



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich



Institut für  
Technische Informatik und  
Kommunikationsnetze

# Towards Low-Power, Timing-Predictable Medical Monitoring

Semester Thesis

**Akos Pasztor**

pasztora@student.ethz.ch

Computer Engineering and Networks Laboratory  
Department of Information Technology and Electrical Engineering  
ETH Zürich

**Advisors:**

Georgia Giannopoulou  
Felix Sutton  
Pengcheng Huang

**Professor:**

Prof. Dr. Lothar Thiele

January 29, 2016

# Abstract

This semester thesis describes the design and implementation of a prototype medical monitoring device. The device can measure human vital signs, such as heart rate and blood oxygen saturation. The measured data is forwarded to a connected device (e.g. PC, smartphone, etc.), moreover received control commands are also interpreted and processed by the device. The two key design goals of the medical device design are timing predictability and low power dissipation. Timing predictability is motivated by the need to provide response and alerts within guaranteed time bounds. Low power dissipation facilitate the portability and mobility of the device. This thesis describes the design and development of the system and its operation in details, and presents the results of the performed timing and power analysis. The final conclusion summarizes the project and my experience, moreover it contains my personal ideas and opinions, as well as an envisioned proposal as a possible future work.

# Acknowledgements

I would like to express my sincere gratitude to my advisors: Georgia Giannopoulou, Felix Sutton and Pengcheng Huang for the continuous support of my semester project. Beside my advisors, I would like to thank the Computer Engineering and Networks Laboratory for providing me a personal workplace with access to laboratories and equipments. My sincere thank goes to Prof. Dr. Lothar Thiele, without whom this project would not have been possible. Last but not least, I would like to thank my family and friends for supporting me spiritually throughout this project and my life in general.

# Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgements</b>	<b>ii</b>
<b>Contents</b>	<b>iii</b>
<b>List of Figures</b>	<b>v</b>
<b>List of Tables</b>	<b>vii</b>
<b>Abbreviations</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Contribution . . . . .	2
1.3 Overview . . . . .	2
<b>2 Background</b>	<b>3</b>
2.1 Embedded Software . . . . .	3
2.2 Equipment and Tools . . . . .	4
2.3 BOLT . . . . .	6
2.4 Pulse Oximetry Sensor . . . . .	7
2.4.1 Measurement Range and Precison . . . . .	7
2.4.2 Sensor Data . . . . .	8
2.4.3 Sensor Control . . . . .	13
2.5 User Interface . . . . .	14
2.6 Microcontroller . . . . .	15
2.6.1 Interrupt System . . . . .	16
2.6.2 Connectivity and Pinout . . . . .	17
2.7 FreeRTOS . . . . .	17
2.7.1 Configuration . . . . .	17
2.7.2 Interrupt Priorities . . . . .	18
2.7.3 Memory Management . . . . .	19
<b>3 System Design</b>	<b>21</b>
3.1 System Architecture . . . . .	21
3.1.1 Communication . . . . .	22
3.2 Software architecture . . . . .	22
3.2.1 Initialization . . . . .	23

3.2.2	Interrupts . . . . .	24
3.2.3	Tasks . . . . .	25
3.3	Low Power System Design . . . . .	29
3.3.1	Microcontroller Low Power Modes . . . . .	29
3.3.2	Sensor Duty Cycling . . . . .	29
<b>4</b>	<b>Evaluation</b>	<b>32</b>
4.1	Timing Analysis . . . . .	32
4.1.1	Medical Sensor . . . . .	32
4.1.2	BOLT . . . . .	34
4.1.3	RTOS Tasks . . . . .	35
4.1.4	RTOS Context Switching . . . . .	38
4.1.5	Data Flow . . . . .	38
4.1.6	Response Times . . . . .	39
4.2	Power Analysis . . . . .	46
4.2.1	Analysis of Individual Interrupts and Tasks . . . . .	47
4.2.2	Analysis of Operating Modes . . . . .	52
4.2.3	Analysis of Duty Cycling . . . . .	58
4.2.4	Idle Task Analysis . . . . .	64
4.2.5	Sensor Power Analysis . . . . .	65
4.2.6	Conclusion of Power Analysis . . . . .	66
<b>5</b>	<b>Conclusion</b>	<b>68</b>
5.1	Future Work . . . . .	68
	<b>Bibliography</b>	<b>70</b>
	<b>A Connectivity and Pinout</b>	<b>71</b>
	<b>B Source Code Organization</b>	<b>72</b>
	<b>C Device Outputs</b>	<b>74</b>
	<b>D Sensor Timings</b>	<b>75</b>
	<b>E BOLT Timing Analysis</b>	<b>76</b>
E.1	Writing Operation . . . . .	76
E.2	Reading Operation . . . . .	77
	<b>F RTOS Task Timings</b>	<b>79</b>

# List of Figures

2.1	Workstation and development tools . . . . .	5
2.2	Overview of the BOLT processor interconnect [1] . . . . .	6
2.3	BOLT read and write operations, as illustrated in [1] . . . . .	7
2.4	MCU UART RX signal captured with an oscilloscope . . . . .	8
2.5	DataFormat #2 Status byte . . . . .	9
2.6	Heart rate value in DataFormat #2 . . . . .	10
2.7	Blood oxygen saturation (SpO2) in DataFormat #2 . . . . .	10
2.8	Byte 1 of packet in DataFormat #1 . . . . .	12
2.9	Byte 2 of packet in DataFormat #1 . . . . .	12
2.10	Byte 3 of packet in DataFormat #1 . . . . .	12
2.11	Interrupt priority values stored in TI MSP432 MCU . . . . .	16
2.12	FreeRTOS interrupt priorities port to Cortex-M core MCU . . . . .	19
2.13	MSP432 memory management . . . . .	20
3.1	System Architecture . . . . .	21
3.2	Software architecture . . . . .	23
3.3	Sensor RX structure that is transmitted in xQueueRx queue . . . . .	25
3.4	Sensor data structure . . . . .	26
3.5	Message structure for xQueueUI queue . . . . .	28
3.6	Bitwise description of leds variable . . . . .	28
4.1	RTOS trace using DataFormat #2 . . . . .	36
4.2	RTOS trace using DataFormat #1 . . . . .	37
4.3	Data Flow . . . . .	39
4.4	RTOS interrupt and task priorities . . . . .	41
4.5	BOLT task response . . . . .	43
4.6	Packet response time . . . . .	44
4.7	Power analysis setup . . . . .	46
4.8	Sensor UART RX ISR and Data task execution analysis . . . . .	48
4.9	Power profile: last byte of packet reception and send data to BOLT . . . . .	48
4.10	Trace: last byte of packet reception and send data to BOLT . . . . .	49
4.11	Power profile: last frame of packet using DataFormat #2 . . . . .	49
4.12	Trace: last frame of packet using DataFormat #2 . . . . .	50
4.13	Power profile of last three frames of a packet using DataFormat #2 . . . . .	51
4.14	Power profile of a packet using DataFormat #2 . . . . .	51
4.15	Power profile of a packet using DataFormat#1 . . . . .	52
4.16	Power profile: basic operation using DataFormat#2 . . . . .	53
4.17	Trace: basic operation using DataFormat#2 . . . . .	53

4.18	Power profile: basic operation with flashing LED after each writing operation to BOLT . . . . .	54
4.19	Power dissipation while LED is on . . . . .	55
4.20	Power profile: basic operation using DataFormat#1 . . . . .	56
4.21	Trace: basic operation using DataFormat#1 . . . . .	56
4.22	Dynamically switch between sensor output data formats . . . . .	57
4.23	Overview of a reading operation . . . . .	57
4.24	Power profile of a reading operation . . . . .	58
4.25	Power profile: duty cycling without periodic reading using DF#2 . . . . .	59
4.26	Trace: duty cycling without periodic reading using DF#2 . . . . .	60
4.27	Power profile: duty cycling with periodic reading using DF#2 . . . . .	60
4.28	Trace: duty cycling with periodic reading using DF#2 . . . . .	61
4.29	Power profile: duty cycling without periodic reading using DF#1 . . . . .	61
4.30	Trace: duty cycling without periodic reading using DF#1 . . . . .	62
4.31	Power profile: duty cycling with periodic reading using DF#1 . . . . .	63
4.32	Trace: duty cycling with periodic reading using DF#1 . . . . .	63
4.33	Difference between sensor on and off during duty cycling . . . . .	64
4.34	Power analysis of idle task . . . . .	65
4.35	Power analysis of the pulse oximeter . . . . .	65
5.1	Smartphone application with measurement data . . . . .	69
B.1	Source code organization . . . . .	73
C.1	Device output when rested . . . . .	74
C.2	Device output after performing 50 push-ups . . . . .	74
D.1	Sensor DataFormat #2 frame . . . . .	75
D.2	Sensor DataFormat #2 packet . . . . .	75
D.3	Sensor DataFormat #1 packet . . . . .	75
E.1	Write to BOLT: Request to ACK response time . . . . .	76
E.2	Write to BOLT: ACK to first byte transfer . . . . .	76
E.3	Write to BOLT: Last byte to REQ line release . . . . .	77
E.4	Read from BOLT: Request to ACK response time . . . . .	77
E.5	Read from BOLT: ACK to first byte transfer . . . . .	77
E.6	Read from BOLT: Last byte to IND line release . . . . .	78
E.7	Read from BOLT: IND to ACK line release . . . . .	78
F.1	Data task execution . . . . .	79
F.2	BOLT task execution . . . . .	79
F.3	BOLT reading operation using DataFormat #2 . . . . .	79

# List of Tables

2.1	Serial Output Formatting Options . . . . .	9
2.2	STATUS byte description in DataFormat #2 . . . . .	10
2.3	DataFormat #2 packet description . . . . .	11
3.1	TI MSP432P401R Power Modes [2] . . . . .	30
4.1	Sensor timings using DataFormat #2 . . . . .	33
4.2	Sensor timings using DataFormat #1 . . . . .	33
4.3	BOLT writing operation timings (average values) . . . . .	34
4.4	BOLT reading operation timings (average values) . . . . .	35
4.5	RTOS timings using DataFormat #2 . . . . .	36
4.6	RTOS timings using DataFormat #1 . . . . .	37
4.7	Context switching times . . . . .	38
4.8	Maximum execution times and task periods . . . . .	41
4.9	Worst-case response times . . . . .	45
4.10	Scenarios for power analysis . . . . .	47
4.11	Summary of current drain of different operating modes . . . . .	66
A.1	Pinout of the microcontroller . . . . .	71



# Abbreviations

<b>API</b>	Application Programming Interface
<b>BPM</b>	Beats Per Minute
<b>DF#1</b>	DataFormat#1
<b>DF#2</b>	DataFormat#2
<b>HR</b>	Heart Rate
<b>ISR</b>	Interrupt Service Routine
<b>IT</b>	Interrupt
<b>LPM</b>	Low Power Mode
<b>LSB</b>	Least Significant Bit
<b>MCU</b>	Microcontroller Unit
<b>MISO</b>	Master In Slave Out
<b>MOSI</b>	Master Out Slave In
<b>MSB</b>	Most Significant Bit
<b>MUX</b>	Multiplexer
<b>NVIC</b>	Nested Vectored Interrupt Controller
<b>PCB</b>	Printed Circuit Board
<b>RTC</b>	Real Time Clock
<b>RTOS</b>	Real Time Operating System
<b>SCK</b>	SPI Clock
<b>SPI</b>	Serial Peripheral Interface
<b>SS</b>	SPI Slave Select
<b>TI</b>	Texas Instruments
<b>UART</b>	Universal Asynchronous Receiver/Transmitter
<b>UI</b>	User Interface
<b>VCP</b>	Virtual Com Port
<b>WDT</b>	Watchdog Timer

# Introduction

---

Embedded systems provide unprecedented opportunities in every field of life. By using state-of-the-art technologies in healthcare, the quality of medical monitoring has been growing steadily. The topic of this thesis is the development of a prototype medical monitoring device which can monitor human vital signs, such as heart rate, blood oxygen saturation, respiratory rate, in combination with ambient characteristics, for instance temperature and humidity. The envisioned device can be used for not only long-term medical monitoring, but also for issuing warnings and alerts when abnormal changes in vital signs are observed. Nowadays there are several devices that exist on market which are able to fulfill the requirements, however two key features - which are timing predictability and low power dissipation - differentiate the device developed in this thesis from commercially available products.

## 1.1 Motivation

Two major goals were focused on during development. When the device is used for monitoring patients with critical or life-threatening health conditions, for instance in emergency rooms, *timing predictability* is crucial. These devices need to be certified, which means that strict bounds on the response time between an abrupt change in vital signs and issuing an alert must be guaranteed. Consequently both hardware and software have to be designed and developed with the respect of timing predictability principles.

The second key goal is to achieve *low power consumption* in order to make the device not only portable, but also to provide long-term monitoring on-the-go. This is extremely important when constant monitoring can be a life-saver although power supply cannot be guaranteed, for instance when a patient is being transported. Therefore the device has to be designed with the evaluation

of low power design principles in order to maximize battery lifetime.

## 1.2 Contribution

The contributions of this thesis can be summarized as follows:

1. A fully functional prototype which fulfills the requirements was developed while focusing on the two major goals.
2. On the hardware side, the device was built from off-the-shelf components.
3. On the software side, FreeRTOS was used with preemptive scheduling policy for small overhead and timing predictability.
4. Sensor duty cycling and low power modes were used to focus on leveraging low power system design to minimize power dissipation.
5. Considering a combination of fixed-priority and dataflow-driven scheduling in FreeRTOS, timing analysis was performed to estimate bounds for the response time of all system tasks.
6. Extensive timing and power measurements were conducted to validate the analytical results and overall system operation.

## 1.3 Overview

Chapter 2 present an overview of system and the *background* of design and development. Then in chapter 3, the *system design* is described in detail in conjunction with the operation of system. In the *evaluation* chapter, the results of meticulous timing and power analysis are presented. The *conclusion* chapter contains a summary of the work along with my personal ideas for future work.

# Background

---

To develop the prototype of the medical monitoring device, off-the-shelf components were used. A state-of-the-art ARM Cortex-M core microcontroller was chosen as the application processor which incorporates high performance along with low power dissipation. The microcontroller is located on a development board called TI MSP-EXP432 LaunchPad which includes an on-board emulator that provides programming and debugging without the need of additional tools. I used a pulse oximetry sensor to measure human vital signs, including heart rate and blood oxygen saturation. The sensor features different output data formats, which provides the possibility to tune the application for maximum efficiency. I evaluated BOLT, a stateful processor interconnect for communication. BOLT provides predictability by ensuring that communication takes a known, bounded time regardless of the attacked processors, while leveraging efficiency with ultra-low power consumption.

## 2.1 Embedded Software

Software development is inseparable from hardware, therefore embedded software should be written with full knowledge of hardware components and their performance in order to leverage efficiency. With keeping the two key goals in mind, I decided to utilize a real-time operating system which not only provides portability, expandability and upgradability for future functions, but also contributes towards developing applications with timing predictability and low power system design.

## 2.2 Equipment and Tools

The prototype consists of the following off-the-shelf hardware components:

- Nonin OEM III pulse oximetry sensor with internal spring finger clip
- Breakout board for pulse oximeter
- TI MSP-EXP432 LaunchPad
- BOLT processor interconnect
- BOLT to USB adapter
- 2-channel 2:1 multiplexer

During development, the following equipment was used for hardware development:

- Saleae Logic 8 logic analyzer
- Keysight Technologies N6705A DC Power analyzer (formerly manufactured by Agilent Technologies)
- Tektronix TBS1102 Digital oscilloscope
- Digital multimeter

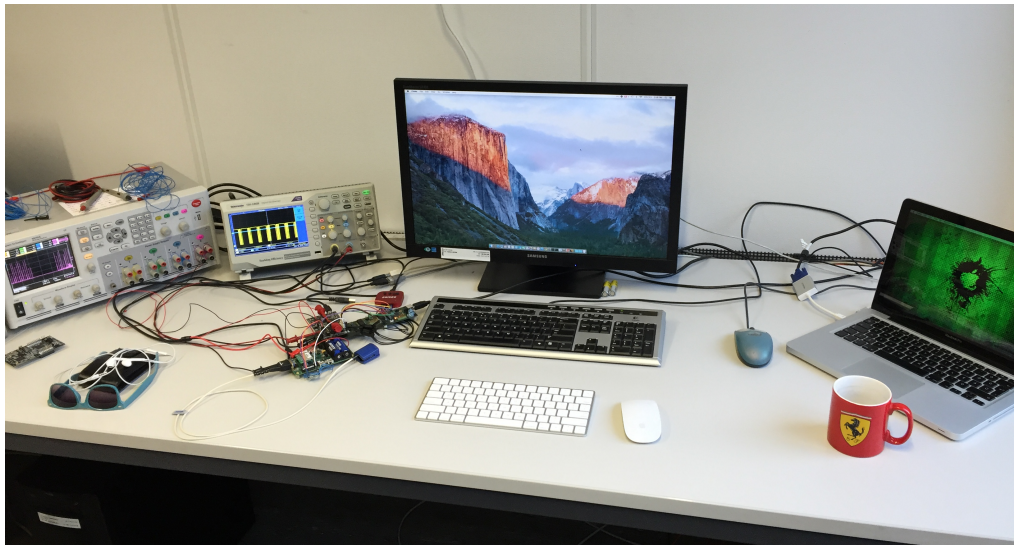


Figure 2.1: Workstation and development tools

The following list summarizes the tools and softwares that I used for software development:

- Texas Instruments Code Composer Studio (TI CCS)
- FreeRTOS
- Putty and Bray's terminal emulator
- MatLab
- Subversion (SVN)

I used a version control system, namely Subversion<sup>1</sup> for collaboration and tracking my own progress. The remote repository is provided by the Department of Information Technology and Electrical Engineering at ETH Zürich.

---

<sup>1</sup>Subversion  
<http://subversion.apache.org>

### 2.3 BOLT

BOLT<sup>1</sup> is an ultra-low power processor interconnect that interconnects application and communication processors while decoupling them with respect to time, power and clock domains. It features asynchronous message passing with predictable timing characteristics. BOLT features two FIFO queues implemented in non-volatile FRAM for both directions [1].

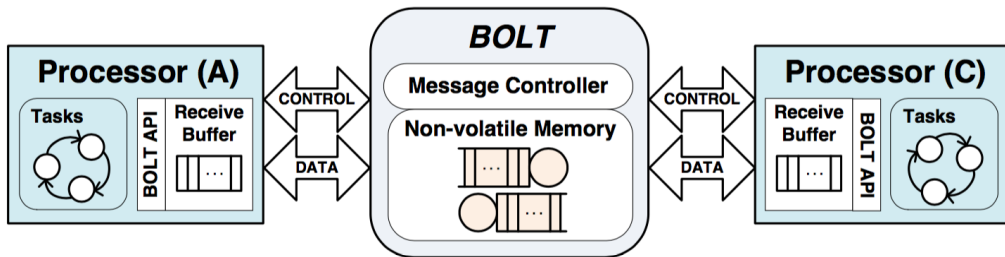


Figure 2.2: Overview of the BOLT processor interconnect [1]

The communication interface is SPI<sup>2</sup> which is a synchronous serial communication interface that became a de-facto standard in embedded systems. SPI devices communicate in full-duplex mode using a master-slave architecture.

BOLT uses an enhanced SPI communication interface. In addition to the four traditional SPI wires (CLK, MISO, MOSI, SS<sup>3</sup>) there are three more control lines which are required to evaluate BOLT in a system. The *MODE* pin is used for indicating whether a Read (*MODE* is pulled low) or Write (*MODE* is pulled high) operation is initiated. The *IND* pin is driven high by BOLT when one or more messages are available for the appropriate processor. The *ACK* pin is also driven by BOLT. After a processor pulls the *REQ* line to initiate communication, BOLT pulls the *ACK* line high when ready to communicate and holds it high during transmission. In Read mode, *ACK* is pulled low at the end of the read message; in Write mode *ACK* is pulled low after a processor finished its writing operation. Read and Write operations are demonstrated in Figure 2.3.

<sup>1</sup> BOLT - Stateful Processor Interconnect  
<http://www.bolt.ethz.ch>

<sup>2</sup> SPI - Serial Peripheral Interface Bus

[https://en.wikipedia.org/wiki/Serial\\_Peripheral\\_Interface\\_Bus](https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus)

<sup>3</sup> SS - Slave Select: in BOLT the SS wire is called *REQ* (Request)

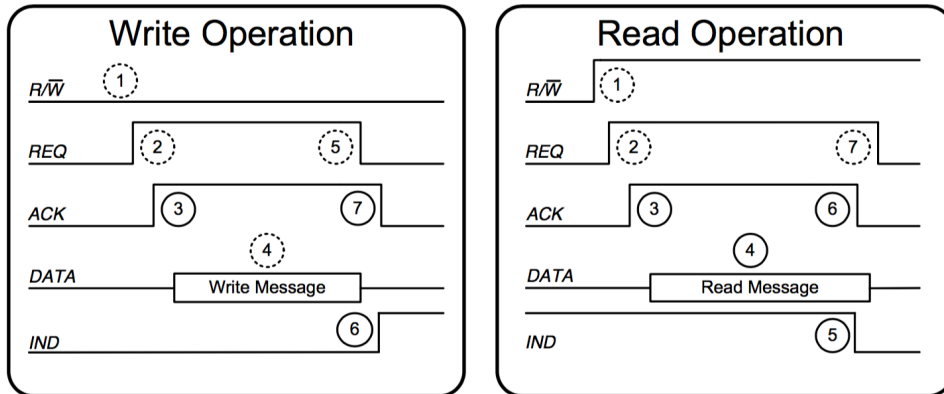


Figure 2.3: BOLT read and write operations, as illustrated in [1]

Both processors that are interconnected operate in SPI master mode and initiate communication with BOLT which is always in SPI slave mode. BOLT limits the transfer speed to 4 MHz SPI clock frequency for communication.

By using BOLT, the application processor can be effectively separated from the communication processor while ensuring data consistency, timing predictability yet consuming several orders of magnitude lower power than the application and communication processors.

## 2.4 Pulse Oximetry Sensor

In this prototype, a pulse oximetry sensor is used to measure heart rate and blood oxygen saturation (SpO<sub>2</sub>). The chosen sensor is a Nonin OEM III Module<sup>1</sup> with an adult articulated internal spring finger clip. The power dissipation of the sensor is not more than 29 mW (at 3.3V input voltage) in normal operation mode [3].

### 2.4.1 Measurement Range and Precision

The sensor can measure pulse rate between 18 and 321 beats per minute (BPM). With the finger clip, the precision of the heart rate is  $\pm 3$  digits while there is

<sup>1</sup>Nonin OEM III Module - Internal OEM Pulse Oximeter  
<http://www.nonin.com/OEM-III-Module>



no motion. The precision during motion is  $\pm 5$  digits.

The displayed oxygen saturation is between 0 and 100%. The precision of SpO<sub>2</sub> is  $\pm 2$  digits regardless of motion.

### 2.4.2 Sensor Data

The sensor uses the UART interface to output measured data. Once the sensor is turned on, it constantly outputs the measured data based on the chosen output data format. The UART interface of the sensor operates from 9600 baudrate with 8bit data, no parity and 1 stop bit (“N81” mode). This means that the maximum theoretical output of the sensor is 1.2 kbyte/second.

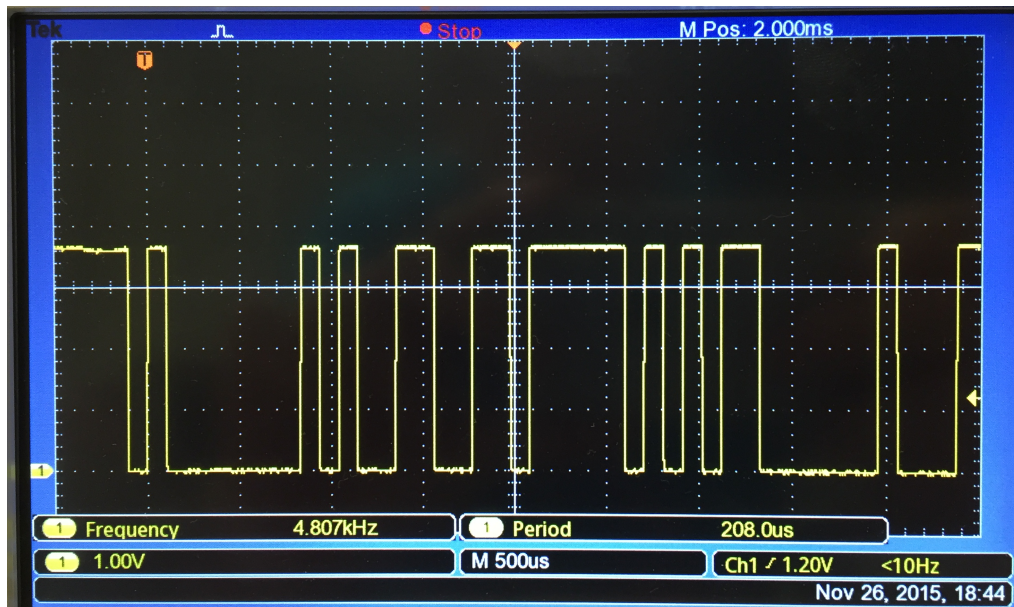


Figure 2.4: MCU UART RX signal captured with an oscilloscope

This sensor supports up to three output data formats. The format can be set directly on the module with the amount of resistance present between Pin9 and GND. If Pin9 is left unconnected, the default data format is set which is called DataFormat #2.

Serial Format	Pin9 Status
#1	$0\Omega$ to $626\Omega$
#2	$297k\Omega$ to $\infty \Omega$
#7	$4.3k\Omega$ , $\pm 5\%$

Table 2.1: Serial Output Formatting Options

**DataFormat #2**

This is the default output data format. In this mode the sensor continuously transmits packets which consist of 125 bytes. A packet comprises 25 frames, therefore each frame contains 5 bytes. Three packets (3 x 125 bytes) are transmitted each second. The detailed packet description is shown on Table 2.3.

Each frame starts with a 0x01 character, then it is followed by the STATUS and PLETH bytes. The fourth byte is different for each frame, for instance the 4th byte of the first two frames contain the upper and lower byte of the heart rate respectively. The 5th byte of a frame is the checksum of the previous four bytes.

PLETH is a 8bit Plethysmographic Pulse Amplitude value. The STATUS byte contains the current state of the sensor. Each bit is active-high.

1	SNSD	ARTF	OOT	SNSA	RPRF	GPRF	SYNC
					YPRF		
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Figure 2.5: DataFormat #2 Status byte

Since the range of hearth rate is from 18 to 321 bpm, two bytes are necessary to transmit the value. In each packet, the 4th bytes of the first and second frames contain the heart rate value. The upper 2 bits are stored in the first frame and the lower 7 bits are stored in the second frame. The combined value is a 4-beat average heart rate value in standard mode.<sup>1</sup>

<sup>1</sup>In Standard mode when HR and SpO2 cannot be computed, HR and SpO2 values are set to missing data values and “OOT” is indicated.

SNSD	Sensor Disconnect	Sensor is not connected to oximeter or sensor is inoperable
ARTF	Artifact	A detected pulse beat didn't match the current pulse interval
OOT	Out Of Track	An absence of consecutive good pulse signals
SNSA	Sensor Alarm	Sensor is providing unusable data for analysis
RPRF	Red Perfusion	Amplitude representation of low signal quality
YPRF	Yellow Perfusion	Amplitude representation of medium signal quality
GPRF	Green Perfusion	Amplitude representation of high signal quality
SYNC	Frame Sync	Occurs every first status byte of packet (1 of 25)

Table 2.2: STATUS byte description in DataFormat #2

x	x	x	x	x	x	HR8	HR7
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

x	HR6	HR5	HR4	HR3	HR2	HR1	HR0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Figure 2.6: Heart rate value in DataFormat #2

The 4th byte of the third frame contains the blood oxygen saturation value. This is also calculated on a 4-beat average in standard mode.

x	SP6	SP5	SP4	SP3	SP2	SP1	SP0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Figure 2.7: Blood oxygen saturation (SpO2) in DataFormat #2

Frames				
Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
01	STATUS	PLETH	HR MSB	CHK
01	STATUS	PLETH	HR LSB	CHK
01	STATUS	PLETH	SpO2	CHK
01	STATUS	PLETH	REV	CHK
01	STATUS	PLETH	reserved	CHK
01	STATUS	PLETH	reserved	CHK
01	STATUS	PLETH	reserved	CHK
01	STATUS	PLETH	reserved	CHK
01	STATUS	PLETH	SpO2-D	CHK
01	STATUS	PLETH	SpO2 Fast	CHK
01	STATUS	PLETH	SpO2 B-B	CHK
01	STATUS	PLETH	reserved	CHK
01	STATUS	PLETH	reserved	CHK
01	STATUS	PLETH	E-HR MSB	CHK
01	STATUS	PLETH	E-HR LSB	CHK
01	STATUS	PLETH	E-SpO2	CHK
01	STATUS	PLETH	E-SpO2-D	CHK
01	STATUS	PLETH	reserved	CHK
01	STATUS	PLETH	reserved	CHK
01	STATUS	PLETH	HR-D MSB	CHK
01	STATUS	PLETH	HR-D LSB	CHK
01	STATUS	PLETH	E-HR-D MSB	CHK
01	STATUS	PLETH	E-HR-D LSB	CHK
01	STATUS	PLETH	reserved	CHK
01	STATUS	PLETH	reserved	CHK

Table 2.3: DataFormat #2 packet description

**DataFormat #1**

The basic output data format is called DataFormat #1. In this output format only the status, heart rate and SpO2 values are transmitted once in every second. Each packet consists of three bytes. The first byte contains status information and the two upper bits of the heart rate. The second byte contains the lower 7 bits of the heart rate and the third byte contains the blood oxygen saturation value. A packet is transmitted periodically every second. At the beginning of the transmission all the three bytes are transmitted without pause, then there is no transmission until the next second.

1	SNSD	00T	LPRF	MPRF	ARTF	HR8	HR7
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Figure 2.8: Byte 1 of packet in DataFormat #1

0	HR6	HR5	HR4	HR3	HR2	HR1	HR0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Figure 2.9: Byte 2 of packet in DataFormat #1

0	SP6	SP5	SP4	SP3	SP2	SP1	SP0
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Figure 2.10: Byte 3 of packet in DataFormat #1

This basic data format can be used when there is no need for all data transmitted in DataFormat #2 or one measured value per second is enough for the system. Moreover this format consumes less power, because the sensor sends data less frequently.

### **DataFormat #7**

This data format is very similar to DataFormat #2 except the PLETH value is extended to two bytes. The first byte of a frame is the STATUS byte, the second and third bytes are for the PLETH value, and the 4th and 5th bytes are the same as in DataFormat #2. Since DataFormat #2 fully satisfies all requirements, there is no need to use this output format in our system. Therefore processing values in DataFormat #7 is not implemented in this prototype.

#### **2.4.3 Sensor Control**

The sensor manufacturer provided a breakout board for the OEM module. On this board there is an independent power supply for the sensor, DIP switches for selecting output data format and USB connectivity for transmitting the measured data directly to PC. This breakout can be easily connected to the LaunchPad with one wire for UART and one wire for GND. However there are two problems with simply connecting the sensor board to LaunchPad. Firstly after booting the system, the sensor must be switched on manually by hand. Secondly, changing the data format also requires manual handling. Therefore, we use a 2-channel 2:1 multiplexer module to be able to switch the sensor on or off and switch between the output data formats easily with the microcontroller.

The multiplexer has ten pins, 2 of which are VDD and GND that are connected to the LaunchPad. The remaining eight pins are for the two multiplexer channel. The first is responsible for switching the sensor on or off and the second channel is used for switching between DataFormat #2 and DataFormat #1. The module can be supplied from 1.65V to 5.5V and can transfer maximum of 100mA. This fulfills the requirements since the current drain of the sensor is much lower. Moreover the multiplexer has very low power dissipation, thus making the multiplexer a suitable choice for the prototype.

For the prototype, the multiplexer was soldered to a small PCB adapter for ICs by hand, then the pins were soldered to the sensor breakout board. Then the appropriate pins can be connected to the LaunchPad.

## 2.5 User Interface

A properly designed user interface is inevitable considering a monitoring device. Although nowadays a high-resolution LCD display with touch controller is widely available for embedded systems, in terms of low power dissipation, currently they are not the most efficient choice. Viable choice would be an ultra-low power consumption display with low resolution, for instance electronic paper (e-paper) displays<sup>1</sup>. These displays can preserve their state without power and only consume current when the content of display is changed. The disadvantages of these displays are they are black and white and they have the so-called effect of “ghost images”. However their greatest drawback is their refresh rate is extremely low compared to LCD displays, which heavily limits their usability in interactive mobile applications.

Since nowadays almost everything is connected and controlled by our smartphone, many ultra-low power devices do not have user interfaces at all. Instead, an application on the smartphone represents the user interface of the device. Therefore the device could be significantly smaller and more compact, moreover communication via Bluetooth LE<sup>2</sup> consumes orders of magnitude lower energy<sup>3</sup> than controlling an interactive user interface. The disadvantage of this method is although the device still remains operational, without a smartphone the user cannot interact with it.

A possible solution to evaluate a proper user interface while maintaining ultra-low power dissipation is to use a Bluetooth LE module for communication and forward data to an appropriate application installed on a smartphone, for instance feed the data into the Health<sup>4</sup> application developed by Apple for iOS.

---

<sup>1</sup> Electronic paper  
[https://en.wikipedia.org/wiki/Electronic\\_paper](https://en.wikipedia.org/wiki/Electronic_paper)

<sup>2</sup> Bluetooth Low Energy  
[https://en.wikipedia.org/wiki/Bluetooth\\_low\\_energy](https://en.wikipedia.org/wiki/Bluetooth_low_energy)

<sup>3</sup> For instance, the ST SPBTLE-RF module draws  $2\mu A$  in standby mode and around  $100\mu A$  when connected to a device.  
[http://www.st.com/web/catalog/sense\\_power/FM2185/SC1898/PF261927](http://www.st.com/web/catalog/sense_power/FM2185/SC1898/PF261927)

<sup>4</sup> Apple Health for iOS  
<http://www.apple.com/ios/health/>

## 2.6 Microcontroller

The selected microcontroller for the application processor is a Texas Instruments MSP432P401R<sup>1</sup> which is a 32-bit ARM Cortex-M4F core MCU. This state-of-the-art microcontroller combines high performance with low power dissipation. It features:

- up to 48MHz frequency core clock with configurable internal bus system and flexible clocking features
- up to 256KB of Flash Memory
- 64KB of SRAM
- 32KB of ROM with built-in MSPWare Driver Library<sup>2</sup>
- Various types of communication interfaces, including SPI, UART, USB and many more
- Ultra Low Power operating modes
- and many more features [2]

The mentioned features makes this microcontroller a perfect choice for this prototype. It provides the possibility for effectively implement the two main goals - low power system design and timing predictability for bounded response times - throughout the system.

To make the development easier, I used the TI MSP-EXP432P401R LaunchPad<sup>3</sup> which includes the TI MSP432P401R MCU along with an onboard emulator which provides programming and debugging features without any external tools. It features:

- TI MSP432P401R 32-bit ARM Cortex-M4F core MCU
- Onboard XDS-110ET emulator for programming and debugging
- Back-channel UART via USB to PC for debug purposes

---

<sup>1</sup>TI MSP432P401R

<http://www.ti.com/product/MSP432P401R>

<sup>2</sup>TI MSPWare Driver Library

<http://www.ti.com/tool/mspdriverlib>

<sup>3</sup>TI MSP-EXP432 LaunchPad

<http://www.ti.com/tool/msp-exp432p401r>



- 40 pin BoosterPack connector and support for 20 pin BoosterPacks
- 2 push buttons and 2 LEDs for user interaction

### 2.6.1 Interrupt System

ARM Cortex-M4 core microcontrollers feature advanced interrupt system, making those largely appropriate for complex systems which have many interrupt sources with different priorities.

For each interrupt, priorities are set by using 3 or 4 bits. This depends on the chip manufacturer, for instance TI's MSP432P401x microcontrollers have 3 bits for interrupt priority. This means there are 8 different priority values that can be assigned to interrupts and a specific priority value can be set to more than one interrupt. The Cortex-M core stores interrupt priority values in the most significant bits of its eight bit interrupt priority register. The highest priority is always a zero value, and the lowest priority means the highest value in the register. More specifically in the MSP432 core, an interrupt has a highest priority with the value of 0 and the lowest possible priority value that can be assigned is 7.

P	P	P	x	x	x	x	x
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Figure 2.11: Interrupt priority values stored in TI MSP432 MCU

ARM Cortex-M core microcontrollers feature a module called Nested Vectored Interrupt Controller (NVIC). NVIC supports priority grouping which provides the possibility to configure the interrupt priority system to be preemptive. This grouping divides each interrupt priority register into two fields. The upper field defines the group priority, and the lower field defines the subpriority within that group. Only the group priorities determine preemption of interrupt exceptions, and priorities with the same (or lower priority) group cannot preempt each other. If multiple interrupts within the same priority group are pending, the subpriority value determines the order they are going to be served.

## 2.6.2 Connectivity and Pinout

The LaunchPad features a 100pin version of TI MSP432P401R microcontroller and most of the pins are available for connectivity. Table A.1 summarizes the connections.

## 2.7 FreeRTOS

FreeRTOS<sup>1</sup> is a market leading de-facto standard and cross-platform Real Time Operating System (RTOS). It provides extensive features to minimize overhead. FreeRTOS is open-source and highly optimized for several processors and microcontrollers. It is officially supported by numerous chip manufacturers, strictly quality controlled, has excellent community support and free to use in commercial products, which makes FreeRTOS the best choice for this application.

### 2.7.1 Configuration

FreeRTOS provides an official port for TI MSP432 microcontroller family. This port is widely configurable depending on the actual system requirements by editing the `FreeRTOSConfig.h` file.<sup>2</sup> To reduce overall code size, it is highly recommended to disable all unused functions and components.

In this prototype, the major configuration steps are the followings:

- System Tick period is set to 1ms
- Maximum priority level is 5 for tasks, and use preemption
- Use essential features: timers, mutexes, semaphores and task notifications
- Use features for debugging and trace purposes: application task tag, check for stack overflow, idle hook callback, tick hook callback and malloc failed hook callback
- Disable all unused modules and API functions

---

<sup>1</sup>FreeRTOS - Real Time Operating System  
<http://www.freertos.org>

<sup>2</sup>FreeRTOS Configuration  
<http://www.freertos.org/a00110.html>

- Tickless Idle Mode is disabled

FreeRTOS features a special power saving mode, called Tickless Idle Mode.<sup>1</sup> According to the description, the Tickless Idle Mode stops the periodic tick interrupt during idle periods (periods when there are no application tasks that are able to execute), then makes a correcting adjustment to the RTOS tick count value when the tick interrupt is restarted. Stopping the tick interrupt allows the microcontroller to remain in a deep power saving state until either an interrupt occurs, or it is time for the RTOS kernel to transition a task into the Ready state. With this mode enabled, the energy consumption can be slightly reduced, especially when the frequency of the tick interrupt is high. The reason why this function is disabled during development is that by suppressing the ticks makes the system unpredictable and incalculable in terms of detailed timing analysis. Suppressing the system tick causes the tick interrupt non-periodic, which makes the calculation of response times ambiguous.

### 2.7.2 Interrupt Priorities

ARM Cortex-M core microcontrollers use a different convention for interrupt priority. Zero means the highest priority and the highest numerical value means the lowest priority. In contrast, FreeRTOS uses 0 for the lowest possible priority, therefore this is important to keep in mind during software development.

After defining the maximum number of different interrupt priorities in the configuration based on the microcontroller core, two unique priority levels should be defined<sup>2</sup>:

- `configKERNEL_INTERRUPT_PRIORITY`
- `configMAX_SYSCALL_INTERRUPT_PRIORITY`

`configKERNEL_INTERRUPT_PRIORITY` sets the interrupt priority used by the kernel. This is always defined as the lowest priority in the system, and usually bound to `configLIBRARY_LOWEST_INTERRUPT_PRIORITY` which always defines the lowest possible priority.

---

<sup>1</sup>FreeRTOS Tickless Idle Mode  
<http://www.freertos.org/low-power-tickless-rtos.html>

<sup>2</sup>FreeRTOS Interrupt priorities  
[http://www.freertos.org/a00110.html#kernel\\_priority](http://www.freertos.org/a00110.html#kernel_priority)

A full interrupt nesting model can be achieved by defining `configMAX_SYSCALL_INTERRUPT_PRIORITY` higher than the kernel interrupt priority. The macro sets the maximum level of priority for interrupt routines that can nest. This means that interrupt routines with the same or lower priority can call FreeRTOS API functions ending with `FromISR`. Interrupts running at higher priorities are never delayed from executing by the FreeRTOS kernel, thus no API calls are allowed from these interrupt routines.

Since Cortex-M core microcontrollers, including the TI MSP432P401R MCU use different convention for interrupt priorities, therefore the interrupt priority macros of FreeRTOS should be defined accordingly. The translation of priority values are described in Figure 2.12.

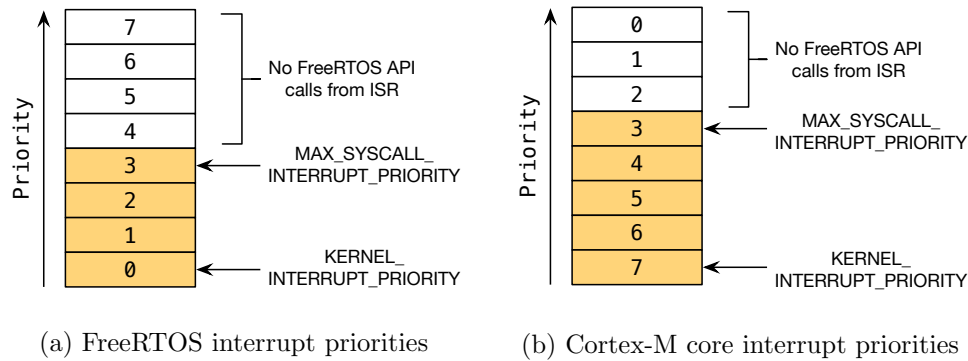


Figure 2.12: FreeRTOS interrupt priorities port to Cortex-M core MCU

Lastly, the Nested Vectored Interrupt Controller (NVIC) of the microcontroller should be configured that the interrupts can preempt each other.

### 2.7.3 Memory Management

FreeRTOS provides numerous memory management options for heap<sup>1</sup>, including the simplest scheme which does not allow dynamic memory allocation to the more complex schemes that allow combining adjacent free blocks into a single larger block. However allocating memory in heap (for instance with `malloc()` or `calloc()` C function) is never deterministic. That means there is no guarantee for returning from allocation procedure in predictable or limited amount of

<sup>1</sup>Heap memory is used for dynamic allocation

time. Considering a medical device where timing guarantee and predictability are essential, using dynamic allocation during runtime is not acceptable. This conveys appropriate heap size allocation for RTOS.

Since the RAM in the microcontroller is limited, therefore calculations were performed to allocate enough heap size for RTOS (which includes memory for tasks, queues, and other objects) while keeping sufficient memory for non-RTOS objects and stack. Moreover, the stack size for each task had to be set properly (without running out of memory) to avoid stack overflow during execution. These calculations require deep understanding how the functions, tasks and RTOS works to avoid sudden system crash, failure and memory corruption that is unacceptable considering a medical device.

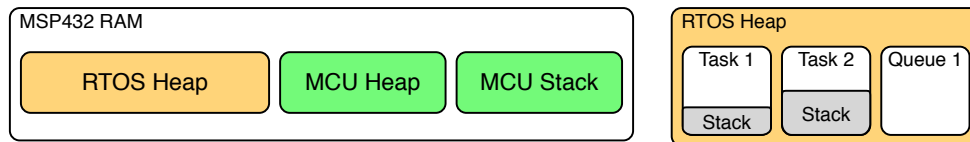


Figure 2.13: MSP432 memory management

FreeRTOS provides a basic memory management scheme called `heap_1` which allocates the maximum amount of memory space defined by the `configTOTAL_HEAP_SIZE` macro, then this area is subdivided into smaller blocks as RAM requested. This scheme does not permit memory to be freed once it has been allocated. The major advantage of this scheme is the memory management always remains deterministic. The drawback of using `heap_1` is that tasks, semaphores, mutexes, etc. cannot be deleted during runtime. Therefore this scheme was used with always keeping an eye out for memory usage during development.

# System Design

---

This chapter describes the design of the system and the required components for the prototype.

## 3.1 System Architecture

Figure 3.1 shows an overview of the system which consists of a *Pulse Oximetry Sensor*, an *Application Processor*, a *Stateful Processor Interconnect* called BOLT, a *Communication Processor* and a *User Interface*.

The application processor handles the user interface, and receives data from the pulse oximeter sensor which can measure heart rate and blood oxygen saturation. After processing the received data, the application processor forwards data to BOLT. The communication processor reads data from BOLT and transmits data to other devices on a preferred communication interface. The communication between the application and communication processor is bi-directional, therefore the received commands via the communication interface is forwarded and processed by the application processor.

In this prototype the communication processor is located on a BOLT-USB adapter which can be connected to a PC. Data can be sent between BOLT and PC with a serial terminal emulator software.

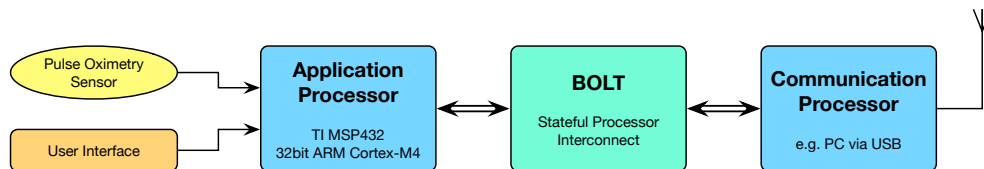


Figure 3.1: System Architecture

### 3.1.1 Communication

Considering an embedded system, usually there is one microcontroller which handles every task. This MCU handles both application-related tasks and communication tasks. In this case there are two major problems which should be handled correctly. The first is data-consistency which means the collected data should be forwarded to another device without data loss and data corruption. Since communication is usually much slower than collecting and processing data, it rather often occurs that new data is ready to be sent while the previous is still being transmitted, which can lead to data corruption or loss. Therefore these systems need to be carefully designed to effectively avoid these problems.

The second major issue is power dissipation. In systems - which aim for low energy consumption - usually the largest drain is caused by communication. Therefore communication should be highly optimized by shutting down the communication interface when it's not in use. The problem is caused by the difference between the durations of data processing and communication. Moreover data processing is usually a periodic task, however communication is often asynchronous or triggered by a specific event. Communication also requires implementations of reading and writing with sophisticated error handling. In a single processor system this leads to unnecessary wake-ups, ineffective scheduling between tasks, therefore significantly higher power dissipation.

The solution is to separate the application-related tasks from communication. By implementing these on different processors, the above mentioned issues can be eliminated. The application processor handles the collection and processing of data without the unwanted interruptions caused by communication. The data is then forwarded to the Communication Processor which only focuses on the effective transmission and reception. However in this case a new problem has to be dealt with - which is the communication and synchronization between the two processors. By using BOLT, this issue can be effectively eliminated [1].

## 3.2 Software architecture

The entire system architecture is shown in Figure 3.2. The architecture can be divided into two major domains. The first contains the hardware-related low-level components, and the second comprises hardware independent modules.

During development, I largely paid attention to write the source code accordingly to make the whole software flexible, portable and easily modifiable.

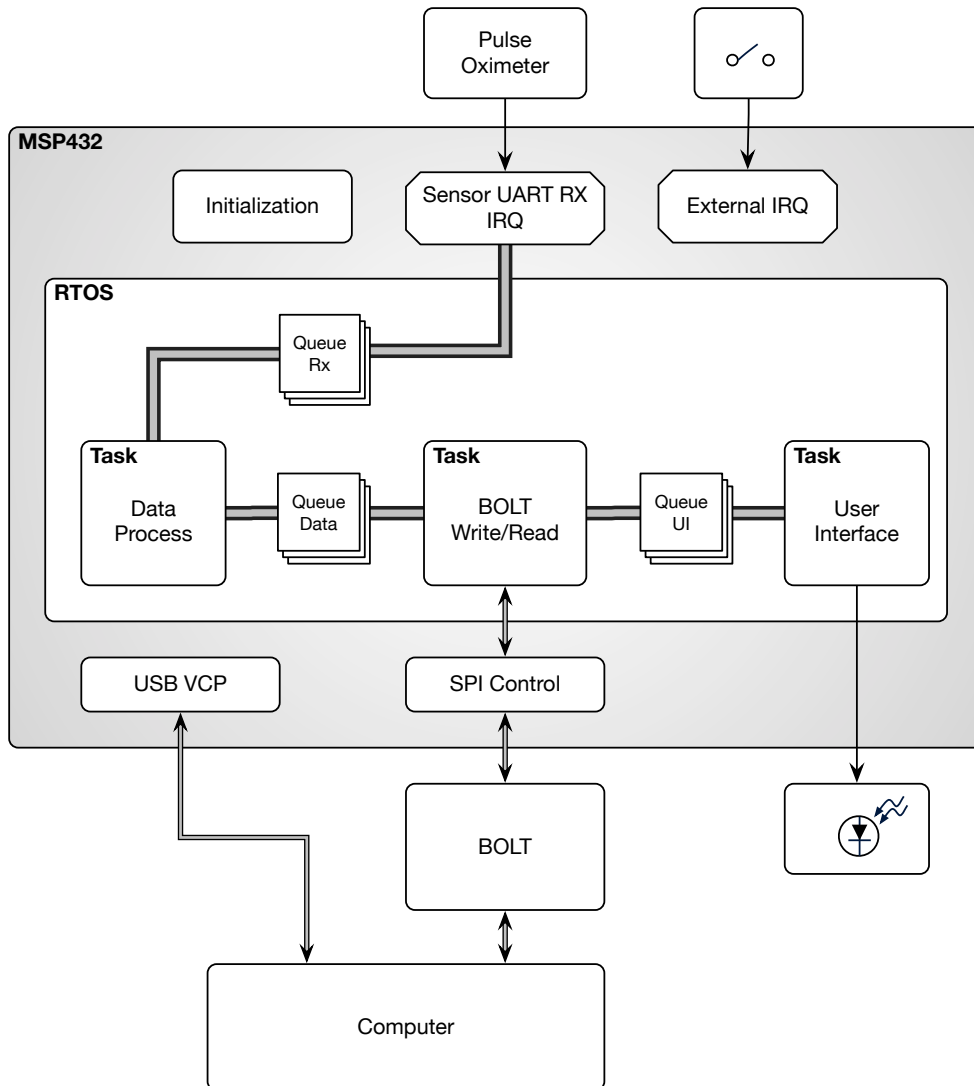


Figure 3.2: Software architecture

### 3.2.1 Initialization

Upon initialization of the system, all low-level components and peripherals are configured:

- GPIOs for LEDs, trace pins, push buttons, peripheral communication in-



terfaces

- UART initialization for the pulse oximeter
- SPI initialization for BOLT
- USB VCP initialization for debug purposes

Then the RX interrupt is set enabled for Sensor UART with maximum priority that RTOS API calls are still allowed from the interrupt service routine. This priority value is defined by `configMAX_SYSCALL_INTERRUPT_PRIORITY` (see Figure 2.12). Priorities for the external interrupts that can triggered by the push buttons are configured to one value lower priority. Therefore it is ensured that RX interrupt of the sensor UART never gets delayed by other interrupts.

The hardware initialization is followed by the RTOS initialization. FreeRTOS main configuration is stored in `FreeRTOSConfig.h` (see Section 2.7.1). After initializing the application tasks, queues and timers, the RTOS scheduler is started.

### 3.2.2 Interrupts

Currently there are two types of interrupts that are served by the system.

#### **Sensor UART RX interrupt**

Depending on the output data format, the pulse oximetry sensor periodically transmits data via UART to the microcontroller. Received bytes generate an interrupt which is processed by the system. To make the interrupt service routine as compact as possible, a structure (see Figure 3.3) is filled which contains the received character and the type of the current output format. Then this structure which contains the received byte and the current output data format is sent to a message queue called `xQueueRx`. The size of the queue can be configured in `main.h` and it is set to 125 messages by default.

Using `DataFormat #2`, an extra identification is needed to identify the very first byte of packet. Since during this output format the sensor constantly transmits data without extra delay between packets, and every frame starts with the same byte, two consecutive bytes have to be investigated to identify the first byte.

There is a special variable in the structure called `sync` which is set when the first byte is identified.

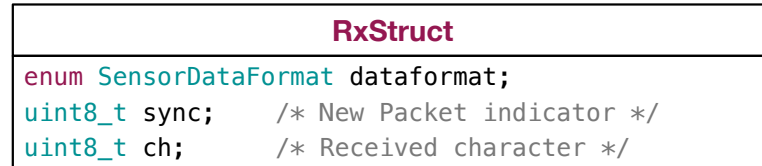


Figure 3.3: Sensor RX structure that is transmitted in `xQueueRx` queue

### External interrupts

The external interrupts can be triggered by the push buttons located on the LaunchPad. These interrupts are served by a common interrupt service routine, and the interrupt source (which button is pressed) is identified inside the routine. The priority of the external interrupts is configured to be lower than the sensor UART interrupt, thus incoming measurement data is served first.

### 3.2.3 Tasks

There are three application tasks running on the RTOS. These tasks are the data processing task for the sensor, the task for writing and reading from BOLT and the task for emulating the user interface. Each task has separate priority defined by macros. The tasks are communicating with each other by using message queues. In case there is no task to run, the RTOS runs a task called *Idle Task* which is automatically created by FreeRTOS after starting the scheduler. This task has the lowest possible priority to ensure it does not use any CPU time if there is a higher priority application task in ready state. The Data Task has the highest priority to minimize data reception delay, the BOLT Task has lower priority and the UI Task has the lowest priority.

#### Data Task

This task is waiting for messages in `xQueueRx` message queue which is filled by the sensor UART ISR. The task is a blocking task which means that it is

suspended until new message arrives from the sensor UART ISR.

Identifying the first byte of the incoming packet is straightforward by investigating the STATUS byte in DataFormat #1 (see Section 2.4.2). However in DataFormat #2, two consecutive bytes have to be investigated. This is performed in the UART ISR and the leading byte is signaled in the `sync` field of the structure.

After receiving the first byte from the sensor, a new data structure is created. This structure (see Figure 3.4) is filled with relevant data (current status<sup>1</sup> of the sensor, heart rate, blood oxygen saturation and plethysmographic pulse amplitude) during packet reception. After receiving a full packet, the data structure is ready to be sent to BOLT with measured data, and the structure is pushed into a queue called `xQueueSensorData`. This queue is used for communication between Data Task and BOLT Task.

This task has the highest priority of all tasks, thus it is ensured that processing data from the sensor is never delayed by other tasks.

<b>SensorStruct</b>	
<code>enum SensorStatus</code>	<code>status;</code>
<code>uint16_t</code>	<code>HR; /* Hearth Rate */</code>
<code>uint8_t</code>	<code>SpO2; /* Oxygen Saturation */</code>
<code>uint8_t</code>	<code>pleth; /* Plethysmographic Pulse * Amplitude */</code>

Figure 3.4: Sensor data structure

### BOLT Task

This task performs writing<sup>2</sup> and reading<sup>3</sup> operations for BOLT. The task waits for a new message arrival to `xQueueSensorData` queue. After reception, the status is checked and in case the structure contains valid data, it is formatted and transmitted to PC via BOLT. However if the finger clip is disconnected or

<sup>1</sup>The STATUS byte received from the sensor contains information about data validity and finger clip connection.

<sup>2</sup>“Writing” means MCU sends data to BOLT which forwards it to the PC

<sup>3</sup>“Reading” means MCU reads data (which was previously sent from PC) from BOLT

there is no finger present, an error message is sent accordingly.

After sending, BOLT is checked if there is a new message waiting to be read from PC. Reading operation is always performed after writing operation in case there is new data waiting. All received messages are transmitted to USB VCP of the LaunchPad for debug purposes. There are three commands which the user is able to configure and control the device.

- ‘‘dataformat1’’
- ‘‘dataformat2’’
- ‘‘duty X/Y’’

With the first two commands the sensor output data format can be configured. The second command sets the sensor duty cycling. X and Y are numeric arguments, both have to be valid numbers. X means the online duration of the sensor in seconds, Y means the duration while the sensor is powered off in seconds. All commands have to be terminated with Carriage Return (0x0D).

#### *Periodic Reading Mode*

When the sensor is switched off for a duration because of duty cycling (see Section 3.3.2), there is no message to be written to BOLT during this period. Therefore when a new message arrives to be read from BOLT, the reading operation has to be delayed until a new writing operation occurs. In case the ratio of sensor on duration and sensor off duration is low, the response within a bounded time frame is not guaranteed, which makes the system unpredictable. To eliminate this delay, the task can be configured to be periodic. This can be done with the `mmdConf_BoltReadPeriodicMode` macro. In this mode the task blocks and waits for new message in `xQueueSensorData` only for a limited time instead of blocking infinitely when there is no message in the queue. After the timeout which can be configured with the `mmdConf_BoltReadPeriod` macro, the task checks whether there is data to be read from BOLT, and performs a reading operation if necessary.

Please note that on timing diagrams the BOLT task is labelled as RW task referring to Reading and Writing operations.

## UI Task

The UI task handles the user interface connected to the system. A possible user interface can be a touch screen with low energy consumption or an electronic paper (e-paper) display for ultra low power consumption. Because of time constraints, the user interface is simulated with LEDs which are a red and an RGB LED located on the LaunchPad.

The task blocks and waits for a new message in `xQueueUI` queue which provides communication for other tasks with the UI task. The queue accepts messages with `UIStruct` type.

<b>UIStruct</b>	
<code>uint8_t</code>	<code>leds; /* Selected LEDs */</code>
<code>uint16_t</code>	<code>ms; /* Duration in millisec */</code>

Figure 3.5: Message structure for `xQueueUI` queue

Focusing on using minimal resources, the structure comprises only two variables. The `leds` variable is a 8bit unsigned integer which contains the required status of the LEDs. More LEDs can be switched on or off at a time: the state of the last 4 bits define whether the appropriate LED will be turned on or off. The `ms` variable contains the duration of the selected LEDs are switched on in milliseconds. After receiving a structure from the queue, a timer is started with timeout defined in `ms` variable. When the timer fires, the LEDs are automatically switched off.

However switching on LEDs for indefinite duration is also possible. The MSB bit in `leds` variable is a control flag. If it is set, the selected LEDs will be turned on or off for indefinite time period. In this case if the `ms` variable is set to 0, then the selected LEDs will be turned off, else the LEDs will be turned on.

Flag	x	x	x	RGB blue	RGB green	RGB red	LED_1 red
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Figure 3.6: Bitwise description of `leds` variable

## 3.3 Low Power System Design

### 3.3.1 Microcontroller Low Power Modes

From the low power design perspective, the various low power modes supported by the microcontroller are greatly important. This MCU supports an active mode which comprises six different operating modes depending on the core voltage and clock source. There are five different low power modes including two shutdown modes. The modes are summarized in Table 3.1.

### 3.3.2 Sensor Duty Cycling

There is no option to configure the sensor, which means it constantly transmits data when it is turned on. In most of the cases this generates unnecessarily high amount of data and consumes a lot of energy. To reduce power dissipation, duty cycling is implemented with a timer called `xSensorWakeUp`. The sensor is switched on for a defined time period, then it is turned off for another defined duration. The default times can be set with `defaultSensorOnlineDuration` and `defaultSensorOfflineDuration` macros. Moreover these values can be changed with commands via BOLT (see Section 3.2.3).

### Idle Task with Low Power Modes

When there is no task running, an idle task is executed in FreeRTOS. The easiest method to save power is to put the microcontroller in a low power mode (see Table 3.1) when the RTOS switches in the idle task.

After investigating the low power modes, it is not suitable for our application to use the *Shut Down* modes (*LPM3.5* and *LPM4.5*), since the flash memory is powered down and there is no state retention.

State retention is possible by using *LPM3* or *LPM4*<sup>1</sup> modes. However the system needs a way to wake itself up to perform a system tick and continue to run the RTOS. Since the high frequency peripherals are turned off in these modes, this means that the SysTick timer is also powered down. There are two possible

---

<sup>1</sup>LPM4 mode is the extension of LPM3, where the low frequency peripherals are disabled, including the real-time clock and the watchdog timer.

Power Mode	Features
Active Mode	CPU is active and full peripheral functionality is available All low and high frequency clock sources can be active Flash memory and all enabled SRAM banks are active
LPM0	CPU is inactive but full peripheral functionality is available All low and high frequency clock sources can be active Flash memory and all enabled SRAM banks are active
LPM3	CPU is inactive and peripheral functionality is reduced Only RTC and WDT modules can be functional All other peripherals and retention enabled SRAM banks are kept under state retention power gating Flash memory is disabled, SRAM banks not configured for retention are disabled Only low frequency clock sources (LXFT, REFO, VLO) can be active All high frequency clock sources are disabled Device I/O pin states are latched and retained DC/DC regulator can not be used
LPM4	Achieved by entering LPM3 with RTC and WDT modules disabled CPU is inactive with no peripheral functionality All peripherals and retention enabled SRAM banks are kept under state retention power gating Flash memory is disabled, SRAM banks not configured for retention are disabled All low and high frequency clock sources are disabled Device I/O pin states are latched and retained DC/DC regulator can not be used
LPM3.5	Only RTC and WDT modules can be functional CPU and all other peripherals are powered down Only Bank0 of SRAM is under data retention, all other SRAM banks and flash memory are powered down Only low frequency clock sources (LXFT, REFO, VLO) can be active All high frequency clock sources are disabled Device I/O pin states are latched and retained DC/DC regulator can not be used
LPM4.5	Core voltage is turned off CPU, flash memory, all SRAM banks and all peripherals are powered down All low and high frequency clock sources are powered down Device I/O pin states are latched and retained

Table 3.1: TI MSP432P401R Power Modes [2]

options to trigger an interrupt that can wake up the system. Firstly an alarm of the RTC can be configured to generate an interrupt, however after checking the documentation, the minimum time duration for setting an alarm is 1 second. Therefore this is not a viable option.

Secondly, the watchdog timer can be configured to run from a low frequency clock source. When the WDT fires, an interrupt is generated which can wake up the system. I tried to evaluate this method, however I did not observe the expected behavior. I was able to send the microcontroller into LPM4 where the drain was  $5\mu A$ , although using this mode the system cannot be configured to wake itself up. Then I successfully sent the MCU to LPM3 with low frequency clock sources enabled for watchdog, and configured the WDT to fire after approximately  $1ms$ . However this time I did not turn on the interrupt of WDT in order to have an overview of current drain which was  $530\mu A$ <sup>1</sup> in this mode. Then I enabled the interrupt and configured the ISR of WDT to wake up and continue to run the RTOS. In this case the system did not work the way I expected. The MCU did not enter LPM3 and the current drain remained in the order of magnitude of milliamperes before entering the low power mode. Although the watchdog timer started and fired, moreover the ISR of WDT did execute, the microcontroller was not able to go into LPM3 for an unknown reason.

There is a possible workaround for this problem by using an external interrupt source to wake up the system. To evaluate this, a configurable, external hardware timer chip is needed with ultra low power consumption which can generate a pulse on a pin to trigger an interrupt. Every time the RTOS enters the idle task, the microcontroller configures the timer to start and generate a pulse. Then the MCU can enter into LPM3 or even LPM4 and the system can wake up and run after receiving an interrupt from the external timer.

Because of the problems I encountered after experimenting with LPM3, I decided to use *LPM0* mode. This is a low power mode where CPU is inactive but full peripheral functionality is available along with all high frequency clock sources. Thus the SysTick timer can run after entering LPM0, and consequently the system automatically wakes up after a tick interrupt and continues its execution in active mode. See Section 4.2.4 for detailed power analysis of the idle task by using LPM0 mode.

---

<sup>1</sup>The datasheet of the microcontroller does not provide data about current consumption of peripherals, because this chip is still in pre-production state.



# Evaluation

---

In this chapter the evaluation of the developed RTOS software are presented along with both timing and power analysis. The execution times of all tasks are measured, then bounded response times are calculated for each task. The analytical results are validated in practice. In the power analysis section, the results of power dissipation measurements performed in different scenarios are presented.

## 4.1 Timing Analysis

To evaluate system behavior and bounded response times in different operation modes and scenarios, I performed a thorough system-wide timing analysis. The execution times of all tasks and runtime overheads were measured with a Saleae Logic 8 analyzer with 10MS/s sampling rate. The `trace.h` and `trace.c` files contain the pins, macros and functions used for RTOS trace. Each interrupt, task, timer and the tick interrupt is assigned to a pin; and when they are triggered or switched in by the RTOS, the appropriate pin is pulled high, then pulled low before before context switch. The trace pins are connected to the logic analyzer, therefore the operation of the system can be extensively analyzed.

### 4.1.1 Medical Sensor

After powering on the sensor, it takes 1812.349ms to receive the first byte on average. However the sensor needs a few pulses to measure the heart rate and blood oxygen saturation level correctly, which means the few first packets are flagged as *Out of Track*. It takes 10 packets using DataFormat #2 and 4 packets using DataFormat #1 on average to receive precise values. In conclusion, the average time elapsed from power up to the first valid data arrives to the micro-

controller is 5.356s using DataFormat #2 and 5.746s using DataFormat #1. The following tables summarize the timings for the sensor.

<b>Event</b>	<b>Min</b>	<b>Avg</b>	<b>Max</b>	<b>Unit</b>	<b>NoM<sup>1</sup></b>
Power on - First UART RX	1808.395	1812.349	1815.700	ms	15
Power on - First valid packet	4.136	5.356	6.809	s	15
Number of received packets before first valid packet	6	10	14	-	15
Time between interrupts	1.024	1.024	1.024	ms	5
Time between frames	9.148	9.148	9.148	ms	5
Whole packet reception	324.166	324.166	324.166	ms	5

Table 4.1: Sensor timings using DataFormat #2

<b>Event</b>	<b>Min</b>	<b>Avg</b>	<b>Max</b>	<b>Unit</b>	<b>NoM<sup>1</sup></b>
Power on - First UART RX	1806.469	1813.987	1818.973	ms	15
Power on - First valid packet	4.812	5.746	6.815	s	15
Number of received packets before first valid packet	3	4	5	-	15
Time between interrupts	1.024	1.024	1.024	ms	5
Time between packets	997.20	997.95	998.50	ms	10
Whole packet reception	2.100	2.101	2.102	ms	10

Table 4.2: Sensor timings using DataFormat #1

According to the measurements, using DataFormat #2 has slightly faster response times after power-up.

---

<sup>1</sup>Number of Measurements

### 4.1.2 BOLT

The SPI interface for BOLT is configured to 3MHz clock speed<sup>1</sup>. The timing figures relating to both writing and reading operations can be found in Appendix E.

#### Writing to BOLT

First the MCU sets the *MODE* line to high for writing operation. Then *REQ* line is pulled high and the MCU is waiting for BOLT to pull *ACK* high. After *ACK*, the MCU starts transmitting the data, then the *REQ* line is pulled down and BOLT acknowledges the transmission by pulling the *ACK* line down. The overheads of all operations related to writing to BOLT are presented in Table 4.3.

Event	Duration [ $\mu s$ ]
REQ pulled high - ACK response	34.0
ACK response - first byte transfer	8.9
Last byte transfer - REQ line pulled low	3.0
REQ pulled low - ACK pulled low	19.1

Table 4.3: BOLT writing operation timings (average values)

#### Reading from BOLT

BOLT notifies the microcontroller that there is new data to read by pulling the *IND* line high. The MCU initiates the reading operation by setting *MODE* to low and pulling *REQ* line high. Then the MCU waits for BOLT to pull *ACK* high as an acknowledgement. After receiving the *ACK*, the MCU starts receiving data. BOLT automatically pulls the *ACK* line low when the whole message is transferred. In case there is no new message in BOLT to read, the *IND* line is also pulled low. Finally the MCU pulls *REQ* line low. The overheads of all operations related to reading from BOLT are presented in Table 4.4.

---

<sup>1</sup> BOLT allows up to 4MHz SPI clock frequency

Event	Duration [ $\mu s$ ]
REQ pulled high - ACK response	31.2
ACK response - first byte transfer	10.1
Last byte transfer - ACK line pulled low	1.0
ACK pulled low - REQ pulled low	6.6

Table 4.4: BOLT reading operation timings (average values)

The analysis of BOLT timing (see Appendix E) fully matches the results described in the paper of BOLT [1].

### 4.1.3 RTOS Tasks

After initialization, the sensor is turned on (*TimerWkUp* trace goes high in Figure 4.1) and the system is ready to receive data from the sensor. As the sensor heats up, it transmits a dummy byte which can be seen as the very first single pulse on *IT.Rx* trace line. Then the sensor starts transmitting data. After receiving a packet, the values are sent to BOLT. This occurs after every packet reception. Because of the duty cycling, the sensor is switched off after 10 seconds by default. This value can be changed in the configuration or via a command sent from PC. After every writing operation, BOLT is checked whether there is a new message to be read, and the MCU performs a reading operation if needed. If the system is configured in a way that when there is no writing operation, then BOLT task<sup>1</sup> does not block indefinitely and it periodically performs a reading operation.

Timing diagrams taken with the logic analyzer can be found in Appendix F.

<sup>1</sup>Please note that on timing diagrams the BOLT task is labelled as RW task referring to Reading and Writing operations.

### Task timings using DataFormat #2

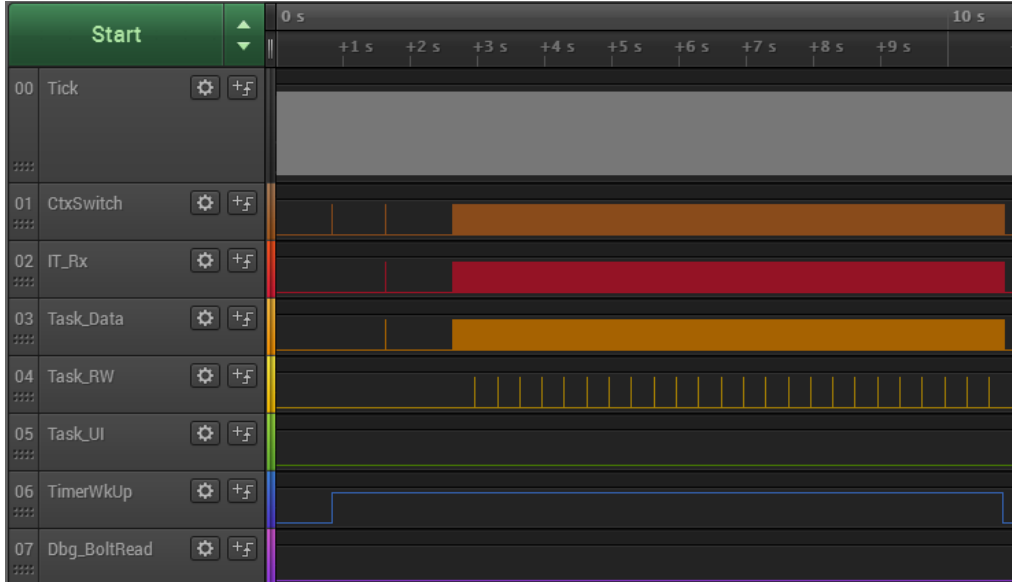


Figure 4.1: RTOS trace using DataFormat #2

Task execution	Min	Avg	Max	Unit	NoM <sup>1</sup>
UART ISR	18.30	18.34	18.50	$\mu s$	10
Data Task	42.90	49.89	57.0	$\mu s$	20
BOLT Task (write only)	227.70	230.17	234.80	$\mu s$	10
BOLT Task (write & read)	53.644	53.672	53.758	$ms$	10
UI Task	193.30	196.61	203.80	$\mu s$	10
Period of BOLT Task	333.308	333.313	333.315	$ms$	10

Table 4.5: RTOS timings using DataFormat #2

The Data Task has slightly larger difference between minimum and maximum values. The reason is after receiving the last byte of a packet, this task fills a structure and pushes it into `xQueueSensorData` queue for the BOLT Task. This requires more CPU cycles.

<sup>1</sup> Number of Measurements

### Task timings using DataFormat #1

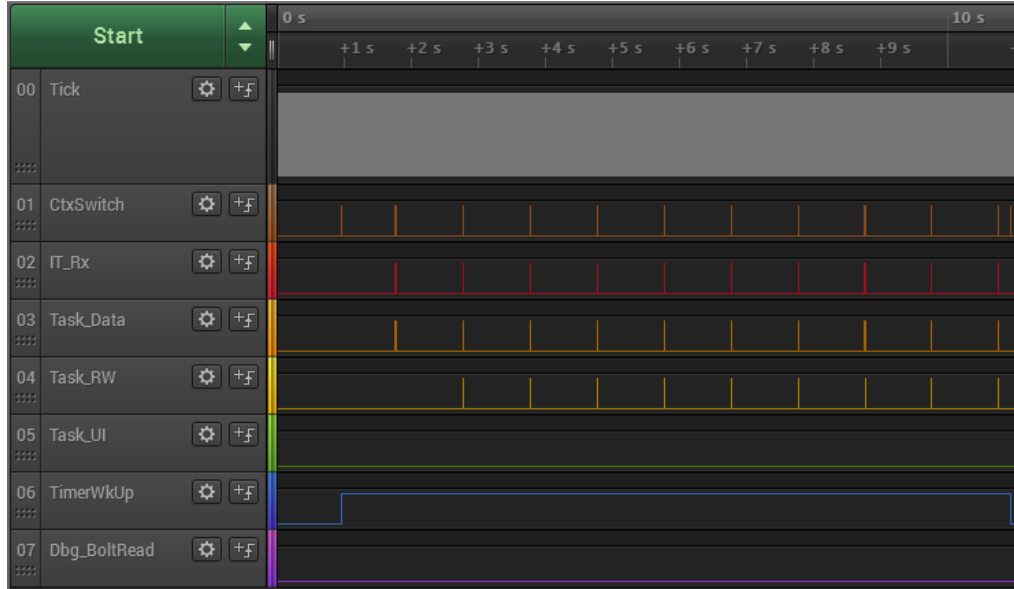


Figure 4.2: RTOS trace using DataFormat #1

Task execution	Min	Avg	Max	Unit	NoM <sup>1</sup>
UART ISR	17.70	17.76	17.80	$\mu s$	10
Data Task	42.80	49.83	56.70	$\mu s$	20
BOLT Task (write only)	225.90	233.69	236.10	$\mu s$	10
BOLT Task (write & read)	53.643	53.670	53.759	$ms$	10
UI Task	194.21	195.42	202.96	$\mu s$	10
Period of BOLT Task	998.80	998.90	999.40	$ms$	10

Table 4.6: RTOS timings using DataFormat #1

All tasks have similar execution time, however the sensor UART RX ISR execution time is slightly higher using DataFormat #2. The reason is an extra condition needed for identifying the first byte of a packet using DataFormat #2.

<sup>1</sup> Number of Measurements

#### 4.1.4 RTOS Context Switching

To measure the context switching times, the RTOS kernel was modified to be able to trace the actual context switches and measure their execution times.

The context switching times are the same for using both data formats. The values are summarized in Table 4.7.

Context switching times	Min	Avg	Max	Unit	NoM <sup>1</sup>
UART ISR → Data Task	4.70	4.76	4.80	$\mu s$	10
Data Task → BOLT Task	2.80	2.86	2.90	$\mu s$	10
BOLT Task → UI Task	2.70	2.71	2.80	$\mu s$	10
SysTick & PendSV execution time	4.81	4.90	5.0	$\mu s$	10

Table 4.7: Context switching times

The scheduler execution time with a context switch (SysTick & PendSV) is significantly larger than the context switching time between tasks. The scheduler execution time is measured from ISR begin to end, which means this is the real overhead value. However during the PendSV routine - which manages the context switch, the trace flag for the upcoming task is set during the actual context switch, and not at the end of the ISR. Therefore the durations of context switches are actually longer than the times directly measured with FreeRTOS trace macros.

#### 4.1.5 Data Flow

The sensor measures vital signs and sends data via UART to the microcontroller. The received bytes are forwarded through the `xQueueRx` message queue to the Data task where the incoming bytes are processed. After receiving a whole packet, a structure is filled with the measured values. The structure is forwarded through `xQueueSensorData` to BOLT task which sends the received data to BOLT.

The whole system is event-driven, which means each task is executed after triggering the corresponding event. Sensor UART RX ISR is triggered after receiving a byte from the sensor, data task is executed when there is a message in

---

<sup>1</sup>Number of Measurements

`xQueueRx` queue, and BOLT task is executed when a structure is pushed into `xQueueSensorData` queue.

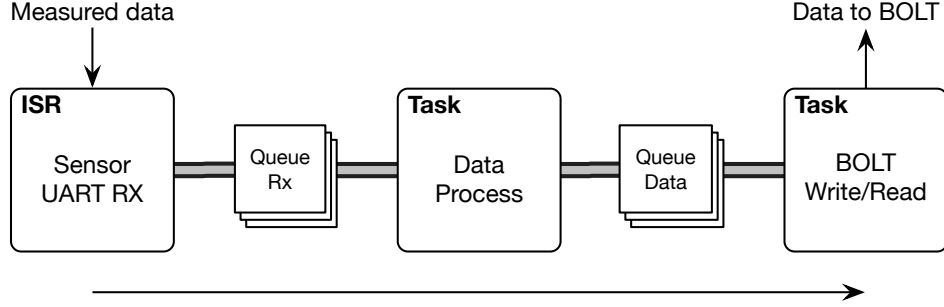


Figure 4.3: Data Flow

#### 4.1.6 Response Times

Considering a medical device, having bounded response times is essential for all tasks and interactions. The greatest challenge in a complex system is to accurately calculate its response time. Tasks have interference with higher priority tasks, interrupts and the RTOS system tick. The calculation is not only complex because of all the delays caused by the system tick, context switches, interrupts and other RTOS modules (i.e. software timers, co-routines, etc.), but also the fact that switching in (and out) a task can occur without a system tick. For instance, a context switch is forced after executing an interrupt service routine regardless of system tick.

Most RTOS, including FreeRTOS, use a fixed priority preemptive scheduling with known context switching time, every task has a different priority in the system and their priorities do not change during operation. Therefore the maximum response time can be calculated with Formula 4.2 as found in [4].

$$R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \quad (4.1)$$



The iterative solution for Formula 4.1 is the following:

$$R_i^0 = C_i, \quad R_i^{(n+1)} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^{(n)}}{T_j} \right\rceil C_j \quad (4.2)$$

For task  $i$ ,  $R_i$  is the response time and  $C_i$  is the execution time and  $hp$  is the set of tasks with higher priority than task  $i$ .  $T_j$  is the period and  $C_j$  is the execution time of a task which has higher priority than task  $i$ .

In a real-time operating system, there are interrupts and there is a special interrupt called system tick (SysTick) interrupt which is triggered periodically by the scheduler. Therefore the above mentioned general formula (4.2) should be enhanced for our system: [4]

$$\begin{aligned} R_i^{(n+1)} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^{(n)}}{T_j} \right\rceil (C_j + C_{sw}) + \sum_{\forall k \in it} \left\lceil \frac{R_i^{(n)}}{T_k} \right\rceil (C_k + C_{sw}) + \\ + \left\lceil \frac{R_i^{(n)}}{T_{clk}} \right\rceil C_{clk} + T_{clk} \end{aligned} \quad (4.3)$$

$C_{sw}$  is the delay caused by executing a context switch, the set of interrupts is labelled as  $it$ ,  $C_{clk}$  is the execution of the system tick interrupt and  $T_{clk}$  is the period of the system tick. The first summation is the delay caused by higher priority tasks and the second summation calculates the cost of handling interrupts. Moreover the response time is delayed by the system tick interrupt and the period of the system tick. In case the task becomes ready before a system tick, then the task has to wait until the next tick when the scheduler switches in the task.

The response times are calculated for using DataFormat#2.

Figure 4.4 shows the priorities in the system. Each task and interrupt has different priority and there is no change of priorities during operation. The external interrupt is only used for debug purposes, and the interrupts of push buttons are disabled during normal operation. Therefore the external interrupts have no effect on the response times.

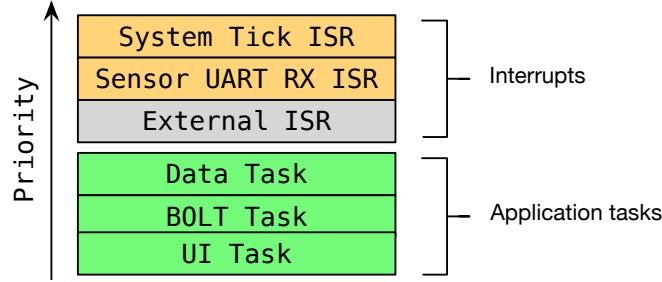


Figure 4.4: RTOS interrupt and task priorities

Table 4.8 summarizes the maximum execution times and minimum periods for each interrupt and task. The calculation of periods are described for better understanding in the followings. Period of system tick is configured and fixed for 1ms. The pulse oximeter operates at a fixed baudrate of 9600. This means the maximum transfer rate of the sensor is 9600 bit/s which equals 1200 byte/s. This means that a byte arrives every  $833.3\mu s$  at most, thus the minimum period of sensor UART RX interrupt is  $T_{rx} = 833.33\mu s$ . Optimally there is no congestion in `xQueueRx` queue and every incoming byte is handled without delay by the Data Task. Therefore the minimum period of Data Task equals the period of sensor UART ISR:  $T_{data} = T_{rx} = 833.3\mu s$ . By using DataFormat #2, the maximum frequency of a whole packet is 3 packet per second. BOLT Task is triggered after receiving an entire packet, thus the minimum period of BOLT Task is  $T_{bolt} = 333.3ms$ .

Task	$C_{max}$ execution time	$T_{min}$ period
System Tick and Context Switch	$5.0 \mu s$	$1000.0 \mu s$
Sensor UART RX ISR	$18.5 \mu s$	$833.3 \mu s$
Data Task	$57.0 \mu s$	$833.3 \mu s$
BOLT Task	$53.759 ms$	$333.3 ms$
UI Task	$203.8 \mu s$	-

Table 4.8: Maximum execution times and task periods

### Sensor UART ISR Response Time

The interrupt of sensor UART RX has the highest priority and only the RTOS scheduler has higher. Moreover interrupts are immediately switched in regardless of the system tick. Therefore the formula for calculating the response time of sensor UART RX ISR is the following:

$$R_{rx}^{(n+1)} = C_{rx} + \left\lceil \frac{R_{rx}^{(n)}}{T_{clk}} \right\rceil C_{clk} \quad (4.4)$$

$R_{rx}^{(0)} = 18.5\mu s$ ,  $R_{rx}^{(1)} = 23.5\mu s$  and  $R_{rx}^{(2)} = 23.5\mu s$ . After performing the calculations, the worst-case response time is  $R_{rx} = 23.5\mu s$ .

### Data Task Response Time

The Data task has the highest priority among all other tasks and only interrupts and system tick can delay its response time. Since FreeRTOS forces a context switch after executing an interrupt routine, moreover the Data task is triggered by the Sensor UART RX ISR, the execution of the Data task is not delayed by the period of the system tick.

$$R_{data}^{(n+1)} = C_{data} + \left\lceil \frac{R_{data}^{(n)}}{T_{rx}} \right\rceil (C_{rx} + C_{sw}) + \left\lceil \frac{R_{data}^{(n)}}{T_{clk}} \right\rceil C_{clk} \quad (4.5)$$

$R_{data}^{(0)} = 57.0\mu s$ ,  $R_{data}^{(1)} = 80.5\mu s$  and  $R_{data}^{(2)} = 80.5\mu s$ . Therefore the maximum response time is  $R_{data} = 80.5\mu s$ .

### BOLT Task Response Time

Based on the data flow (see Figure 4.3), BOLT task execution can be interrupted by Sensor UART RX interrupt routine, the execution of Data task and the system tick interrupt, moreover it is delayed by the period of the system tick. Therefore the calculation for the response time is the following:

$$\begin{aligned}
R_{bolt}^{(n+1)} = C_{bolt} + \left[ \frac{R_{bolt}^{(n)}}{T_{data}} \right] (C_{data} + C_{sw}) + \left[ \frac{R_{bolt}^{(n)}}{T_{rx}} \right] (C_{rx} + C_{sw}) + \\
+ \left[ \frac{R_{bolt}^{(n)}}{T_{clk}} \right] C_{clk} + T_{clk}
\end{aligned} \tag{4.6}$$

$R_{bolt}^{(0)} = 54.7590ms$ ,  $R_{bolt}^{(1)} = 59.9365ms$ ,  $R_{bolt}^{(2)} = 60.4145ms$ ,  $R_{bolt}^{(3)} = 60.4950ms$  and  $R_{bolt}^{(4)} = 60.4950ms$ . The worst-case response time is  $R_{bolt} = 60.495ms$ .

This is the most relevant response time from a system-wide perspective. This means the system can measure, process and forward data to a connected device or PC within this time frame. This is a worst-case value and the response is guaranteed within this time frame. Moreover BOLT also has a fixed response time, therefore the overall response time of data from sensor measurement to the arrival to a communication interface of a specific device can be easily calculated.

In case the reading operation is completely disabled in the system, the BOLT task has significantly lower maximum execution time  $C_{bolt_{writeonly}} = 236.10\mu s$ . In this case the response time of BOLT task can be calculated in the following:  $R_{bolt}^{(0)} = 1236.1\mu s$ ,  $R_{bolt}^{(1)} = 1316.6\mu s$  and  $R_{bolt}^{(2)} = 1316.6\mu s$ . The worst-case response time in a write-only scenario is  $R_{bolt} = 1.3166ms$ .

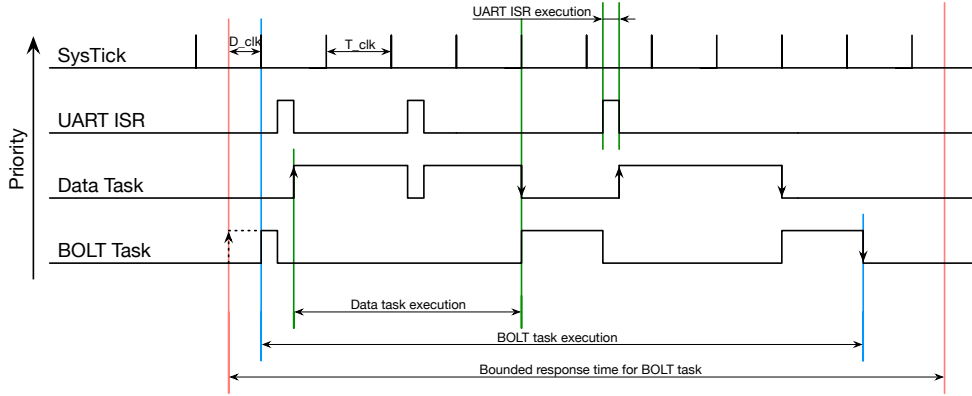


Figure 4.5: BOLT task response

### Packet Response Time

The system performs a writing operation after a whole packet is received from the sensor. By using DataFormat #2, the BOLT task is triggered after receiving 125 bytes. However there are different delays between each received byte in a frame, and between the frames. (See Table 4.1.) Each byte follows each other with a delay of  $1.024ms$  within a frame, and there is a delay of  $9.148ms$  between frames. These delays has to be taken into account when calculating the response time of a whole packet. The response time can be calculated as follows:

$$R_{packet} = R_{data} \cdot 125 + R_{bolt} + \sum D_{bytes} + \sum D_{frames} \quad (4.7)$$

where  $D_{bytes}$  means the delay between bytes in a frame, and  $D_{frames}$  implies the delay between frames.

$$\sum D_{bytes} = \sum_1^{25} (4 \cdot 1.024ms) = 102.40ms$$

$$\sum D_{frames} = \sum_1^{24} 9.148ms = 219.552ms$$

$$R_{packet} = 80.5\mu s \cdot 125 + 1.3166ms + 102.40ms + 219.552ms = 333.3311ms.$$

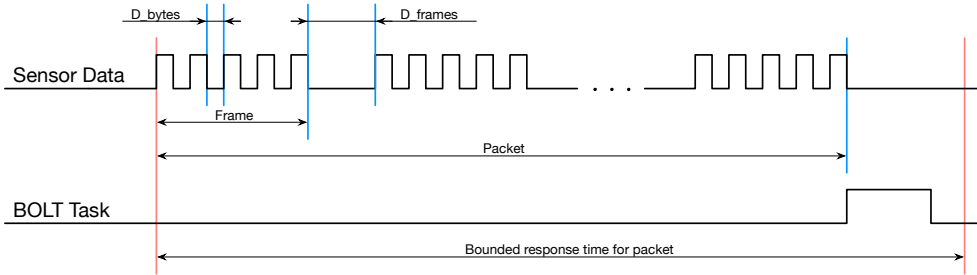


Figure 4.6: Packet response time

### UI Task Response Time

This task has the lowest priority, thus every other task, interrupts and system tick can delay its execution. The formula develops as the following for the UI

task:

$$\begin{aligned}
 R_{ui}^{(n+1)} = & C_{bolt} + \left\lceil \frac{R_{ui}^{(n)}}{T_{bolt}} \right\rceil (C_{bolt} + C_{sw}) + \left\lceil \frac{R_{ui}^{(n)}}{T_{data}} \right\rceil (C_{data} + C_{sw}) + \\
 & + \left\lceil \frac{R_{ui}^{(n)}}{T_{rx}} \right\rceil (C_{rx} + C_{sw}) + \left\lceil \frac{R_{ui}^{(n)}}{T_{clk}} \right\rceil C_{clk} + T_{clk} \quad (4.8)
 \end{aligned}$$

In case the BOLT task only performs write-only operations, the response time can be calculated as below.  $R_{ui}^{(0)} = 203.8\mu s$ ,  $R_{ui}^{(1)} = 1520.4\mu s$  and  $R_{ui}^{(2)} = 1520.4\mu s$ . The UI task has the highest response time of all tasks, with the value of  $R_{ui} = 1.5204ms$ .

However if the calculations are performed with the maximum response time of BOLT task that performs a reading operation in every execution, the limit of UI task response time does not exist. This is because the execution time of BOLT task is high while its period is low, therefore in every iteration the response time grows and does not converge to a number.

### Summary of Response Times

Table 4.9 summarizes the worst-case response times for the tasks in the system. These values are guaranteed and validated in practice, therefore each task and operation has bounded response times. This conveys predictability and execution in a maximum time frame.

Task	Theoretical Bound	Measurements	Deviation
Sensor UART ISR	23.5 $\mu s$	18.3 $\mu s$	22.12 %
Data Task	80.5 $\mu s$	64.7 $\mu s$	25.20 %
BOLT Task	60.495 $ms$ / 1.3166 $ms$ <sup>1</sup>	0.3012 $ms$	77.12 %
Packet (DF#2)	333.3311 $ms$	324.4501 $ms$	2.66 %
UI Task	— / 1.5204 $ms$ <sup>1</sup>	—	—

Table 4.9: Worst-case response times

<sup>1</sup>In the case where the BOLT task only performs write operations

## 4.2 Power Analysis

I set up different scenarios and made several measurements to obtain power dissipation of the whole system. I used a Keysight Technologies N6705A<sup>1</sup> (formerly manufactured by Agilent) DC power analyzer to perform power profiling. I used the maximum sampling rate of device to obtain precise real-time consumption data. The sampling period was  $T_s = 2.048 \cdot 10^{-5}s$ .

To perform detailed power analysis, the LaunchPad and the pulse oximetry sensor were connected to different channels of the power analyzer in order to measure their power dissipation separately.

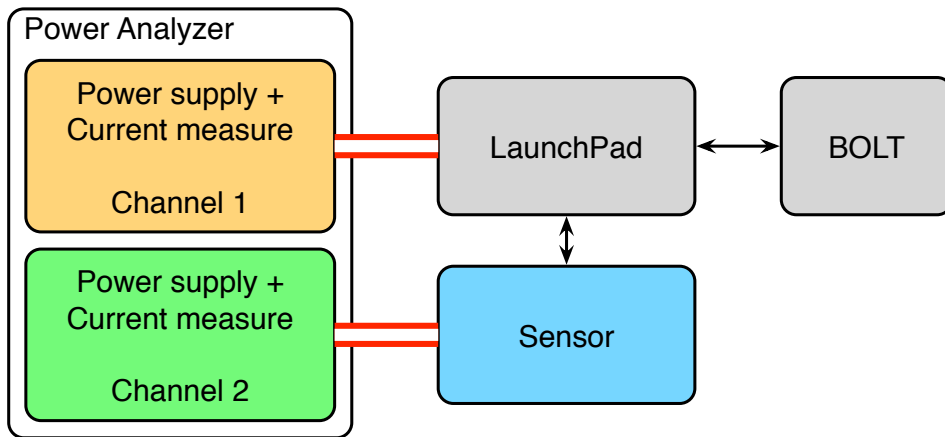


Figure 4.7: Power analysis setup

Table 4.10 summarizes the eight scenarios chosen to evaluate the prototype device. The scenarios were constructed by adjusting the following setup parameters:

- The sensor output data format.
- If a command was sent from PC via BOLT which was read and processed by the MCU. This is indicated in 'BOLT operation' column.

<sup>1</sup> Keysight Technologies N6705A DC Power analyzer  
<http://www.keysight.com/en/pd-1123271-pn-N6705A/dc-power-analyzer-modular-600-w-4-slots?cc=US&lc=eng>

- If the system was configured to check BOLT periodically for reading, regardless of writing operation.
- If a LED was flashed after every successful BOLT write operation.

#	DataFormat	BOLT operation	Periodic read	UI	Duration
1	DF#2	Write			12s
2	DF#2	Write		•	12s
3	Mixed	Write & Read		•	12s
4	DF#1	Write			12s
5	DF#2	Write			30s, DC <sup>1</sup>
6	DF#2	Write	•		30s, DC <sup>1</sup>
7	DF#1	Write			30s, DC <sup>1</sup>
8	DF#1	Write	•		30s, DC <sup>1</sup>

Table 4.10: Scenarios for power analysis

#### 4.2.1 Analysis of Individual Interrupts and Tasks

In this section the basic individual interrupts and tasks which are most common in all scenarios are analyzed.

##### Sensor UART RX ISR, Data and BOLT Task Execution

During the service of an interrupt request that had arrived from the sensor, a message is pushed into the `xQueueRx` queue which is going to trigger the Data task. Since this task has the highest priority, it will immediately execute after the ISR. Figure 4.8 shows the power profile of execution using DataFormat #2. The average execution time of UART ISR and Data task with the context switches is  $18.5\mu s + 57.0\mu s + 2 \cdot 5.0\mu s = 85.5\mu s$ , which match the width of the power pulse which is approximately  $100\mu s$ .

<sup>1</sup>Duty cycling with 10s on and 15s off; 30s of total measurement duration



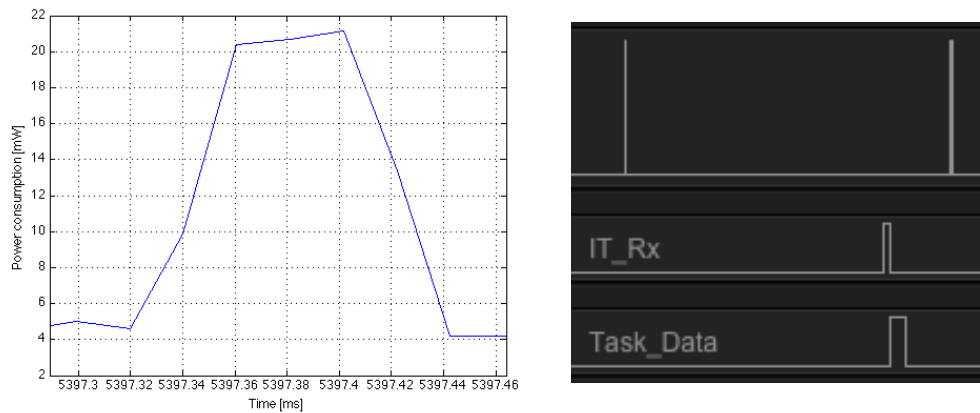


Figure 4.8: Sensor UART RX ISR and Data task execution analysis

In case the received byte is the last byte of a packet, a structure is pushed into `xQueueSensorData` queue filled with data extracted from the packet in Data task. The BOLT task has the highest priority after Data task, therefore BOLT task is going to be triggered immediately after execution. The chain of execution is shown in Figure 4.10, which is confirmed with the power analysis shown in Figure 4.9. The width of power trace is approximately  $330\mu s$ , while the average duration of executing RX ISR, Data and BOLT task with the context switches is  $18.5\mu s + 57.0\mu s + 234.8\mu s + 3 \cdot 5.0\mu s = 325.3\mu s$ .

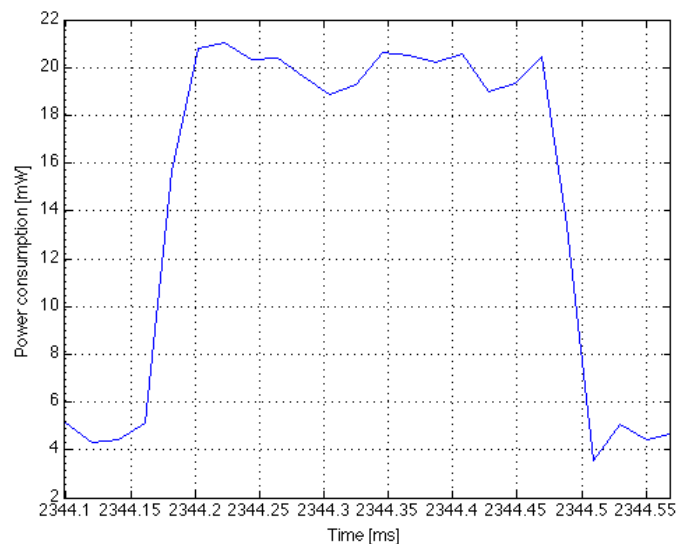


Figure 4.9: Power profile: last byte of packet reception and send data to BOLT

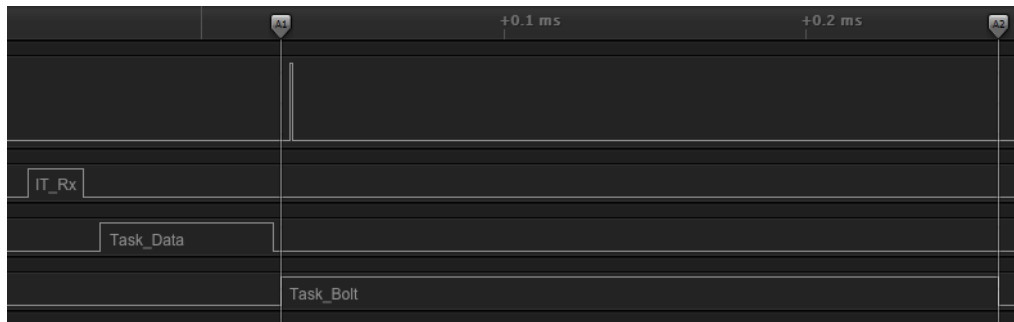


Figure 4.10: Trace: last byte of packet reception and send data to BOLT

#### Analysis of packet reception using DataFormat #2

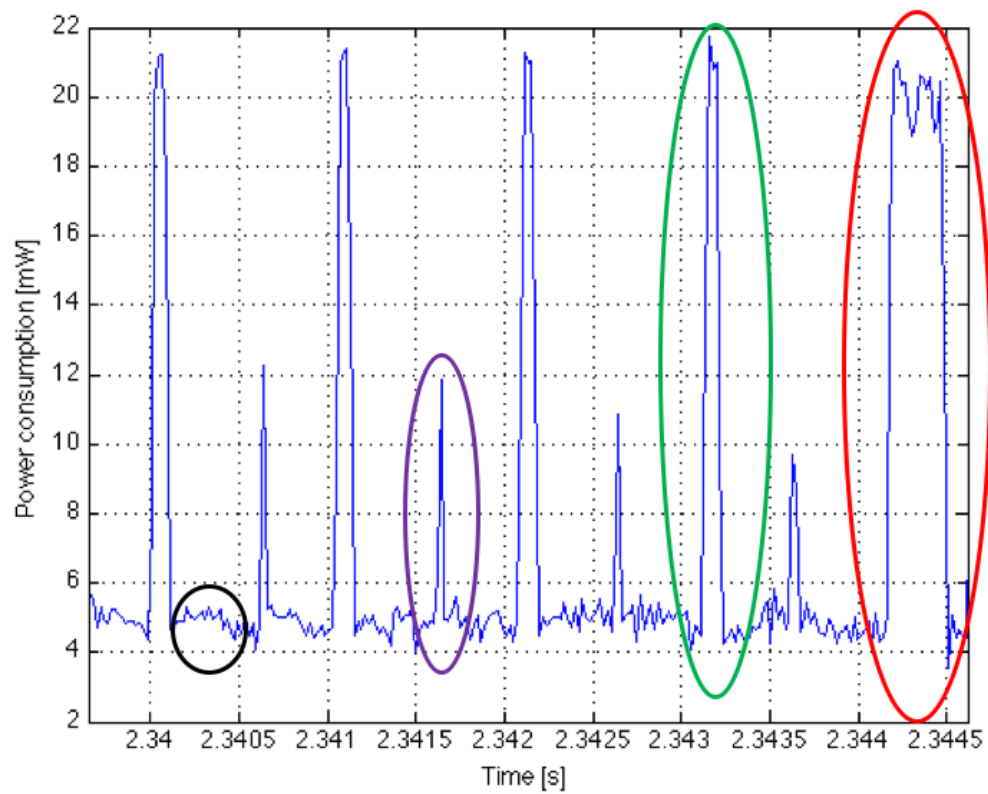


Figure 4.11: Power profile: last frame of packet using DataFormat #2

Figure 4.12 shows the last frame of packet using DataFormat #2. A frame consists of 5 bytes. The power profile is shown in Figure 4.11. The idle task execution (marked with black) has an average of 4.2mW power dissipation using LPM0. In the case LPM0 is not used and the microcontroller runs in Active Mode in idle task, the power dissipation raises to 16.5mW. The system tick interrupt execution has an average of 10.5mW power dissipation (marked with purple). The reception of a byte (marked with green) triggers the UART ISR and the Data task for execution. The reception of the last byte (marked with red) triggers the UART ISR, the Data task and the BOLT task for execution. During task execution, the average power dissipation is 21mW.

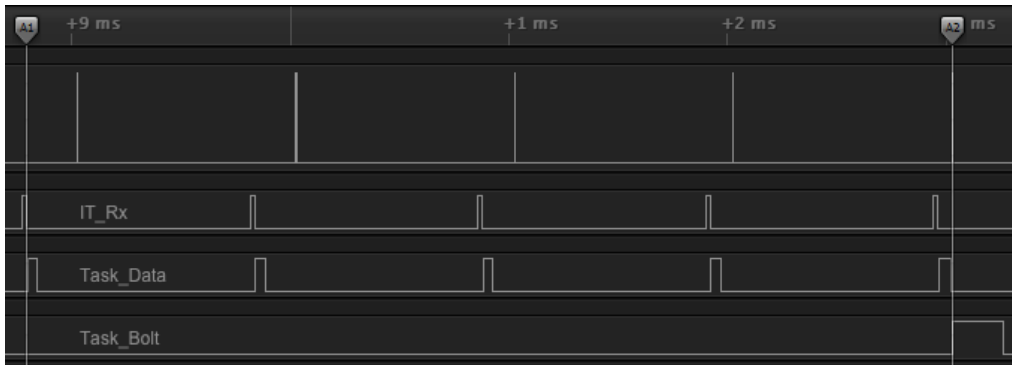


Figure 4.12: Trace: last frame of packet using DataFormat #2

The reception of the last three frames of a packet using DataFormat #2 is shown in Figure 4.13. The power profile of a full packet that comprises 125 bytes is shown in Figure 4.14. End of packet is easily noticeable because after reception, BOLT task is triggered, resulting in slightly wider spike.

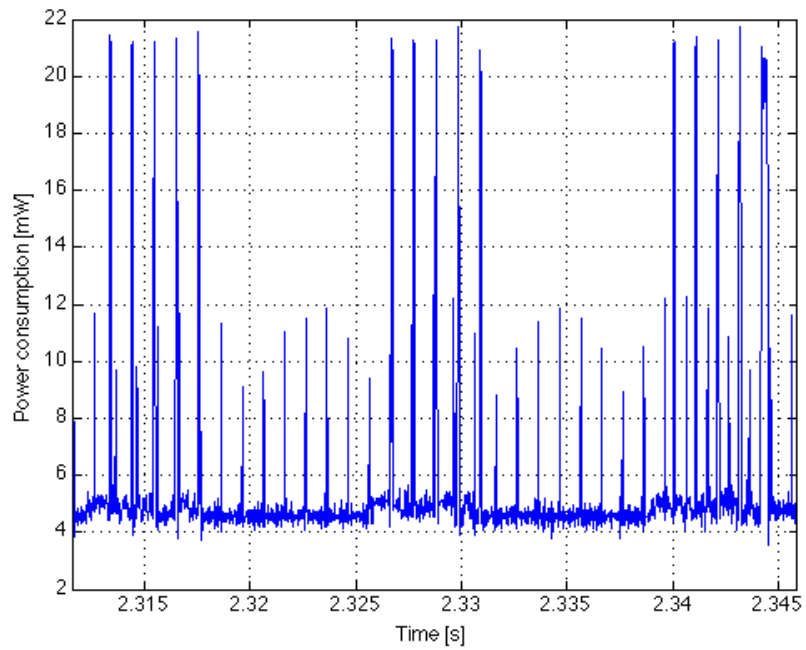


Figure 4.13: Power profile of last three frames of a packet using DataFormat #2

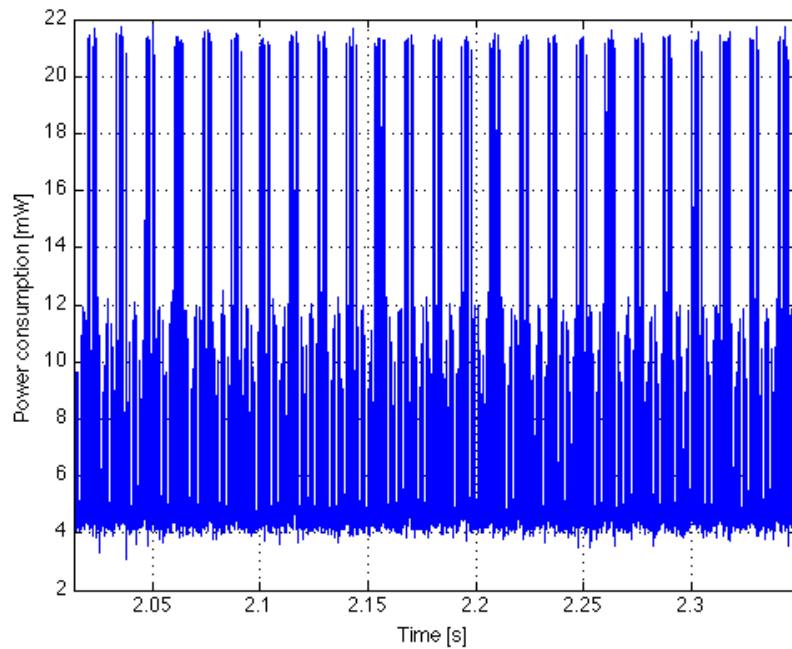


Figure 4.14: Power profile of a packet using DataFormat #2

### Analysis of packet reception using DataFormat #1

A packet consists of only 3 bytes using DataFormat #1. The bytes follow each other with a period of  $1ms$  and the frequency of packets is 1 per second. As seen at DataFormat #2, BOLT task is triggered after receiving the last byte of a packet. The execution of BOLT task after reception manifests in a slightly wider pulse on power profile.

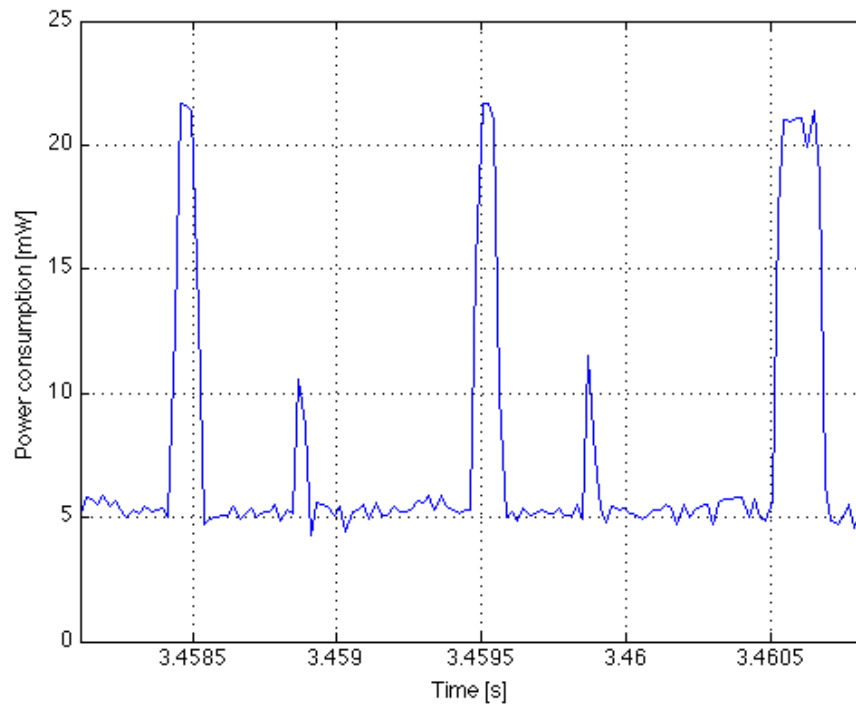


Figure 4.15: Power profile of a packet using DataFormat#1

### 4.2.2 Analysis of Operating Modes

In this section the different scenarios are evaluated and summarized based on their power dissipation. Table 4.11 shows the comparison of energy consumption for each scenario.

Figure 4.17 shows the first scenario (see Table 4.10). This is a basic operating mode using DataFormat #2: the sensor is switched on for 10 seconds and during this time the data received from sensor is transmitted to PC via BOLT without

flashing the LED after each execution. Figure 4.16 shows the power profile of this mode. The small spikes at the top represent the writing operation to BOLT.

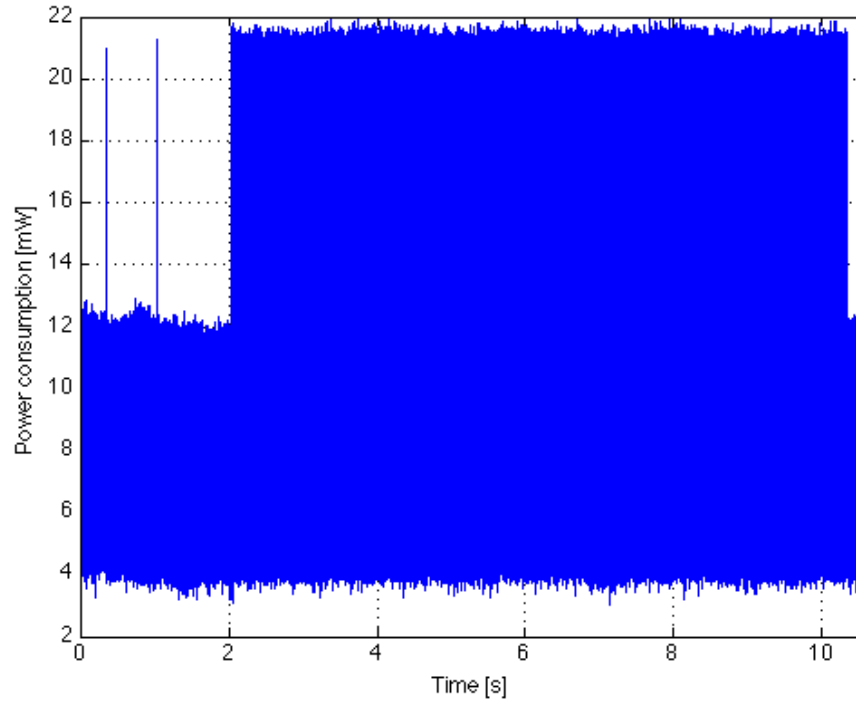


Figure 4.16: Power profile: basic operation using DataFormat#2

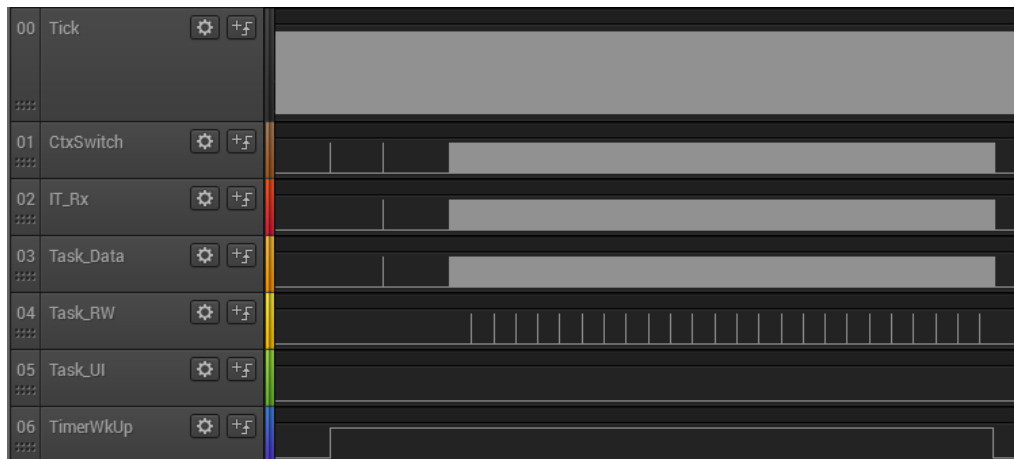


Figure 4.17: Trace: basic operation using DataFormat#2

The only difference in the second scenario is that after every successful writing operation to BOLT, a LED is flashed. This results in noticeable increase in power dissipation which can be seen on the power profile (see Figure 4.18).

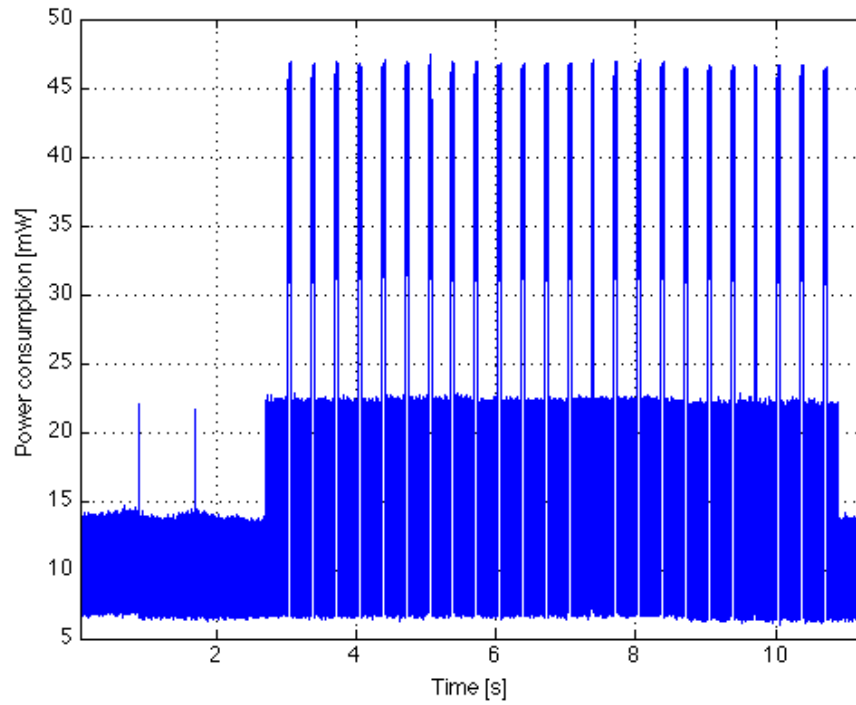


Figure 4.18: Power profile: basic operation with flashing LED after each writing operation to BOLT

For the time the LED is on, the baseline of dissipation clearly increases to 32mW as shown in Figure 4.19. The width of this pulse is 250ms which concurs the duration of LED is switched on. During this interval the consumption of interrupts and tasks remain the same, however the overall consumption increases by the power demand of the LED.

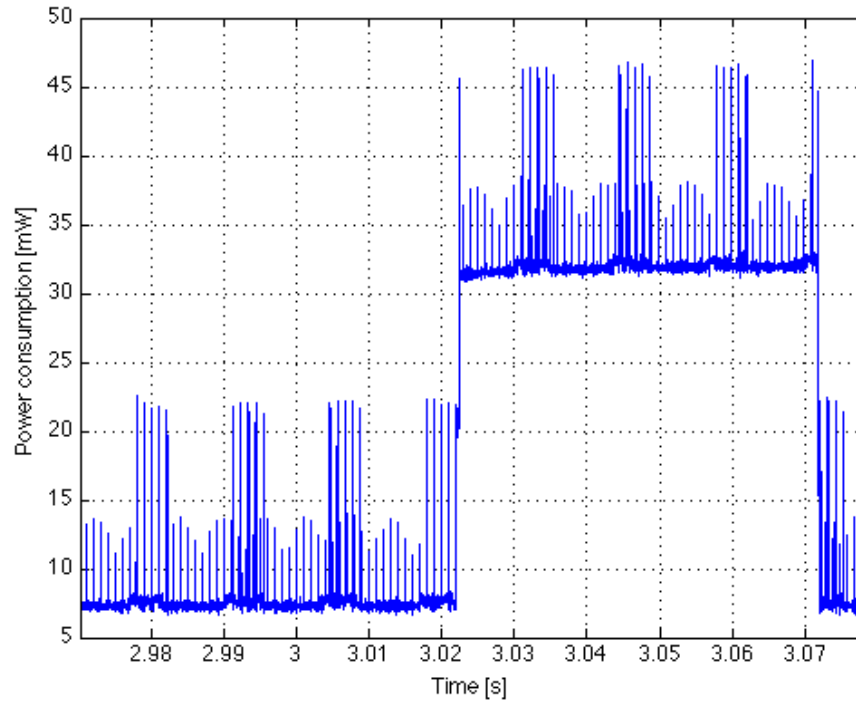


Figure 4.19: Power dissipation while LED is on

In Figure 4.20 the power profile of the basic scenario is shown using DataFormat #1. The measurement lasts for 12 seconds while the sensor is turned on for 10 seconds. Data is forwarded to BOLT once per second without blinking the LED, and during this period no reading operation is performed. The timing diagram of this scenario is shown in Figure 4.21. The average drain is 1.6027mA which is 4% lower compared to the same scenario when DataFormat #2 was used. This implies that if the requirements can be fulfilled with using DataFormat #1, it should be used to achieve lower power dissipation.



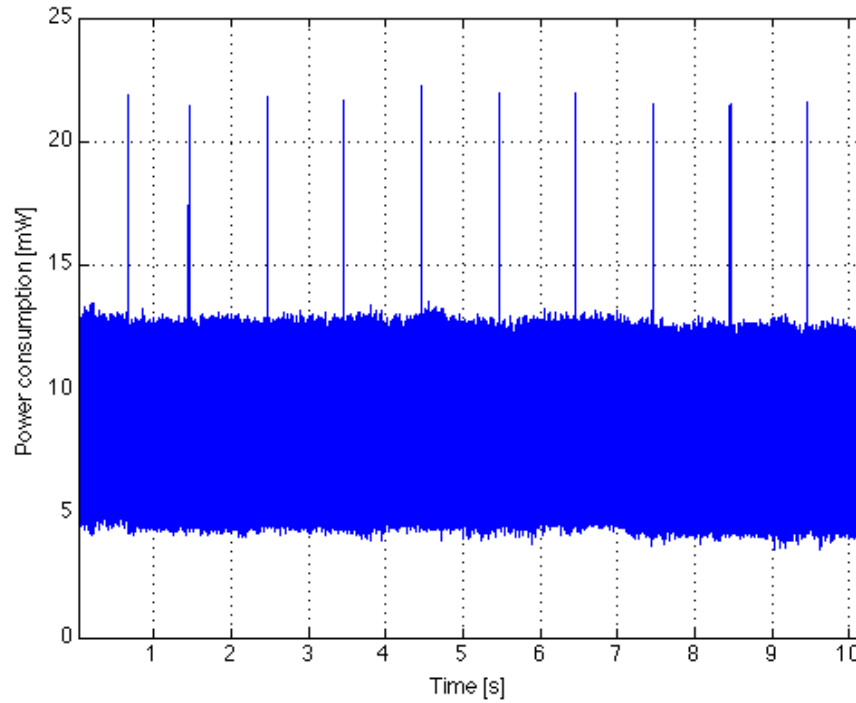


Figure 4.20: Power profile: basic operation using DataFormat#1

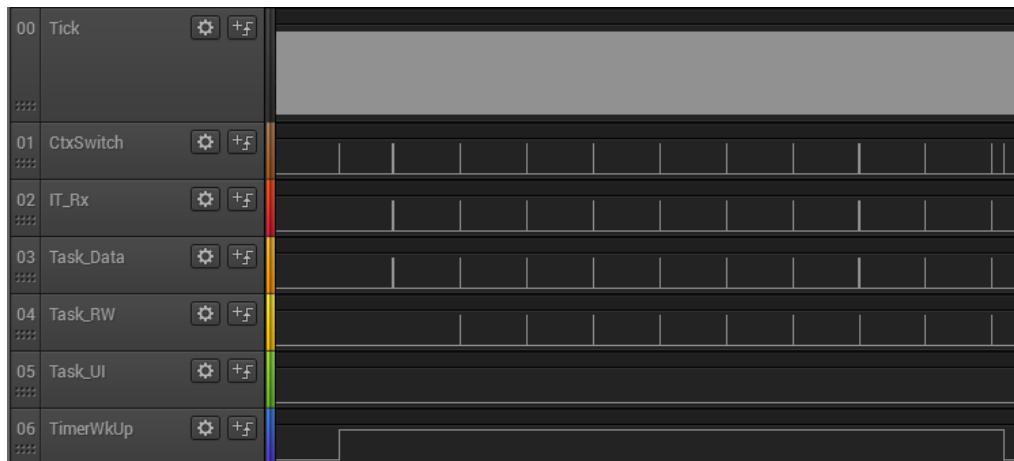


Figure 4.21: Trace: basic operation using DataFormat#1

Complex scenarios involve reading operations from BOLT. Scenario 3 (see Table 4.10) starts with using DataFormat #1, then a command is received from PC via BOLT to change the output data format. After reading out the message,

the MCU dynamically switches the sensor to DataFormat #2 and continues processing data received from sensor.

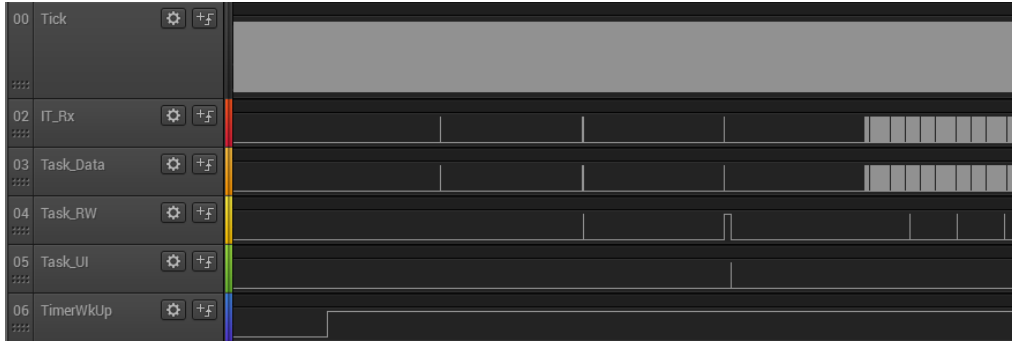


Figure 4.22: Dynamically switch between sensor output data formats

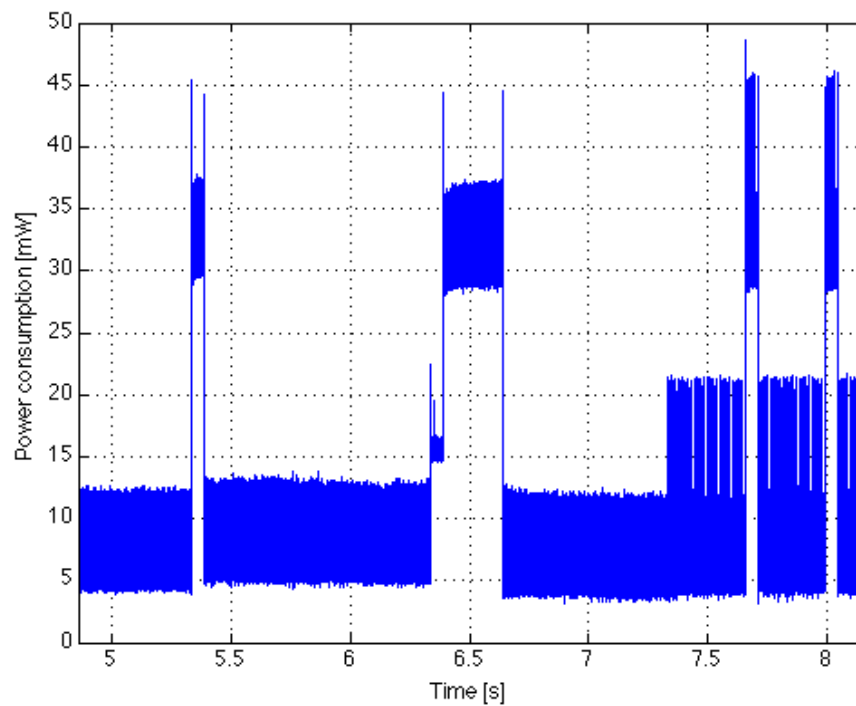


Figure 4.23: Overview of a reading operation

As seen in Figure 4.22, the execution of BOLT task (labelled as *Task\_RW*) is significantly longer when reading message from BOLT. The power profile of reading operation is shown in Figure 4.23 which is magnified in Figure 4.24.

Reading a message from BOLT that falls in the order of magnitude of 10 bytes takes 53ms on average, although it heavily depends on the size of the message. Currently all messages that are interpreted as commands are between 10 and 15 bytes. After processing a valid command during a reading operation, a LED is flashed for 250ms. In Figure 4.24 the first pulse which has 15mW power dissipation represents the BOLT task that performs a reading operation. After the reading operation a LED is switched on, therefore the consumption doesn't fall back to the consumption of idle mode. When the LED is on, the energy consumption raises to an average of 30mW. The pulse of BOLT task is approximately 50ms wide, furthermore after 250ms the energy consumption goes back to the level of consumption during idle mode, which fully supports our timing analysis.

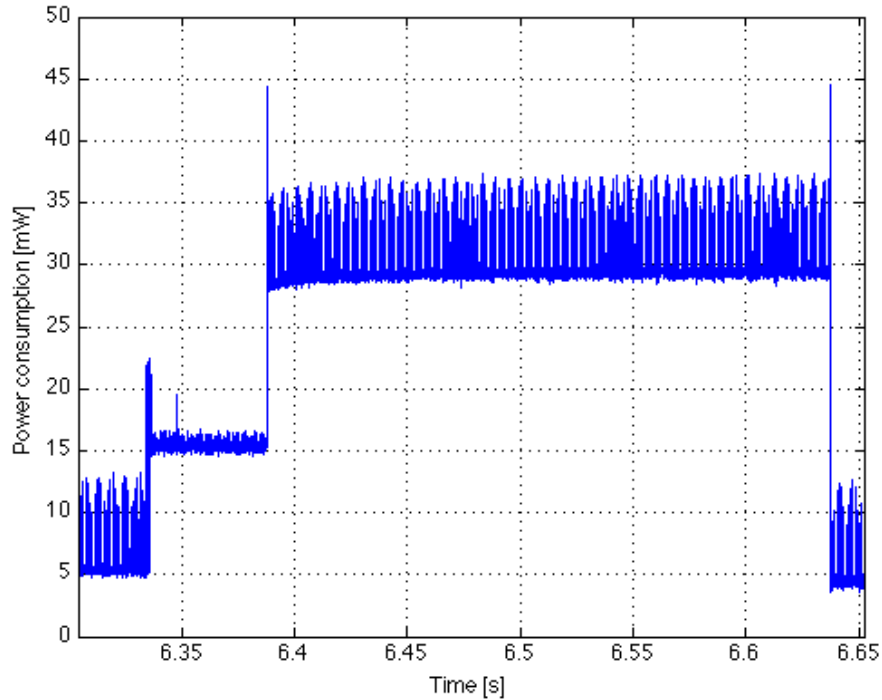


Figure 4.24: Power profile of a reading operation

### 4.2.3 Analysis of Duty Cycling

During scenario 5 to 8 (see Table 4.10), the effect of duty cycling is analyzed in terms of power dissipation. Two major scenarios are tested using both output

data formats. First the periodic reading mode (see Section 3.2.3) is switched off, then influence of 1s periodic reading is investigated. When the periodic reading mode is switched off, the command sent from the communication processor to the MCU via BOLT has to wait until a writing operation occurs. In case the ratio of sensor on duration and sensor off duration is low, the response within a bounded time frame is not guaranteed, which makes the system unpredictable.

Figure 4.25 shows the power profile of operation with sensor duty cycling of 10s on and 15s off using DataFormat #2 without periodic reading mode. The average current drain is  $1.3367mA$ .

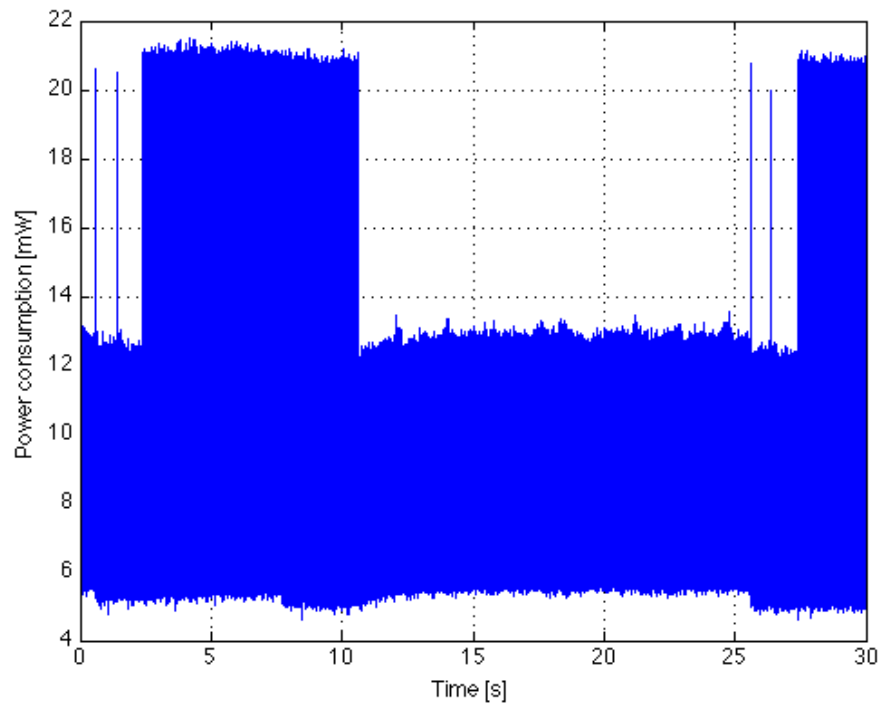


Figure 4.25: Power profile: duty cycling without periodic reading using DF#2



Figure 4.26: Trace: duty cycling without periodic reading using DF#2

The result of the same case with periodic reading mode on is shown Figure 4.27. It is clearly visible that the BOLT task wakes up every second to check BOLT whether a reading operation needs to be performed. The average current drain is  $1.3861\text{mA}$ , which is only  $50\mu\text{A}$  more than operating without periodic reading mode.

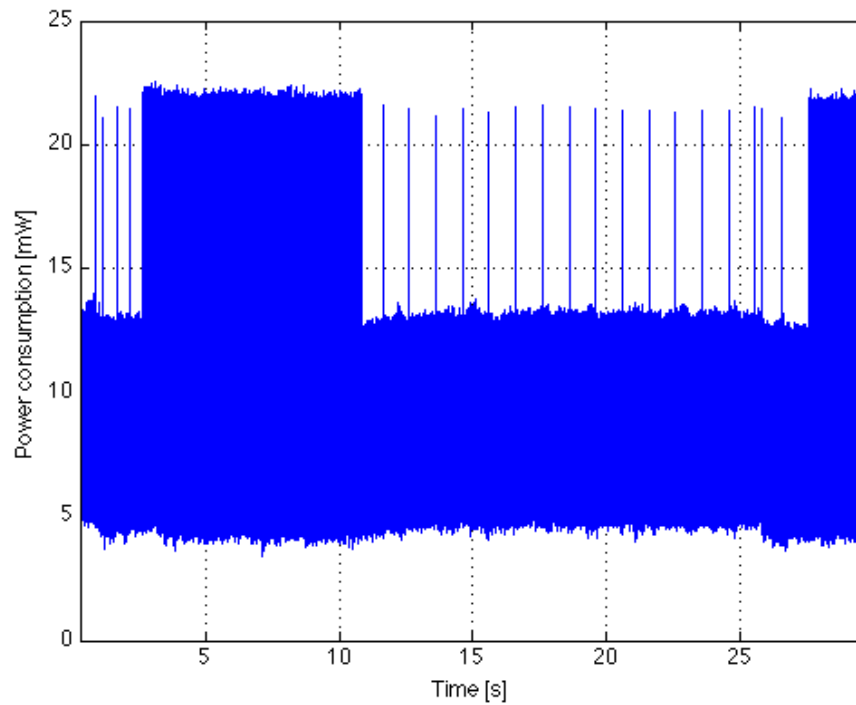


Figure 4.27: Power profile: duty cycling with periodic reading using DF#2



Figure 4.28: Trace: duty cycling with periodic reading using DF#2

The above mentioned measurements were also performed with using DataFormat #1. The conditions remained the same: sensor was switched on for 10s, then it was switched off for 15s. Analysis was performed for 30s.

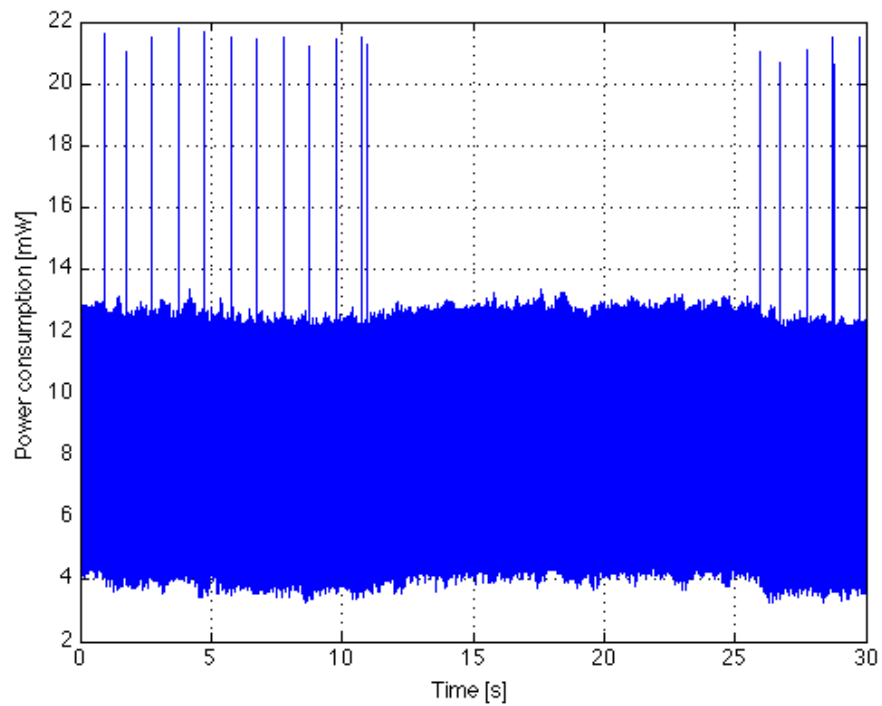


Figure 4.29: Power profile: duty cycling without periodic reading using DF#1

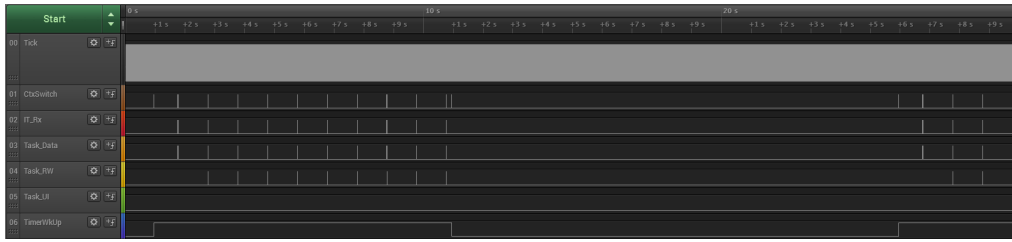


Figure 4.30: Trace: duty cycling without periodic reading using DF#1

Figure 4.29 shows the result of power analysis with using periodic reading mode. Since the frequency of receiving data from sensor is one per second, moreover the period of the periodic reading mode is configured to one second, it is difficult to distinguish whether the sensor is switched on or off when having an overview of the power profile. The only way to clearly differentiate is to zoom in and investigate the pulses.

The difference is shown in Figure 4.33. When sensor is switched on, in every second there are three pulses which follow each other with a period of  $1ms$ . In the case sensor is turned off, there is only one pulse in every second caused by the BOLT task which periodically wakes up and performs a check whether there is a message to be read from BOLT.

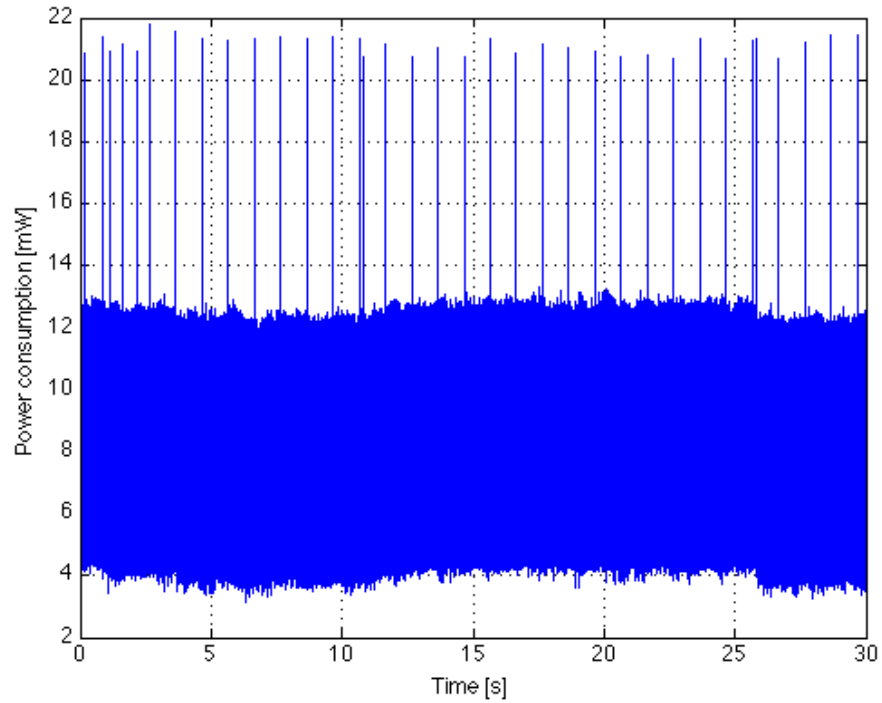
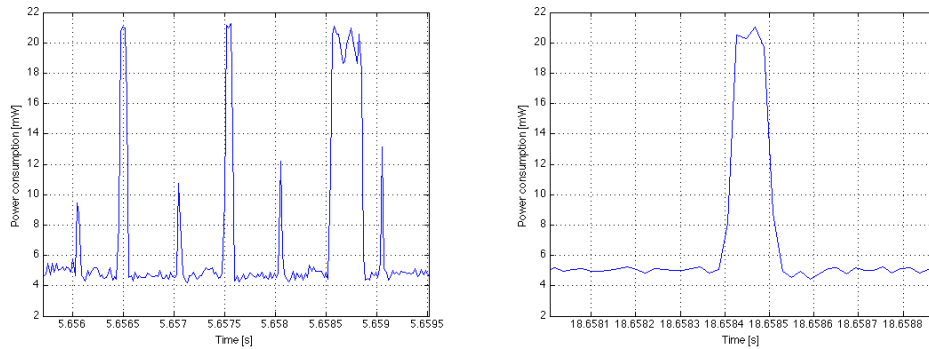


Figure 4.31: Power profile: duty cycling with periodic reading using DF#1



Figure 4.32: Trace: duty cycling with periodic reading using DF#1





(a) Packet reception and writing operation      (b) Periodic check when sensor is off

Figure 4.33: Difference between sensor on and off during duty cycling

#### 4.2.4 Idle Task Analysis

FreeRTOS switches in a special task called idle task when there is no task to run. This task is called every time there is no task in ready state, therefore the microcontroller can be put into sleep mode to reduce energy consumption. Evaluating different sleep modes in the idle task is described in Section 3.3.2.

After entering the idle task, the microcontroller is put into *LPM0* mode to save power. In this mode the high frequency clock sources and all peripherals are active while the CPU is inactive, therefore the SysTick timer can remain functional. The interrupt generated by the SysTick wakes up the system and puts the microcontroller back into active mode, therefore the RTOS can continue to operate.

Figure 4.34 shows the power profile of the system when there is no task to run. The spikes represent the execution of system tick interrupt. The system wakes up, performs a system tick and goes back to *LPM0* as there is no task to run. The power dissipation in *LPM0* is  $4.2mW$  on average, while the execution of the SysTick ISR dissipates  $10.5mW$  for  $5\mu s$ . In the case the microcontroller is in Active Mode instead of *LPM0* during idle task, the average power dissipation raises to  $16.5mW$ .

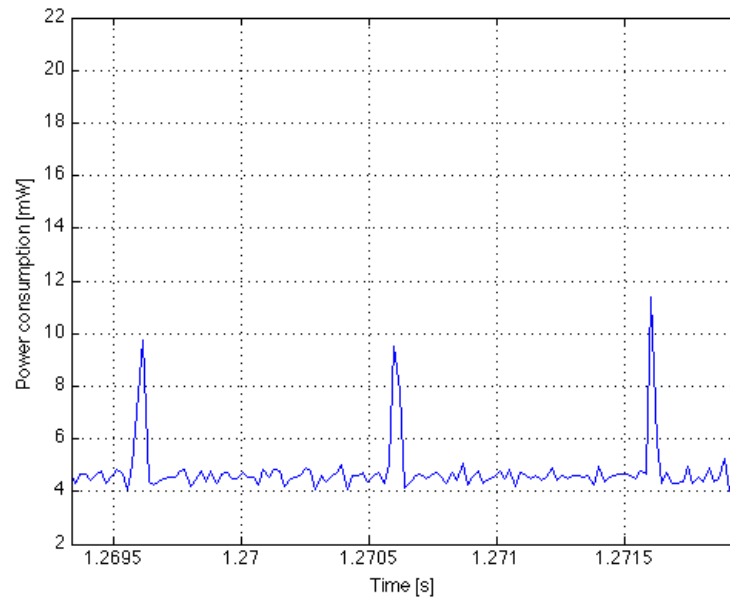


Figure 4.34: Power analysis of idle task

#### 4.2.5 Sensor Power Analysis

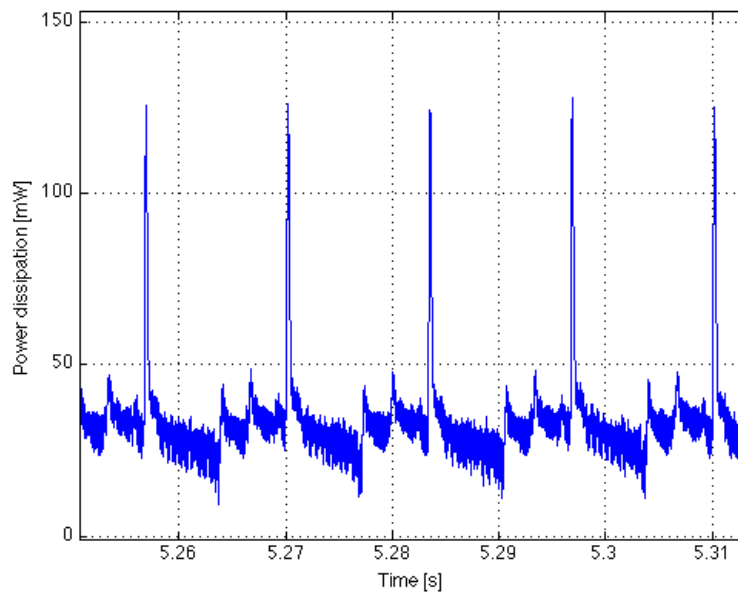


Figure 4.35: Power analysis of the pulse oximeter

When the sensor is on, it performs measurements periodically with a period of  $125\mu s$  and with a peak of  $40mA$  at  $3.0V$ . The average power dissipation is around  $30-33mW$ . Figure 4.35 shows the power profile of the pulse oximetry sensor.

#### 4.2.6 Conclusion of Power Analysis

The system dissipates  $4.2mW$  during idle by using LPM0 low power mode. Executing a system tick dissipates approximately  $10.5mW$ . Using communication interfaces (UART, SPI) in tasks requires  $21.5mW$  at most.

Table 4.11 illustrates the current drain measured in different scenarios. Using a user interface represented by LEDs shows significant increase in power dissipation. In scenario 3, where a reading operation was performed the average consumption increased, because the microcontroller needs to perform a reading operation, moreover it has to process the received commands. The difference between using DataFormat #2 and DataFormat #1 without duty cycling is  $65\mu A$ . By using 33% duty cycling without periodic reading mode, the difference between the data formats increased to  $208\mu A$ , which suggest using DataFormat#1 to save power in the long run.

Scenario <sup>1</sup>	Min	Avg	Max	Unit
1	0.9221	1.6676	6.6449	<i>mA</i>
2	0.8326	3.1041	14.3833	<i>mA</i>
3	0.9523	2.7297	14.7327	<i>mA</i>
4	0.9643	1.6027	6.7558	<i>mA</i>
5	0.8978	1.3367	6.5955	<i>mA</i>
6	0.7938	1.3861	6.6301	<i>mA</i>
7	0.8805	1.1280	6.6988	<i>mA</i>
8	0.8104	1.1521	6.6133	<i>mA</i>

Table 4.11: Summary of current drain of different operating modes

Comparing the scenarios where periodic reading mode is turned on to the scenarios where it is switched off, the results show that by using the periodic reading

<sup>1</sup>The different scenarios are summarized in Table 4.10.

mode only increases the average current drain by  $50\mu A$  and  $24\mu A$  using DataFormat #2 and DataFormat #1 respectively. This conveys that by turning on the periodic reading mode when the system is configured to use duty cycling, system predictability and bounded response times can be guaranteed with tiny additional energy consumption.

# Conclusion

---

In the previous chapters I described the system design in detail, then explained the operation of the system. I performed detailed timing and power analysis and finally presented the results of evaluation. Throughout system design, I was constantly focusing on the two main goals - timing predictability and low power system design. I set up bounds on response times, thus system predictability and responsibility is guaranteed within those frames. As discussed in the evaluation chapter, I had some difficulties with using ultra-low power modes in FreeRTOS, therefore currently in idle state the system uses the basic low power mode, which results in a few milliamperes of consumption on average. I am certain about this number can be significantly lowered as my attempts with LPM3 and LPM4 ultra-low power modes were successful without FreeRTOS. I also suggested a workaround for this problem, which utilize an external hardware timer to wake up the microcontroller when needed. Despite the difficulties I encountered, I can say that the prototype which I developed is a fully functional device, which fulfills the requirements that had been set for this semester.

## 5.1 Future Work

I truly enjoyed working on this project. During development, a few ideas came to my mind which I would gladly carry out as a project or even a master thesis. The next step would be to discard off-the-shelf components, breakout boards, and develop a specific hardware from scratch. Let me quote Alan Kay<sup>1</sup>, who said the followings at Creative Talk seminar in 1982: *“People who are really serious about software should make their own hardware.”* With designing the schematics and the printed circuit board, not only an ergonomic, market-ready device could

---

<sup>1</sup> Alan Curtis Kay  
[https://en.wikipedia.org/wiki/Alan\\_Kay](https://en.wikipedia.org/wiki/Alan_Kay)

be developed, but also optimization and efficiency could be leveraged in terms of timing predictability and ultra low power consumption.

I envision a compact device with its own hardware and PCB equipped with extremely minimal or even without a user interface. Therefore it would be as compact as possible and it could be portable and comfortably wearable. The device would maintain connection with smartphones or medical equipments in hospitals via Bluetooth Low Energy to save power, and the data would be fed to an application. Since people have smartphones in their pockets or within reach all day long, looking back old data, tracking measurements and overall health condition would take only a few taps.



Figure 5.1: Smartphone application with measurement data

Not to mention, smartphones are becoming the central control interface of our life: we want to control our houses, cars, smart devices - basically everything with our phones. Why wouldn't we track our health condition, receive notifications, warnings and possible threats to the only device which is around us 24/7?

# Bibliography

- [1] F. Sutton, M. Zimmerling, R. Da Forno, R. Lim, T. Gsell, G. Giannopoulou, F. Ferrari, J. Beutel, and L. Thiele, “Bolt: A Stateful Processor Interconnect”, in *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pp. 267–280, ACM, 2015.
- [2] “TI MSP432P401x Datasheet.”,  
<http://www.ti.com/lit/ds/symlink/msp432p401r.pdf>, 2015.
- [3] “Nonin OEM III Module Specification and Technical Information.”,  
<http://www.nonin.com/documents/OEM%20III%20Module%20Specifications.pdf>, 2007.
- [4] P. Mellgren, “Response Time Calculation, Priority Assignment and Holistic Scheduling with Integer Programming Methods”, Master’s thesis, Mälardalens University, Sweden, 2001.

APPENDIX A

# Connectivity and Pinout

MSP432P401R pin	Connection
P1.0	LaunchPad "LED1"
P1.2	LaunchPad USB VCP RX line
P1.3	LaunchPad USB VCP TX line
P1.5	BOLT SPI Clock
P1.6	BOLT SPI MOSI
P1.7	BOLT SPI MISO
P2.0	LaunchPad RGB LED - Red
P2.1	LaunchPad RGB LED - Green
P2.2	LaunchPad RGB LED - Blue
P2.3	BOLT REQ line
P2.4	BOLT MODE (R/W) line
P2.5	BOLT IND line
P2.6	BOLT ACK line
P2.7	BOLT IND_OUT line (not used)
P3.0	RTOS trace: timer wakeup
P3.5	RTOS trace: context switch
P3.7	RTOS trace: system tick
P5.1	RTOS trace: Sensor IT RX
P5.4	MUX channel 2: Sensor data format switch
P5.5	MUX channel 1: Sensor power switch
P5.6	RTOS trace: UI task
P6.4	RTOS trace: debug pin
P6.5	RTOS trace: debug pin
P6.6	RTOS trace: BOLT task
P6.7	RTOS trace: Data task

Table A.1: Pinout of the microcontroller



# Source Code Organization

---

The source code is written in embedded C and assembly. The organization of the project is highly structured to make it flexible, modular and portable:

- `driverlib` folder contains all peripheral libraries for the TI MSP432 microcontroller family
- `FreeRTOS` folder contains the source of the RTOS, including memory management and port specific codes
- `system` folder contains the startup files for the microcontroller
- project root contains all configuration and source files of the system

RTOS related configurations are stored in `FreeRTOSConfig.h`. `main.h` contains the system specific configurations. Low level and hardware specific functions are written in `hardware.c`; trace macros and functions are implemented in `trace.h` and `trace.c` respectively. All tasks are stored in separate `.c` files starting with `t_`, and `main.c` contains system initialization and RTOS callback functions.

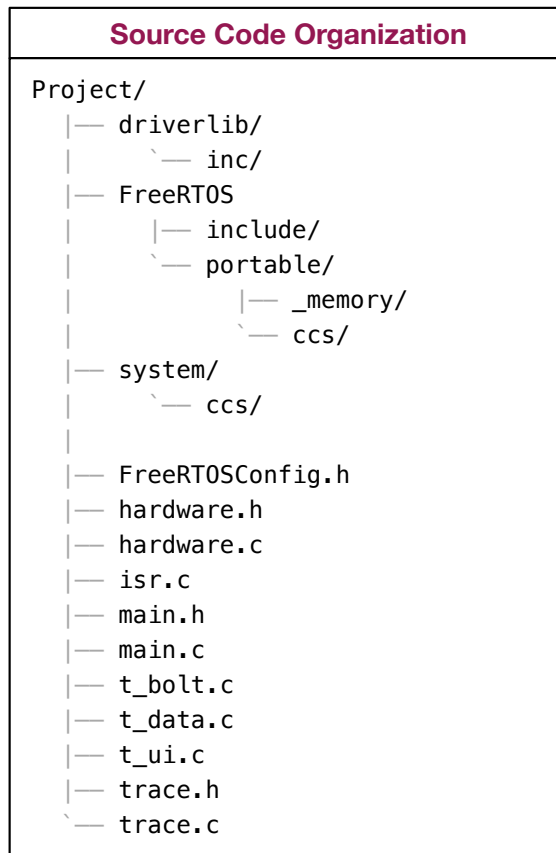


Figure B.1: Source code organization

## APPENDIX C

# Device Outputs

The device forwards the actual heart rate and blood oxygen saturation value to the PC via USB VCP. The output is captured with a terminal emulator software.

```
20 bytes received:
HR: 75 | SpO2: 98%

48 52 3a 20 37 35 20 7c 20 53 70 4f 32 3a 20 39 38 25 0d 00
20 bytes received:
HR: 75 | SpO2: 98%

48 52 3a 20 37 35 20 7c 20 53 70 4f 32 3a 20 39 38 25 0d 00
20 bytes received:
HR: 75 | SpO2: 98%

48 52 3a 20 37 35 20 7c 20 53 70 4f 32 3a 20 39 38 25 0d 00
20 bytes received:
HR: 75 | SpO2: 98%

48 52 3a 20 37 35 20 7c 20 53 70 4f 32 3a 20 39 38 25 0d 00
```

Figure C.1: Device output when rested

```
21 bytes received:
HR: 127 | SpO2: 98%

48 52 3a 20 31 32 37 20 7c 20 53 70 4f 32 3a 20 39 38 25 0d 00
21 bytes received:
HR: 127 | SpO2: 98%

48 52 3a 20 31 32 37 20 7c 20 53 70 4f 32 3a 20 39 38 25 0d 00
21 bytes received:
HR: 128 | SpO2: 98%

48 52 3a 20 31 32 38 20 7c 20 53 70 4f 32 3a 20 39 38 25 0d 00
21 bytes received:
HR: 128 | SpO2: 98%

48 52 3a 20 31 32 38 20 7c 20 53 70 4f 32 3a 20 39 38 25 0d 00
```

Figure C.2: Device output after performing 50 push-ups

## APPENDIX D

# Sensor Timings

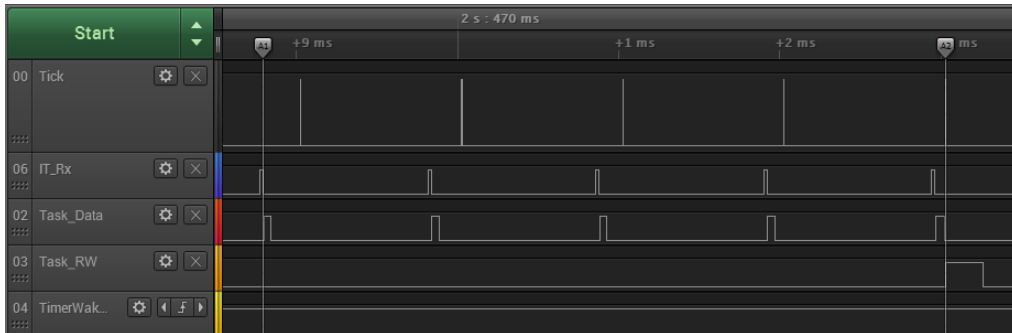


Figure D.1: Sensor DataFormat #2 frame

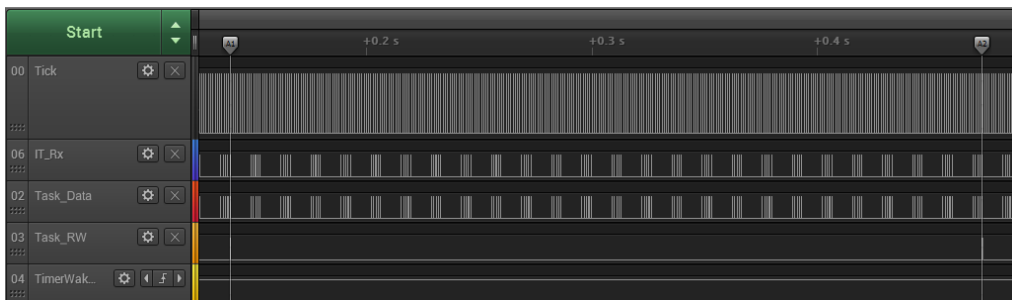


Figure D.2: Sensor DataFormat #2 packet

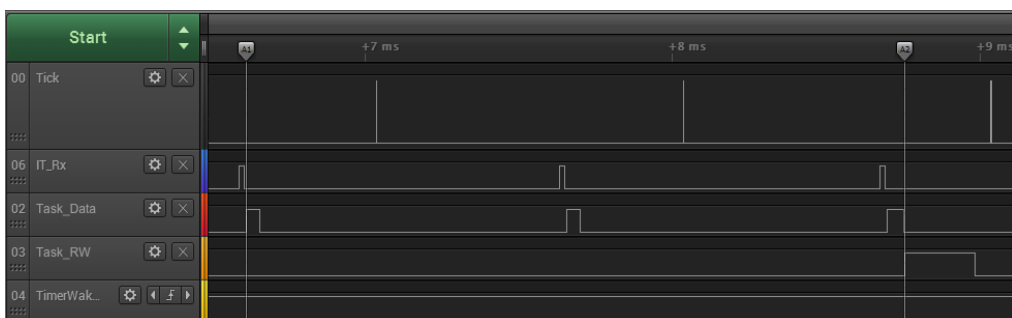


Figure D.3: Sensor DataFormat #1 packet

## APPENDIX E

# BOLT Timing Analysis

### E.1 Writing Operation

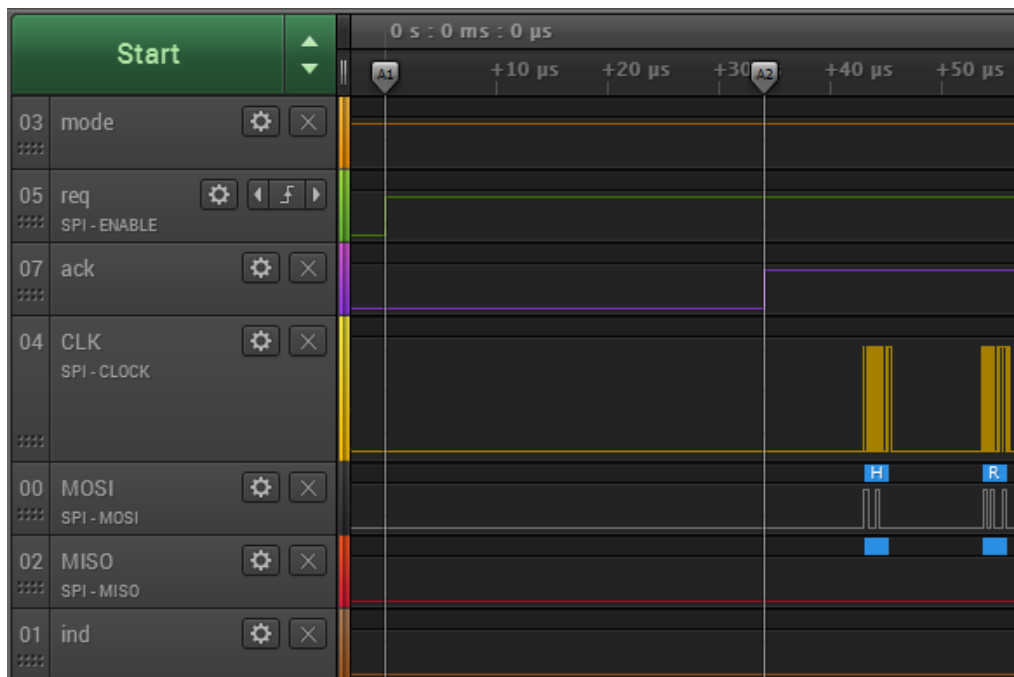


Figure E.1: Write to BOLT: Request to ACK response time

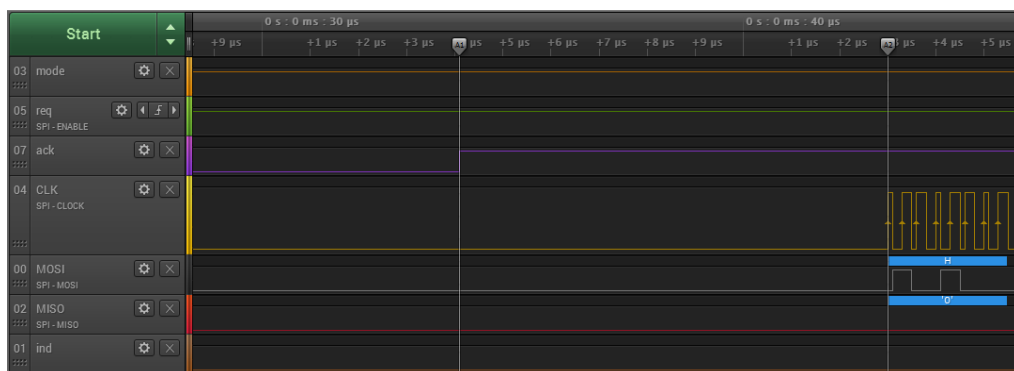


Figure E.2: Write to BOLT: ACK to first byte transfer

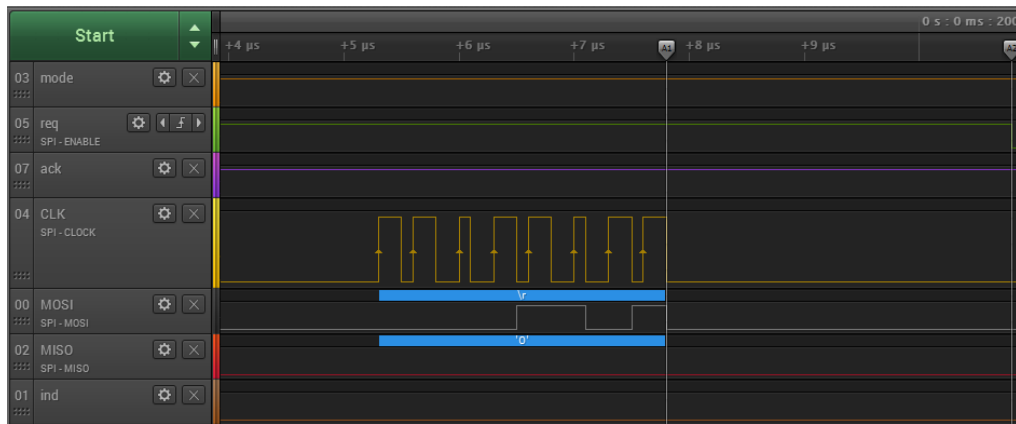


Figure E.3: Write to BOLT: Last byte to REQ line release

## E.2 Reading Operation

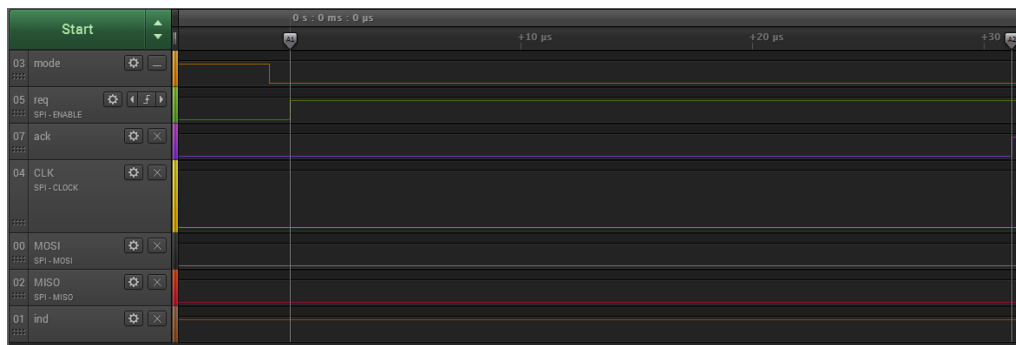


Figure E.4: Read from BOLT: Request to ACK response time

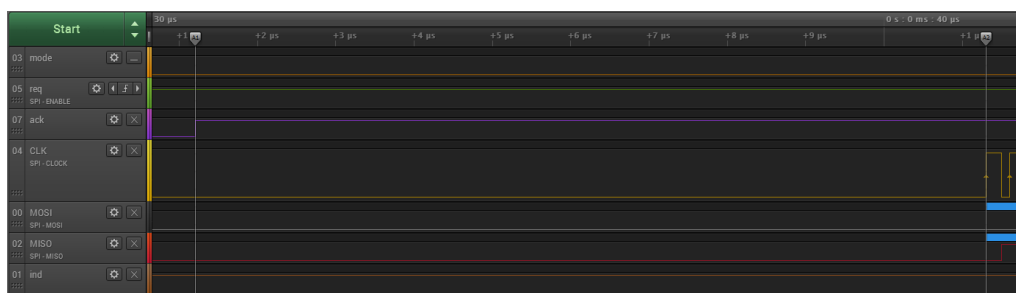


Figure E.5: Read from BOLT: ACK to first byte transfer

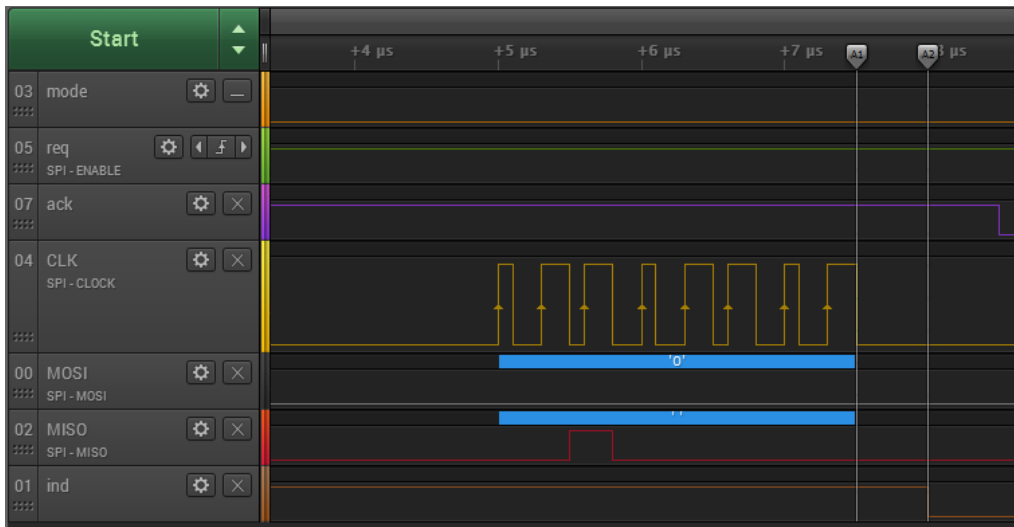


Figure E.6: Read from BOLT: Last byte to IND line release

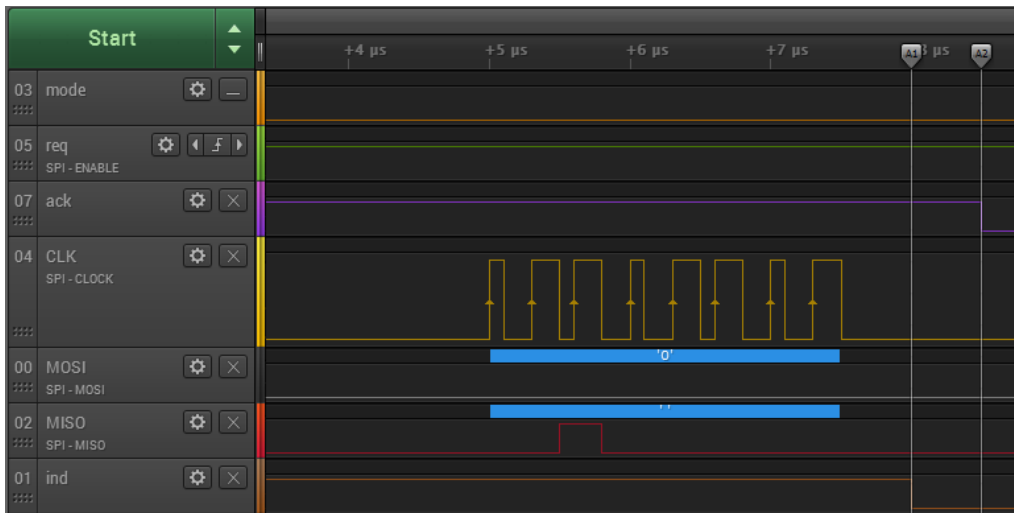


Figure E.7: Read from BOLT: IND to ACK line release

## APPENDIX F

# RTOS Task Timings

---



Figure F.1: Data task execution

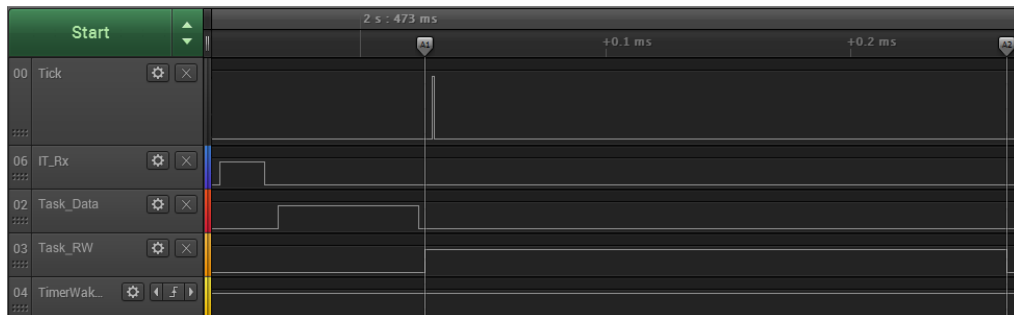


Figure F.2: BOLT task execution

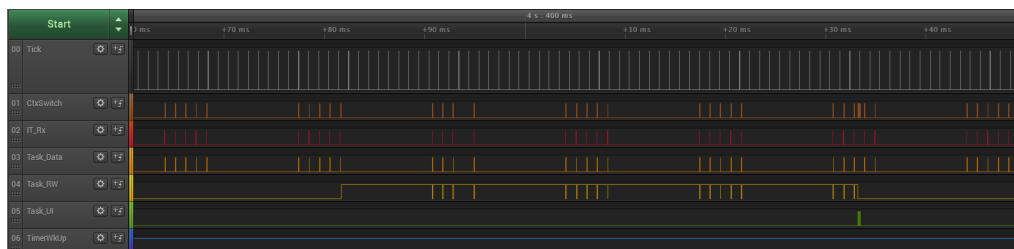


Figure F.3: BOLT reading operation using DataFormat #2



**ETH** zürich



# Towards Low-Power, Timing-Predictable Medical Monitoring

**Akos Pasztor**  
D-ITET, Semester Thesis  
Fall Semester 2015

**Supervisors:**  
Georgia Giannopoulou  
Felix Sutton  
Pengcheng Huang  
Prof. Dr. Lothar Thiele

Akos Pasztor | 20.01.2016 | 1

**ETH** zürich

Introduction   Design   Timing analysis   Power analysis   Summary

## Introduction

### Concept

- Medical Monitoring Device
- Monitoring vital signs
  - Heart rate (HR)
  - Blood oxygen saturation (SpO2)
- Decision model for critical situations

### Goals

- Timing guarantee and predictability
  - Safety-critical device
- Low-power design
  - Portable

Akos Pasztor | 20.01.2016 | 2

**ETH** zürich

Introduction
Design
Timing analysis
Power analysis
Summary

## Key Design Decisions

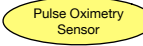
- **BOLT – stateful processor interconnect**
  - Developed at ETH TIK
  - Interconnect two platforms to provide deterministic communication
  - Ultra-low power consumption
  - Asynchronous message passing with FIFOs
  
- **Real Time Operating System**
  - Scalability, modularity
  - FreeRTOS – de-facto standard RTOS
    - Minimal overhead
    - Highly supported (by community, chip manufacturers)
    - Open source, Free for commercial use

Akos Pasztor | 20.01.2016 | 3


**ETH** zürich

Introduction
Design
Timing analysis
Power analysis
Summary

## System Design

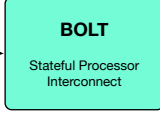


Pulse Oximetry Sensor




User Interface


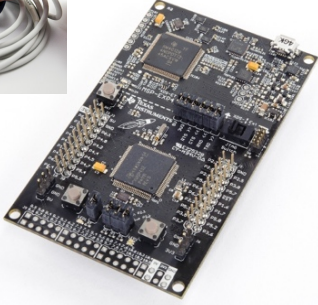
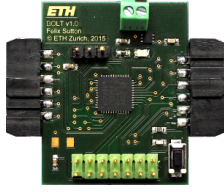
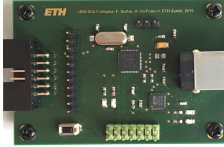
**Application Processor**  
TI MSP432  
32bit ARM Cortex-M4



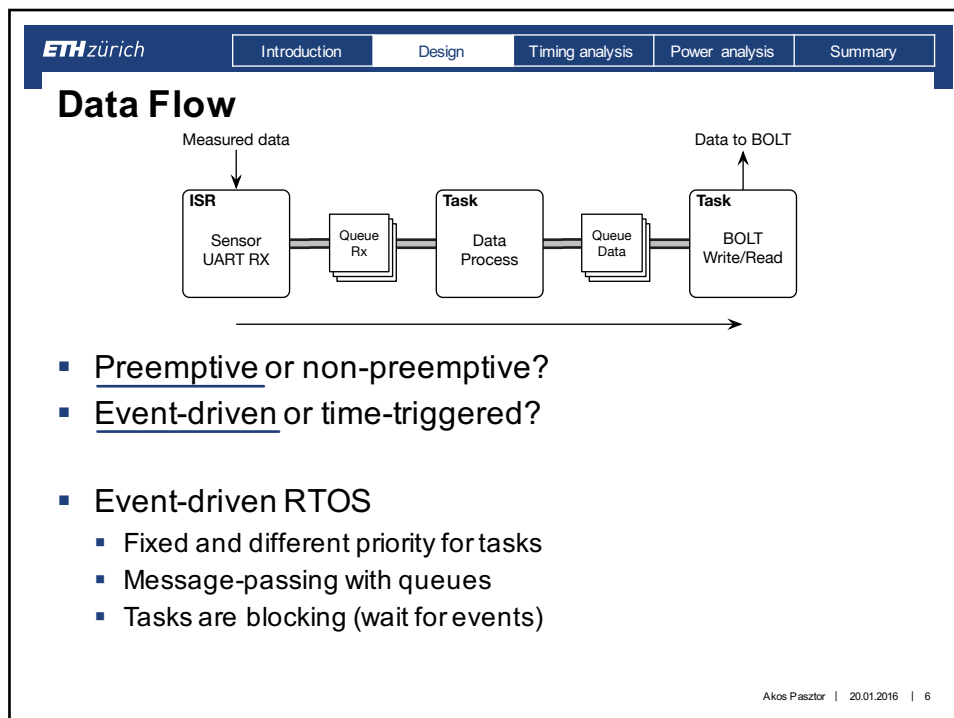
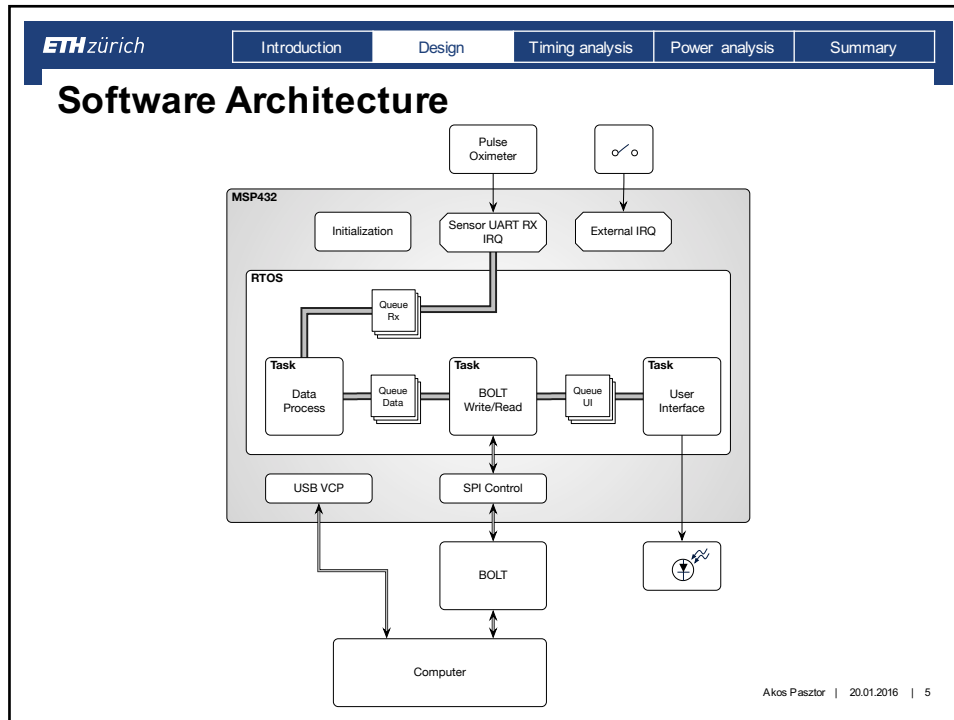
**BOLT**  
Stateful Processor Interconnect

**Communication Processor**  
e.g. PC via USB



Akos Pasztor | 20.01.2016 | 4



ETH zürich

Introduction Design Timing analysis Power analysis Summary

## RTOS Configuration

- Cortex-M core IT system vs. FreeRTOS
- Interrupt priorities → translate to Cortex-M core system, including special FreeRTOS interrupt priority values
- Task priorities → based on importance to avoid data collision in queues

The diagram shows a vertical stack of components. On the left, a vertical arrow labeled 'Priority' points upwards. The stack consists of six boxes: Scheduler IT (yellow), Sensor UART RX IT (yellow), External IT (grey), Data Task (green), BOLT Task (green), and UI Task (green). Brackets on the right group the top three boxes as 'Interrupts' and the bottom three as 'Application tasks'.

Akos Pasztor | 20.01.2016 | 7

ETH zürich

Introduction Design Timing analysis Power analysis Summary

## RTOS Configuration

- Memory handling
  - Only deterministic operations allowed
  - Allocation before system start, no memory-free or re-allocating operations during operation
  - Appropriate heap size for RTOS
  - Proper stack size for tasks

The diagram illustrates memory handling. On the left, a box labeled 'MSP432 RAM' contains three sub-boxes: RTOS Heap (yellow), MCU Heap (green), and MCU Stack (green). A red arrow points from the RTOS Heap in the MSP432 RAM to a detailed view of the RTOS Heap on the right. This detailed view shows three columns: Task 1, Task 2, and Queue 1. Below each column is a 'Stack' box.

- Thoughtful Software Design!

Akos Pasztor | 20.01.2016 | 8

**ETH** zürich

Introduction
Design
Timing analysis
Power analysis
Summary

## RTOS Configuration

```

18 /* Constants related to the behaviour of the scheduler. */
19 #define configUSE_PORT_OPTIMISED_TASK_SELECTION 1
20 #define configUSE_PREEMPTION 1
21 #define configUSE_TIME_SLICING 1
22 #define configMAX_PRIORITIES ( 5 )
23 #define configIDLE_SHOULD_YIELD 1
24 #define configUSE_16_BIT_TICKS 0 /* Only for 8 and 16-bit hardware. */
25
26 /* Constants that describe the hardware and memory usage. */
27 #define configCPU_CLOCK_HZ MAP_CS_getMCLK()
28 #define configMINIMAL_STACK_SIZE ( ( uint16_t ) 100 )
29 #define configMAX_TASK_NAME_LEN ( 12 )
30
31 /* Memory: heap configuration */
32 #define configTOTAL_HEAP_SIZE ( ( size_t ) ( 50 * 1024 ) )
33
34 /* Constants that define which hook (callback) functions should be used. */
35 #define configUSE_IDLE_HOOK 1
36 #define configUSE_TICK_HOOK 1
37 #define configUSE_MALLOC_FAILED_HOOK 1
38
39 /* Constants provided for debugging and optimisation assistance. */
40 #define configCHECK_FOR_STACK_OVERFLOW 2
41 #define configASSERT( x ) if ( x == 0 ) { taskDISABLE_INTERRUPTS(); for( ;; ); }
42 #define configUSE_REGISTRY_SIZE 0
43
44 /* Software timer definitions. */
45 #define configUSE_TIMERS 1
46 #define configTIMER_TASK_PRIORITY ( 3 )
47 #define configTIMER_QUEUE_LENGTH 5
48 #define configTIMER_TASK_STACK_DEPTH ( configMINIMAL_STACK_SIZE )
49
50 /* Cortex-M3/4 interrupt priority configuration ..... */
51 #ifndef _NVIC_PRIO_BITS
52 #define configPRIO_BITS _NVIC_PRIO_BITS
53 #else
54 #define configPRIO_BITS 3 /* 8 priority levels */
55 #endif
56
57 /* The lowest interrupt priority that can be used in a call to a "set priority"
58 function. */
59 #define configLIBRARY_LOWEST_INTERRUPT_PRIORITY 0x07
60
61 /* The highest interrupt priority that can be used by any interrupt service
62 routine that makes calls to interrupt safe FreeRTOS API functions. DO NOT CALL
63 INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY INTERRUPT THAT HAS A HIGHER
64 PRIORITY THAN THIS! (higher priorities are lower numeric values. */
65 #define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 3
66
67 /* Interrupt priorities used by the kernel port layer itself. These are generic
68 to all Cortex-M ports, and do not rely on any particular library functions. */
69 #define configKERNEL_INTERRUPT_PRIORITY ( configLIBRARY_LOWEST_INTERRUPT_PRIORITY << ( 8 - configPRIO_BITS ) )
70 /* !!!! configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero !!!!
71 See http://www.FreeRTOS.org/RTOS-Cortex-M3-M4.html. */
72 #define configMAX_SYSCALL_INTERRUPT_PRIORITY ( configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << ( 8 - configPRIO_BITS ) )
73
74 /* MY EXTENSION: macro for setting IT Priority relative to SYSCALL Priority.
75 * Can be used for setpriority function. */
76 #define configSET_IT_PRIO_SYSCALL(x) ( ( configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY + x ) << ( 8 - configPRIO_BITS ) )
77
78 /* Constants that build features in or out. */
79 #define configUSE_MUTEXES 1
80 #define configUSE_TIME_SLICING 0
81 #define configUSE_APPLICATION_TASK_TAG 1
82 #define configUSE_NRTOS_REENTRANT 0

```

- Many other challenges
- FreeRTOS: wide configurability
- Developer's task to implement a reliable and timing-predictable system

Akos Pasztor | 20.01.2016 | 9

**ETH** zürich

Introduction
Design
Timing analysis
Power analysis
Summary

## Timing Analysis Challenges

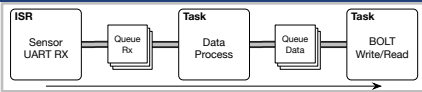
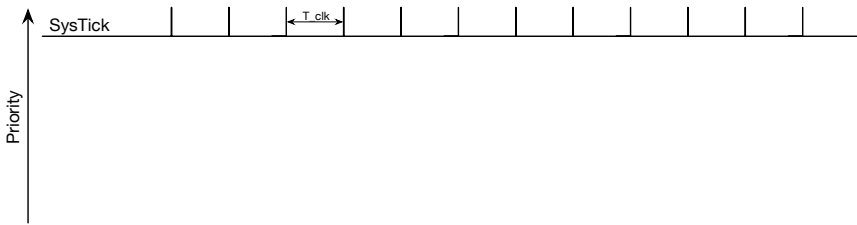
- Task activation depends on task priority and availability of data in task's incoming queues
- Context switch can occur
  - At system tick
  - After executing an interrupt service routine
  - After finishing a task
- Interrupts can also be preempted

Akos Pasztor | 20.01.2016 | 10

ETH zürich

Introduction Design **Timing analysis** Power analysis Summary

## Timing Analysis

Priority

System Tick

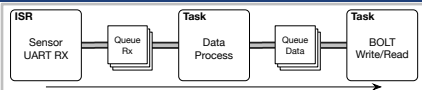
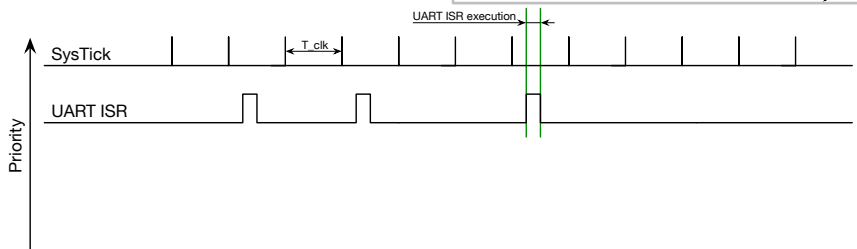
- Special interrupt generated every ms ( $T_{clk} = 1\text{ ms}$ )
- In case there is a ready task in task queue which has higher priority than the current one → Context Switch

Akos Pasztor | 20.01.2016 | 11

ETH zürich

Introduction Design **Timing analysis** Power analysis Summary

## Timing Analysis

Priority

UART RX Interrupt Service Routine

- Interrupts are processed regardless of system tick
- Forced context switch can occur at the end of ISR
- Interrupted only by SysTick

Akos Pasztor | 20.01.2016 | 12

ETH zürich | Introduction | Design | **Timing analysis** | Power analysis | Summary

## Timing Analysis

**Data task: highest priority task**

- Can be interrupted by SysTick, UART ISR
- Cannot be delayed by the period of tick
  - Triggered only by UART ISR
  - After ISR: forced context switch

Akos Pasztor | 20.01.2016 | 13

ETH zürich | Introduction | Design | **Timing analysis** | Power analysis | Summary

## Timing Analysis

**BOLT task**

- Can be interrupted by SysTick, UART ISR, Data task
- Can be delayed by the period of tick

Akos Pasztor | 20.01.2016 | 14

ETH zürich Introduction Design Timing analysis Power analysis Summary

## Timing Analysis

$$R_{bolt}^{(0)} = C_{bolt}$$

$$R_{bolt}^{(n+1)} = C_{bolt} + \left\lceil \frac{R_{bolt}^{(n)}}{T_{data}} \right\rceil (C_{data} + C_{sw}) + \left\lceil \frac{R_{bolt}^{(n)}}{T_{rx}} \right\rceil (C_{rx} + C_{sw}) + \left\lceil \frac{R_{bolt}^{(n)}}{T_{clk}} \right\rceil C_{clk} + T_{clk}$$

asztor | 20.01.2016 | 15

ETH zürich Introduction Design Timing analysis Power analysis Summary

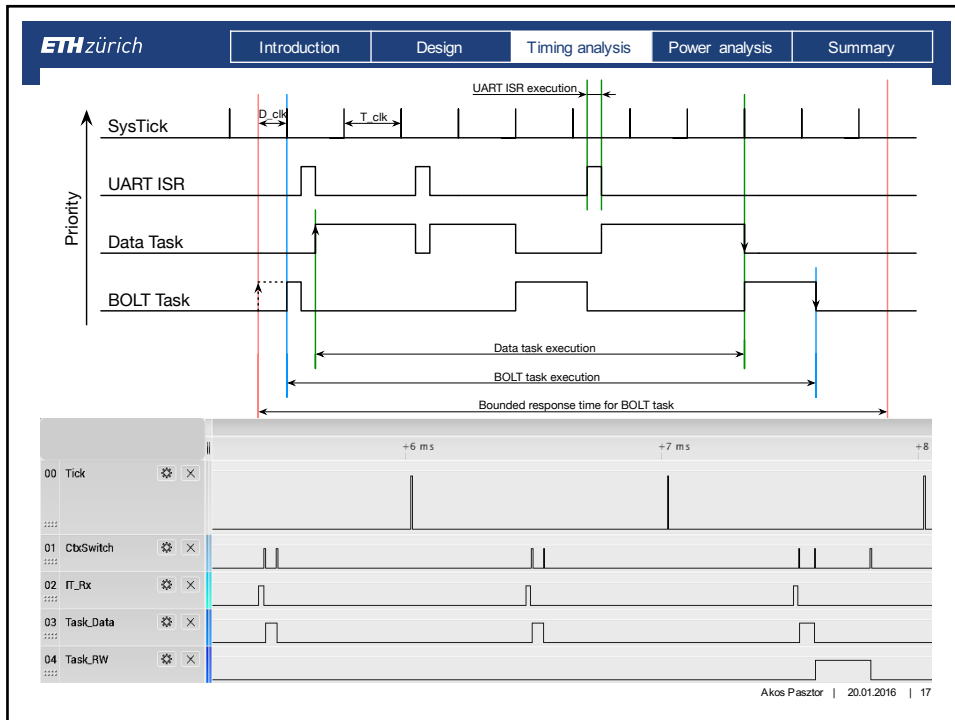
## Timing Analysis

$$R_{bolt}^{(0)} = C_{bolt}$$

$$R_{bolt}^{(n+1)} = C_{bolt} + \left\lceil \frac{R_{bolt}^{(n)}}{T_{data}} \right\rceil (C_{data} + C_{sw}) + \left\lceil \frac{R_{bolt}^{(n)}}{T_{rx}} \right\rceil (C_{rx} + C_{sw}) + \left\lceil \frac{R_{bolt}^{(n)}}{T_{clk}} \right\rceil C_{clk} + T_{clk}$$

Akos Pasztor | 20.01.2016 | 16



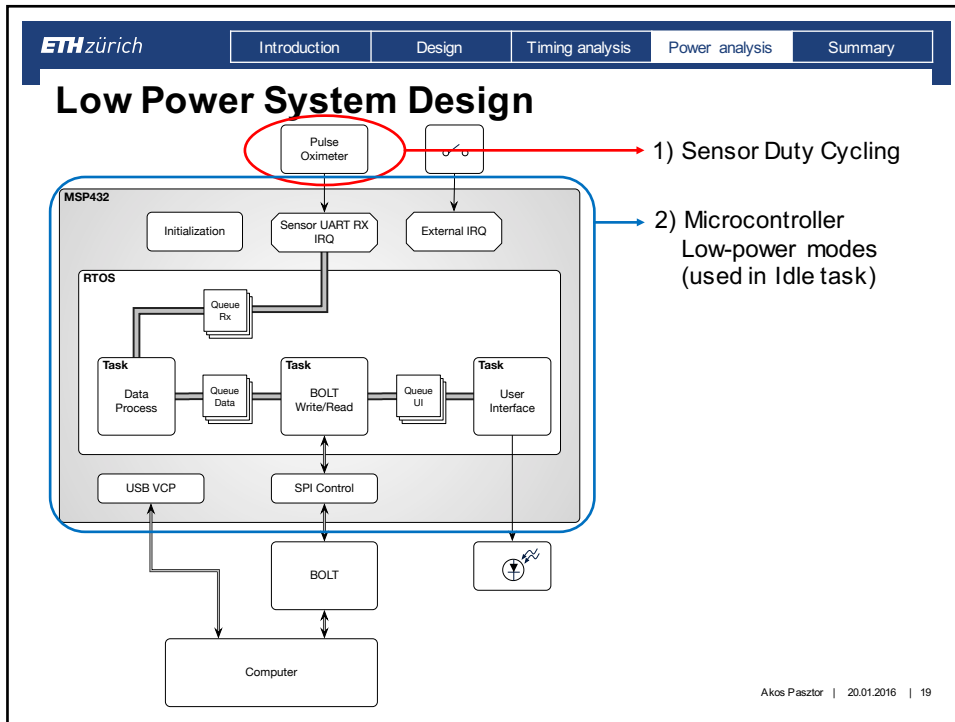


## Timing Analysis | Response times

Task	Theoretical bound	Measurements	Deviation
Sensor UART ISR	23.5 $\mu s$	18.3 $\mu s$	22.12 %
Data Task	80.5 $\mu s$	64.7 $\mu s$	25.20 %
BOLT Task	1.3166ms	0.3012 ms	77.12 %
Packet	333.3311 ms	324.4501 ms	2.66 %

- The analytic results bound safely the execution times of the tasks in practice
- Sensor data have enough time to be written to BOLT, therefore there is no data loss
- Result of timing analysis: system leverages deterministic computation and communication

ETH zürich | Introduction | Design | Timing analysis | Power analysis | Summary  
 Akos Pasztor | 20.01.2016 | 18



ETH zürich | Introduction | Design | Timing analysis | Power analysis | Summary

## Microcontroller Low Power Modes

Power Mode	Features
Active Mode	CPU is active and full peripheral functionality is available All low and high frequency clock sources can be active Flash memory and all enabled SRAM banks are active
LPM0	CPU is inactive but full peripheral functionality is available All low and high frequency clock sources can be active Flash memory and all enabled SRAM banks are active
LPM3	CPU is inactive and peripheral functionality is reduced Only RTC and WDT modules can be functional All other peripherals and retention enabled SRAM banks are kept under state retention power gating Flash memory is disabled, SRAM banks not configured for retention are disabled Only low frequency clock sources can be active
LPM4	Achieved by entering LPM3 with RTC and WDT disabled CPU is inactive with no peripheral functionality All peripherals and retention enabled SRAM banks are kept under state retention power gating Flash memory is disabled, SRAM banks not configured for retention are disabled All low and high frequency clock sources are disabled
LPM3.5	Only RTC and WDT modules can be functional CPU and all other peripherals are powered down Only Banks of SRAM is under data retention, all other SRAM banks and flash memory are powered down Only low frequency clock sources can be active
LPM4.5	Core voltage is turned off CPU, flash memory, all SRAM banks and all peripherals are powered down All low and high frequency clock sources are powered down

Challenges: system wakeup

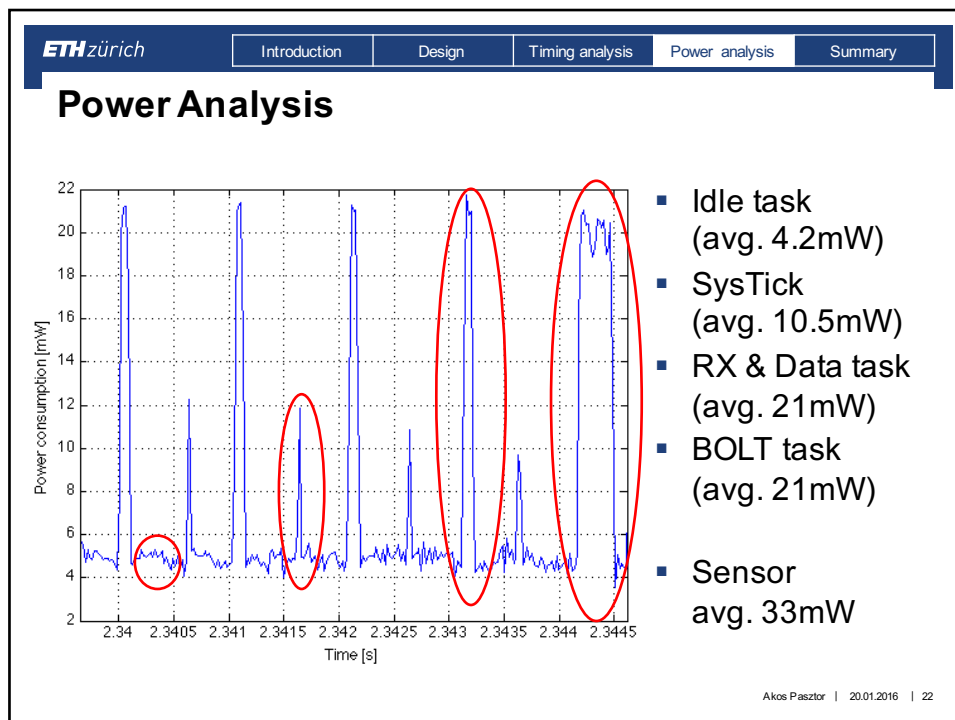
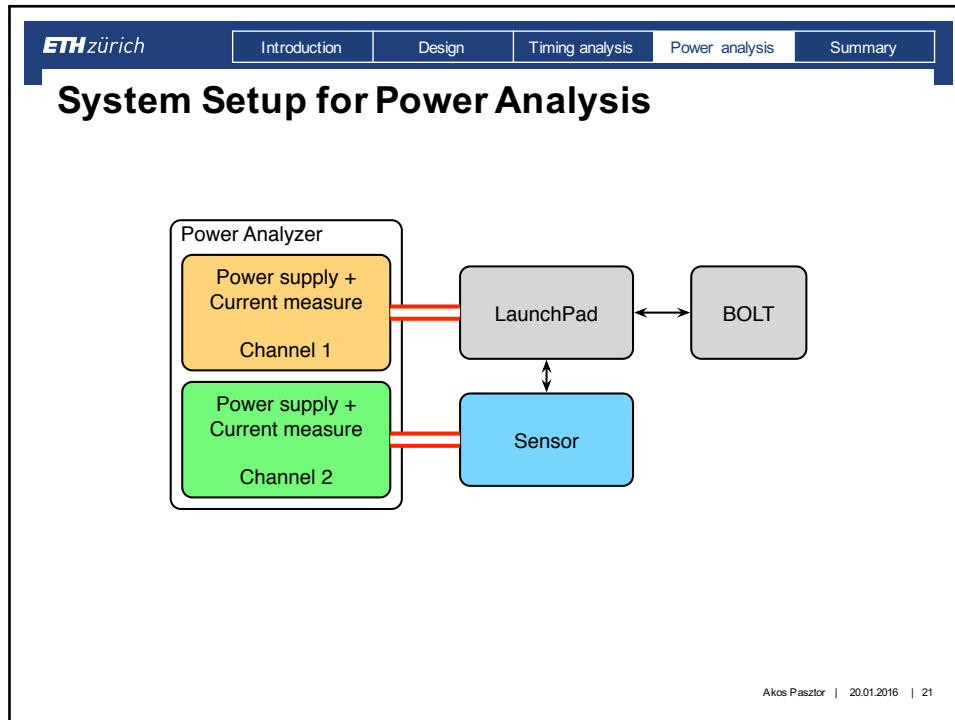
- RTC<sup>1</sup> → min. 1sec
- WDT<sup>2</sup> → unlikely high power consumption
- External IT → external hardware is required

Solution:

- LPM0

1 RTC: Real Time Clock  
2 WDT: WatchDog Timer

Akos Pasztor | 20.01.2016 | 20



<b>ETH zürich</b>	Introduction	Design	Timing analysis	Power analysis	Summary
-------------------	--------------	--------	-----------------	----------------	---------

## Future work

Suggestions for further reducing power consumption

- Design dedicated hardware
- Use new revision of microcontroller
- Use external hardware to trigger external interrupt to wake up from low power mode

Akos Pasztor | 20.01.2016 | 23

<b>ETH zürich</b>	Introduction	Design	Timing analysis	Power analysis	Summary
-------------------	--------------	--------	-----------------	----------------	---------

## Conclusion

- Working prototype of medical monitoring device
- Focused on timing predictability and low power system design
- Timing analysis:
  - Bounded response times, timing guarantee for task execution
  - Validation
- Power analysis:
  - Aimed for low power consumption
  - Duty cycling
  - Use of low power modes
  - Performed detailed measurements and power analysis of system

Akos Pasztor | 20.01.2016 | 24