DEGREE PROJECT IN ELECTRICAL ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

Timing delay characterization of GNU Radio based 802.15.4 network using LimeSDR

SAPTARSHI HAZRA

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Contents

List of Figures

List of Tables

Abbreviations

SDR Software Defined Radio

CSMA Carrier Sense Multiple Access

MAC Medium Access Control

TDMA Time Division Multiple Access

CPU Central Processing Unit

ACK Acknowledgement

Chapter 1

Introduction

Why 802.15.4?

Software Defined Radio (SDR) are flexible radio platforms where most of the communication systems functionality is designed in software. Typically, SDR platforms have on board radio front-end equipped with wide band antennas and analog signal processing chain for tuning the carrier frequency and desired bandwidth. High speed data converters convert the incoming analog signals into the digital domain and vice-versa. In traditional radios, the digital processing chain of a wireless protocol physical layer is implemented on the same chip as the radio front-end and analog signal processing functions. SDR, on the other hand, in *host-PHY([3])* architecture transfers the converted data to a general purpose computing platform using bus transfer (USB, PCIe). The digital processing chain is designed in software, thus allowing for flexibility in the protocol design, enabling experimentation in decoding and modulation techniques. SDR also allows for careful analysis of RF signals as the raw sample data is made available to the host.

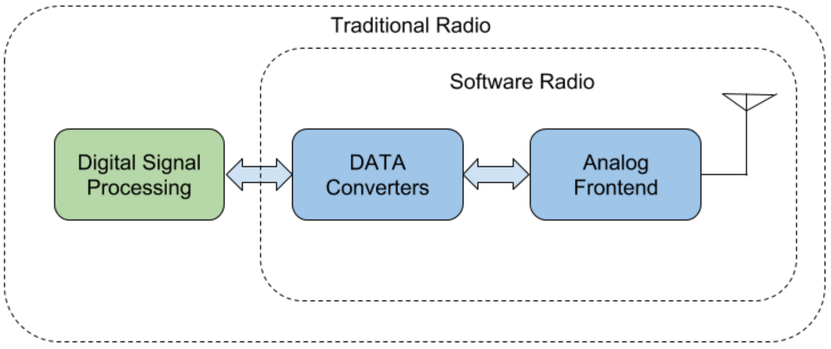


Figure 1.1: Software Radio and Traditional Radio Architecture.

Communication systems needs to have longer service time in military sector as compared to commercial sector. SDR helps to protect investments by facilitating change of protocols on already existing system. A major motivation within the commercial communications arena, is the rapid evolvement of communications standards, making software upgrades of base stations a more attractive solution than the costly replacement of base stations[4]. SDR also opens up the possibility of Cognitive Radios, a context sensitive radio system that can adapt depending on the radio channel conditions and applications.

A fundamental challenge of SDR system is computational horsepower, because it needs to process complex data waveforms in a reasonable time-frame. Since SDR involves transferring of signals and data from one system to another, this introduces considerable communication delays. Finally, general purpose processing systems introduces non-determinism in data processing and communication times.

Wireless devices share the wireless channel with other devices. Wireless protocol *Medium Access Control* (MAC) layer is responsible for moderating access to the wireless channel. It typically uses *Time Division Multiple Access* (TDMA) and *Carrier Sense Multiple Access* (CSMA) to allocate the use of the channel. TDMA protocols schedule the allocation of the entire channel to one of the devices for a particular time duration. This requires global time synchronization among the devices so that the devices can understand when to transmit and receive. CSMA, on the other hand uses the channel on an opportunistic basis, with the devices sensing if the channel is free or not. When it senses the channel to be free, it can start using it.

1.1 Problem Context

1.1.1 CSMA

As highlighted by [3], SDR based systems don't comply the stringent timing constraints imposed by modern MAC protocols. Furthermore, the presence of long bus communication and latency delays creates *blind spots*[3] in carrier sensing. In fig 1.2, a packet is being transmitted through the air medium which is received by the SDR system. There is a delay being end of transmission (t_0) on the air medium and complete reception of packet (t_1) by the *Central Processing Unit* (CPU) of the SDR system. This delay is caused by processing and communication delay in a SDR system. Once the packet has been received, the system wants to let the transmitting system about the successful reception using the *Acknowledgement* (ACK) packet. Using carrier sensing it detects the medium is free, but this information is delayed by $t_1 - t_0$, hence the system is blind towards the real time channel situation when making the decision to transmit and might lead to a collision.

Hence when designing MAC protocols, these delays needs to be taken into account, which necessitates a closer understanding of these delays and how different system

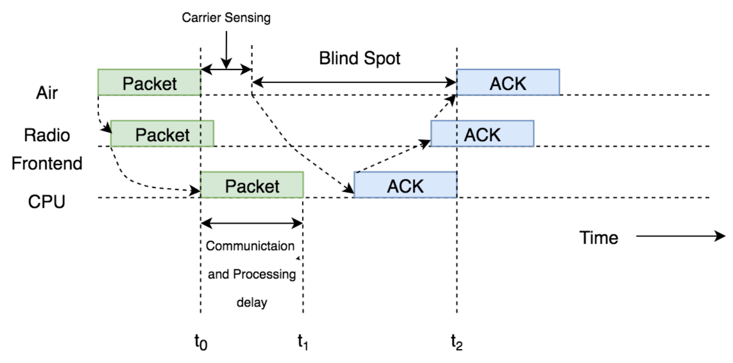


Figure 1.2: Blind Spots Illustration(adapted from [3]).

parameters affect these delays.

1.1.2 TDMA

TDMA based protocols are controlled by time slots, hence there is need for precise scheduling to ensure that the transmissions happen in the correct time-slot. The delays and imprecise scheduling can be tolerated by making longer time-slots but that degrades the efficiency of the overall network. Modern contention based protocols(CSMA) also require precise timing to implement inter-frame spacing.

Hence methods to implement precise time scheduling needs to studied.

1.2 Research Questions

- Quantitative evaluation of timing delays in SDR systems.

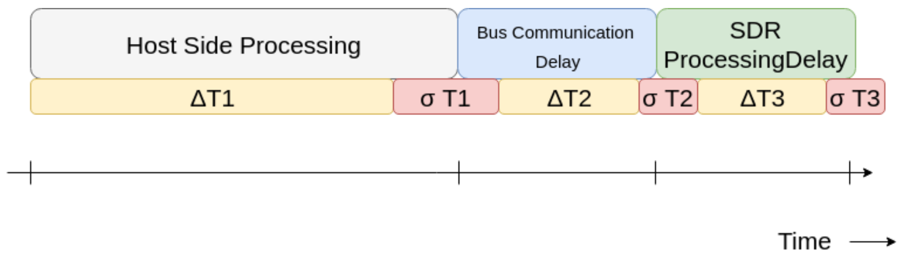


Figure 1.3: Quantitative evaluation of timing delays.

- Methods to implement precise scheduling in SDR systems.

1.2.1 Report Outline

The remainder of the report is structures as follows.*Chapter 2* introduces the previous work in this field, as well the needed background information on the SDR platform, the base system design and the relevant tools used in methods section. *Chapter 3* introduces the experimental setup and the methods used in the measurement of the timing delays. *Chapter 4* presents the experimental results, which are analyzed in *Chapter 5*. Finally, *Chapter 6* includes the concluding remarks and scope of future work.

Chapter 2

Background

This chapter presents the relevant background information that are useful in the understanding of the report. First, it introduces USBMon kernel utility and how it provides event updates. Second, it explains the LimeSDR software defined radio platform and specifically its USB communication architecture.

2.1 LimeSDR-USB

The LimeSDR-USB uses a USB 3.0 interface for communicating with the host computer. It supports MIMO operations with 2 RX and TX channels operating simultaneously. The maximum sampling rate supported by the ADC and DAC of LMS7002M is 160 Mhz, but the USB 3.0 restricts it to 61.44 MSPS when all the RX and TX channels are used simultaneously.

2.1.1 LimeSDR USB dataflow.

- *TX Data Path:* Stream data from GNURadio is passed through Soapy drivers to the LimeSDR Streaming API. The API unwraps the data and the control flags and pushes it to the TX-FIFO. It also does the necessary data representation translation depending on the data format of the streamed data. For example, in case of complex data, it changes 32 bit I and Q value representation of GNU Radio to 16 bit I and Q values representation. The values are pushed to the respective TX stream channel buffers(TXFIFO). The connection stream initializes the TX buffers and fills them with data from the TXFIFO. The Write function structures the data into FPGA data packet structure((Figure 2.3)) and combines multiple such packets into predefined batch size(initially 4). The buffer is processed by libusb to create bulk transfer packet and finally streamed to output data endpoint.

- *RX Data Path:* The USB data is continuously streamed from the LimeSDR to the Connection stream buffers through libusb. The Read function waits for data to be available from a usb context for the endpoint it is listening to, then it transfer

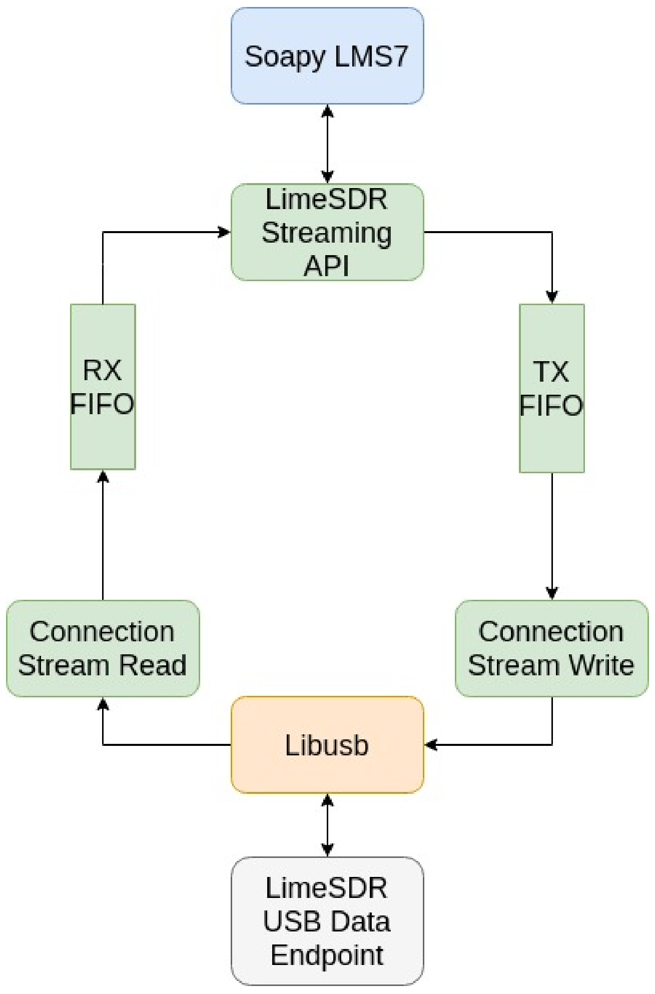


Figure 2.1: LimeSDR USB software architecture

data from the endpoint, parses the FPGA packets (Figure 2.3) to collect the data and pushes them to the RXFIFO. If the rx stream is configured to have particular receive time, it checks if that condition is satisfied. The LimeSDR streaming API collects the data from the RXFIFO and does the necessary data interpretation translation (reverse translation to the TX Data Path), finally streams the data to GNURadio.

2.1.2 LimeSDR USB packets and endpoints

LimeSDR uses four different endpoints for USB data transfer, these endpoints as Data Endpoint and Control Endpoint for both input and output directions. The control endpoints are used for configuring and retrieving data from the LMS7002M and NIOS Core on the FPGA. Data packets are used for the streaming data. It uses two different packet structures for the LMS7002M Control Packets and the Stream Data Packets. Depending on the control command, different number of bytes are packed into one data element and the maximum number of blocks in a single packet is defined. One

Endpoint No.	Function
0x01	Stream Data Output
0x81	Stream Data Input
0x0F	Control Data Output
0x8F	Control Data Input

Table 2.1: LimeSDR USB transfer endpoints

LMS64C protocol packet (Figure 2.2) is maximum 64 bytes, if the data to be sent is larger than that then the data field is segmented into several packets. The block count gives the number of data element in a single packet. The FPGA contains 4080 bytes of data along with 8 bytes of counter data that can be used for timestamp on the TX packets. The Lime driver uses synchronous bulk transfer for LMS Control packets and asynchronous bulk transfers for the FPGA packets.

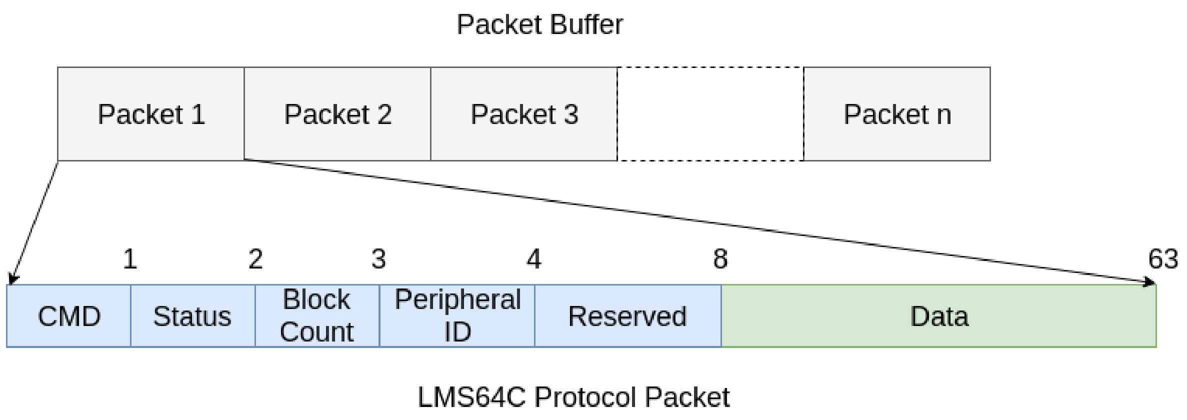


Figure 2.2: LMS Control Packet Structure

2.2 GNU Radio

2.3 USBMon

It is kernel facility provided to collect I/O traces on the USB Bus[7]. USBMon reports the requests made to and by the USB Host Controller Drivers(HCD). It provides two kinds of API's : binary and character. The binary API is accessed by character devices located in the /dev namespace. The character API provides human readability and uniform format for the traces.The kernel data from the USBMon text data is made available to the userspace using debugfs[2] utility.

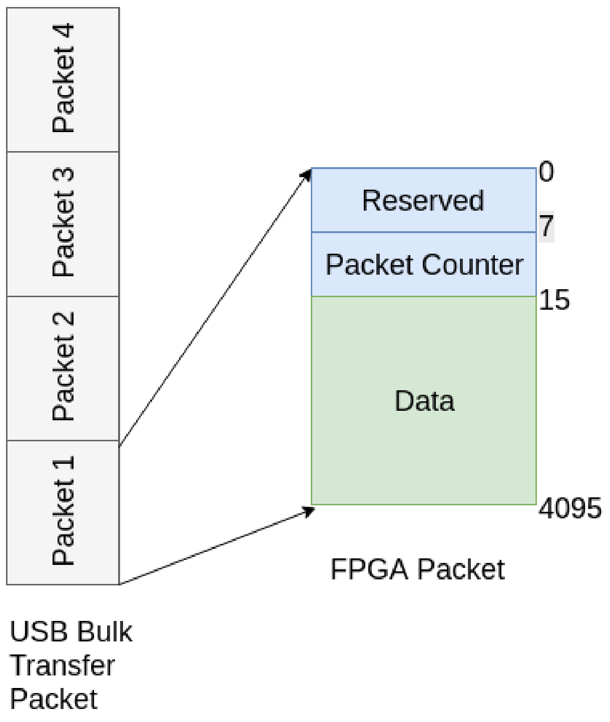


Figure 2.3: FPGA Packet Structure

URB Tag	Timestamp	Event Type	Address	URB Status
ffff8fbdbbae4000	2942307806	S	Bo:3:008:15	-115

Data Length	Data Tag	Data
64	=	21000100 00000000 002a0484 00000000 000000

Table 2.2: Text USB Trace Example.

2.3.1 Text Data Format

- *URB Tag:* URB Identification number, it is usually the in kernel adress of the URB structure.

- *Timestamp:* The timestamp for the URB event at the HCD in microseconds. It is measured by the usbmon main utility using *gettimeofday()* function of *time.h*.

- *Event Type:* It specifies the event type of the HCD event. S - Submission C - Complete E - submission error.

- *Address:* It consists of four fields separated by colons. The URB type and direction, bus number, device number, endpoint number. The URB type and direction specifies the type of USB transfer(can be both synchronous and asynchronous).

  The USB device transfers data through a pipe to a memory buffer on the host and

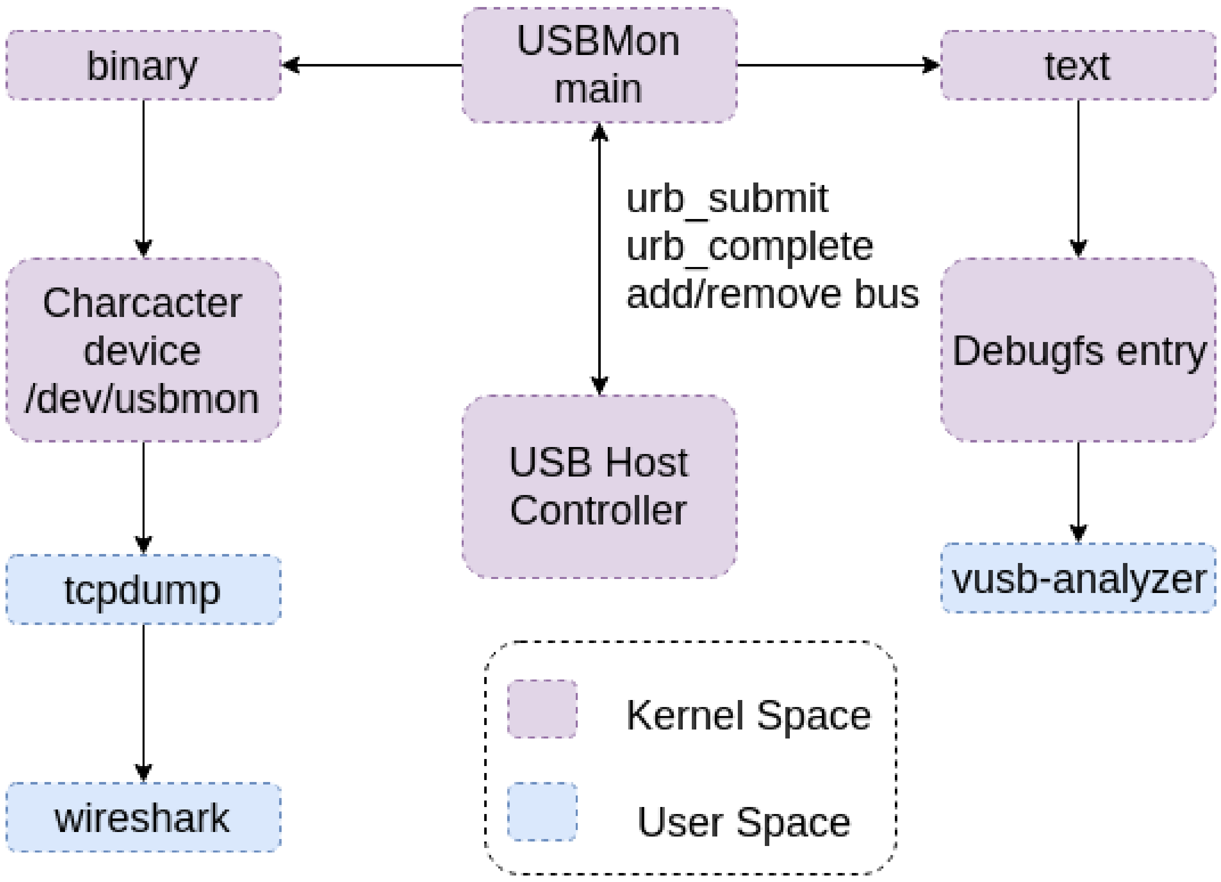


Figure 2.4: USBMon Architecture(Adapted from [1]).

Bi	Bo	Bulk Input and Output.
Ci	Co	Control Input and Output.
Ii	Io	Interrupt Input and Output.
Zi	Zo	Isochronous Input and Output.

Table 2.3: URB Type and Direction.

endpoint on the device. The type of data transfer depends on the endpoint and the requirements of the function. The transfer types are as follows[5]:

- **Control Transfers:** It is mainly used for configuration, command and status operations.
- **Bulk Transfers:** Bulk Transfer are used for bulky,non-periodic non time-sensitive burst transmissions.
- **Interrupt Transfers:** It is used for mainly sending small amounts of data infrequently or asynchronously.
- **Isochronous Transfers:** Isochronous transfers are mainly used for periodic, continuous streams of time sensitive data.

USB endpoint as explained by [6] , refers to the buffers on the USB device. The host computer irrespective of the host operating system can communicate by reading and writing to these buffers. They can be data endpoints and control endpoints.Data endpoints are used for transferring data whereas the control endpoint is used for configuration and device specific control.

- *Data Length:* For urb_submit it gives the requested data length and for callbacks it is the actual data length.
- *Data tag:* If this field is '=' then data words are present.
- *Data:* The data words contains in the USB transfer packet.

2.3.2 Raw Binary

The overall data format is same as the text data, the data is available in raw binary by accessing character devices at /dev/usbmonX. The data can be read by using *read* with *ioctl* or by mapping the buffer using *mmap*. The usbmon events are buffered in the following format:

```
struct usbmon_packet {
        u64 id;                 /*  0: URB ID - from submission to callback */
        unsigned char type;     /*  8: Same as text; extensible. */
        unsigned char xfer_type; /*    ISO (0), Intr, Control, Bulk (3) */
        unsigned char epnum;    /*     Endpoint number and transfer direction */
        unsigned char devnum;   /*     Device address */
        u16 busnum;             /* 12: Bus number */
        char flag_setup;        /* 14: Same as text */
        char flag_data;         /* 15: Same as text; Binary zero is OK. */
        s64 ts_sec;             /* 16: gettimeofday */
        s32 ts_usec;            /* 24: gettimeofday */
        int status;             /* 28: */
        unsigned int length;    /* 32: Length of data (submitted or actual) */
        unsigned int len_cap;   /* 36: Delivered length */
        union {                 /* 40: */
                unsigned char setup[SETUP_LEN]; /* Only for Control S-type */
                struct iso_rec {                /* Only for ISO */
                        int error_count;
                        int numdesc;
                } iso;
        } s;
        int interval;           /* 48: Only for Interrupt and ISO */
        int start_frame;        /* 52: For ISO */
        unsigned int xfer_flags; /* 56: copy of URB's transfer_flags */
        unsigned int ndesc;     /* 60: Actual number of ISO descriptors */
};
```

2.4 pidstat

2.5 802.15.4

Chapter 3

Methods

This chapter introduces the quantitative methods used in the measurement of the timing delays.

3.1 Timing Analysis

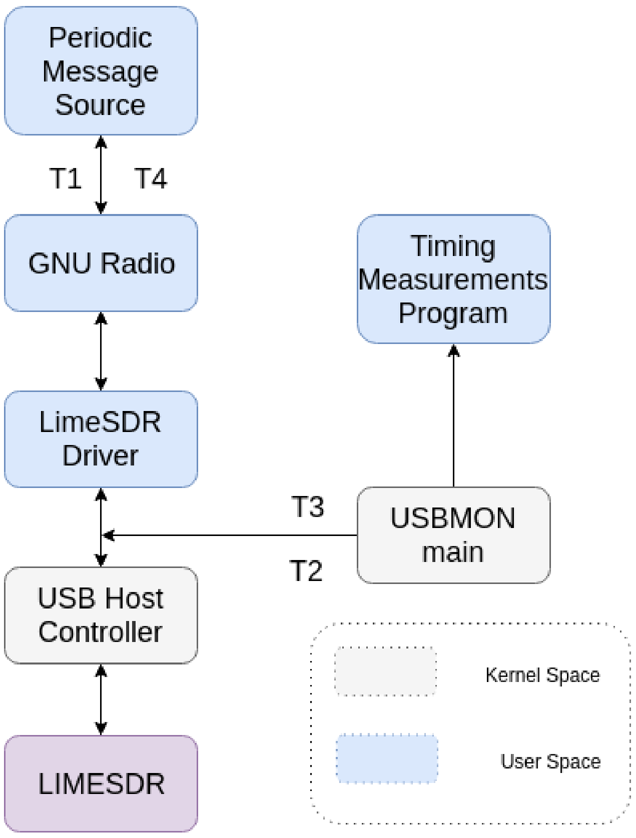


Figure 3.1: Overview of measurement setup

The project uses the Wime Project implementation of 802.15.4 MAC and PHY layers in GNU Radio. For the purpose of measurement of round trip latency, a loop back experimentation setup(Figure 3.1) was implemented. A periodic message source generates messages and notes down the time T1. It is then processed and modulated by the 802.15.4 MAC and PHY respectively and sent through the OSMOCOM transreceiver to the LimeSDR. The RX and TX ports of the LMS7002M has been shorted and hence the original sent message loopbacks through the FPGA and comes back to the GNU Radio and is demodulated and processed by the PHY and MAC blocks respectively and is ultimately received by the periodic message source and the time is noted as T4.

The usbmon kernel utility continuously monitors bus activity between the LimeSDR USB driver and USB Host Controller. It timestamps the transfers and generates event queues to be accessed from the user space. The timing measurements program parses the event queue to find the relevant packets and notes down their usbmon timestamps as T3 and T4 for transmit and receive packets respectively.

3.1.1 Message Source

A periodic message source block was implemented in the GNU Radio, it takes in the message length and time period as parameters. Figure 3.2 shows the working of the message source with respect to time. The data length controls the duty cycle of the signal by varying ΔT_{tx}, which is the time is requires to transmit the message through USB.

Everytime a message is sent and it receives a loopback, the block notes down the global time as T1 for transmit and T4 for receive respectively. Since the time noted should be compared with those from usbmon, *gettimeofday* was selected as the preferred method. The time period was set such that the transmitted message is received before sending the next message.

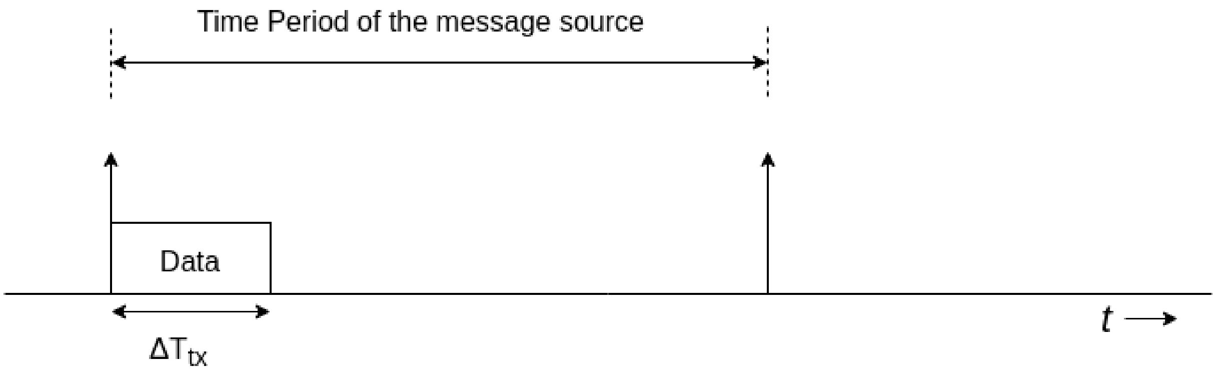


Figure 3.2: Periodic Message Source

USB Transfer Direction	Threshold Value
|I| TX	0.8
|I| RX	0.2

Table 3.1: Transfer Direction and Threshold Value

3.1.2 Timing Measurements Program

The timing measurements program uses ioctl to access the /dev/usbmonX character device. This allows the program to access the usbmon kernel utility event queue. The events are filtered to find packets with **0x01** & **0x81** device endpoints. The data streams are parsed to find the relevant data fields from FPGA packets(Figure 2.3), following that the data is converted from integer representation to complex floating point representation. The modulus of In Phase Sample's amplitude is used to determine if the data contained in the packet is useful or not.

Analyzing the samples in the data stream, the samples threshold for actual data packets to as shown in Table 3.1. Once the packets have been analyzed, the sequence of events was studied to generate a state machine representation for the timing functionality.
The sequence follows the structure shown in figure 3.3 if the condition mentioned about the time period in 3.2 is satisfied. Since we want to measure the round trip delay, the time instant of the first TX and last RX packet as noted by T2 and T3 respectively needs to be measured. The difference between them gives the Kernel round-trip delay as measured by usbmon.

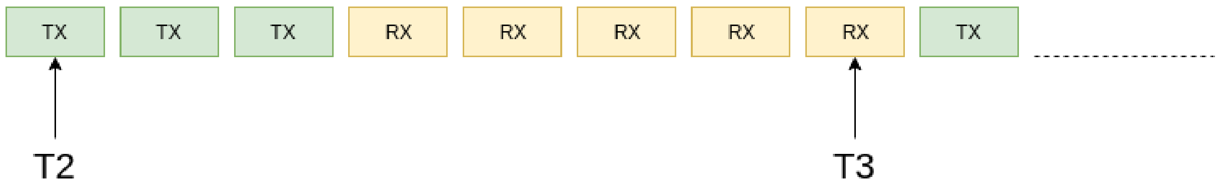


Figure 3.3: Sequence of valid data packet with time

State Machine shown in Figure 3.4 controls the timing measurement function. It starts with state S0 and when it receives a TX event it sets T2 and moves to S1, further TX events don't update the value of T2 as we want the first TX event time. The state machine moves from S1 to S2 on a RX event, it sets the value of T3, further RX events updates the value of T3 as we want the time instant of the last RX event. On receiving TX event when at S2, it moves to S1, calculates $T3 - T2$ and sets the value of T2.

3.1.3 Results Correlation Method

All the time instants are stored in Unix Time Format, a python script stores the values in separate arrays t1, t2, t3, t4 for GNU Radio Transmit Time, Kernel Transmit Time, Kernel Receive Time and GNU Radio Receive Time respectively.

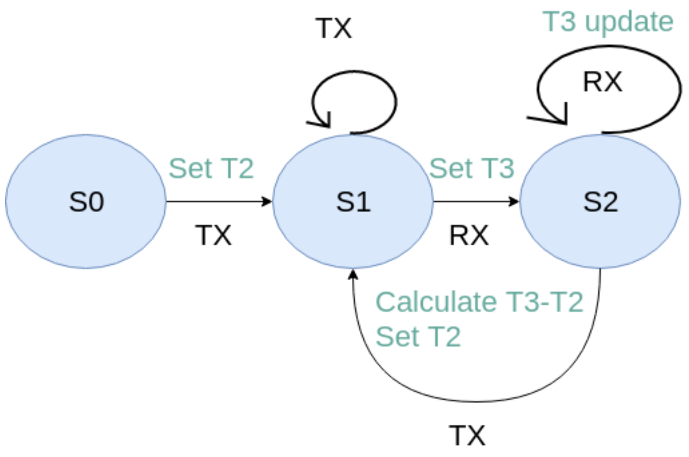


Figure 3.4: State Machine

Algorithm 1 Time Data Correlation

```
$T \leftarrow$ Time Period of Message Source
$l \leftarrow$ min(length of time arrays)
$i \leftarrow 0$
**for** $i < l$ **do**
    **if** $(t1[i] > t2[i]$ or $t3[i] > t4[i])$ **then**
        delete $t1[i], t4[i]$
    **else if** $(t2[i] - t1[i]) > T$ **then** delete t2[i]
    **else if** $(t4[i] - t3[i]) > T$ **then** delete t3[i]
    **else** $i \leftarrow i + 1$
        $l \leftarrow$ min(length of time arrays)
    **end if**
**end for**
```

Once the arrays have been compared to remove corrupt data, the mean and standard deviation of the respective arrays are found.

Chapter 4

Results and Analysis

4.0.1 Analytical Method

The 802.15.4 PHY layer expands 1 byte of message data to 128 bytes, so the maximum packet length of 127 bytes becomes produces sample data of size $127 * 128 = 16256 bytes = 15.875 KB$

. The FPGA packet format adds 16 bytes overhead for every 4080 bytes so the overhead for 16256 bytes would be 64bytes. So the overall transfer size would be 16320 bytes. This would require four FPGA packets so the actual size of the USB transfer would be 16384 bytes Now for sampling rate of 1MHz $\equiv$ 1MSPS, the actual data transfer is 1.5 MBps since the LMS7002M has 12 bits ADC and DAC.

Sampling Rate	USB Transfer delay
5 MHz	4369.07 μs
10 MHz	2184.53 μs
15 MHz	1456.35 μs
20 MHz	1092.27 μs

Table 4.1: Analytical USB Transfer Delay

4.0.2 Experimental Results

Figure 4.1 shows the different terminology used in the results, with the TX & RX software delay is the delay caused by the GNU Radio and LimeSDR driver processing, the Kernel RTT Time includes the buffer delay in the LimeSDR and the USB communication delay. Total RTT Time = Kernel RTT Time + TX Software Delay + RX Software Delay. All the timing measurements are done on Lenovo Thinpad X240 with Dual-Core Intel® Core^{TM} i5-4300U CPU @ 1.90GHz and 4GB RAM. The setup use Limesuite version 17.12.0 and gateware version 2.12.

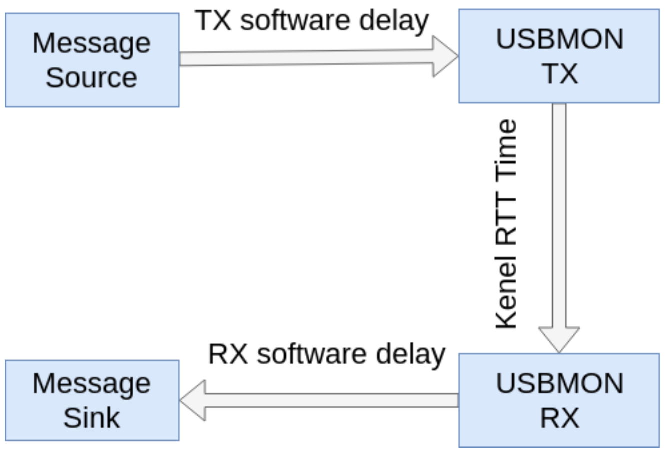


Figure 4.1: Results Setup

Sampling Rate(MHz)	5	10	15	20
Total RTT mean (μs)	5360	3606	3065	4485
Total RTT std deviation (μs)	326	472	262	1273
Kernel RTT mean (μs)	4113	1937	1354	762
Kernel RTT std deviation (μs)	1201	330	232	298
TX chain mean (μs)	470	675	831	1586
TX chain std deviation (μs)	60	1165	186	860
RX chain mean (μs)	1100	1122	871	1769
RX chain std deviation (μs)	472	1764	262	1036

Table 4.2: Experimental results

4.0.3 Analysis

- The results show a monotonic drop in the kernel USB timings with increase in sampling rate and monotonic increase for RX and TX delay (Exception: 15 MHz). This indicates with increase in sampling rate, the buffers are getting overloaded and hence an increase in processing delay compared to bus communication delay.
- Another thing that I noticed was at high sampling rate the round trip time increases with time, again pointing to buffer delay on the RX chain.
- My measurement program becomes highly unstable at higher sampling rates, for example for 20MHz, I captured 610 packets of which I could correlate only 160 packets. This is mainly because the usbmon event queues overflow and hence my timing measurement program misses some relevant events and reports wrong timing information. One method I plan on using is flushing the buffers before the message source generates the message since each measurement is independent of the previous in a TDMA protocol.
- The analytical values for the RTT time(Table 4.1) is more than the actual value that usbmon reported(Table 4.0.2), my hypothesis is that this happens due to my assumption that all the data in the LimeSDR TX buffer is popped before the relevant RX data is popped back in the RX buffers on the LimeSDR side. But in actual operation even before all the TX data has been popped, the loopback data is being pushed to the RX buffers. This is demonstrated using figure 4.2 where its shows the state of the RX and TX buffers with respect to time. *t1* shows the instant when all the relevant data has been popped from the TX buffers and *t2* shows the instant when the relevant RX data is popped from the buffers. For my assumption *t1* should be equal to *t2*, but here $t2 < t1$ hence the reported values are less than those of the analytical model. With increase in sampling rate the difference between t2 and t1 increases. There is a need to address this issue to ensure reliability of the measurement method.

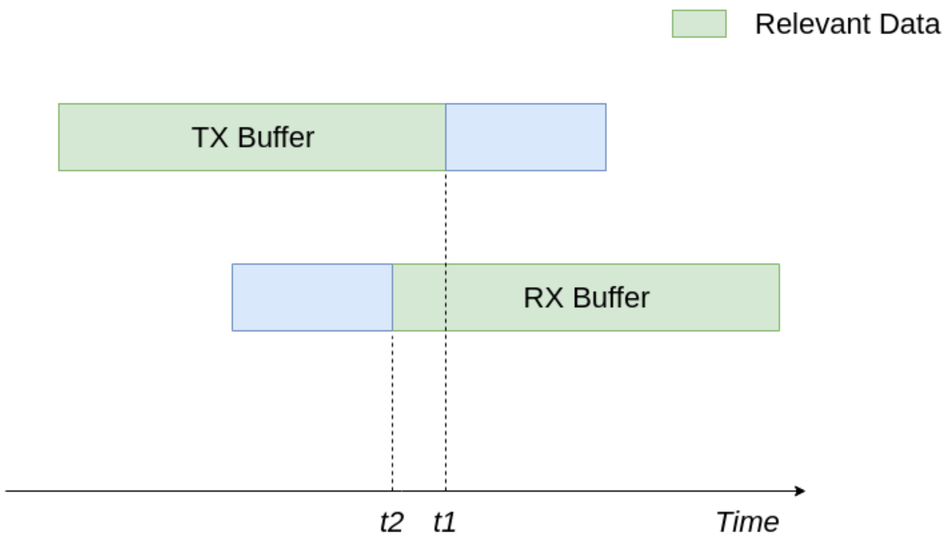


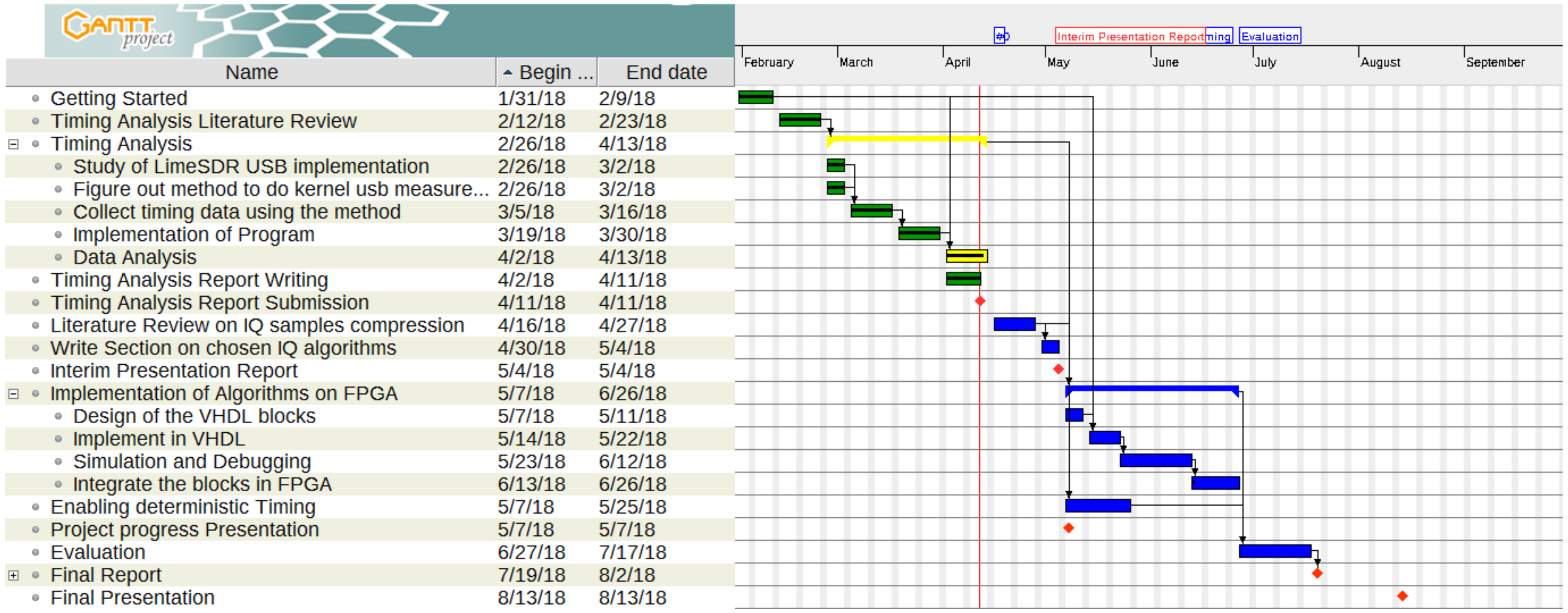
Figure 4.2: Overlapping of buffers on LimeSDR

Chapter 5

What's next

The project plan is presented in Gantt Chart format in figure 5.1. According to my estimate I am one week behind the project plan.

- Timing Analysis:
  - More data points to pinpoint the buffer delays.
  - Off-line processing of the USB data to pinpoint the time instant of maximum correlation between RX and TX data
  - Make individual measurements independent.
  - Solve the buffer overlap problem.
- Enabling deterministic timing: A key element for enabling TDMA protocols would be deterministic send and receive time of packets. So I would look at possible strategies.
- Evaluation: Since I am looking at compression algorithms, it would be essential to find the compression ratio I am able to achieve with lossy and lossless compression. Also, I should concentrate on figuring out how these compression algorithms affect the performance of SDR systems. The strategies for deterministic timing needs to be evaluated for jitter to actual TDMA schedules.



Name	Begin ...	End date
Getting Started	1/31/18	2/9/18
Timing Analysis Literature Review	2/12/18	2/23/18
Timing Analysis	2/26/18	4/13/18
Study of LimeSDR USB implementation	2/26/18	3/2/18
Figure out method to do kernel usb measure...	2/26/18	3/2/18
Collect timing data using the method	3/5/18	3/16/18
Implementation of Program	3/19/18	3/30/18
Data Analysis	4/2/18	4/13/18
Timing Analysis Report Writing	4/2/18	4/11/18
Timing Analysis Report Submission	4/11/18	4/11/18
Literature Review on IQ samples compression	4/16/18	4/27/18
Write Section on chosen IQ algorithms	4/30/18	5/4/18
Interim Presentation Report	5/4/18	5/4/18
Implementation of Algorithms on FPGA	5/7/18	6/26/18
Design of the VHDL blocks	5/7/18	5/11/18
Implement in VHDL	5/14/18	5/22/18
Simulation and Debugging	5/23/18	6/12/18
Integrate the blocks in FPGA	6/13/18	6/26/18
Enabling deterministic Timing	5/7/18	5/25/18
Project progress Presentation	5/7/18	5/7/18
Evaluation	6/27/18	7/17/18
Final Report	7/19/18	8/2/18
Final Presentation	8/13/18	8/13/18

Figure 5.1: Project Plan

Bibliography

[1] Partha Basak and Kishon Vijay Abraham I. "USB Debugging and Profiling Techniques". In: (Oct. 4, 2018). URL: https://elinux.org/images/1/17/USB_Debugging_and_Profiling_Techniques.pdf.

[2] *Debugfs Documentation*. Linux Kernel Archives. URL: https://www.kernel.org/doc/Documentation/filesystems/debugfs.txt.

[3] Thomas Schmid, Oussama Sekkat, and Mani B. Srivastava. "An experimental study of network performance impact of increased latency in software defined radios". en. In: ACM Press, 2007, p. 59. ISBN: 978-1-59593-738-4. DOI: 10.1145/1287767.1287779. URL: http://portal.acm.org/citation.cfm?doid=1287767.1287779 (visited on 07/31/2018).

[4] T. Ulversoy. "Software Defined Radio: Challenges and Opportunities". In: *IEEE Communications Surveys Tutorials* 12.4 (2010), pp. 531–550. ISSN: 1553-877X. DOI: 10.1109/SURV.2010.032910.00019.

[5] *USB Data Transfer Types*. URL: http://www.jungo.com/st/support/documentation/windriver/10.2.0/wdusb_manual.mhtml/USB_data_transfer_types.html.

[6] *USB endpoints and their pipes*. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/usb-endpoints-and-their-pipes (visited on 04/11/2018).

[7] *USBMon Documentation*. The Linux Kernel Archives. URL: https://www.kernel.org/doc/Documentation/usb/usbmon.txt.