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

Acronyms

6LoWPAN	IPV6 over Low-Power Wireless Personal Area Network
ACK	Acknowledgement
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit.
ASK	Amplitude Shift Keying
BPSK	Binary Phase Shift Keying
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSMA	Carrier Sense Multiple Access
CSMA/CA	Carrier-sense multiple access with collision avoidance
CSMA/CD	Carrier-sense multiple access with collision detection
CSS	Chirp Spread Spectrum
DAC	Digital Analog Converter
DDR	Double Data Rate
DDS	Direct Digital Synthesis
DMA	Direct Memory Access
DSP	Digital Signal Processor
FCS	Frame Control Sequence
FFD	Full Function Device
FIFO	First In First Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FPRF	Field Programmable RF
FSM	Finite State Machine
GPIF	General Programmable Interface
GTS	Guaranteed Time Slot
I2C	Inter-Intergrated Circuit
IO	Input Output
IoT	Internet of Things

IQ	In-Phase Quadrature.
L2	Layer 2
LNA	Low noise amplifier
LPWAN	Low Power Wide Area Network
LQI	Link Quality Information
LR-WPAN	Low-Rate Wireless Personal Area Network
MAC	Medium Access Control
MIMO	Multiple Input Multiple Output.
NCO	Numerically Controlled Oscillator
NIC	Network Interface Controller
O-QPSK	Offset Quadrature Phase Shift Keying
OSI	Open Systems Interconnection
PAN	Personal Area Network
PCIe	Peripheral Component Interconnect Express
PDU	Packet Data Unit.
PGA	Programmable Gain Amplifier.
PHY	Physical
PLL	Phased Lock Loop
PN	Pseudo Noise
RF	Radio Frequency
RFD	Reduced Function Device
RTL	Register Transfer Level
SDR	Software Defined Radio
SFD	Start-of-Frame Delimiter
SHR	Synchronization Header
SPI	Serial Peripheral Interface.
TDMA	Time Division Multiple Access
TSP	Transreceiver Signal Processor
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language

Chapter 1

Introduction

Internet of Things (IoT) is enabling communication among huge numbers of diverse low power devices. According to estimate by Ericsson [3], there will be 20 billion connected IoT devices by 2023 . The communication needs for a field temperature sensor is different from an industrial controller. Hence, there is need for research and development of communication protocols that satisfy diverse device communication needs. The evaluation of these experimental protocols is difficult because of the need of specialized radio hardware. SDR devices can be a powerful platform for enabling the real-world evaluation of these protocols.

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 [5]* 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.

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 soft-

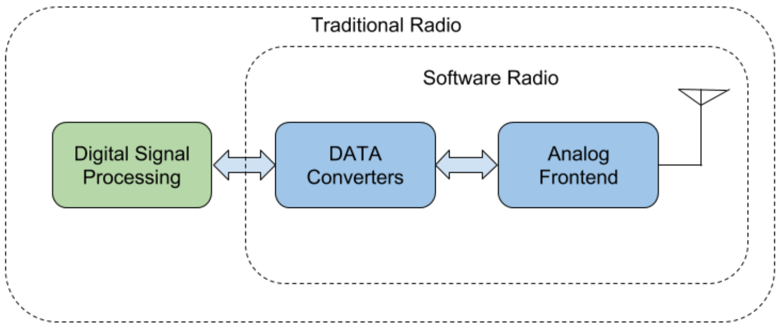

Figure 1.1: Software Radio and Traditional Radio Architecture.

ware upgrades of base stations a more attractive solution than the costly replacement of base stations[7]. 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 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.

IEEE 802.15.4 [**noauthor_ieee_nodate**] is a network specification deigned specially for Low Power Wide Area Network (LPWAN) i.e IoT devices. It specifically defines the Physical Layer and the MAC layer of the network stack. With the aim to be enabler of SDR testbeds for IoT protocols, this project uses IEEE 802.15.4 based physical layer implementation for the evaluation of the SDR platform.

LimeSDR [4] is a new low-cost SDR platform, which supports the desired frequency bands. The lack of research on the characteristics of the platform made it the ideal choice for my SDR platform.

1.1 Problem Context

1.1.1 CSMA

As highlighted by [6], SDR based systems don't comply with the stringent timing constraints imposed by modern MAC protocols. Furthermore, the presence of long bus communication and processing delays create *blind spots*[6] in carrier sensing. In fig 1.2, a packet is being transmitted through the air medium which is received by the SDR system. Since there is communication and processing delays, the Central Processing Unit (CPU) of the SDR system receives the packet completely at t_1 delayed from t_0 when the packet transfer ends on the air medium.

Once the packet has been received, the system wants to let the transmitting system about the successful reception by sending the Acknowledgement (ACK) packet. It detects the medium is free using carrier sensing, but this information is actually past information that has been 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.

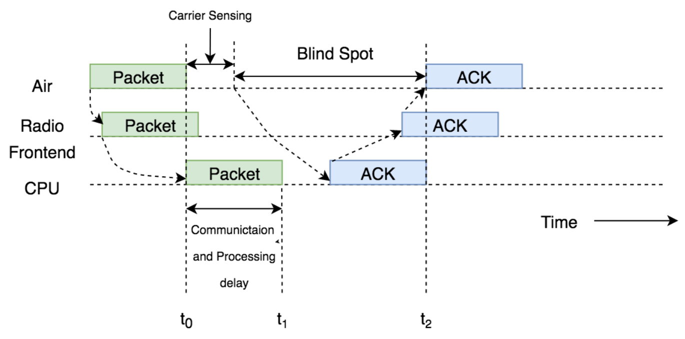

Figure 1.2: Blind Spots Illustration(adapted from [6]).

Hence when designing MAC protocols, these delays needs to be taken into account to avoid collisions. This necessitates a closer understanding of these delays and how different system 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 the time-slots longer 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

- What are the timing delays in LimeSDR based IEEE 802.15.4 network ?

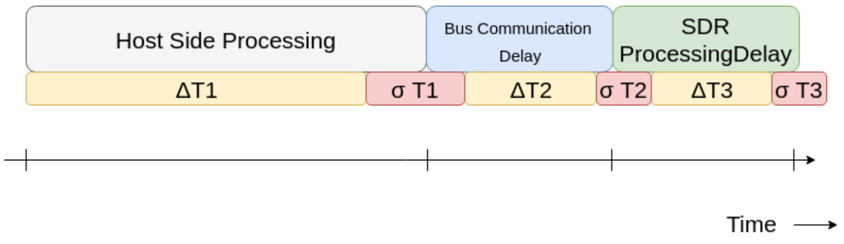

Figure 1.3: Research Question 1.

- How to implement precise scheduling in LimeSDR based IEEE 802.15.4 communication system?

1.3 Report Outline

The remainder of the report is structured as follows.*Chapter 2* introduces the previous work in this field, as well the needed background information on the LimeSDR 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. It also discusses on methods for precise scheduling in LimeSDR based systems. *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 introduces the necessary background information needed for understanding the rest of the report. First section, provides a broad introduction to SDR systems, GNU Radio Software Tool and PHY and MAC layers of the network stack. The second section introduces the LimeSDR platform and its hardware and software architecture. Wime Project and IEEE 802.15.4 is explained in the third section. Finally, the fourth section introduces the tools used in the methods chapter.

2.1 Essential Concepts

2.1.1 PHY and MAC Layers

Open Systems Interconnection (OSI) Model (ISO/IEC 7498-1:1994) presents the abstract model for networking, that is used for most communication systems design. The abstract model is divided into 7 layers, where entity in each layer implements the functionality of the layer and interacts directly with the layer beneath it. This added functionality can be used by the upper layers. Data from the user application is encapsulated by each subsequent layers of the OSI model into their frame format. These frames carry meta-data in the form of frame headers. Different protocols use different frame format and headers as it helps differentiate one protocol from another. These headers help the receiver in learning where the incoming data packet is coming from, who is it meant for, how to decode and arrange the contents of the data packets etc.

PHY layer is the lowest layer (L1) of the OSI model,it interacts with the physical communication channel directly. It specifies the type of data transfer(serial/parallel) and data rate of the protocol. PHY layer defines the process of transmitting raw bits through the physical medium. The bit-stream is grouped into code words and converted to symbols, which are then modulated to a physical signal for transmission over the transmission medium. PHY layers also provides physical transmission link information like carrier sense, collision detection and Link Quality Information (LQI) to the upper layers.

MAC layer, Layer 2 (L2) of the OSI model, is responsible for defining the the methods for sharing and using the common transmission medium among multiple devices. In case of outgoing packets, the MAC layer adds the MAC address of the destination device to the packet header. It adds the synchronization preamble and Frame Control Sequence (FCS) for checking transmission error. Retransmission in case of dropped packets and acknowledgement to successfully received packets are handled by this layer.

CSMA is L2 protocol of the OSI Model. It mainly comes in two varieties: Carrier-sense multiple access with collision detection (CSMA/CD) and Carrier-sense multiple access with collision avoidance (CSMA/CA). The flow graphs of both are shown in Figure 2.1. In the older CSMA/CD, the nodes checks the idleness of the channel after the frame is ready. If idle it starts transmission. During transmission, it monitors the medium for collision. If collision is detected, it employs a collision recovery process, where it sends a jam signal to signal other nodes that a collision has occurred. Then it waits for a random delay and starts transmission again.

CSMA/CA tries to avoid collision, it starts off similar to CSMA/CD where it senses to check when the channel is idle. If found idle, it starts transmission. It is difficult for wireless nodes to detect collisions simultaneously during transmission. Therefore, it relies on an ACK message from the receiving node to check if the data packet was received. If ACK is not received, the node assumes a collision has occurred and uses exponential back-off to determine when the next time to re initiate transmission.

TDMA is also a L2 protocol, where a coordinator schedules medium access to the nodes in a periodic manner. Communication happens in time-slots. Each node in the network is given exclusive access to transmit during its time slot. The coordinator generates beacon signals periodically to maintain relative time synchronization. On receiving the beacons, the nodes adjust their transmit clocks so that they have the correct estimate of their time-slots.

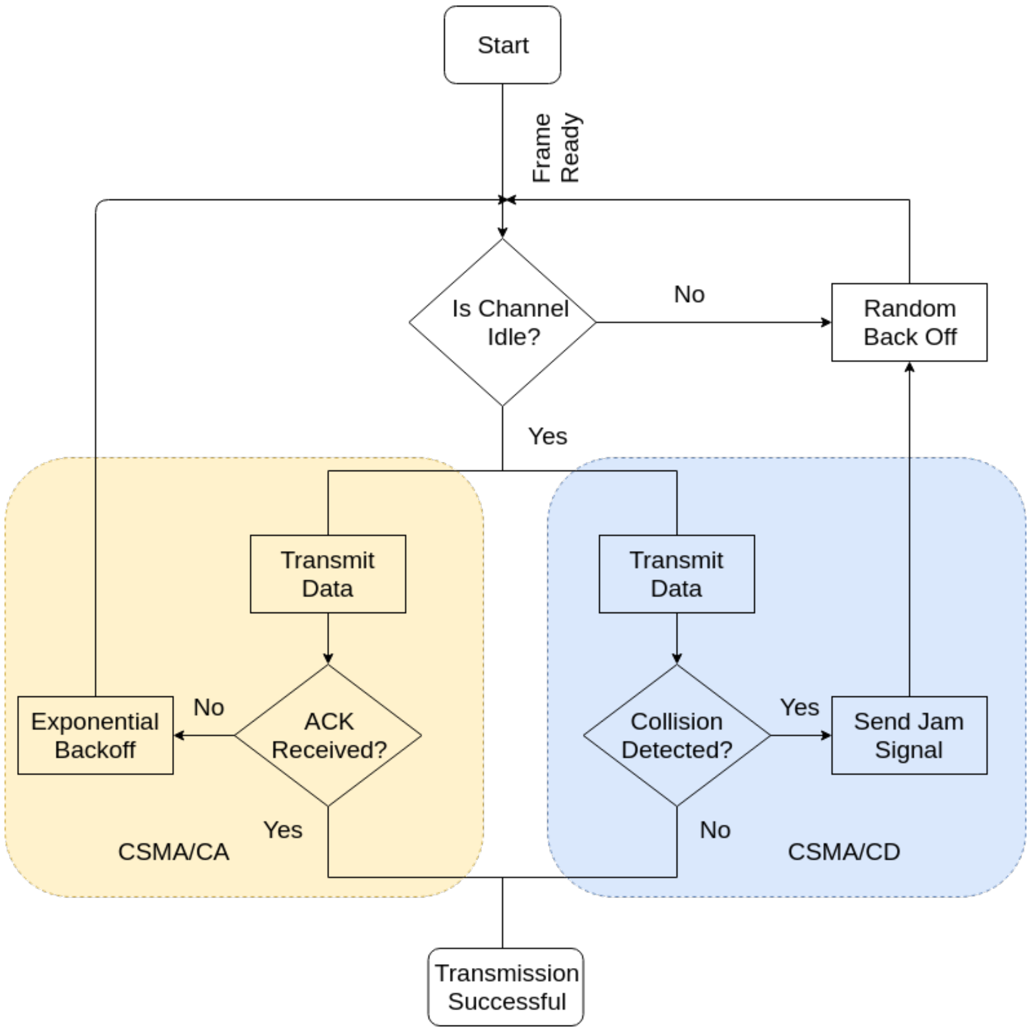

Figure 2.1: CSMA flow graph.

2.1.2 SDR Platforms

SDR represents a new paradigm of communication system design where the system is flexible to adapt to the needs of the end-user as also the radio channel conditions. Nychis et.al [5] classifies SDR based communication systems into two main architectures.

- *Host-PHY Architecture:* This is the most common architecture, enabling design and development of the entire system in software. It provides the maximum flexibility in terms of design and implementation choices, also there is added benefit of easy upgrades. But, since the system is designed is software only, the processing and communication delays make most modern

MAC protocols infeasible in this architecture.

- *NIC-PHY Architecture:* In this architecture most of the PHY layer functionality is implemented in Field Programmable Gate Array (FPGA) and Digital Signal Processor (DSP). The closer proximity to the radio hardware and specialized parallel hardware processing makes this architecture most suitable for running the modern MAC protocols. But the design process for this architecture based systems is time consuming and difficult, as traditionally hardware programming is harder than software programming. However, they are much more flexible compared to commercial Network Interface Controller (NIC). *Wireless Open Access Research Platform*(WARP) [10] is an example of system based on this type of architecture.

Since host-PHY [5] is the most commonly used architecture (**cite sources**), the report concentrates on explaining the functionality of SDR systems using this architecture. Figure 2.2 shows the typical design of communication systems in this architecture. The system can be broadly divided into two main components:

1. SDR Platform.

2. Host Computer.

Since the process for transmission and reception processes are very similar , this report concentrates on the explaining the reception process of SDR platforms.

SDR Platform: It is the hardware that provides access to the wireless medium in a flexible manner. RF signals are transmitted and received by the platform, it converts these analog signals to digital samples and transfers them to the host computer. The main building blocks of these platforms are shown in Figure 2.2.

- **Software Configurable RF trans-receiver** This is the heart of SDR platforms and provides RF modulation and demodulation capability. They are attached to wide-band antennas for receiving and transmitting over a broad range of frequencies. Taking the case of reception of RF signal, the signal received from the antenna is amplified by a Low noise amplifier (LNA). The LNA amplifies a low power signal without significantly degrading the signal to noise ratio. Once amplified, the signal to passed to a RF receiver, where the RF signal is demodulated either to a intermediate frequency or baseband signal depending on whether the the receiver is a Zero-IF receiver or Super-heterodyne receiver respectively.

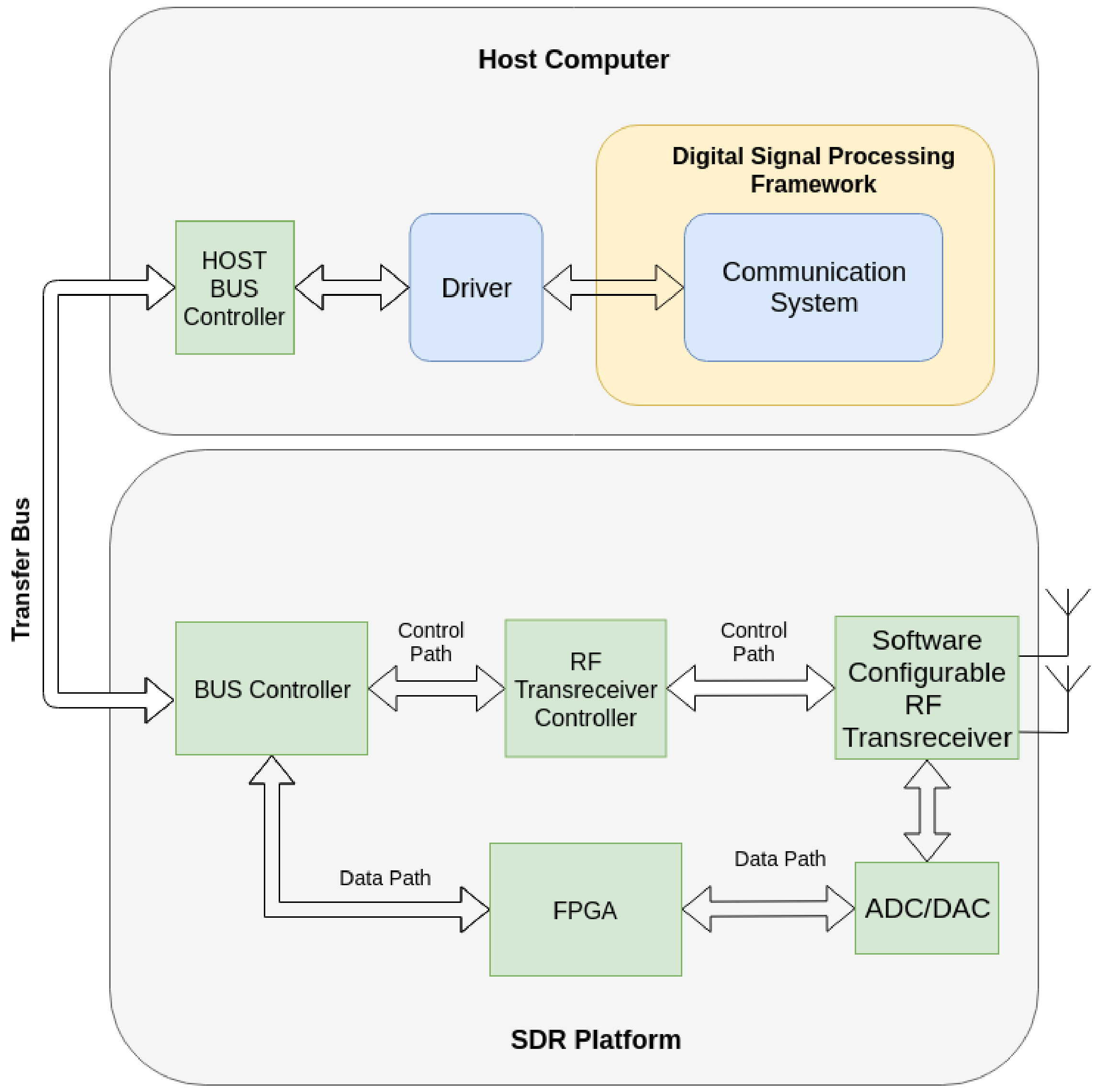

Figure 2.2: Host-PHY [5] SDR architecture.

Figure 2.3 shows a simple RF receiver in which the LNA output signal is fed to a mixer. The mixer is a signal processing block used for translating the input signal to another frequency range. The mixer uses a locally generated carrier frequency for the translation. If the input RF signal has a frequency of f_{rf}, and the local oscillator frequency is f_{lo} then the mixer will produce a signal with frequency components $f_{rf} - f_{lo}$ and $f_{rf} + f_{lo}$. This output signal is then passed through a band-pass filter with $f_{rf} - f_{lo}$ as center frequency, this will reject the unwanted $f_rf + f_lo$. If we assume $f_{rf} = f_{lo}$, the

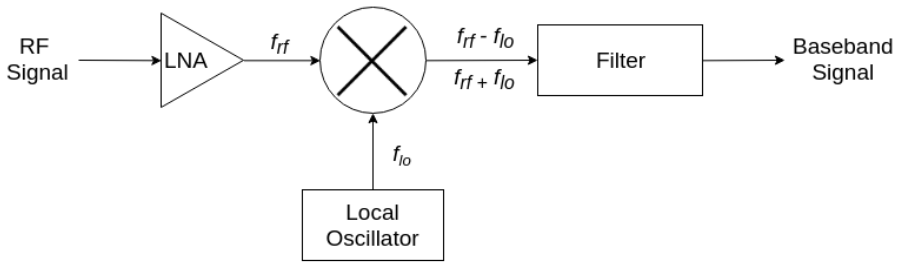

Figure 2.3: Simple RF receiver.

bandpass filter will become a low pass filter and the output signal will be the baseband signal. This is the case in Zero-IF receiver architecture. In super heterodyne receiver architecture, a number of intermediate frequency stages are used before generation of the baseband signal.

In traditional trans-receiver, a crystal oscillator is used. This results in good stability of the local oscillator signal but the system is now tuned to a particular frequency. With the goal of flexibility in mind, SDR platforms use frequency synthesizers to generate the local clock signal. Frequency synthesizers are used for creating arbitrary waveforms from a single frequency clock. Most SDR use DDS as the frequency synthesizer, which uses a highly stable oscillator used as a reference signal.

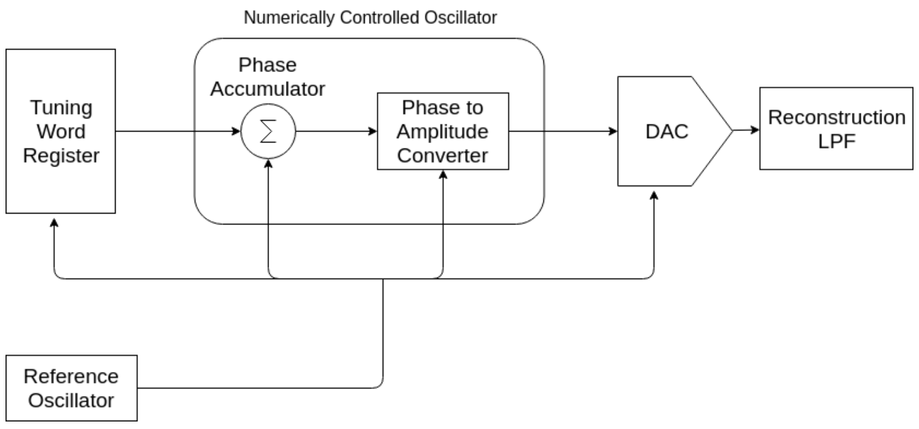

Figure 2.4: DDS.

The main components of a DDS are Numerically Controlled Oscillator (NCO) which is made of the Phase Accumulator and Phase to Amplitude Converter, Digital Analog Converter (DAC) and a Tuning Word Register as shown in Figure 2.4.

In each clock cycle, the phase accumulator increases the output(phase) by the value stored in the Tuning Word Register. This output is the input to the Phase to Amplitude Converter, which basically is a lookup table containing the amplitude for a particular phase. The output of the Phase Accumulator is basically the address(phase) for the lookup table. The output is then converted to an analog signal by the the DAC and then passed through a low pass filter to smoothen out the waveform.

Since the Tuning Word Register is responsible for how fast the phase changes, the output sinusoid frequency can be controlled by controlling the Tuning Word Register, this is how SDR platforms are able to generate a wide range of local oscillator frequencies.

In SDR platforms, the filter (shown in Figure 1.1 is implemented using Finite Impulse Response (FIR) filters. FIR filters are convolutional filters where the output response of a particular input is finite. The output of a FIR filter can be adjusted by readjusting the parameters of the filter's impulse response. This allows for the SDR to allow certain range of frequencies in the output, which can be adjusted by external control.

- **RF Trans receiver Controller:** The controller provides the interface to control the RF Transceiver. The communication system running on the host computer, provides the desired RF parameters to the controller. It then translates those instructions to digital signals for configuring the RF trans-receiver modules like the FIR filter weights, the tunable word register for selecting the desired frequency and also the desired gain in the programmable gain amplifiers.

- **ADC/DAC:** The filtered analog RF signal needs to be converted to digital domain before transferring to the host computer. Fast data converters are used on the SDR platforms for this purpose. The sampling rate of these data converters determine the available RF bandwidth of the system according to the Nyquist criteria. The resolution of the SDR data converters are important so as to ensure high dynamic range of the received signal, ensuring that the system is capable is receiving very weak signals as well very strong signals without saturation. The sampling rate of these data converters are controlled by the RF trans-receiver controller.

- **FPGA:** The FPGA provides acts as the glue logic between the data converters and the bus controller. In most cases the bus communication is bursty in nature, whereas the DAC produces a stream of samples. FPGA provides for efficient buffering of these samples, packs them into bursts to be sent over the bus. In some platforms, additional information like a sample clock is also packed into these bursts. Some applications might need additional signal processing, FPGA provides for a efficient way for implementing these filters. In NIC-PHY architecture most of the communication system is designed using the FPGA.

- **Bus Controller:** It is the bridge for the bursts of data crossing over from the SDR platform to the host computer and vice versa. It takes in data packets from the FPGA encodes them with bus transfer protocol, then initiates transfer. The flow control and routing for different packets is also handled by the bus controller.

Host Computer: The host computer, a general purpose computer, is the brain of the communication system. It runs the software implementation of the baseband processing for the desired protocol, taking the digital samples from the SDR as input. During initialization, it configures the SDR platform. Depending on implementation, it can have the full network stack and an application running on top of it. From architectural viewpoint, the host computer has three main components as shown in Figure 2.2.

- **Bus Controller:** The bus controller on the host computer controls the other end of the bus communication link. It decodes the received data bursts and sends them to the driver for further processing. If the bus communication involves a master slave relationship, then the host computer bus controller is designated as the master. It initiates the data transfer on the bus, and the slave bus controller responds to the requests placed by this bus controller.

- **Driver:** The driver is the abstraction layer for the SDR platform communication and configuration. The communication system designer should be able to provide high level instructions to the SDR platform. It is left to the driver to handle the translation of high level instructions to low level register control data words. For example, when tuning the RF trans-receiver, the system designer would be much more comfortable to say set the center frequency to 1.8 GHz, rather than saying set register at address "x" to value "data". The driver handles this translation. It also is responsible for ensuring a reliable data transfer link. Since the communication system and the incoming data may be running at different rates, the driver buffers the incoming data and provides it to the the running communication system at its data processing rate.

- **Digital Signal Processing Framework:** The hardware baseband processing of communication systems are generally designed with concurrent execution in mind. Whereas, general purpose computing platform are sequential in nature. Many core processors add the capability of concurrent execution but at a much smaller scale than what can be acheived with hardware processing platforms like FPGA and Application Specific Integrated Circuit. (ASIC). So when designing systems on general purpose computing platforms, this change in execution model needs to be taken into account.

  Threads provide a software method to implement concurrent execution model. But, efficient thread management and synchronization produces significant overhead. Digital Signal Processing frameworks are designed to help system designers to only design their signal processing modules without worrying about thread synchronization and management. They are designed with concurrent execution of signal processing algorithms and modularity of system design in mind. Two of the most popular frameworks are GNU-Radio and Labview. In the next section, the report goes into detail about GNU Radio.

2.1.3 GNU Radio

GNU Radio is an open-source digital signal processing framework, which has been rapidly evolving with a large active community. The software framework can be used for both simulation and prototyping real-world application scenarios. It provides a graphical interface for designing signal processing chains as well as extensive library of signal processing blocks like filters, synchronizers, demodulators etc. The ease of use of the framework, extensive library as well as hardware support for most SDR platforms has led to diverse application use cases such as RFID, 802.11, cellular networks.

GNU Radio is designed to stream large amounts of data in real-time between parallel computational nodes. The data flow between the nodes from the source to the sink is described by the flow-graph, while the flow of data is controlled by the GNU Radio Scheduler. Figure 2.5, shows a simple flowgraph where the data from the Sources node is processed by the signal processing chain. The processed data is sent to the Sink nodes. The signal processing chain itself is composed of multiple data processing nodes, for example the flow graph in Figure 2.5 has 4 nodes in the processing chain. The scheduler is in task of scheduling the execution of these blocks. From the block designer point of view, these blocks can be viewed to be executing concurrently.

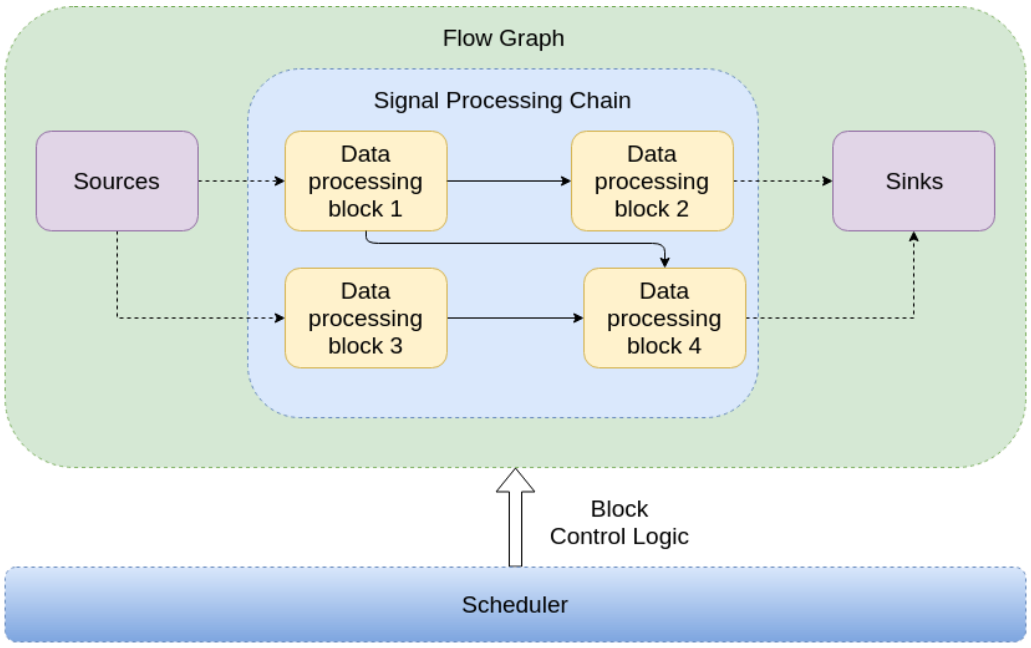

Figure 2.5: GNU Radio Software Architecture.

Blocks and Flow Graphs The computational nodes in the flow-graph are the GNU Radio processing blocks. Each block describes how the input elements to the block, are converted to output elements in the *work* function.

Number of input elements	Number of output elements	Name
N	0	Sink Block
0	N	Source Block
N	1	Interpolation block
1	N	Decimation block
M	N	General Block

Table 2.1: GNU Radio Block Types

The relationship of input and output elements defines the type of the GNU Radio processing block as shown in Table 2.1. The type of block indicates the scheduler on how the block processes information. There are two types of blocks: *Synchronous block* and *block*. For synchronous blocks, there is a rational relationship between the input and output elements. The sink, source, interpolation block and decimation block in Table 2.1 are synchronous blocks. The key difference between different block types is in how the scheduler handles the input and output buffers of each block. For the synchronous blocks, the scheduler implicitly handles the input and output pointers to the buffers. For general blocks, the *work* function needs to explicitly pass the information on how many elements it consumed and produced.

Since in GNU Radio flow graph, data is passed from one node to another, the method of passing the data among different blocks needs to be defined. This method is defined in the block interfaces. Stream Interfaces are intended to stream large amounts of data between blocks with variable processing rates. They use large buffers to pass the data from one node to another. Stream interfaces work well for samples, bits etc. but they are not the right method to pass metadata, control information or bursts of data between blocks as it involves significant overhead.

GNU Radio recently added the message passing interface for handling asynchronous message passing. GNU Radio also supports stream tags for handling metadata as it is closely associated with the stream data samples. Stream tags are attached with stream data samples and provide additional information associated with the sample. It can be used both for passing control flags as well as metadata information like the Packet Data Unit. (PDU) size, timing information etc. These stream tags are propagated to the next blocks and is updated by the data rate changes. For example, if the block takes it 2 samples as input and produces 4 samples as output, its data rate is 2. In this case, if the input stream had a stream tag at position "x" then the the location in the output stream would be "2x".

Figure 2.6 shows the general architecture of a GNU Radio block. Each block has associated buffers for the stream interfaces, two computational components, namely the *work* function and the *message handler* function. A run time scheduler is associated with each block for controlling the execution of block during runtime. The runtime scheduler has its own signaling mechanism for interacting with schedulers of other blocks. This signaling mechanisms are hidden to the flow graph designer, and are used for flow control. The blocks providing the inputs to the current block are called upstream blocks, while those that are fed by the output of this block are called downstream block.

For the sake of simplicity, it is assumed that the current block in Figure 2.6 has one upstream block and one downstream block. This implies that the block

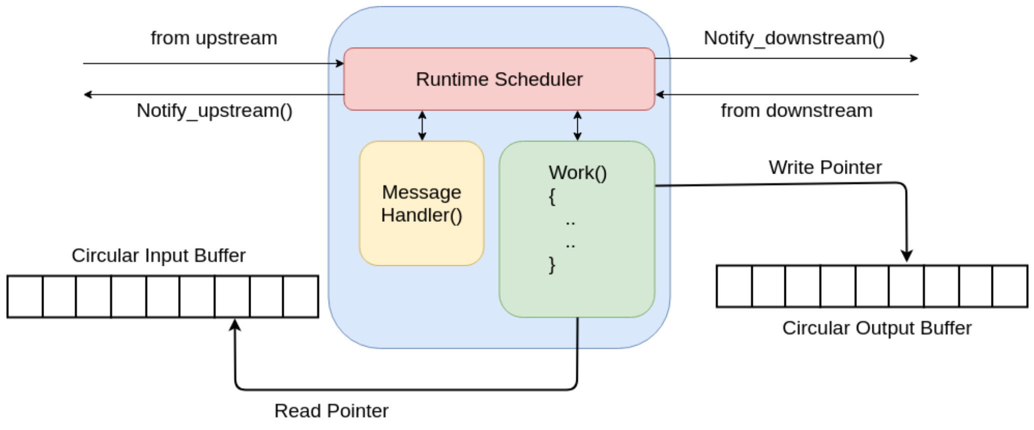

Figure 2.6: Architecture of GNU Radio block

has a single input buffer and a single output buffer. The *work* function describes the implementation of the signal processing algorithm. On starting execution, the *work* function accesses data from the circular input buffer, which is same as the output buffer for the upstream block. Both the upstream block and the current blocks maintains a pointer to the last used data element position. Once completion of execution, the upstream block writes new elements to the input buffer of the current block and updates its the write data pointer. The upstream block notifies the current block using the notify_ downstream() method that new input data elements have been written. The scheduler for the current block checks if there is sufficient new data for a single execution. It then starts the execution of the current block once the input buffer has sufficient data. On successful execution, it updates the read data pointer, writes the data elements into the output buffer and updates the write pointer.

Since, most filter design rely on history of previous inputs, the scheduler also notifies the upstream block on reading the data from the buffer to notify that its output might have been modified. The current block scheduler notifies the downstream block that new data elements are available when it writes to the output buffer. The *message handler* function works similarly, in this case the upstream block scheduler uses notify_msg() method for signaling that a new message may be available.

The flow graph describes the flow of data between different blocks. Flow graphs make it easier to design complex signal processing algorithms by combining simpler blocks. This provides modularity and scalability to the algorithm development process. Generally blocks are designed in C++ to enable fine grained control and faster execution. Python's QT framework defines a signals and slot mechanism for communicating events between different objects. Slots are instantiation of C++ objects, which in this case are the the processing blocks. When the internal state of the block is changed, it emits a signal which notifies other slots an event has occurred. This mechanism is used for describing the flow graphs.

Scheduler The scheduler is the control unit for the flow graph. At initialization, the scheduler allocates the buffers and instantiates each block in its own thread. At runtime, it does memory management for each block, determines the requirements that are set by the block such as number of items to be processed in one execution, alignment of data in the buffers etc. Once the requirements are satisfied, it passes the read and write pointers to the *work* function and starts the one execution. Once the *work* function finishes its execution, the scheduler takes in the returned information and updates the state of the block and the appropriate pointers.

2.2 LimeSDR-USB

The SDR platform used in this project is LimeSDR-USB. It follows the architecture of the SDR platform shown in Figure 2.2. The technical specifications of LimeSDR-USB and the component description in reference to Figure 2.2 has been summarized in Table 2.2. In the next subsections, the report discusses the hardware and software architecture of LimeSDR-USB.

Feature	Description
Software Configurable RF Transreceiver	LMS7002 MIMO Field Programmable RF (FPRF)
FPGA	Altera Cyclone IV EP4CE40F23
Bus Controller	Cypress USB 3.0 CYUSB3014-BZXC

Table 2.2: LimeSDR-USB specifications

2.2.1 LimeSDR-USB Hardware Architecture

Figure 2.7 shows the block diagram of a LimeSDR-USB board. For the sake of simplicity the diagram shows the major components, namely the LMS7002M FPRF, FPGA and the Cypress FX3 Bus Controller. Other components will be introduced in correspondence to their application with these major components.

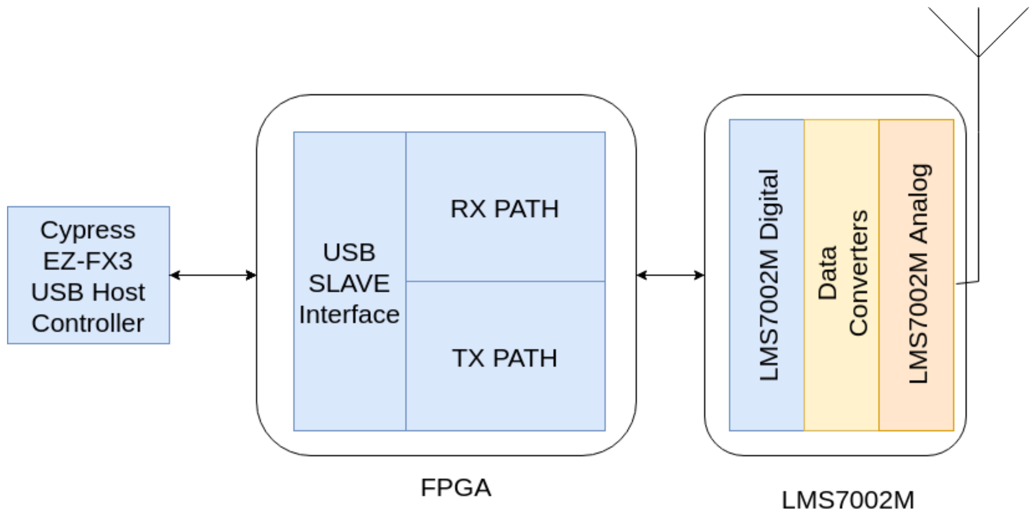

Figure 2.7: Block Diagram of LimeSDR-USB.

LMS7002M

LMS7002M is a fully intergarted FPRF transreceiver providing 2×2 Multiple Input Multiple Output. (MIMO) functionality. It provides continuous coverage in the 100kHZ- 3.8GHz frequency range, with on chip data converters providing 160 MHz RF bandwidth. It is designed for a broad range of applications, ranging from broadband wireless communication, cellular communications, SDR applications etc. The hardware mainly consists of three main sub-components: analog processing chain, data converters and digital processing chain.

LMS7002M offers full duplex communication link on both the TX and RX chains. Each of the RX chains has three separate RF ports tuned for narrow band low frequency, narrow band high frequency and wide band operations. Similarly, the TX Chains are connected two separate RF ports tuned for high frequency and low frequency operations. This separation is done for better impedance matching at the boundary of the antennas.

Figure 2.8 shows the functional block diagram for a LMS7002M FPRF. Since, both the RX and TX paths are identical, the report concentrates on only one RX path. The output from the RF RX ports are fed into the LNA inorder to minimize injecting too much noise at the beginning of the chain. The receiver follows the architecture shown in Figure 2.3, with a RX mixer, followed by filter and a Programmable Gain Amplifier. (PGA) combined in a Zero-IF architecture. The RX PGA outputs the analog baseband signal.

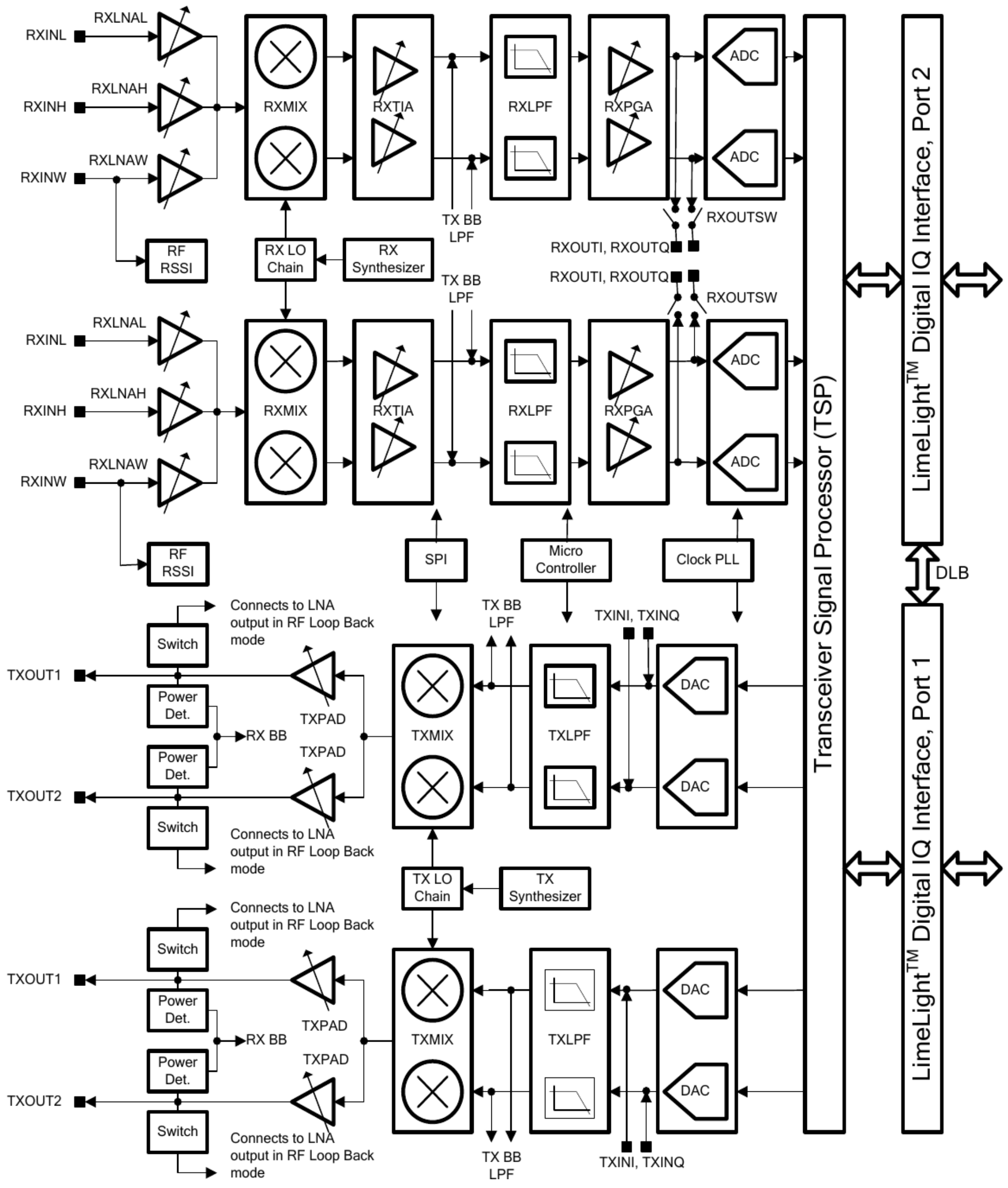

Figure 2.8: Block Diagram of LMS7002M.

LMS7002M uses a fractional N-Phased Lock Loop (PLL) architecture for the local oscillator frequency synthesis. PLL is used extensively in RF circuits for making sure the generated local oscillator signal and the reference signal have the same phase and frequency. PLLs are essentially negative feedback systems, so when the input signal differs a lot from the output signal, the control logic tries to lower the error (*input-output)*.

Integer N- PLL architectures are used to generate high frequency signals from low frequency reference clocks, by using a frequency divider in the negative loop-back path. The frequency divider is basically a counter, that outputs every "N" (division factor of the loop) clock cycles of the output signal. But since the output signal frequency will be multiples of the reference clock, the output signal resolution is determined by the reference clock.

So to have a high frequency as well as a high resolution, the divider counter should be very large in size. To counter the problem, fractional N-PLL architectures were designed where the output signal frequency can also be a fractional multiple of the input signal frequency. This helps in increasing the frequency resolution without the need for a large divisor counter. The input and output frequency relationship for a fractional N-PLL can be summarized by: $f_{out} = f_{ref}(N + k/M)$, where N is the integer divider factor, k is the fractional divider factor and $1/M$ gives the output frequency resolution. Both the integer and fractional divider factor are determined by the size of the counters used. In case of LimeSDR, the reference signal fed to the PLL varies from to 10 to 52 MHz. The output signal can vary from 30 to 3800 MHz, with a frequency resolution of 24.8 Hz.

Once the RF demodulation is completed by the analog processing chain, the analog signal is sent to the data converters and converted to digital data samples. The sampling rate for the data conversion is determined by the required RF channel bandwidth. The digital samples are sent to the Transreceiver Signal Processor (TSP) for further processing.

The TSP uses advanced signal processing algorithms like IQ DC offset correction, IQ phase correction for correcting the received samples. An interpolation and decimation filter is added to the TSP for the TX and RX chains respectively. These filters are implemented with a chain of five fixed co-efficient half band FIR filters, which allows interpolation and decimation factors of 1,2,4,8,16. Interpolation and Decimation allows the baseband to run at a lower data rate while still running the data converters at higher sampling rates, enabling the quantization noise to be spread over larger frequency range. Automatic Gain Control is also implemented by the the TSP.

LMS7002M interfaces can be segmented into control interfaces and data interfaces. The control interfaces are used for initialization, calibration and on the fly reconfiguration of the LMS7002M parameters. The data interface is used for exchanging In-Phase Quadrature. (IQ) samples with the baseband modem. For the data interface, LMS7002M uses the LimeLight interface which implements a 12 bit JESD Double Data Rate (DDR) interface for each RX/TX chain as shown in Figure 2.8. The LMS7002M has a on-chip micro-controller which can be used for configuration and control of the LMS7002M chip. It also provides a Serial Peripheral Interface. (SPI) interface for offloading the control and configuration functionality to the baseband modem.

FPGA

The LMS7002M is designed to stream data continuously, whereas the Cypress FX3 uses USB 3.0 protocol transmits packets of data. LimeSDR-USB uses an Altera Cyclone IV FPGA to buffer the streaming data, converts them to packets and adds meta-data to each packet as recommended by Nychis et.al [5]. The architecture of the FPGA data path blocks is shown in Figure 2.7. The TX path is responsible for moving data from the USB interface to the LMS7002. The RX Path on the other hand controls the reception of samples from the LMS7002M and subsequent sending to the Host-Computer via the USB interface. The FPGA also has on board PLLs which are configured to be the same as the sample clock

The FPGA interfaces are designed to handle the segregation of control and data paths by LMS7002M. The data paths (TX path and RX path) uses a 12-bit parallel interface to stream data to and from the LimeLight interface of the LMS7002M. The control path of the FPGA has a NIOS processor which interacts directly with the USB Slave Interface and controls the RF parameters through an SPI interface.

The FPGA RX path converts samples to packets which takes significant buffering time. As this report concentrates on study of timing delays, it is necessary to take a closer look at the FPGA implementation and understand the buffering process. In contrast, FPGA TX path converts packets to samples and there is no need for buffering. So in the next paragraph, the report describes the FPGA RX path in detail.

RX Path: VHSIC Hardware Description Language (VHDL) entities are primary level of Register Transfer Level (RTL) abstraction, they define the hardware functionality in response to input signals. VHDL signals on the other hand, carry

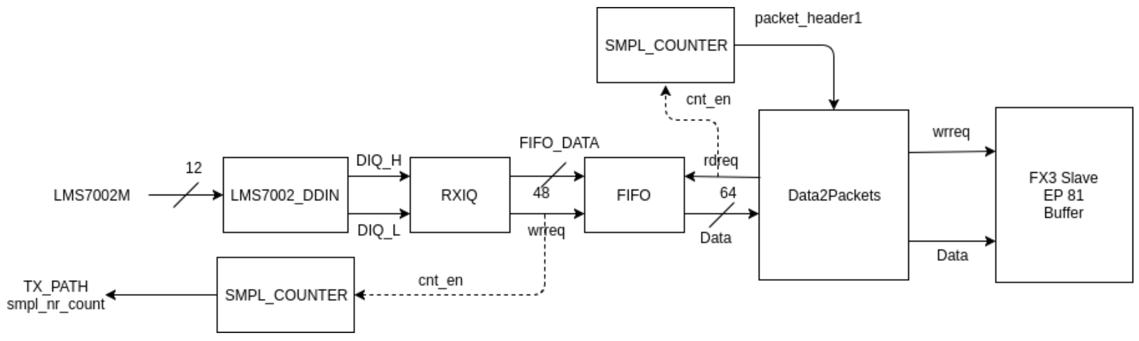

Figure 2.9: LimeSDR FPGA RX Path

information from one entity to another. This paragraph uses quoted text and italics to represent entities (blocks) and signals respectively.

Figure 2.9 shows the overview of the RX Path implementation on the LimeSDR-USB FPGA. Sample data from the LMS7002M is captured by the "LMS7002_DDIN", which uses an ALT DDIO IP block to capture in-phase and quadrature samples at double data rate. This means that incoming data stream is latched both at the positive and negative edges of the clock. "LMS7002_DDIN" separates the sample data collected at the postive and negative edge into two different components: *DIQ_H* and *DIQ_L* repectively. It introduces one clock delay between the input,LMS7002M Sample Stream, and the *DIQ_L*, *DIQ_H* outputs.

The "RXIQ" block is responsible for arranging the individual components into a single data stream of In-phase component of a sample followed by its Quadrature component. It takes four component data coming from the "LMS7002_DDIN" block and arranges arranges them into the sequence *DIQ_L*,*DIQ_H*, *DIQ_L*, *DIQ_H*. "RXIQ" takes one clock cycle for latching the inputs. It takes three clock cycles for producing the structure. Finally, the output is latched after one clock cycle from the internal output register. It writes this structure every two clock cycles to the "FIFO" by enabling the *wrreq* signal. This means effectively one IQ sample is then sent to the "FIFO" every clock cycle.

The first "SMPL_COUNTER" is a 64-bit counter which increases its count every time "RXIQ" asserts the *wrreq* for writing new data samples to the "FIFO". The count is sent to the TX Path to track the number of produced samples. The second "SMPL_COUNTER" is also a 64 bit counter which is enabled when the data is read from the "FIFO" by the "DATA2PACKETS" block. It increases its count every clock cycle when the *rdreq* signal is '1'.

The "FIFO" is implemented using the same read and write clocks, generated by the FPGA RX PLL. The read enable is controlled by the "RXIQ" using the *wr-req* signal, so two new samples are written in two clock cycles of the RX PLL. On the write side, the "DATA2PACKETS" controls the write enable signal (*rdreq*). The "FIFO" receives 48-bit data samples, stores it a First In First Out (FIFO) structure and keeps track of how many samples have been loaded into the FIFO. The "FIFO" takes one clock cycle for latching the input. The counter for the number of elements is updated five clock cycles after the data has been latched.

The "DATA2PACKETS" block is responsible for converting data samples into packets. It mainly consists of two Finite State Machine (FSM). The first FSM controls the read and write signal for the input and output blocks. It monitors the amount of data in the "FIFO", and when it is greater than the amount of data in one packet it asserts the *rdreq* signal. It takes two clock cycles for the "DATA2PACKETS" to register that the number of data elements in the "FIFO" is sufficient for one FPGA packet. The first FSM takes two more clock cycles to generate the *rdreq* signal after that.

The second FSM arranges the data in the FPGA data packet structure (Figure 2.12). It latches the value of the sample count from the second "SMPL_COUNTER" when the first FSM asserts the *rdreq* signal. It adds the sample count and different flags as meta-data for each FPGA packet. Once, the first FSM determines there is enough space to write the data in the FX3 buffer, it enables the *wrreq* signal and writes 64 bits of data every clock cycle. The second FSM takes 6 clock cycles to output the first data element read from the "FIFO" after the *rdreq* has been generated by the first FSM.

RX Path Delays The delays introduced by the individual blocks has been highlighted in the previous paragraph.

- Delay introduced by the "LMS7002_DDDIN" block : 1 clock cycle
- Delay introduced by the "RXIQ" block: 5 clock cycles ; 1 clock cycle each for input and output latching, and 3 clock cycles for the structure formation.
- Delay introduced by the "FIFO" block: 6 clock cycles ; 1 clock cycle for input latching; 5 clock cycles for the increment of internal counter

- Delay introduced the the "DATA2PACKETS" block: 10 clock cycles ; 4 clock for generation of *rdreq* signal and 6 clock cycles for output of the first data element read from the "FIFO"

The report will refer to these block delays as constant delays. The constant delay is 22 clock cycles for even index samples, and 21 clock cycles for odd index samples. It can be summarized as:

$$sampl_{rel} = sampl_{abs} \; mod \; N \tag{2.1a}$$

$$packet\_number = \left\lfloor \frac{sampl_{abs}}{N} \right\rfloor \tag{2.1b}$$

$$\Delta_{constant} = \frac{22 - \lfloor sampl_{rel} \; mod \; 2 \rfloor}{f_s} \tag{2.1c}$$

N = Number of samples in one packet; N= even, f_s is the FPGA RX PLL frequency, $sampl_{rel}$ is the relative sample index and $sampl_{abs}$ is the absolute sample index in Equation 2.1

For example, with N=1020, $sampl_{abs}$ = 8000, packet_number would be $\lfloor \frac{8000}{1020} \rfloor =$ 7, and $sampl_{rel}$ would be $8000 \; mod \; 1020 = 860$.

In addition to the constant delays, there are two more delays that need to considered. They are Queuing Delay and the Streaming Delay. The data elements in the "FIFO" has to wait until there is sufficient data for a single packet. This waiting time is being referred to as Queuing Delay. The Queuing Delay is dependent on the absolute sample number. If the element is the N^{th} sample, it has to wait for N clock cycles, where N is the number of samples in one packet. If the element is the $N - 1^{th}$ sample, it needs to wait only for one more cycle, that is two clock cycles.

$$\Delta_{queuing} = \frac{N - 2 \times \lfloor \frac{sampl_{rel}}{2} \rfloor}{f_s} \tag{2.2}$$

In Equation 2.2, N = Number of samples in one packet, f_s is the FPGA RX PLL frequency, and $samp_{rel}$ is defined by equation 2.1a.

The "DATA2PACKETS" block outputs two samples in one clock cycle, hence the data will be streamed out sequentially. The time a element has to wait to be streamed is being referred as the streaming delay. The streaming delay is also dependent on the arrival time of the sample which is equivalent to the absolute sample number. If the element is the N^{th} sample, it is the first data element of a packet, it doesn't need to wait for any sample to be streamed ahead of it, so it has zero streaming delay. On the other hand, the $N-1^{th}$ sample needs to wait for N-3 samples to streamed first, before it is outputted together with the $N-2^{th}$ sample.

$$\Delta_{streaming} = \frac{\left\lfloor \frac{sampl_{rel}}{2} \right\rfloor}{f_s} \tag{2.3}$$

In Equation 2.3, N = Number of samples in one packet, f_s is the FPGA RX PLL frequency, and $samp_{rel}$ is defined by equation 2.1a.

The total delay can be calculated as :

$$\Delta_{total} = \Delta_{constant} + \Delta_{queuing} + \Delta_{streaming} \tag{2.4a}$$

$$\Delta_{total} = \frac{22 + N - \left\lfloor \frac{sampl_{rel}}{2} \right\rfloor - \lfloor sampl_{rel}\ mod\ 2 \rfloor}{f_s} \tag{2.4b}$$

Cypress EZ-FX3

LimeSDR-USB uses a Cypress EZ-FX3 as Universal Serial Bus (USB) 3.0 peripheral controller. It has a fully configurable, general programmable interface called the General Programmable Interface (GPIF) II which allows integration with any processor like ASIC, FPGA, Image Sensors etc. It also provides low speed interfaces like Inter-Intergrated Circuit (I2C), SPI for the low-speed Input Output (IO) operations.

The hardware architecture of EZ-FX3 is shown in Figure 2.10. The "ARM926EJ" is a 32-bit microprocessor operating at 200 MHz. It is responssible for the configuring and controlling the distributed Direct Memory Access (DMA) controller and the peripherals. The "USB" block is a USB 3.0 peripheral controller. It implements the USB 3.0 Physical Layer, and is responsible for handling the communication with the Host Computer. The "GPIF interface" block handles the data streams to and from the FPGA of the LimeSDR-USB. It implements a Synchronous Slave FIFO Interface which is controlled by "Slave FIFO" block of the LimeSDR FPGA.

For the purpose of this project, the implementation details for the USB and GPIF II interface needs to be explored further. The next two paragraphs, concentrates on presenting the concrete understanding of these two important interface in the context of LimeSDR-USB.

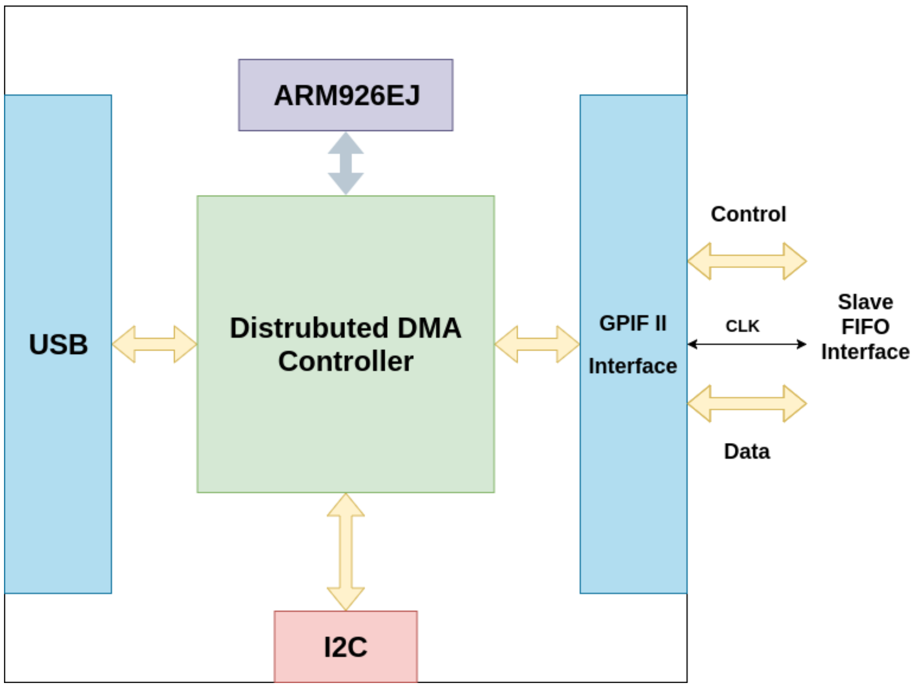

Figure 2.10: EZ-FX3 architecture

LimeSDR USB endpoints USB endpoint are buffers on the USB device. They are logical abstraction for having multiple parallel data streams use a single physical channel. The LimeSDR-USB uses four different endpoints for the USB data transfer, these endpoints are divided into control and data endpoints for both the input and output directions. The control endpoints are used for configuring and retrieving data from the LMS7002M, whereas data endpoints are used for streaming data to and from the LMS7002M data converters through the FPGA. The different endpoint address and their associated function in LimeSDR-USB is shown in Table 2.3

Endpoint address	Function
0x01	Stream Data Output
0x81	Stream Data Input
0x0F	Control Data Output
0x8F	Control Data Input

Table 2.3: LimeSDR USB transfer endpoints

The USB protocol specifies different transfers mechanisms depending on the requirement of the reliability and deterministic transfers. Bulk Transfers are used for large bursty data. They don't have any guarantees on the bandwidth and are transmitted when there is bandwidth available after other types of transfers. They offer guaranteed delivery of data.

The LimeSDR-USB uses Bulk Transfers for the control transfers. It sends bursts of data which are moved in the endpoint buffer of the Host Computer USB Controller which transfers it to the endpoint point buffer of the "USB" block shown in Figure 2.10.

For the data transfers, the LimeSDR-USB uses bulk streams. USB 3.0 protocol introduced stream pipes where a single USB endpoint can be allocated to multiple data streams. Each stream endpoint has their associated Stream ID which is used for tagging the data in the USB endpoint buffer. Each of the data packets in the endpoint buffer are transmitted as a regular bulk transfer and the USB Controller on the other end decodes the Stream ID and sends it to the appropriate Stream endpoint.

USB endpoints also specifies the size of the data packet. For USB 3.0 the maximum data packet size is 1024 bytes. If the data sent to the USB Controller is greater than that, it segments that into multiple data packets and sends them as multiple bursts of data in a single USB Transaction.

GPIF II The GPIF II is a programmable state machine, that provides the flexibility of implementing a custom interface. The GPIF II segments the functionality into control and data intefaces. For the data interface, the LimeSDR-USB defines a 32-bit interface running at 100 MHz. The state transitions of the state machine are based on the input control signals from the GPIF II interface. The output control signals are driven by the state transitions of the internal state machine.

LimeSDR-USB implements a Slave FIFO interface using the GPIF II interface. The Slave FIFO interface allows the FPGA to perform read/ write operations on the EZ-FX3 internal FIFO buffers directly. The control interface is used for addressing the FX3 FIFOs and signaling read and write operations. In addition, the control interface has flag signals to indicate events to the FIFO Slave Interface. The 32-bit data interface transfers the data according to the signaling and addressing used on the control interface.

The distributed DMA controller uses sockets and threads for allowing data transfer between the FX3 RAM and the peripherals. They are explained in the next paragraph to help understand the Slave FIFO interface in details.

Sockets and Threads Sockets are connections between a FX3 peripheral like GPIF II, I2C and FX3 RAM. The FX3 microprocessor initializes DMA buffers, which are used for intermediate storage of data transferred through the FX3. The microprocessor keeps the address and size of these DMA buffers in a linked list like structure. The elements of this structure are referred to as DMA descriptor. The sockets are implemented as a structure with a pointer to a DMA descriptor and interrupt flags.

The sockets can signal other sockets automatically without CPU intervention or signal the CPU through interrupts. Automatic signaling is used when the FX3 CPU does not need to modify any data in the data stream. The socket which is writing data to the buffer is called producer socket and the socket reading data from the buffer is called consumer socket according to the terminology defined by (refer document.)

One producer socket and one consumer socket accessing the same DMA buffer can be encapsulated in a configuration referred to as DMA channel. The DMA channel can have multiple DMA buffers. Each DMA buffer has its own empty/full flag to signal the producer and consumer sockets. For example, if a 1024 bytes socket is allocated to the DMA channel, when the producer sockets write 1024 bytes, the full flag will be enabled. In case of LimeSDR-USB, two DMA channels are defined: TX DMA channel and RX DMA channel, with a GPIF II socket and USB socket as producer and consumer sockets. Each of channel have multiple DMA buffers, and each buffer is allocated 4096 bytes.

The sockets are internal to the FX3, so for external devices to access the DMA buffers, GPIF threads are defined. They connect sockets with external pins, like for Slave FIFO interface, they connect the GPIF II interface with GPIF II sockets.

2.2.2 LimeSDR USB Software Architecture.

In this subsection, the report explains the driver of the LimeSDR-USB. The software architecture of the LimeSDR-USB is shown in Figure 2.11.

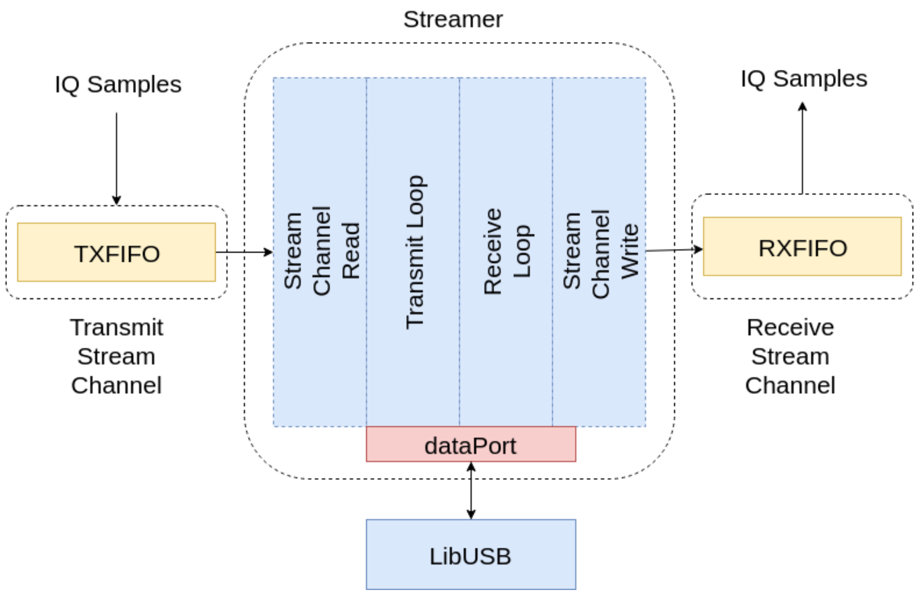

Figure 2.11: LimeSDR USB software architecture

The "Streamer" block is the main software component responsible for interacting with the LimeSDR-USB. LimeSDR defines "dataPort" as an abstract interface, whose methods can be mapped to one specific interface implementation like Peripheral Component Interconnect Express (PCIe) or USB depending on the configuration defined by the "Streamer" block. In case of LimeSDR-USB, the dataPort is configured to use USB bus. For interacting with application software like GNU Radio "Stream Channels" are defined. They are basically FIFO's with control logic for receiving or sending data from the application software. To understand the functionality of the "Streamer" block, let's take a look at the TX data path. The RX Path is vice-versa of the TX Path.

TX Data Path The application software configures the TX Stream Channels through the Lime Application Programming Interface (API). Then the TX FIFO's are initialized and the control parameter like batchsize(shown in Figure 2.12) of the FPGA packets and the data format for the data to be stored in the FIFO is sent to the "Streamer". The Streamer parses these information and configures the hardware data path with the provided configuration. The "Streamer Transmit Loop" initalizes internal buffers which are used for sending data through the "dataPort".

The application software pushes data to the TX "Stream Channel" shown as

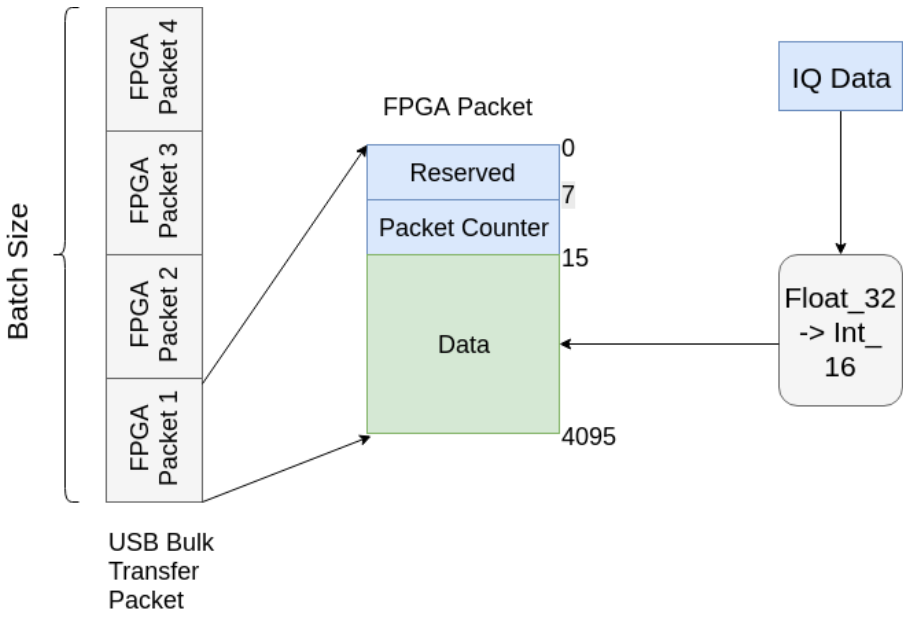

Figure 2.12: Streamer Transmit Loop

"IQ Data" in Figure 2.12. The "Streamer Transmit Loop" runs continuously and once there is data in the "Stream Channel" FIFOs it reads the data using the "Stream Channel Read" function. It converts the data, for example from 32-bit float data to 16 bit integer data, and copies the data to its internal buffers. It then packs the data into a FPGA data packet format shown as the middle component of Figure 2.12. Each FPGA data packet contains flags and packet counter field followed by 4080 bytes of data. These FPGA data packets are batched together as specified by the batchsize. This batch is sent to the USB Host Controller to be transmitted to the LimeSDR-USB using libusb.

TX Control Path The LimeSDR-USB is configured at initialization using the lms7 API. The lms7 API provides an abstraction layer for configuring the LimeSDR-USB. It provides the means to control the LMS7002M, Si5351 clock controller through the NIOS SPI interface. The lms7 API packs the control data into LMS64 control packets shown in figure 2.13. One LMS64C protocol packet (Figure 2.13) is maximum 64 bytes. If the data to be sent is larger than that the data field is segmented into several packets. The "LMS64C_Protocol" keeps a packet buffer and adds the prepared packet to this buffer. These packets are then sent to the "connectionFX3" module which writes it to the control endpoint using the "data-Port".

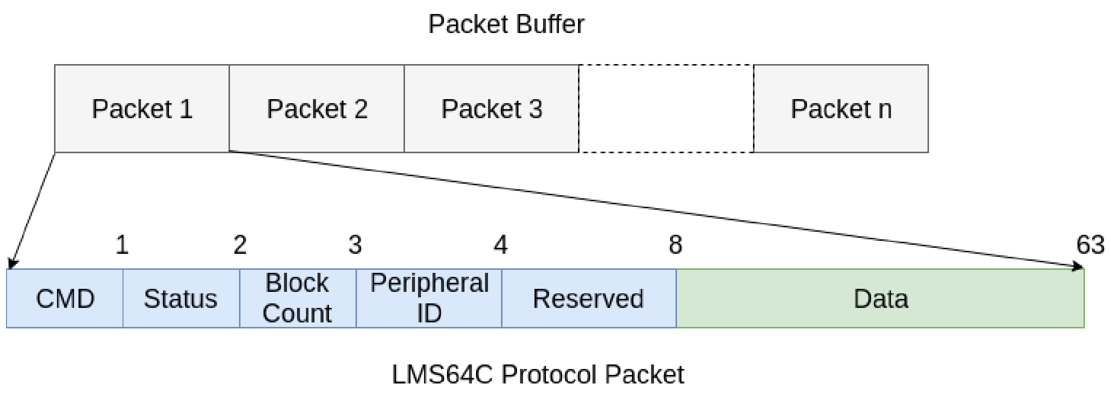

Figure 2.13: LMS Control Packet Structure

2.3 Wime Project

Wime Project was created with the objective of providing experimentation as a wireless communication system performance evaluation tool. The project focuses on evaluation of physical layer strategies using USRP SDR platform. It provides a 802.15.4 testbed which is inter-operable with the the off-the shelf TelosB nodes running the Contiki operating system.

2.3.1 IEEE 802.15.4

The IEEE 802.15.4 is a Low-Rate Wireless Personal Area Network (LR-WPAN) specification aiming to be a ultra low-complexity, ultra-low power and low datarate specification. It specifies the PHY and MAC for LR-WPANs. It forms the basis for network protocols like IPV6 over Low-Power Wireless Personal Area Network (6LoWPAN), Zigbee, WirelessHart. It is intended to provide a data-rate of 250 kbps at 10 meters communication range.

The IEEE 802.15.4 network topologies provide ways in which devices can talk to different nodes in the network. The main two network topologies used in IEEE 802.15.4 are: star and mesh. In star all the devices communicate to one another through the central device called the Personal Area Network (PAN) coordinator. In mesh, the devices can talk to one another without the need for a central coordinator.

These network topologies helps in defining the role and type of device. Reduced Function Device (RFD) can communicate without routing functionality so they usually are the end devices in a network. Full Function Device (FFD) can route information in addition to regular communication. Coordinator is a special FFD which sets up the network and acts as the manager of the network.

The IEEE 802.15.4 PHY specifies defines different radio channels and modulation techniques for these network nodes. It specifies Offset Quadrature Phase Shift Keying (O-QPSK), Binary Phase Shift Keying (BPSK), Amplitude Shift Keying (ASK), Chirp Spread Spectrum (CSS) etc. as physical layer modulation techniques. The main frequency band of interest is ISM band at 2.4 GHz where 20 channels are available globally. The channels are spaced at 5 MHz, having channel bandwidth of 2 MHz.

The MAC layer offers handshake for reliability. CSMA with collision avoidance, TDMA with synchronization beacons and Guaranteed Time Slot (GTS) are defined as methods of medium access. There are four different MAC frames defined for different unique functions: Data, Acknowledgement, Beacon and MAC Command. Data and Acknowledgment frames are used for data communication, whereas Beacon and MAC command are used for network maintenance.

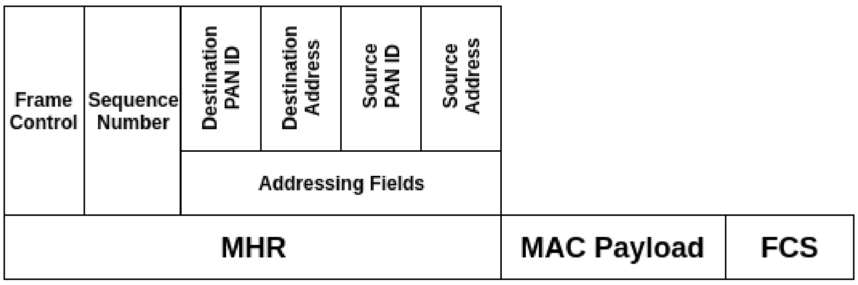

Figure 2.14: MAC Data Frame Structure

The relevant MAC data frame structure has been shown in Figure 2.14. The function of each field is described below:

- **MHR** defines the MAC data frame header. It consists of the following subfields:
  - **Frame Control** specifies the type of the frame and how the rest of frame is structured like the source and destination addressing modes.
  - **Sequence Number** specifies the sequence identifier for the frame

  - **Destination PAN ID** is an unsigned integer which is the unique PAN ID for the destination node.
  - **Destination Address** is the MAC address of the intended recipient.
  - **Source PAN ID** is the PAN ID of the sender.
  - **Source Address** is the MAC address of the originator of the frame.

- **MAC Payload** is the link layer frame which includes the data payload.
- **FCS** is a 16-bit ITU-T Cyclic Redundancy Check (CRC) calculated over the MHR and MAC Payload fields.

Since, O-QPSK modulation is being used in the Wime project, the next paragraph elaborates on the O-QPSK modulation.

O-QPSK The O-QPSK PHY specifies the PHY Data Packet as shown in Figure 2.15.

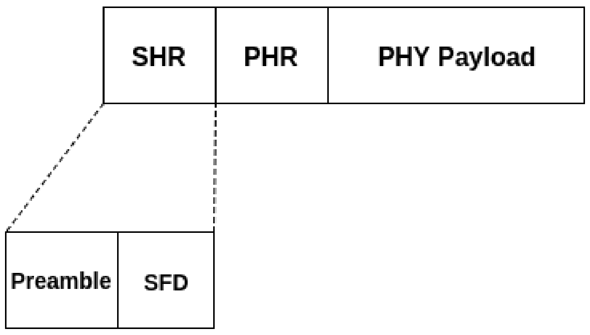

Figure 2.15: O-QPSK PHY Packet

- **Synchronization Header (SHR)** is used for indicating the start of the frame. It is composed of two sub subfields : Preamble and Start-of-Frame Delimiter (SFD). For the O-QPSK, the preamble is four bytes with decimal value of "0000". The SFD is one byte long as is formatted as "A7" in hexadecimal.
- **PHR** is the header for the physical layer packet and it specifies the frame length.
- **PHY Payload** is the data payload, which for the purpose of this project is the MAC data frame. The PHY payload has a maximum size of 127 bytes.

O-QPSK PHY uses 16-ary quasi orthogonal modulation. The data packet is converted to data symbols, the symbols subsequently are converted to chips, which are modulated onto the carrier signal. The octets of the PHY data packets are converted into symbols starting from the Preamble and ending with the last octet of the PHY Payload. The least significant bits of an octet is mapped to one symbol and the most significant bits mapped to another symbol.

Each symbol is mapped to a 32-bit Pseudo Noise (PN) sequence, which are related to each other through cyclic shifts and/or conjugation. The even indexes of the chip sequence are modulated onto the In-Phase component of the carrier as shown in Figure 2.16. On the other hand, the odd indexes of the chip sequences are modulated onto the Quadrature component of the carrier. The chips are represented by a half-sine pulse shape shown in equation 2.5

$$p(t) = \begin{cases} sin(pi \times \frac{t}{2T_c}) \ ,0 \geq t \leq 2T_c \\ 0 \ , otherwise \end{cases} \tag{2.5}$$

$2T_c$ is the period of the half sine wave pulse in equation 2.5. The quadrature component is delayed by T_c as shown in Figure 2.16.

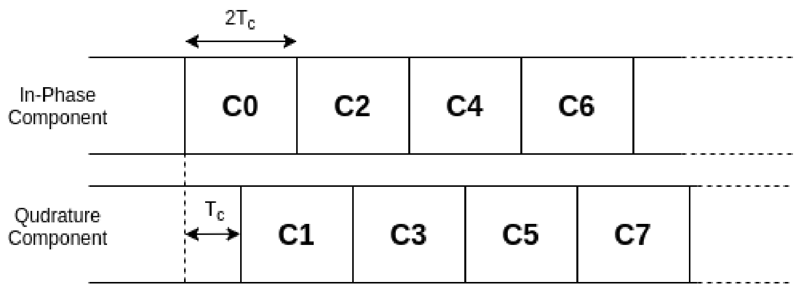

Figure 2.16: In-Phase and Quadrature Chip Sequences

2.3.2 Wime Project GNU Radio Implementation

Wime Project is build on the UCLA 802.15.4 PHY implementation designed by Thomas Schmid. It ports the existing implementation to the modern GNU Radio software framework. It furthermore extends the PHY implementation with a MAC layer and adds Rime Stack as a network layer. The implementation details of the PHY layer and the MAC layer is described below.

- **Modulation** The Wime MAC works with asynchronous mesages, whereas the PHY is implemented with stream interfaces. The GNU Radio modulation flow graph is shown in Figure 2.17. At first, these asynchronous MAC messages are converted to a tagged stream by the "PDU to Tagged Stream block".

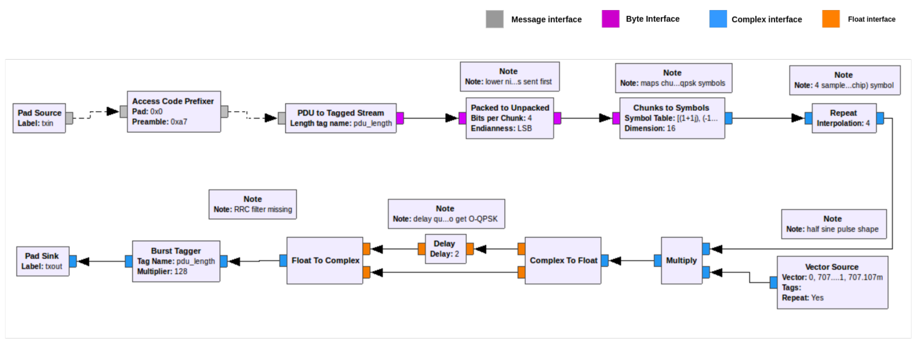

Figure 2.17: GNU Radio Modulation flow graph

The individual octets of the stream are then converted to symbols as described previously in the O-QPSK paragraph by the "Packed to Unpacked Block". Instead of converting the symbols to chips directly and then segmenting into In-Phase and Quadrature components as in UCLA PHY. Wime Project implementation directly converts a symbol through a symbol table directly into a sequence of complex in-phase and quadrature values. Each symbol input results in 16 complex chips. These individual chips are interpolated by repeating a single chip four times by the "Repeat" block. The repeated chips are then multiplied with a sine function at different phases, differing by $\pi/4$. This process makes a single chip to be four samples wide. The quadrature component is delayed by two samples to satisfy the offset requirement (T_c).

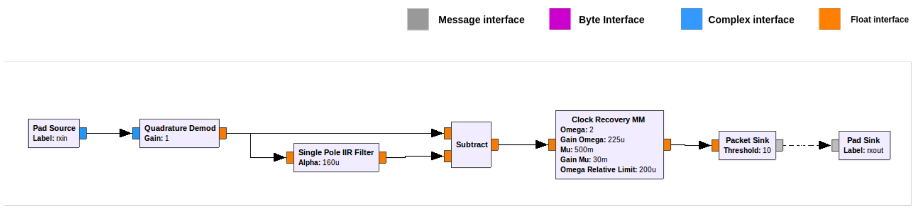

Figure 2.18: GNU Radio Demodulation flow graph

- **Demodulation** The data stream from the USRP is passed through a "Quadrature Demodulation" block as shown in Figure 2.18. It does FM demodulation of the received signal stream. "Clock Recovery" module takes this demodulated signal stream and performs decimation first, followed by Mueller and Müller discrete-time error-tracking synchronizer. The "Clock Recovery" block outputs the chip sequences obtained from the incoming signal stream. These chip sequences are then sliced and converted to symbols by the "Packet Sink" block. The decoded valid message is then outputted to the Wime MAC.

- **MAC** The Wime Project adds a very simple MAC which encapsulates upper layer packets with the MAC header and adds the FCS. When receiving messages from the PHY, it does a validity check on the received message. On successful validation of the received message, it strips away the MAC header. The MAC layer transmits the message immediately without any carrier sensing.

2.4 Tools Used

2.4.1 USBMon

It is kernel facility provided to collect I/O traces on the USB Bus[9]. 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.

Text Data Format

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 2.4: Text USB Trace Example.

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

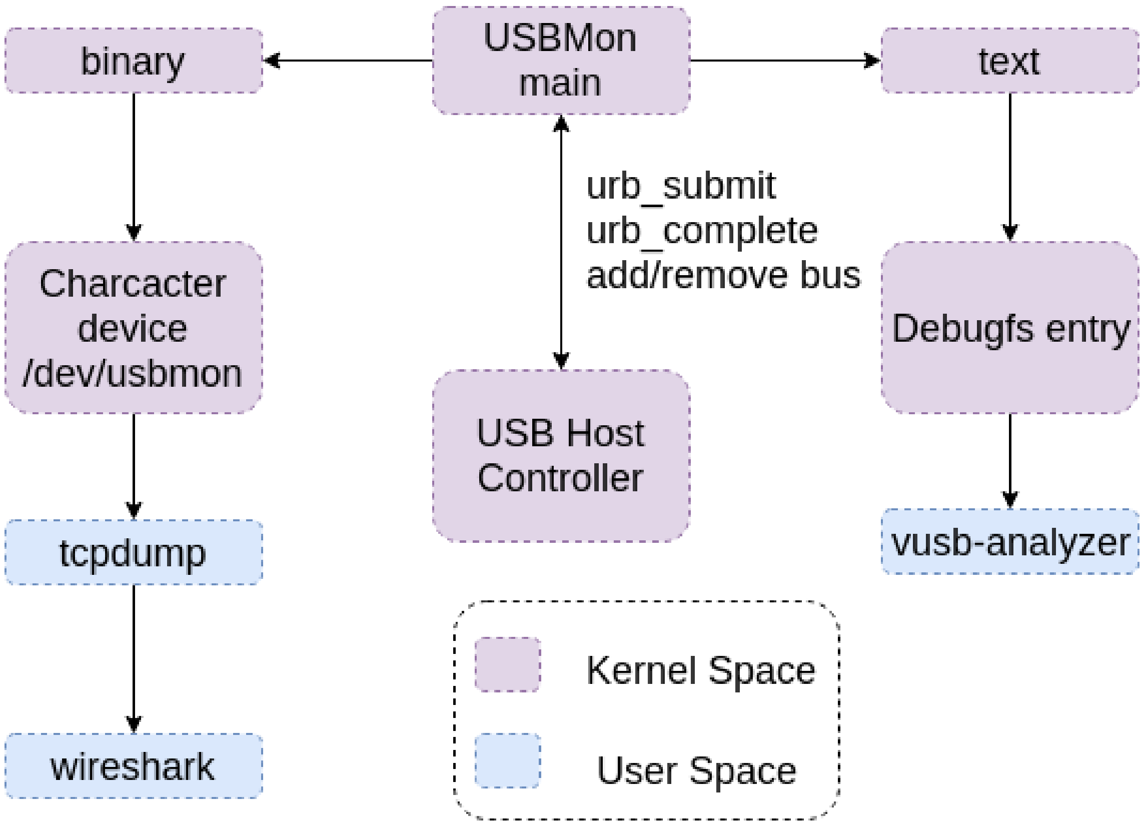

Figure 2.19: USBMon Architecture(Adapted from [1]).

- **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 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[8]:

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 2.5: URB Type and Direction.

  - **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.

- **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.

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 */
};
```

2.4.2 pidstat

Pidstat is a Linux tool provided the sysstat package. It can be used to monitor the resource usage of a Linux process/processes. It can be configured to collect information on CPU utilization, memory utilization, context switches and the time used by the process at application level and the time used in kernel level. It collects the information at one second intervals and the time duration for one collection can be specified during the process invocation.

Chapter 3

Literature Study

Loren Ipsum

Chapter 4

Methods

This chapter introduces the system architecture followed by the experimental designs for the quantitative analysis. The system architecture describes the GNU Radio flow-graph description and the software and hardware used in this project. The experiments are presented in chronological order. In the first experiment, the performance of the system is measured with respect to broader parameters like sampling rate, data payload size and the number of message sent per second. The second experiment was designed was look at the impact of these parameters on the the different subsections of the data path. Then, the experiments are more focused on the USB Bus transfer. In experiment 3, the project looks at the impact of the size of the bus transfer on the overall delay. Finally the modifications introduced in hardware and software are introduced. It is followed by the the experimentation on the impact of finer bus transfer size and the GNU radio buffer sizes on the overall round trip times and how the process utilizes the system resources.

4.1 System Architecture

The experimental system is primarily based on the WIME project implementation of 802.15.4 protocol. It is adapted to use the LimeSDR board instead of USRP as the SDR platform. The adaptations can be grouped as

1. GNU Radio blocks
2. 802.15.4 PHY layer
3. Periodic Message Source

GNU Radio Blocks For using the LimeSDR platform the USRP Sink and Source blocks are replaced by the grlimesdr project source and sink blocks. Another alternate used in this project is the gr-osmosdr project sink and source blocks which used soapysdr to access the Lime API. The former was chosen as it directly interacts with the LimeAPI without using the adaptation layer presented by the soapysdr project. This gives much better control of the board control parameters and also saves subsequent memcpy operations used by the soapysdr glue layer. (**Maybe present the RTT values for using the two different blocks and show that the limesdr block is better.**

)

802.15.4 PHY layer The WIME project PHY layer has been designed to only work with 4 MHz as the sampling rate, in this project, the PHY layer has been modified to accommodate different sampling rates. **Describe the change**

Describe the timing probe

Periodic 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 4.2.1 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.

The block notes down the global system time as T_1 when it publishes a message to its output port. When the block receives the message the time T_1 is written to a file for analysis with the reception time measured in the PHY layer. As only valid messages are received by this block writing T_1 only on valid receive helps in measuring the time delay only for valid data points.

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.

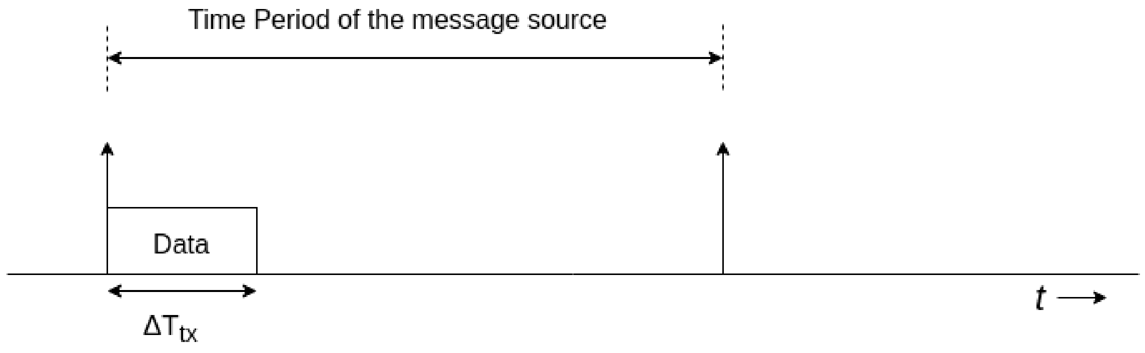

Figure 4.1: Periodic Message Source

4.1.1 System Description

4.1.2 Software Description

4.1.3 Hardware Description

4.2 Timing Analysis

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 4.2) 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.

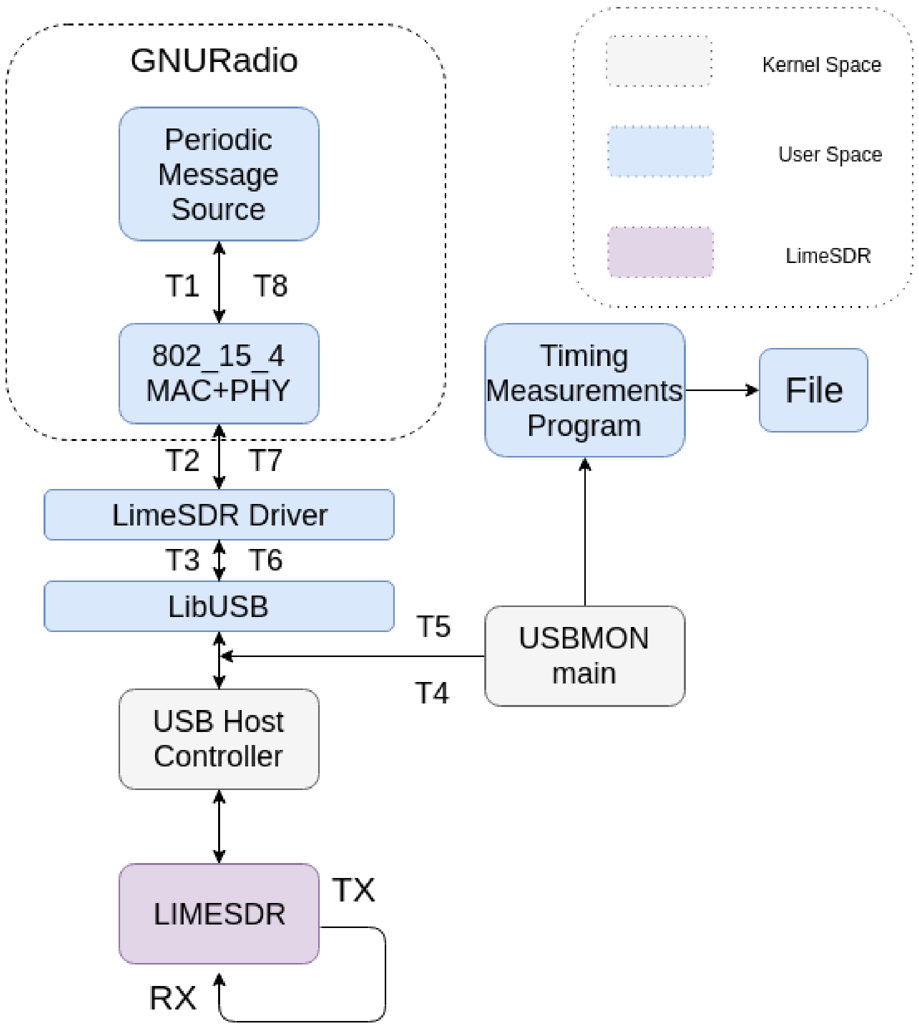

Figure 4.2: System Description

4.2.1 Message Source

4.2.2 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 **??**), 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.

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

Table 4.1: Transfer Direction and Threshold Value

Analyzing the samples in the data stream, the samples threshold for actual data packets to as shown in Table 4.1. 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 4.3 if the condition mentioned about the time period in 4.2.1 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.

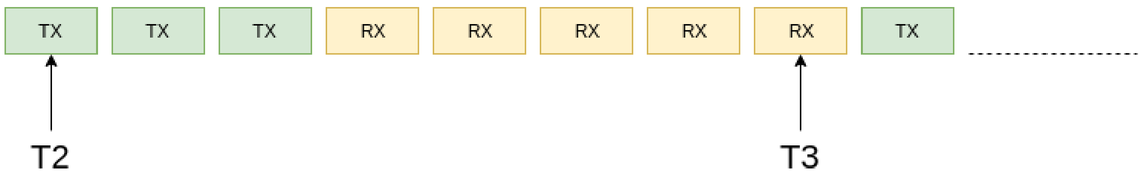

Figure 4.3: Sequence of valid data packet with time

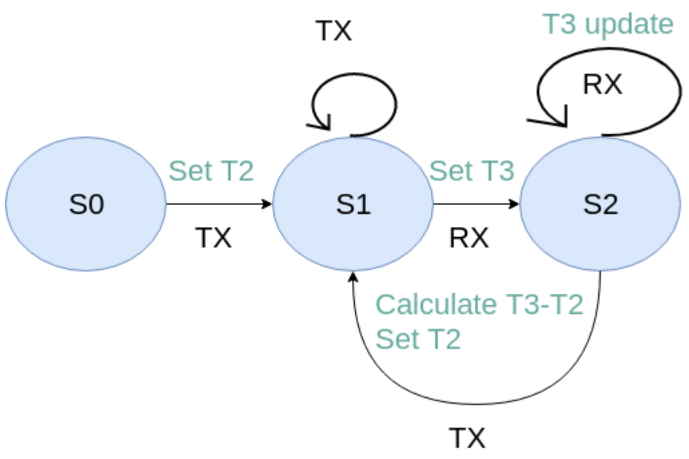

Figure 4.4: State Machine

State Machine shown in Figure 4.4 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.

4.2.3 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.

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 5

Results and Analysis

5.0.1 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 5.1: Analytical USB Transfer Delay

5.0.2 Experimental Results

Figure 5.1 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

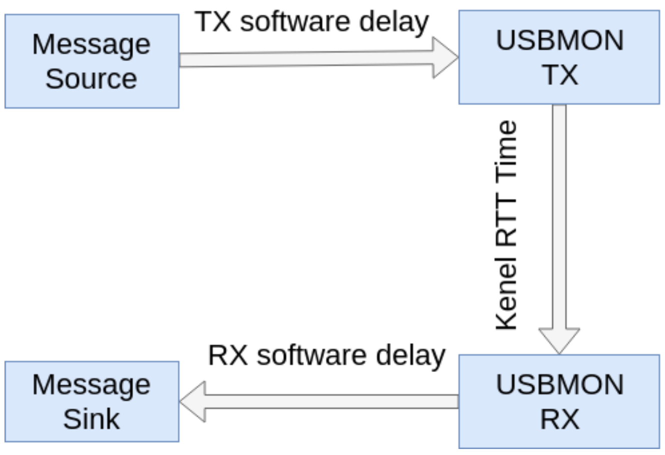

Figure 5.1: Results Setup

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 5.2: Experimental results

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.

5.0.3 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.

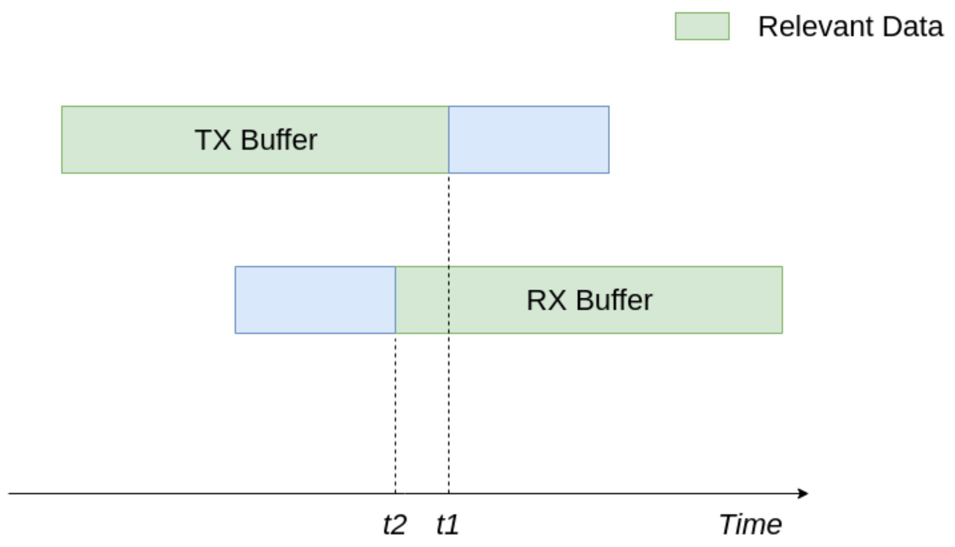

Figure 5.2: Overlapping of buffers on LimeSDR

- 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 5.1) is more than the actual value that usbmon reported(Table 5.0.2), 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 5.2 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.

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] *Internet of Things outlook – Ericsson*. en. SectionStartPage. Nov. 2017. URL: https://www.ericsson.com/en/mobility-report/reports/november-2017/internet-of-things-outlook (visited on 08/01/2018).

[4] *LimeSDR*. en-GB. URL: https://myriadrf.org/projects/limesdr/ (visited on 08/01/2018).

[5] George Nychis and Thibaud Hottelier. “Enabling MAC Protocol Implementations on Software-Defined Radios”. en. In: (), p. 23.

[6] 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).

[7] 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.

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

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

[10] *WARP Project*. URL: https://warpproject.org/trac (visited on 08/02/2018).