2 to GND
8   AT91C_BASE_PIOB->PIO_CODR = AT91C_PIO_PB0|AT91C_PIO_PB1|AT91C_PIO_PB2;
9 }
```

Listing 3.2: Configure the LED-Pins as output

Listing 3.2 displays the Led_init()-function. The PIO_PER (PIO Enable Register) is set with the pins of the LED (PortB0-2). The output at the LED-pins is enabled via the PIO_OER (PIO Output Enable Register). To finish the initialization all LEDs are activated through a set in the PIO_CODR (PIO Clear Output Data Register). This is done to achieve a definite output state after initialization. The complex IO-register features via the clear and set-registers an adjustment of output pins without a read command of the current output state, followed by a dis- or conjunction and a final store command. This reduces needed processor clock cycles and registers for such small IO-operation to an (if possible) atomic instruction.
Table 3.1 lists the available LED-functions:

Function Call	**Parameters**	**Return**	**Description**
Led_init	None	None	Initialize the LED-pins, turns on all LEDs
toggleLed	Symbolic Name	None	Toggles the output of the specified LED(s)
setLed	Symbolic Name	None	Turns on the specified LED(s)
resetLed	Symbolic Name	None	Turns off the specified LED(s)
getLed	Symbolic Name	State	Returns LED(s) state

Table 3.1: LED-functions

Figure 3.2 is an extract of the development board schematic. The LEDs are supplied with 3.3V and a LED independent series resistor of 220Ω. The assignment to the corresponding hardware pis is also indicated.

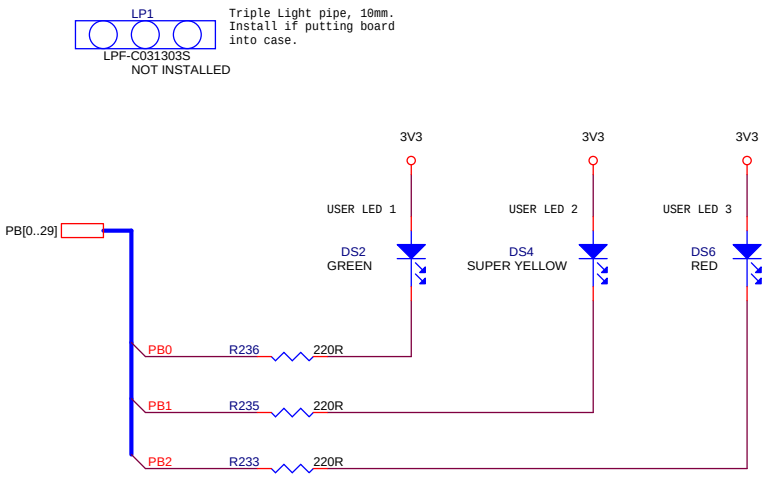


Figure 3.2: LED Schematic AT91RM9200_EK [ATM06]

3.2 UART

The Universal Asynchronous Receiver Transmitter (UART) is a widely used interface in embedded systems. It's simple to configure and features flexible data rates as well as wiring length. Also different physical mediums can be used. The protocol transmit short messages containing only several bits. How ever there is is no clock synchronization, therefor the specified clock frequency has to have such a precision, to stay with in all bit timings of a transmitted frame. The interface features handshaking mechanism, but a simple serial bidirectional communication can be achieved with only 3 wires.
The used protocol in this project is the RS232. The features are transmission between 5 and maximally 9 bits of data. The frame itself begins with a start bit followed by the data. A one bit parity field allows error detection of all odd number of bit errors. The frame is ended 1,1.5 or 2 bits known as stop bit.
The RS232 specifies a negative logic for the computer to computer connection. The voltage levels need to be converted via a level converter into a positive logic with the processor voltage levels.
The Table 3.2 shows the communication settings of the user as well as the debug interface. The handshaking mechanism is not used.

Baud Rate	**Data bits**	**Parity**	**Stop Bit**	**Handshake**	**Type**
115200	8	None	1	Not used	Bidirectional

Table 3.2: User RS232 settings (8-N-1)

There are different methods for sending and receiving the data. The controller features a fully in hardware realized UART transceiver. Therefor the received data is available for the application in a message buffer. For transmission a register used as a buffer to send out the data as soon as the there is no other data in transmission. The following enumeration explains 4 possible ways:

- **Blocking:**
  In case of receiving the application looks with a blocking function call if there is a new message in the buffer. If not it just loops the receiving status bit until there is new message available. This is called blocking, because the processor is fully occupied with the task of waiting till the status bit indicates successful received data. For sending the core is blocked while waiting, before the data is stored in the transmission buffer. This is not suitable for an application, because if there is no data incoming or the transmission buffer is full, this will lead to a dead lock.

- **Polling:**
  An other possibility is the polling approach. This differs between the blocking in the way of testing if transmission or new data is present. The application can check time or event based and if the communication interface is not applicable, continue with its other tasks. The polling frequency needs to be higher as the baud rate of the UART. How ever if the application load increases the the polling frequency (if not interrupt handled) decreases. This may lead to data loss on the receiving site or an decrease in data throughput on the sending side.

- **Interrupt:**
  Interrupt handling for the receive case ensures no loss in data an a fast processing. As soon as valid data is available the generated interrupt executes the corresponding ISR. The data is processed there. The core is only interrupted if new data is waiting for processing. In the case of no communication the processor has the full processing power to handle other tasks. To send data also an interrupt is generated when transmission is possible. When there is long communication ongoing, the processors load is increased. The core has to serve the ISR for every byte. This creates especially with high baud rates a high overhead for entering an exiting the ISR.

- **DMA:**
  To reduce the processor load, in the case of heavy communication, the DMA controller is used. The feature of the memory access enables in the sending case to only specify a memory pointer and a data length. The peripheral DMA controller then starts with the transmission till all data is transmitted. The processor therefor has to be interrupted only once. For receiving the DMA can be configured in a way, such that all incomming data is stored into a specific location in the memory. An interrupt then could be triggered if the buffer is full or all data is available.
  This method is used to transmit the result of the ILMS. The AT91RM9200 features also a transmit next register to start a transmission right after the first one has finished.[ATM09c]

3.3 Digital Input

For demo purpose (subsection 4.3.5) the duration of the high pulse of an digital input is measured. The distance measuring sensor unit (subsection 3.3.1) is connected to the development board.
The microcontroller includes with the Parallel Input/Output (PIO) a fully programmable input/output controller. Other as in section 3.1 the pin has to be configured as an input. The controller provides features like, programmable pull up resistors, input glitch filter as well as an input change interrupt. Because of the connected input sensor neither input filtering nor a pull up resistor (connected on the sensor side) is necessary.
Listing 3.3 gives an overview how the hardware input is configured. The InitDemoInterrupt function call (line 2) displays all function parameter by there mnemonic. At first the Power Management Controller (PMC) has to be configured (line 5). A special feature of the processor is PMC peripheral clock configuration. The activation of the corresponding pin enables the update of the pin status with the master clock frequency. If not enabled the input pin stores the last level when a the clock was enabled. Best practice is to disable the input clock for not used inputs, after they have executed their last operation.[ATM09d]
To be able to read input signals the pin has to be configured as input. This is done with the AT91F_PIO_CfgInput-function. The function performs two operations. The output of this pin is deactivated and the pin is enabled.
Line 7 configures the AIC (see section 4.1 for details) with the interrupt priority, the interrupt type and also the address where the ISR is located. Line 8 and 10 enables the interrupt. First in the PIO-controller and then in the AIC.
Reading the PIO_ISR (Interrupt Status Register) is necessary because the hardware only features an change interrupt. So when ever the interrupt occurs before the programming of the AIC or was not served the last time it needs to be cleared. This is done via reading the status register causing a clear of all pending PIO interrupts. When then the next PIO interrupts occurs the AIC can serve the change and run the corresponding ISR. An pending not served interrupt would be equivalent to a constant high level and would be never processed because of no change in the status register. Also any new interrupt would not affect this, because the status would just keep its value.[ATM09e]

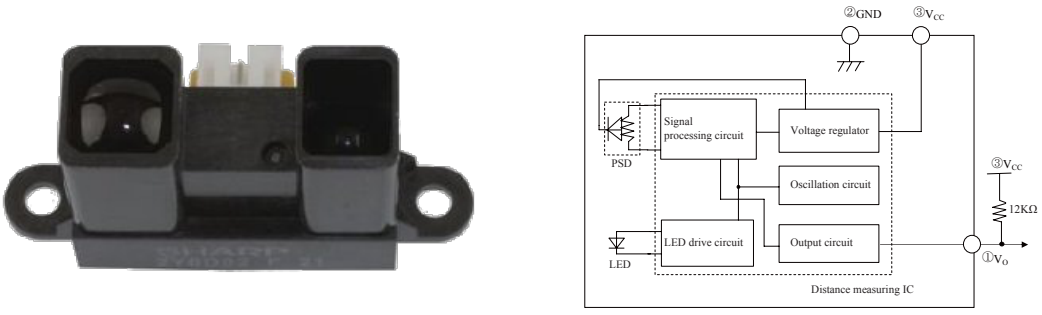
```
1 // Function call with parameters
2   InitDemoInterrupt(AT91C_BASE_PIOB,AT91C_ID_PIOB, MY_INT_PIN,
        AT91C_AIC_PRIOR_LOWEST, AT91C_AIC_SRCTYPE_EXT_POSITIVE_EDGE);
3
4 void InitDemoInterrupt(AT91PS_PIO PIOptr, unsigned int ParallelID, unsigned int
      MyIOpin, unsigned int priority, unsigned int intType){
```

```
5       AT91F_PMC_EnablePeriphClock (AT91C_BASE_PMC, ((unsigned int) 1 <<
            ParallelID));
6       AT91F_PIO_CfgInput (PIOptr, MyIOpin);
7       AT91F_AIC_ConfigureIt(AT91C_BASE_AIC,ParallelID,priority,intType,
            Measured_Interrupt_Lowlevel);
8       AT91F_PIO_InterruptEnable (PIOptr, MyIOpin);
9       {volatile unsigned int dummy; dummy = PIOptr -> PIO_ISR;}
10      AT91F_AIC_EnableIt (AT91C_BASE_AIC, ParallelID); }
```

Listing 3.3: Demo interrupt at PB15 configured as rising

3.3.1 Sharp sensor

The Sharp sensor GP2Y0D02YK0F (figure 3.3(a) is a distance measuring sensor with a range up to 80 cm. It provides an digital output high if an object is with in the range of the sensor. The measuring principle is based on an infrared emitting diode. Figure 3.3(b) illustrates the different blocks of the sensor. The emitted light is processed in a position sensitive detector (PSD) and the signal processing circuit provides the digital output. To avoid toggling the sensor has a hysteresis of about 10 cm. Also because of the sensor design, measurements closer than approximately 4 cm are not possible. In conclusion with the in accuracy of the sensor the detecting range is between 4 cm and 80 $\pm$ 10 cm. The sensor needs a power supply of around 5V with maximally 50 mA of current. The output stage is a so called open collector output and needs an external pull up resistor around 12 kΩ. The sensor is wired according to the recommended values. The power supply is connected to a USB port of the development board. The output is connected to PB15 which is located at pin C42 of the expansion slot.[SHA06]



(a) Sharp GP2Y0D02YK0F

(b) Schematic of the sensor supply

Figure 3.3: Sharp GP2Y0D02YK0F Distance Measuring Sensor Unit [SHA06]

The sensor provides a stable output signal of around 40 ms. Figure 3.4 displays the timing behavior of the sensor. The sensor update frequency is according to the timing limited by around 25 Hz. This is beneficial for the input of the microcontroller, because there is no need to filter this preprocessed, stable input signal. Line 4 in the Table 4.4 is a measurement of the shortest possible sensor high level duration. It is around 40.232 ms and with that in the range of the specified 38.3 $\pm$ 9.6 ms. Due to this fact all demo measurements of this table, are multiple of this duration. For example line 6 is a duration of 1167.025 ms and is approximately 29 times the minimum duration.

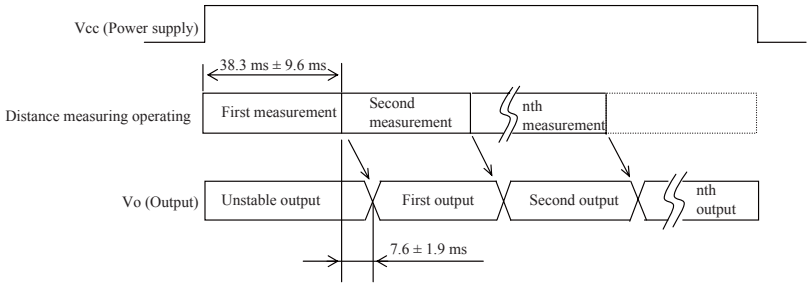


Figure 3.4: Sharp timing chart[SHA06]

3.4 RTC

The controller features a hardware Real Time Clock (RTC). It's used inside this project to supply the date and time stamp when an measured interrupt has occurred. The RTC has low power consumption and 200 year calendar. Also different interrupts can be triggered via programmable interrupts. The device uses the SLCK (32768 Hz) and divides this clock by 32768 to get a high precision 1 Hz clock. Time and date counting is done in hardware. The values are present in the Binary Coded Decimal (BCD) format. This means each decimal is represented via 4 bits. Lower 4 bits encodes the unit and the upper 4 bits the tens.[ATM09b]

Listing 3.4 lists the different commands to setup and initialize the RTC. Line 3 checks and waits till the SEC flag is set. This is done because the user has to wait at least one second after the last update (only necessary for high frequent updating). Next the flags for updating time and date are set in the RTC Control Register. The user also has to wait here, until the acknowledge is set in the RTC Status Register. Line 6 selects the 24 hour mode. The next commands fills the date and time structures with the user defined settings. These two 32 bit values are stored in line 18 & 19 in the RTC time and date

registers. These values are checked if they contain the right format and possible values. Otherwise the values are silently discarded. The clear in the next line disables the update request for time and date. The RTC starts running again. And SEC flag is cleared with the next command to wait at least one second before re updating the time.[ATM09b]

```
1  void Rtc_init(void)
2  {
3    while (!(AT91C_BASE_RTC->RTC_SR & AT91C_RTC_SECEV) );
4    AT91C_BASE_RTC->RTC_CR      = (AT91C_RTC_UPDTIM | AT91C_RTC_UPDCAL);
5    while (!(AT91C_BASE_RTC->RTC_SR & AT91C_RTC_ACKUPD) );
6    AT91C_BASE_RTC->RTC_MR      = 0;
7    rtc_time.time_bits.second   = 0x00;
8    rtc_time.time_bits.minute   = 0x35;
9    rtc_time.time_bits.hour     = 0x16;
10   rtc_time.time_bits.merid    = 0x00;
11
12   rtc_cal.cal_bits.century    = 0x20;
13   rtc_cal.cal_bits.year       = 0x14;
14   rtc_cal.cal_bits.month      = 0x06;
15   rtc_cal.cal_bits.day        = 0x06;
16   rtc_cal.cal_bits.date       = 0x02;
17
18   AT91C_BASE_RTC->RTC_TIMR    = (uint32)rtc_time.time_data;
19   AT91C_BASE_RTC->RTC_CALR    = (uint32)rtc_cal.cal_data;
20
21   AT91C_BASE_RTC->RTC_CR      = 0;
22   AT91C_BASE_RTC->RTC_SCCR    = AT91C_RTC_SECEV;
23 }
```

Listing 3.4: Minimal RTC initialization

Chapter 4

Interrupt Latency Measurement Service

The Interrupt Latency Measurement Service (ILMS) provides a service to measure the computation time of interrupts. This could be used to check for real time capabilities of a measured task or record the worst case execution time in a test scenario. This is done by observing and recording any duration the interrupt needed with different load scenarios. Also long run times in terms of weeks or months may elongate interrupts and increase the used stack. This latency measurement is calculated with the ILMS and transmitted over UART (Table 4.3) to a running and receiving database. A timestamp specifies the point in time when the execution of the interrupt started. For better readability this frame is transmitted as ASCII so also a standalone logging via a serial recorder is possible. The service is implemented in a way to use as less as possible resources of the processor and if possible compensate the measured result (see 4.3.1 for details). In order to archive for the application or programmer an easy way to specify a task to be measured, only the ISR needs to be set to the "Measured_Interrupt_Lowlevel". The programmers ISR then is automatically available as "Measured_Interrupt_Highlevel".
The main idea behind this is to start a timer, as soon as the low level routine is called through the AIC. At this point also the time stamp is saved. After this the high level routine is loaded and executed. After the task is completed the timer is stopped and based on the timer and master clock frequency a duration of the task is calculated. The result is converted in to ASCII and transmitted via the DMA through the UART interface. This method allows an on chip measurement of the latency as well as serving nested interrupts.

4.1 AIC

The microcontroller includes also an Advanced Interrupt Controller (AIC). This allows the user to assign a priority form 0 to 7 for each interrupt source. That permits that higher prioritized interrupts to be executed even if a lower prioritized ISR is executed at that time (interrupt nesting). The controller features different interrupt types for internal and external interrupts. Besides the priority, Fast Interrupt Request (FIQ) and regular Interrupt Request (IRQ) can be processed. The FIQ is always connected to the interrupt source 0. Banked registers enables a short context switch (section 2.1 and Table 2.1). Source 1 is reserved for system peripherals and source 2 to 31 for embedded peripheral or external interrupts. In this project the interrupt sources are, 3 for the digital input and 17 for the timer. The AIC is not affected by the PMC and no interrupt clock needs to be configured. Also vectoring provides an optimized way for branch and execution.[ATM09f]

4.2 ILMS initialization

The ILMS provides 3 core functions. An extract of the "InterruptMeasurmentService.h" in Listing 4.1 displays these with their parameters. Before using the ILMS the service needs to be initialized.

```
1 //InterruptMeasurmentService.h
2 void Init_Latency_Measurement (unsigned char TimerClockBase,unsigned char
      TimerInterruptCompensation);
3 void Start_Latency_Measurement(void);
4 void Stop_Latency_Measurement (void);
```

Listing 4.1: InterruptMeasurmentService.h functions

The init-command initialize the timer with the specified clock frequency. The mnemonics and resulting timer frequencies are specified in Table 4.1. Also the last column indicates the maximum measurement duration without triggering an timer overflow (16 bit counter) at master clock speed of 60 MHz. Both input parameters are stored in the ILMS environment to be used later on. Also a reconfiguration during run time would be possible. Similar to the I/O-configuration (Listing 3.3) the timer interrupt is configured and enabled. Compared to the lowest priority for the demo interrupt, the timer overflow requires the highest interrupt. This permits the overflow ISR to be executed even if nested interrupts are being processed at this time (subsection 4.3.4.

Mnemonic	**Value**	**Description**	**Max. timer w/o ovfl @60MHz**
TIMER_CLOCK1	0x00	MCK/2	2184.53 μs
TIMER_CLOCK2	0x01	MCK/8	8738.13 μs
TIMER_CLOCK3	0x02	MCK/32	34.95 ms
TIMER_CLOCK4	0x03	MCK/128	139.81 ms
TIMER_CLOCK5	0x04	SLCK (32768Hz)	2.00 s
INT_COMP_OFF	0x00	Correction disabled	
INT_COMP_ON	0x01	Correction enabled	

Table 4.1: mnemonic for timer and interrupt compensation

The second function parameter describes if the time compensation for the timer overflow interrupted should be activated or is not required. Subsection 4.3.1 describes the used time approximation. The following enumeration gives examples when to use the compensation and also application where there is no need or a compensation would sophisticate the result:

- **INT_COMP_OFF:**
  Should be used for all sorts of hardware inputs. In general where there are two separate events when the task is started and ended. E.g. input pulse width measurement. Like in the demo interrupt the measurement is started with the rising edge and ended by the falling edge of the digital I/O.

- **INT_COMP_ON:**
  For all measured long tasks which fully use the full processor power from start to finish. They could be interrupted by higher prioritized interrupts. How ever a scheduled task can be measured, but depending on the processor load the task may is not elongated through the timer overflow ISR and the compensation should be turned off.

In general there is no difference for short tasks whether the compensation is active or not. This is only true as long as there is no timer overflow. The clock frequency of the timer (Table 4.1) therefor should be a compromise between accuracy and the maximum measurable time span. To just pick the SLCK as the slowest frequency would enable measurements up to 2 s without an timer overflow. However the accuracy per tick is with 30.52 μs (master clock 60 MHz) rather low.

4.3 Interrupt concept

As mentioned in chapter 4, the ILMS provides the low level routine with a fixed callback to the application / programmer specified high level ISR. Listing 4.2 is an extract of the "ISR.S". This part provides low level function. The interrupt entry macro in line 10 switches the processor mode and pushes the user register in the user stack. Also the link register is updated.
The "bl" command starts the interrupt latency measurement. The mnemonic branch with link adjust the PC to the specified address and updates the link register to continue after the start with the "ldr" command (line 15). The start of the measurement clears the timer overflow counter, starts the timer and saves the time stamp.
The lines 15-17 are the equivalent to the bl command. However they display each step of loading the to branch address, adjusting the return address and updating the PC.

```
1  @-------------------------------------------------------------------------
2  @ Measured_Interrupt_Lowlevel
3  @-------------------------------------------------------------------------
4    .global Measured_Interrupt_Lowlevel
5    .extern Measured_Interrupt_Highlevel
6    .extern Start_Latency_Measurement
7    .extern Stop_Latency_Measurement
8
9  Measured_Interrupt_Lowlevel:
10   IRQ_ENTRY
11
12   @Branch with Link to start the the interrupt latency measurement
13   bl      Start_Latency_Measurement
14
15   ldr     r1, =Measured_Interrupt_Highlevel
16   mov     r14, pc
17   bx      r1
18
19   @Branch with Link to Stop the interrupt latency measurement
20   bl      Stop_Latency_Measurement
21
22   IRQ_EXIT
23 @-------------------------------------------------------------
```

Listing 4.2: Measured interrupt low level service routing

The after the high level routine was executed the "Stop_Latency_Measurement" is processed. At first the timer is stopped and the the overflow interrupt is disabled. A get time function combines the current 16 bit timer status together with the overflow to an 32 bit

timer value. The unit here is ticks and needs to be converted with the knowledge of the timer frequency into μs. This is done with the "Convert_Ticks_To_us"-function (Listing 4.3). The number of ticks was recorded in case "TIMER_CLOCK2" with 1/8 MCK. With one arithmetic logical shift left (multiplication with the factor 2) and the devise by 15 a resulting division of 7.5 is realized. The timer clock frequency oscillates with 7.5 MHz therefore 7.5 ticks are equivalent to one μs. This is also done for the other clocks. The return value of the function is the measured time in microseconds.

```
1 unsigned int Convert_Ticks_To_us (unsigned int Ticks,unsigned char
      TimerClockBase)
2 {
3    switch (TimerClockBase)
4    {
5    case TIMER_CLOCK1:
6       return Ticks/30;         //Tested [v]
7
8    case TIMER_CLOCK2:
9       return (Ticks<<1)/15;    //Tested [v]
10   //. . .
```

Listing 4.3: Extract Convert_Ticks_To_us-function

4.3.1 Time correction

Then depending on the state of the state of the stored variable, the compensated time is calculated by subtracting a correction factor.

Scope	Part	Assembly lines	Processor cycles
Lowlevel:			
	IRQ_Entry	8	13
	Other	3	3
	IRQ_Exit	7	12
Highlevel:			
	Acknowleg int	8	8
	resetLED(RED)	3	3
	TimerOverflowCnt++	3	3
			42 (approx.)

Table 4.2: Additional processors cycles per timer overflow

The correction tries to calculate the time spend to load, process and exit the timer ISR, elongating the measured interrupt. To calculate this the assembly code for the interrupt

entry and exit macros (Listing 4.4 and Listing 4.5) together with the timer overflow assembly code is used. Most of the assembly instructions are single cycle operations. Table 4.2 estimates the required clock cycles. The timer ISR it self contains no branches so no pipeline drop is expected. The start and stop of the timer are almost atomic instructions and only needed once and are therefor neglected. Due to simplicity and with a master clock of 60 MHz the correction assumes 60 clock cycles for a full service of the timer overflow interrupt. This leads that if the compensation is enabled for every overflow 1 μs is subtracted from the measurement result.

```
1  @- IRQ Entry
2  @-----------
3  .macro  IRQ_ENTRY
4  sub     lr, lr, #4
5  stmfd   sp!, {lr}
6  ldr     r14, =AT91C_BASE_AIC
7  str     r14, [r14, #AIC_IVR]
8  mrs     r14, SPSR
9  stmfd   sp!, {r14}
10 msr     CPSR_c, #ARM_MODE_SYS
11 stmfd   sp!, { r0-r3, r12, r14}
12 .endm
```

Listing 4.4: IRQ_Entry

```
1  @- IRQ Exit
2  @-----------
3  .macro  IRQ_EXIT
4  ldmia   sp!, { r0-r3, r12, r14}
5  msr     CPSR_c, #I_BIT |
6          ARM_MODE_IRQ
7  ldr     r14, =AT91C_BASE_AIC
8  str     r14, [r14, #AIC_EOICR]
9  ldmia   sp!, {r14}
10 msr     SPSR_cxsf, r14
11 ldmia   sp!, {pc}^
12 .endm
```

Listing 4.5: IRQ_EXIT

4.3.2 Timestamp and result conversion

The stored date and time has to be converted into ASCII characters. Due to the fact that all values are stored as BCD the conversion can be made rather simple compared to the result value. The index inside the costant arry Dec2ASCII (Listing 4.6) is used to select the correct corresponding ASCII value of each number. This is done for the unit and tens separately by masking and shifting.

```
1 const unsigned char Dec2ASCII[]="0123456789";
```

Listing 4.6: Constant Dec2ASCII

This could be also done by just adding the offset between number and ASCII value (decimal 48). Each value is then stored in the correct place inside the date and time string. The whole conversion is repeated until time and date stamp is completed. The time and date separator between the digits could be chosen individual to enhance readability.
The measured interrupt latency is stored in an unsigned integer (32 bit) and has to be converted into ASCII as well to be transmitted via the serial connection. The function

"Dec2ASCII_Ticks" (Listing 4.7) also takes care of storing the converted values into the "ASCII_UART_Buffer". The constant MEASVALOFF is the offset between the position of the time stamp and the result inside the buffer. The core concept of the function is to convert the value into ASCII and if the value is smaller as the maximum number of characters to fill this blank with the blank symbol (second function parameter). To achieve the conversion the unsigned integer value is divided via the Divider variable. The result of the integer division is then compared if it's greater than zero. And if so the value is converted via the addition of decimal 48 and stored into the buffer. Also the indication number has occurred ("numberoccurred") is set to true. If the value is zero the blank symbol is filled into the buffer. The remaining integer number is calculated via the modulo operator. Also the Divider is down scaled by the factor 10. In the second round of the for loop the integer division is calculated. The if statement does a disjunction with the now may set "numberoccoured". This is necessary that now the zero value instead of the blank symbol should be used if there was a previous other digit than zero.
The whole for loop is run exactly 10 times and therefor has a almost constant conversion time independence from the value that needs to be converted. The blank symbol helps to achieve a constant length to ensure the frame format and readability. A demo recording result is shown in Table 4.4.

```
1  void Dec2ASCII_Ticks(unsigned int value,unsigned char blanksym)
2  { unsigned char numberoccoured=0;
3    unsigned int num;
4    unsigned int ValToWork=value;
5    unsigned int i;
6    unsigned int Divider=1000000000;
7
8    for(i=0;i<10;i++)
9    {
10     num=ValToWork/Divider;
11       if(num|numberoccoured)
12       {     ASCII_UART_Buffer[i+MEASVALOFF]=(unsigned char)num+48;
13             numberoccoured=1;                                  }
14       else
15       {     ASCII_UART_Buffer[i+MEASVALOFF]=blanksym;      }
16     ValToWork\%=Divider;
17     Divider/=10;
18   }
19 }
```

Listing 4.7: Result to ASCII conversion

4.3.3 Measurement transmission

The complete string is after the update of time stamp and measurement result ready to be transmitted. This is done via the DMA. The transmission routine there selects the user RS232 and sets the message address as well as the number of bytes to be transmitted. The routine needs this information to program the DMA register. Depending on the current transmission state, either the transmit pointer register or if a transmission is on going the transmit next pointer register is set[1].
The full string is then transmitted. Equation 4.1 calculates the transmission time for one full message:

$$\begin{aligned} String\_length \times Bits\_per\_character \div Bits\_per\_second &= Transmission\_time \\ 34 \times 10 \div 115200 &= 2.86\,ms \end{aligned} \tag{4.1}$$

Each of the 34 characters (Table 4.3) is transmitted with one start and one stop bit. This results in the number "bits per character". With the baud rate of 115200 bits per second one message needs less than 3 ms to be sent. With the whole process of converting the time and date to ASCII as well as the value transmission, the ILMS is limited by around 4 ms. Therefor sporadic interrupts could be handled once via the "'next transmission DMA register. But the whole ILMS is not able to serve higher interrupt frequencies than approximately 25 Hz without improvements (see section 5.1 for suggestions).

4.3.3.1 RS232 frame format

Table 4.3 specifies the transmitted string with all sections to be received and processed with the database or stand alone software. The Datestamp contains only numbers and has a part length[2] of 11 characters. The Timestamp contains 9 characters[2] followed by the result with a length[2] of 11. The string is completed by the unit μs and the "\n" with 3 characters.

Frame	**Datestamp**	**Timestamp**	**Measurement**	**Unit**	**Delimiter**
Symbolic	DD/MM/YYYY	HH:MM:SS	Value	Unit	\n
Example	18/05/2014	15:57:30	0000123456	μs	\n

Table 4.3: RS232 Frame Components

[1] If no register is available, the transmission is dropped. The function returns with an error code (not handled)

[2] Including space character at the end (see Table 4.4 for details)

4.3.4 Measurement scenarios

There are various scenarios of interrupt occurrence possible. Figure 4.1 gives an overview about 3 different measured interrupts. In general the time axis of the figures features no time unit and is only used to reference for the event description. The time between two timestamps doesn't have to be of the same duration. The left column descries different tasks. The application has in this example no priority[3]. A rising arrow from one task indicates an interruption of the this task. An arrow in downwards direction indicates the ISR of the higher priority task was completed and the interrupted task continue to run. In terms of the ILMS there is no additional interrupt after starting the measurement and the measured interrupt. This is a normal (callback) function call and is indicated with a line with no arrowhead. The whole chart is organized from top down starting with the highest priority.

In figure 4.1(a) a short interrupt is measured. The application is in running state until at time point 2 an interrupt for the measured interrupt occurs. The ILMS starts its service by saving the RTC values and starting the timer. With a normal function call the ISR, which has to be measured, is loaded. The measured ISR then is active in the example till moment 5. The ILMS then stops the Timer and triggers the the time calculation and correction as well as the conversion to ASCII. The service sets the DMA for the auto transmission of the result string and returns to continue the main application. The message then is transmitted in parallel while the application already is executed. Because of the time constraint due to a transmission time of around 4 ms the ILMS is not able to measure interrupts which occur and finish in less than 4 ms in a short time range.

For a visual explanation it is assumed that timer stop and the conversion of the measurement result takes around 1 ms (time between point 5 and 6). The application becomes running again and the transmission of the 34 byte string takes according to Equation 4.1 less then 3 ms. So the complete string would be transmitted until point 9 on the scale. This would allow a new result transmission. With the DMA next transmission buffer this would be possible once[4] in a short time period, until both buffers are available again. Therefor especially high frequent, low computing time interrupts are not measurable.

In the figure 4.1(a) the interrupt occurs at point in time 10 so every 8 instant of time and is handled and measured in the same way as the interrupt at time 2.

In figure 4.1(b) a long ISR has to be measured. Like in the previous example the application is running and is interrupted at time point 2 to execute a measured interrupt. After the start of timer and the store of the start time the measured interrupt is executed. It is

[3]Priorities are referenced by the AIC priority

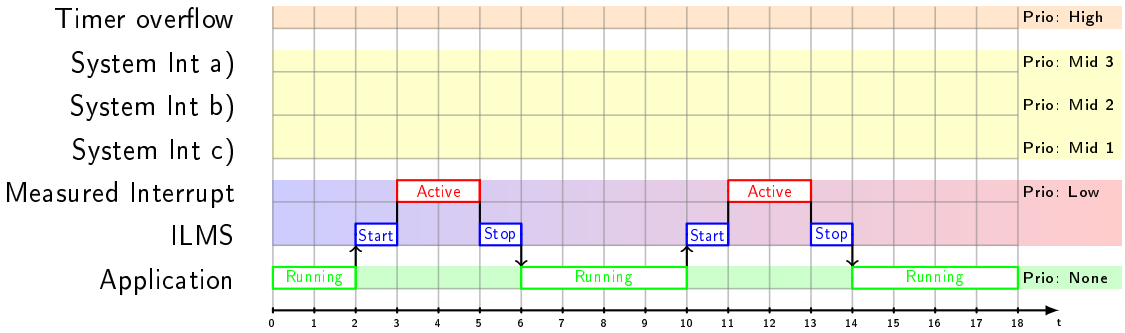
[4]ILMS used only one message buffer

active until point in time 5.5. At this moment an timer overflow occurs. The timer is configured with a high priority permitting the timer ISR to interrupt the measured interrupt. The timer overflow then is active till point 6 of the scale. While being active the interrupt is acknowledged and an counter, counting the number of overflows, is increased. As well as for demo purpose the "RED" LED is turned off. The falling arrow indicates that the prior ISR was served. The metered interrupt then becomes active and continues until time point 8. This example indicates an clock frequency of the timer which leads to an overflow every 2.5 time units with a timer overflow computation time of 0.5 units. This influences especially for high clock frequency an elongation of the measured interrupt. However this is necessary due to the fact that the timer registers are 16 bit. The interrupt continues and is one more time interrupted by the timer overflow until it stops the measurement at point 10. Like in the previous example the time is calculated and converted into μs. Now it depends on the timer configuration if the time in the timer overflow ISR is subtracted from the result (see subsection 4.3.1). The result then is send via the RS232 while the application continues.

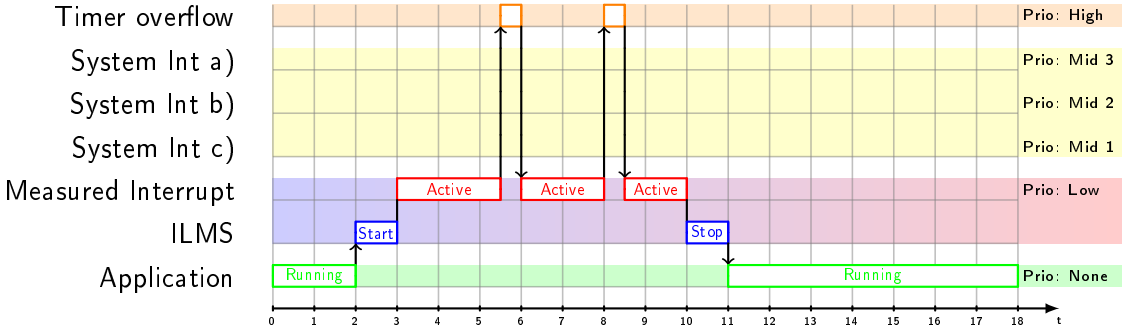
Figure 4.1(c) is an example of an long measured interrupt with timer overflows and other higher prioritized interrupts intercepting the ISR of the measured interrupt. Also in this example the application is running and interrupted at point 2. The ILMS starts the measurement and continues with the metered interrupt. This one is active (ACT) until point in time 4. The Example contains multiple system interrupts (System Int a,b,c). These interrupts could be also other user or peripheral interrupts with higher priorities compared to the measured interrupt. After the intercept System Int c) is active. This one is also interrupted by the timer overflow interrupt having the highest priority in the whole scenario. This one is like previous configured to occur every 2.5 time units. After the timer ISR has been executed the System Int c) continues for 0.5 time slice and is interrupted by the higher prioritized System Int b). At time point 7.5 then the highest System Int a) occurs and is served. After the timer interrupt at time 8 the whole chain then follows in descending order the measured interrupt then is executed again. Between time 10 and 15 two more timer interrupts and one System Int b) is processed. The time is then stopped and converted and send to the stand alone or database server software. In this example the measured interrupt would only need 3.5 time units to be completed. But this execution time is not reached due to the higher priorities competing for processor time. Instead in this case the interrupt has an execution time of 12 units without compensation. In case of normal operation there would be then no timer interrupt and the execution time would be 10 units.

In longtime measurements the execution time of different scenarios could be measured and recorded to figure out the worst case execution time of one task. The results then can be

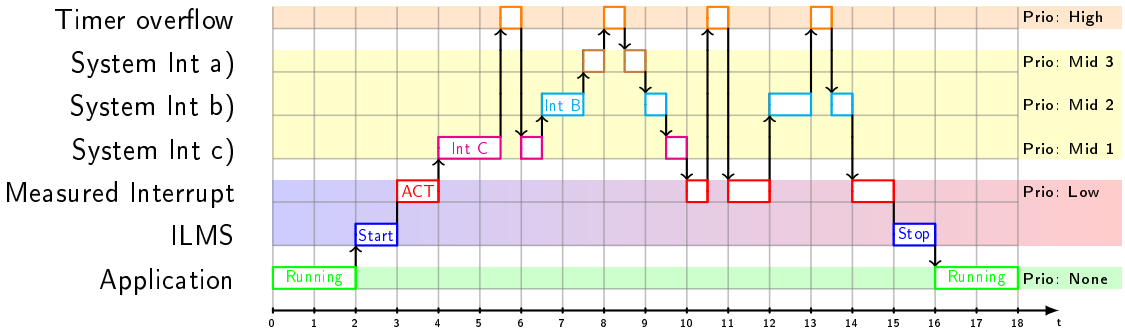
checked against possible critical deadlines for this specific task



(a) Short measured interrupt, no timer overflow or competing other interrupts



(b) Long measured interrupt, with timer overflow but no competing other interrupts



(c) Long measured interrupt, with timer overflow and other higher prioritized interrupts

Figure 4.1: Example of possible interrupts occurrence and the corresponding handling

4.3.5 Demo interrupt function

The demo interrupt uses the output signal of the sharp distance sensor (subsection 3.3.1) to trigger an interrupt when an obstacle is in range of the sensor. The high level routine is executed after the start of the ILMS. Listing 4.8 gives an overview of the main part of the high level ISR. For demo purpose and as an user feed the GREEN LED is turned on. With the while loop (line 5) the controller stays in that interrupt as long as the obstacle is not removed in front of the sensor. This serves as measurement of the high period of the sensor. After the while loop is break the LED is turned off[5] and the high level is completed. During this the ILMS clocked the execution (in this case high duration time). After completion the value is then converted an send to the Database or stand alone software.
In the case of a duration measurement the compensation for the timer overflow interrupt should be turned of. How ever this waiting for low could be also seen as demonstration what could happen if other task interrupt the removing of the obstacle. In terms of the processor this means what happens if there are other higher priorities interrupting and elongating the measured interrupt like in figure 4.1(c).

```
1 void  Measured_Interrupt_Highlevel (void)
2 {     // . . .
3
4       setLed(GREEN);
5       while(AT91F_PIO_GetInput (AT91C_BASE_PIOB) & MY_INT_PIN); // Wait if high
            !
6       resetLed(GREEN);
7
8       // . . .          }
```

Listing 4.8: Extract of the demo interrupt high level ISR

Table 4.4 is an extract of an record of interrupt measurements created with the demo interrupt and the sharp distance sensor.

Measurement Result
18/05/2014␣16:59:39␣0000080460␣us\n
18/05/2014␣16:59:41␣0000080456␣us\n
18/05/2014␣16:59:43␣0011866587␣us\n
18/05/2014␣17:00:07␣0000040232␣us\n
18/05/2014␣18:28:58␣0000120731␣us\n
18/05/2014␣18:28:59␣0001167025␣us\n

Table 4.4: Demo interrupt results

[5] Toggling of the YELLOW LED is done every time after the ILMS is stopped

Chapter 5

Conclusion

The ILMS enables a precise time measurement of the interrupt execution time. By using the internal timer there is no need for external hardware or a co-processor. The time spent in the timer overflow can be estimated and corrected. By using the highest priority for the timer overflow, also nested interrupts can be served. It is then also possible to measure the execution time of a FIQ-ISR if no timer overflow occurs[1]. The timer frequency is configurable and should be set as a trade of between accuracy or time without an timer overflow. In general a precision of 1 μs and a maximum duration of around 5 s can be achieved. The RTC provides the start time and date stamp when an interrupt occurred. Together with the value the whole measurement is transmitted as a 34 byte string. The frame format then can be processed in a database software. An ASCII character format ensures also readability via a serial monitor but also limits the ILMS in terms of the ability to measure high frequent interrupts. A demo interrupt was used to check the measured results for plausibility.

5.1 Further improvements

- **Flexible configuration**
  The ILMS could be configured via RS232 by the user or the database software.
  - Messages with or without time and/or date stamp.
  - Timer frequency selection.

[1]The time between an interrupt becoming pending until the start of execution (including the context switch) is not measured with the ILMS and would require a major change in the interrupt concept

  - Accuracy selection (selection of the base time unit or a flexible configuration e.g. 3 fractional digits).
  - Enable or disable the time correction.
  - Configuring whether the RTC is saved at interrupt occurrence or completion.
  - RAW mode for database high speed transmission. (4 byte UNXI timestamp, 4 byte result and fixed unit ticks).
  - Flexible date and time transmission (only included if there was a change in the values), requires reliable message transmission.

- **Message buffering**
  Message buffering and reliable transmission with message acknowledge / handshaking mechanism. Scalable message buffer.

- **Resynchronizing of the RTC**
  The RTC time could be synchronized via DCF77 receiver. Through an DCF77 decoder time and date is coded via a pulse width modulation. After a frame is successfully received the time could be updated every minute. This would require one more system timer.

- **Result plausibility check**
  Use a second timer with a slow frequency to compute the measured time to. This would allow to the detect if e.g. when metering the FIQ timer overflow were missed. Also if the results would differ if 32 aren't enough to store the value.

- **Timer selection and priority**
  Flexible configuration of the used timer and warning when application uses the same timer. Also e.g. macro based checking if the timer functionality can be achieved (has the timer a higher priority than the measured interrupt or is the timer in use).

List of Figures

2.1 System Overview 10

3.1 Projekt HAL layout 11
3.2 LED Schematic AT91RM9200_EK 13
3.3 Sharp GP2Y0D02YK0F Distance Measuring Sensor Unit 17
3.4 Sharp timing chart 18

4.1 Example of possible interrupts occurrence and the corresponding handling . 30

List of Tables

2.1 AT91RM9200 Registers 2
2.2 Processor Mode 3
2.3 Example of r/w access of the RTC time register 8
2.4 Extract BDI2000 command list 9

3.1 LED-functions 13
3.2 User RS232 settings (8-N-1) 14

4.1 mnemonic for timer and interrupt compensation 22
4.2 Additional processors cycles per timer overflow 24
4.3 RS232 Frame Components 27
4.4 Demo interrupt results 31

List of Listings

2.1 Extract of the rm9200dk.cfg-file 6

3.1 Color definition for the LED-functions 12
3.2 Configure the LED-Pins as output 12
3.3 Demo interrupt at PB15 configured as rising 16
3.4 Minimal RTC initialization 19

4.1 InterruptMeasurmentService.h functions 21
4.2 Measured interrupt low level service routing 23
4.3 Extract Convert_Ticks_To_us-function 24
4.4 IRQ_Entry 25
4.5 IRQ_EXIT 25
4.6 Constant Dec2ASCII 25
4.7 Result to ASCII conversion 26
4.8 Extract of the demo interrupt high level ISR 31

Bibliography