



# **LTE-UE-L1-Processing-On-Parallel**

By

Basma Magdy Hussien Ali

Dina Magdy Mohamed Mohamed

Omayma Gomaa Abdelazem

Somaia Hussien Rashad Mohamed

A Thesis submitted to the  
Faculty of Engineering at Cairo University  
In partial fulfilment of the requirements  
For the Degree of Bachelor of Science in  
**ELECTRONICS AND ELECTRICAL COMMUNICATION  
ENGINEERING**

**FACULTY OF ENGINEERING, CAIRO UNIVERSITY  
GIZA, EGYPT**

2016

# LTE-UE-L1-Processing-On-Parallel

---

By

Basma Magdy Hussien Ali

Dina Magdy Mohamed Mohamed

Omayma Gomaa Abdelazem

Somaia Hussien Rashad Mohamed

Under the Supervision of

Dr. Ahmed Hesham

Dr. Hassan Mostafa

A Graduation Project Report Submitted to  
the Faculty of Engineering at Cairo University  
In Partial Fulfillment of the Requirements for the  
Degree of  
Bachelor of Science  
in  
Electronics and Communications Engineering  
Faculty of Engineering, Cairo University

Giza, Egypt

July 2016

# Table of Contents

List of Tables .....	viii
List of Figures .....	ix
List of Symbols and Abbreviations .....	xi
Acknowledgments .....	xii
Abstract.....	xiii
Chapter 1: Introduction.....	0
1.1 Problem Definition.....	0
1.2 Problem Solution with Project Provides.....	2
1.3 Requirements .....	3
1.4 Project Vision.....	3
Chapter 2: Parallella .....	4
2.1 Overview .....	4
2.2 Parallella System Architecture.....	6
2.2.1 Zynq Memory Map .....	6
2.2.2 Epiphany Memory Map.....	7
2.3 Parallella Features Description .....	8
2.3.1 CPU .....	8
2.3.2 Epiphany Coprocessor.....	9
2.3.3 SDRAM.....	10
2.3.4 LED Indicators.....	10
2.3.5 Serial Port .....	11
2.3.6 Powering the Board.....	11
2.4 Parallella Start & Booting.....	11
Chapter 3: Epiphany SDK .....	14
3.1 Introduction.....	14
3.1.1 SDK (Software Development Kit) Overview .....	14

3.1.2	Epiphany Memory Model.....	15
3.1.3	Epiphany Programming Framework .....	16
3.2	Building SDK .....	17
3.2.1	Building Steps.....	17
3.2.2	Building check .....	19
3.3	Epiphany SDK Tool chain.....	20
3.3.1	C/C++ Compiler (E-GCC) .....	20
3.3.2	Linker (E-LD).....	21
3.3.3	Instruction Set Simulator (E-Run) .....	23
3.3.4	Debugger (E-GDB) .....	24
3.3.5	Hardware Connection Server (E-SERVER).....	27
3.4	Epiphany SDK utilities (E-UTILS).....	28
3.4.1	Introduction.....	28
3.4.2	Reset Utility (E-RESET) .....	29
3.4.3	Loader Utility (E-LOADER) .....	29
3.4.4	Memory Read Utility (E-READ).....	30
3.4.5	Memory Write Utility (E-WRITE) .....	30
3.5	Epiphany Hardware Utility Library (e-Lib) .....	31
3.6	Epiphany Host Library (eHAL) .....	32
Chapter 4:	Frame Structure.....	33
4.1	Introduction.....	33
4.2	Frame Structure Type 1 .....	35
4.3	Frame Structure Type 2 .....	39
4.4	Downlink Frame Structure .....	42
4.5	Uplink Frame Structure .....	43
Chapter 5:	Fixed Point Representation.....	45
5.1	Motivation.....	45

5.2	Ways of Fixed Point Representation .....	46
5.2.1	Sign/magnitude .....	46
5.2.2	One's complement .....	46
5.2.3	Two's complement .....	47
5.3	Q-Format number representation .....	47
5.4	Converting Floating Point to Fixed Point Example .....	49
Chapter 6:	Scrambler .....	51
6.1	Introduction.....	51
6.2	Implementation ways: .....	52
Chapter 7:	Modulation & Precoder .....	58
7.1	Introduction:.....	58
7.1.1	QPSK.....	59
7.1.2	16QAM.....	59
7.1.3	64QAM.....	60
7.2	Implementation Ways.....	60
7.3	Precoder .....	62
Chapter 8:	DFT .....	64
8.1	Introduction.....	64
8.2	Implementation way .....	64
8.2.1	Theory and Output Generated Signals .....	64
8.2.2	Error Measurement Ways .....	68
8.2.2.1	Comparing the Constellations.....	68
8.2.2.2	Calculating SNR .....	69
8.2.2.3	Calculating EMV .....	70
8.3	IFFT.....	70
Chapter 9:	Hardware Prototype .....	73
9.1	Motivation.....	73
9.2	FreeRTOS .....	73

9.2.1	Background.....	73
9.2.2	Problem formulation .....	74
9.2.3	Message .....	75
9.2.4	Message-passing between Cores.....	75
9.2.4.1	FreeRTOS Queues.....	76
9.2.4.2	Creating the Message Box .....	77
9.2.4.3	Mutual Exclusion .....	78
9.2.5	Simulation.....	78
9.2.6	Hardware .....	79
9.3	ARM Cross Compiler.....	79
9.4	Hello world on epiphany core.....	80
9.4.1	Introduction.....	80
9.4.2	Implementation .....	80
9.4.3	Results: .....	82
9.5	Test Functionality of Communication Blocks .....	82
9.5.1	Introduction.....	82
9.5.2	Modulation block .....	82
9.5.3	Scrambler & DFT Blocks .....	83
9.6	Optimization Levels .....	84
9.6.1	Introduction.....	84
9.6.2	Running each Block on Single Core .....	84
9.6.3	Running all Blocks on 12-core in parallel.....	84
	Conclusion .....	86
	Future Work.....	87
	References.....	88
	Appendix I Supported Tutorials .....	90
	Appendix II.....	91
	Appendix III Budget.....	116



## List of Tables

Table 1-1: Comparison between different systems architecture and epiphany system	4
Table 2-1: Parallella Feature Summary .....	5
Table 2-2: Zynq Memory Map .....	8
Table 2-3: Epiphany Memory Map .....	9
Table 3-1: General Compiler Options .....	21
Table 3-2: Optimization Options .....	21
Table 3-3: Memory Management Scenarios.....	24
Table 3-4: Linker Sections.....	24
Table 3-5: Simulator Command Line Options.....	25
Table 3-6: Debugger Command Line Options.....	28
Table 3-7: eServer Command Line Options .....	29
Table 3-8: Loader Command Line Options .....	30
Table 3-9: e-read Command Line Options .....	31
Table 3-10: e-write Command Line Options .....	32
Table 4-1: Frame units V.S. their times .....	35
Table 4-2: Cyclic Prefix Types V.S Lengths .....	37
Table 4-3: Subcarriers , Resource blocks corresponding to the B.W .....	39
Table 4-4: Open Source LTE survey .....	42
Table 7-1: The number of bits corresponding to each modulation scheme.....	60
Table 7-2: QPSK modulation mapping .....	60
Table 7-3: 16-QAM modulation mapping .....	61
Table 8-1: IFFT Sizes Corresponding to each B.W .....	72



## List of Figures

Figure 1-1: CDMA voice call vs. VoIP.....	2
Figure 2-1: Parallella board (top view). ....	6
Figure 2-2: Parallella board (bottom view). ....	6
Figure 2-3: Zynq connectivity diagram. ....	7
Figure 2-4: Parallella accessories.....	12
Figure 2-5: Login in Parallella.....	13
Figure 3-1: Epiphany SDK. ....	14
Figure 3-2: Epiphany Program build flow.....	17
Figure 3-3: Simulate Hello-World simple code.....	19
Figure 3-4: The eServer Client-Target Connection Concept.....	27
Figure 4-1: PUSCH Bit Processing Chain.....	33
Figure 4-2: PUSCH Symbol Processing Chain. ....	33
Figure 4-3: FDD Frame Structure Type 1. ....	35
Figure 4-4: Uplink resource grid.....	36
Figure 4-5: LTE FDD Frame of 1.4 MHz, Normal CP.....	38
Figure 4-6: TDD Frame Structure type 2 ....	38
Figure 4-7: LTE resource grid for FDD . ....	41
Figure 4-8: LTE Uplink Subframe 2-3 of 5 MHz, Normal CP ....	44
Figure 5-1: Progress of implement the codes of blocks to generate test cases.....	48
Figure 5-2: Showing How the conversion of floating point to fixed point happens....	51
Figure 6-1: The output From Scrambler Code Versus Test cases. ....	57
Figure 7-1: Signal constellations for: (a) QPSK; (b) 16QAM; (c) 64QAM.....	60
Figure 7-2: The output From Modulation Code Versus Test case.....	64
Figure 8-1: shows the DFT/IDFT reference design blocks. ....	65
Figure 8-2: The output from DFT code versus test cases.....	68
Figure 8-3: constellation plot of test case of modulated signals.....	68
Figure 8-4: constellation plot of IFFT of test case DFT signals. ....	69
Figure 8-5: Difference between case 1 and case 2.....	69
Figure 8-6: Localized Mapping V.S. Distributed Mapping.....	72
Figure 9-1: State diagram of FreeRTOS tasks [12].....	74
Figure 9-2: Hello-world with ARM cross compiler.....	80

Figure 9-3: The way to Implement code on single core on parallella .....	82
Figure 9-4: The way to Implement code on single core on parallella.....	83
Figure 9-5: First modulation output and time consumption .....	83
Figure 9-6: First scrambler output and time consumption . .....	83
Figure 9-7: First DFT output and time consumption . .....	85
Figure 9-8: First output and time of each block and over all time of application .....	85
Figure 9-9: First DFT output and over all time of application . .....	85

## List of Symbols and Abbreviations

LTE	The Long-Term Evolution
3GPP	the Third Generation Partnership Project
GPP	general purpose processor
VOLTE	voice over LTE
NoC	Network-on-Chip
UE	User Equipment
L1	Physical Layer
SDK	Software Development Kit
$M_{sc}^{\text{PUSCH}}$	Scheduled bandwidth for uplink transmission, expressed as a number of subcarriers
$M_{RB}^{\text{PUSCH}}$	Scheduled bandwidth for uplink transmission, expressed as a number of resource blocks
$Q_m$	Modulation order: 2 for QPSK, 4 for 16QAM and 6 for 64QAM transmissions
PUSCH	Physical Uplink Shared Channel
RB	Resource Block
CP	Cyclic Prefix
MIMO	Multi Input Multi Output
PA	Power Amplifier

## **Acknowledgments**

First of all, we would like to express our deep sense of respect and gratitude towards our advisors and guides DR. Ahmed Hesham and DR. Hassan Mostafa for their continuous support, advice, and guidance throughout our work.

We are grateful to AXXCELERA EGYPT for supporting some of Graduation Projects for the first year.

Next, we want to express our respects to Eng. Mohamed Taha, Eng. Karim Osama, Eng. Wael Elbreqy, and other Technical References from AXXCELERA EGYPT for their great effort support.

Great thanks to our parents who support us with their valuable patience and love.

## **Abstract**

As the ubiquitous wireless communication devices consume a lot of processing power, therefore we need to decrease the power consumed by LTE processing units in UEs, also decrease the development time while increasing the throughput the thing that maximize the performance.

The main purpose of that project is to compare experimentally between the power consumption of general purpose processor (GPP) such as Parallella platform and Digital Signal processor (DSP).

The experimental results are collected from Parallella board containing epiphany system of 16 cores and compared with past results found in literature regarding DSP's used in communication applications. The results show that the Parallella board consumes 92 % less power than the DSP.

Therefore, we can consider Parallella platform as a good prototype which minimize power consumption and achieve high performance.

In this documentation we can find the illustration of each block of symbol chain and all experiments which have been done to achieve high level of optimization such as power and time optimization.

# **Chapter 1: Introduction**

## **1.1 Problem Definition**

We can say we have enormous mobile data revolution, to adapt the mass-market expansion of smart phones, tablets, notebooks, and laptop computers. This great growth in data mobile services and applications such as Video streaming, social networking and Web browsing, will develop the next generation of wireless standards and also becomes a very great force for the development. As a result, new standards are evolved to provide network capacity and data rates necessary to support worldwide delivery of these types of rich multimedia application.

Nowadays, it is important to understand how cellular systems have developed and also to understand the mobile-communication systems, its complexity and from where they came. The task of developing mobile technologies has also changed, from regional or national concern, to become an increasingly complex task controlled by the global organizations for developing standards such as the "3GPP" (Third Generation Partnership Project) and involving thousands of people.

Mobile communication technologies have a lot of generations, started from 1G which is analog mobile radio system, 2G which is considered the first digital mobile system, and 3G which is the first mobile system used to handle the broadband data. And finally The LTE (Long-Term Evolution) and it's commonly called "4G", but actually it's 3.9G and the upgrade of it is called LTE-Advance and it's actually the 4G. This continuing race of increasing sequence numbers of mobile system generations is in fact just a matter of labels.

LTE and LTE-Advanced have been developed to realize the goal of achieving global broadband mobile communications and to respond to the requirements of this area. The objectives and goals of this developed system include higher radio access data rates, improved system capacity and coverage, flexible bandwidth operations, significantly improved spectral efficiency, low latency, reduced operating costs, multi-antenna support, and seamless integration with the Internet and existing mobile communication systems.

So as obvious LTE phones are fast, but the battery is sucked quickly in just some few hours, based on some studies made by Nokia Siemens Networks, they found that LTE devices consume from 5 percent to 20 percent more than previous-generation phones, and also it depends on the used application. In Samsung Galaxy Nexus' review, they found that the Google Navigation running over the LTE network ate battery power faster than the Nexus' car charger could restore it [34].

Every mobile carrier wants to replace their old voice services with new VoIP-based systems utilizing their 4G networks, but it looks like they've got some big kinks to iron out in the technology first. Wireless testing and measurement vendor Spirent Communications has identified a big problem with voice over LTE (VoLTE): it consumes twice as much power as a traditional 2G call, which could have big implications for mobile phone battery life.

Metrico Wireless, a radio field testing company Spirent acquired in September, conducted voice trials on a commercial VoLTE-enabled network in two U.S. cities, comparing the power consumption of VoIP calls made over LTE against the power used by the same carrier's CDMA systems as shown on Figure (1-1) [33].

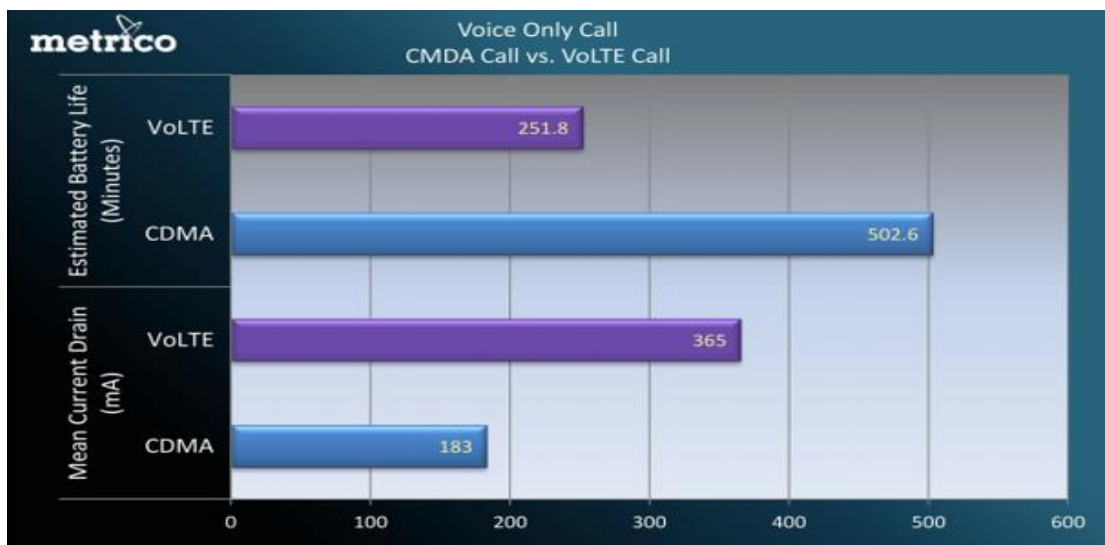


Figure 1-1: CDMA voice call vs.VoIP

It's important to know why LTE is considered battery killer, the first reason is **Phones with LTE has rabbit ears**, All LTE devices sold today use a technology called MIMO, which doesn't just send or receive a single signal, but rather multiple parallel transmissions. Today's devices support two such paths – future devices will support more — which means each phone has two antennas,

each of which requires its own power amplifier. Another reason for LTE to be greedy that **LTE devices are co-dependent**. Phone screen may be dark, but phone is constantly pining for the network. That means it's periodically scans the airwaves around it to determine which tower it should tether itself to. The more networks there are to choose from the more scans it must make. With the typical operator sporting some combination of GSM, HSPA, CDMA and EV-DO systems —often multiple version of each in different frequency bands — there are a lot of other networks for an LTE device to flip between.

So the major ultimate goal of LTE L1 processing in UEs is to provide maximum performance while consuming minimal power. A lot of platforms are optimized to achieve this goal but there still some challenges in this area, like achieving 64QAM while minimizing the processing power.

## **1.2 Problem Solution with Project Provides**

Implementing LTE UE-L1 processing on a system with many-cores general-purpose processors (GPP) will decrease both the cost and development time, plus it is eco-friendlier as it will consume less power than the complex DSP processors using now in implementing LTE.

To evaluate implementing LTE UE-L1 processing on GPP platform, we introduce the Epiphany system which consists of GPP clustered in a Network-on-Chip (NoC) that range from 4x4 to 64x64 cores. The Epiphany combines fully-featured floating point C/C++ programmable RISC processors, as each core has separate CPU, each CPU can run a separate and independent program (MIMD not SIMD).

Epiphany introduces a high bandwidth distributed memory system, a low latency Network-On-Chip, and low overhead off-chip IO to bring an unprecedented level of processing to power constrained systems. Table (1-1) below show comparison between Architecture Comparison of different systems and epiphany systems.



Table 1-1: comparison between different systems architecture and epiphany systems.

<b>Technology</b>	<b>FPGA</b>	<b>DSP</b>	<b>Epiphany</b>
<b>Process</b>	28nm	40nm	28nm
<b>Programming</b>	VHDL	OCL/C++/C	OCL/C++/C
<b>Area (mm<sup>2</sup>)</b>	590	108	10
<b>Chip Power (W)</b>	40	22	2
<b>Compile Time</b>	Hours	Minutes	Minutes
<b>L1 Memory</b>	6MB	512KB	2MB

### 1.3 Requirements

It is required to evaluate the feasibility of implementing one of the LTE UE-L1 (R10) data processing chains (PDSCH or PUSCH) on the Epiphany system represented by parallella platform which consist of 16 epiphany co-processors.

The following steps are required to achieve this:

- 1-Study the LTE UE-L1 system (PDSCH and/or PUSCH), and choose a design that would maximize parallelization gain.
- 2-Choose the blocks that will be implemented.
- 3-Implement C code for these blocks as processing kernels.
- 4-Port these kernels as jobs for the processing cores on the HW platform.
- 5-Synchronize between those jobs running on different cores.
- 6-Implement a testing mechanism that would involve LTE transmission on wire (Ethernet).

### 1.4 Project Vision

The vision of the company on the long term to use the platform prototype as a complete transmitter of LTE system using min power and with high rate. That low power consumption will help in the far places which is hard to be supported by huge amount of power ex the submarines in petroleum field where I need to coverage it with efficient communication system, low power consumption and far periodic maintenance planning (cost).

## Chapter 2: Parallella

### 2.1 Overview

Parallella board is a high performance computing platform based on a dual-core ARM-A9 Zynq System-On-Chip and Adapteva’s Epiphany multicore coprocessor.

Current commercially available models Table (2-1):

Table 2-1: Parallella Feature Summary

Model	P1600	P1601	P1602
Mnemonic	“Microserver”	“Desktop”	“Embedded”
Host Processor	Xilinx Zynq Dual-core ARM A9 XC7Z010		Xilinx Zynq Dual-core ARM A9 XC7Z020
Coprocessor	Epiphany 16-core CPU E16G301		
Memory	1 GB DDR3		
Ethernet	Gigabit Ethernet		
Boot Flash	128Mb QSPI Flash		
Power	5V DC		
Storage	Micro-SD		
USB	No	USB 2.0 Host Port	
HDMI	No	Micro HDMI	
GPIO Pins	0	24	48
eLink Connectors	0	2	2
FPGA Logic	28K Logic Cells 80 DSP Slices	28K Logic Cells 80 DSP Slices	80K Logic Cells 220 DSP slices
Weight	1.3 oz (36 grams)	1.4 oz (38 grams)	
Size	3.5” x 2.1” x 0.625” (90mmx55mmx18mm)		
SKU	P1600-DKxx	P1601-DKxx	P1602-DKxx
HTS Code (Schedule B )	8471.41.0150	8471.41.0150	8471.41.0150

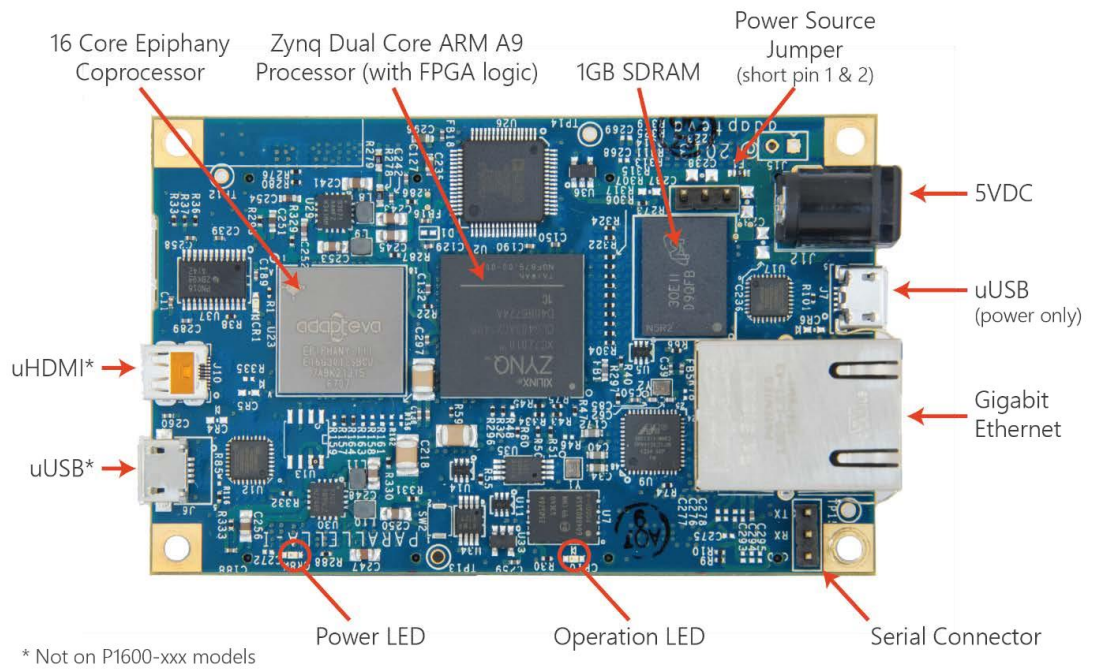


Figure 2-1: Parallella Board (top view)

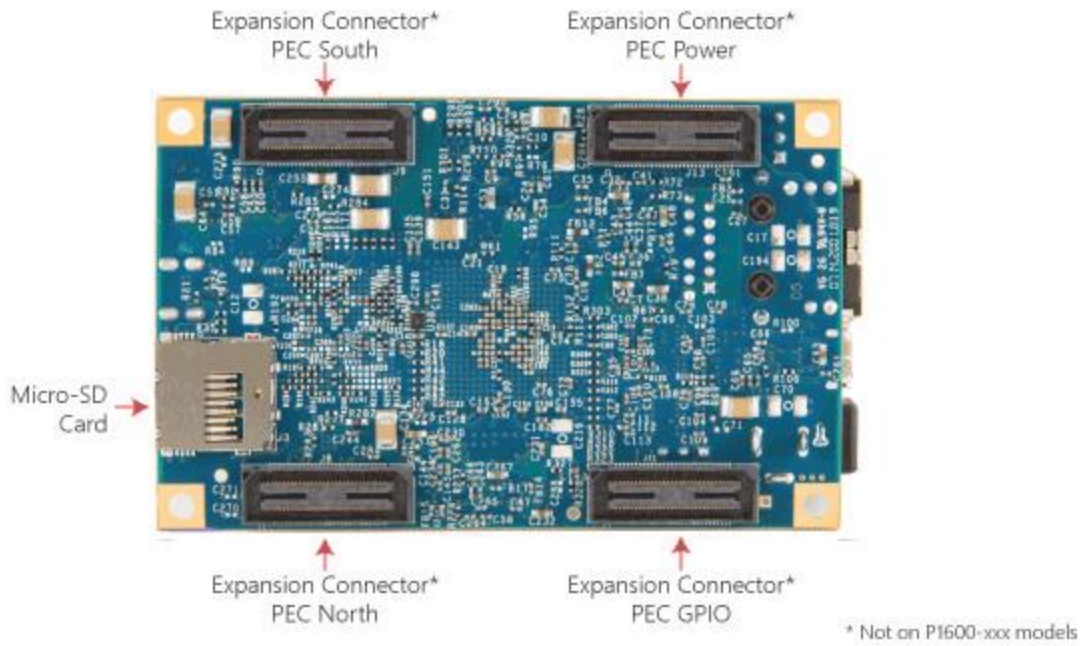


Figure 2-2: Parallella Board (bottom view)

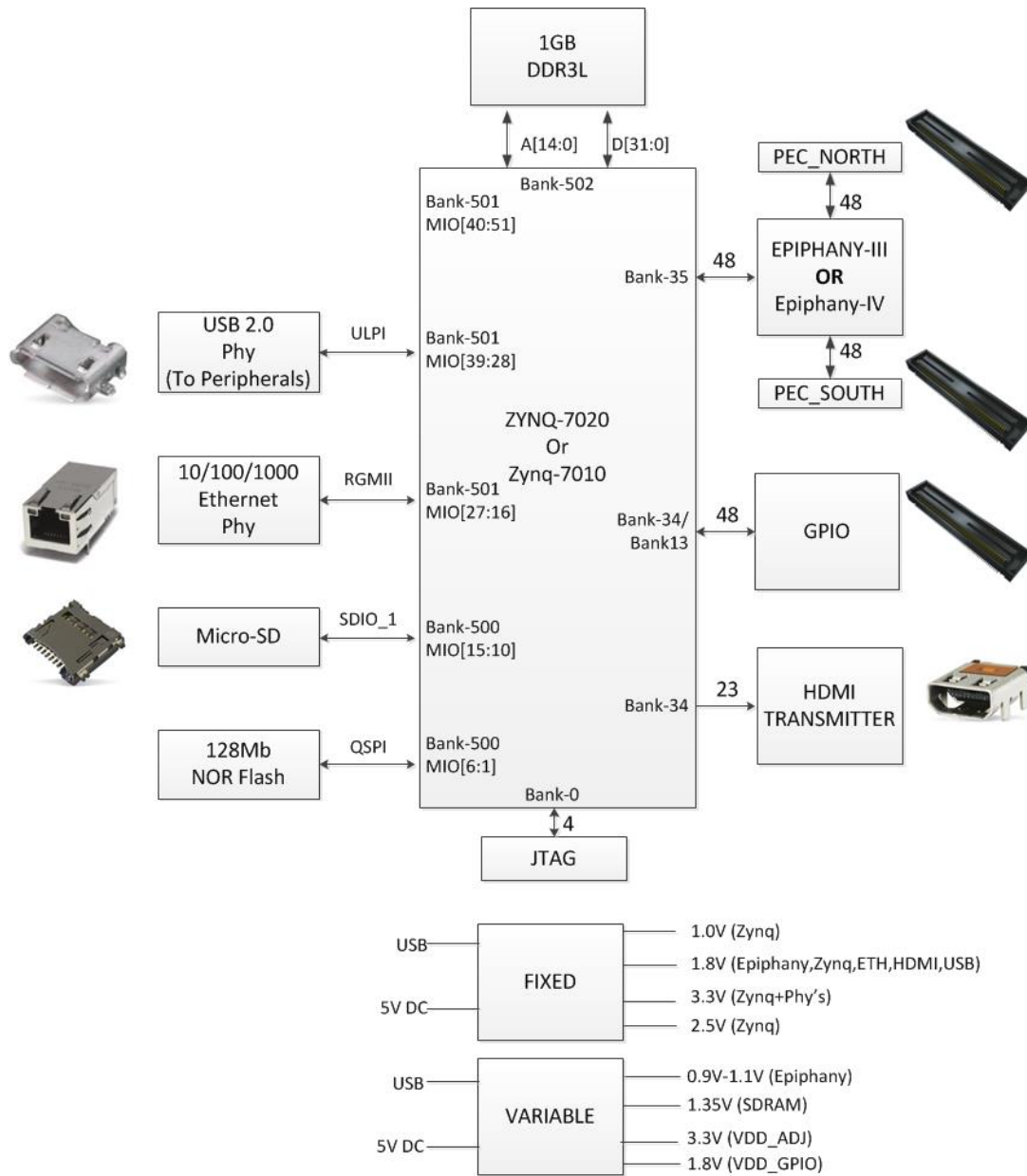


Figure 2-3: Zynq connectivity Diagram

## 2.2 Parallella System Architecture

### 2.2.1 Zynq Memory Map

Table (2-2) shows the hard-coded memory architecture of the Zynq architecture most relevant to the Parallella architecture.

Table 2-2: Zynq memory map

Address Start	Address End	Size	Function	Note
0x0010_0000	0x3FFF_FFFF	1GB	DRAM	Accessible to all interconnect masters
0x4000_0000	0x7FFF_FFFF	1GB	PL	Custom logic address range
0x8000_0000	0xBFFF_FFFF	1GB	PL	Epiphany address range
0xFC00_0000	0xFCFF_FFFF	16MB	FLASH	Quad-SPI linear address for linear mode
0xFFF0_0000	0xFFFF_FFFF	252KB	OCM	OCM upper address range

The ARM communicates with programmable logic, GPIO connected to the programmable logic, and the Epiphany by accessing the memory ranges shown in the table.

The Epiphany 32-bit memory space is mapped into the Zynq memory space allowing for easy sharing of data and resources between the ARM and the Epiphany. The Epiphany address range is a matter of convention and depends on the appropriate AXI master and slave interfaces being implemented within the programmable logic on the Zynq.

### 2.2.2 Epiphany Memory Map

The Epiphany chip is situated within a 1GB section within the Zynq host processor memory map. The offset within the 1GB space occupied by an Epiphany coprocessor is set by the ROWID and COLID pins on the Epiphany chip. The ROWID and COLID can be individually set on boards through the PEC\_POWER connector enabling direct board to board connection through the PEC\_NORTH and PEC\_SOUTH connectors. By default, the address locations of the Epiphany cores on Parallella-16 are as shown in Table (2-3).

Table 2-3: Epiphany memory map

Core Number	Start Address	End Address	Size
(32,8)	80800000	80807FFF	32KB
(32,9)	80900000	80907FFF	32KB
(32,10)	80A00000	80A07FFF	32KB
(32,11)	80B00000	80B07FFF	32KB
(33,8)	84800000	84807FFF	32KB
(33,9)	84900000	84907FFF	32KB
(33,10)	84A00000	84A07FFF	32KB
(33,11)	84B00000	84B07FFF	32KB
(34,8)	88800000	88807FFF	32KB
(34,9)	88900000	88A07FFF	32KB
(34,10)	88A00000	88A07FFF	32KB
(34,11)	88B00000	88B07FFF	32KB
(35,8)	8C800000	8C807FFF	32KB
(35,9)	8C900000	8C907FFF	32KB
(35,10)	8CA00000	8CA07FFF	32KB
(35,11)	8CB00000	8CB07FFF	32KB

## 2.3 Parallella Features Description

### 2.3.1 CPU

The central processor on the Parallella board is the [Zynq™-7000 AP SoC](#). The Zynq represents a new class of processor product which combines an industry-standard ARM® dual-core Cortex™-A9 MPCore™ processing system with Xilinx 28nm programmable logic. The Zynq SoC includes the following set of features:

Dual-core ARM® Cortex™-A9 CPU:

- Coherent multiprocessor support.
- ARMv7-A architecture.
- 32 KB Level 1 4-way set-associative instruction/data caches (independent for each CPU).
- 512 KB 8-way set-associative Level 2 cache shared between CPUs.
- TrustZone® security.
- Jazelle® RCT execution Environment Architecture.
- NEON™ media-processing engine.
- Single and double precision Vector Floating Point Unit (VFPU).
- CoreSight™ and Program Trace Macrocell (PTM).

- Three watchdog timers, one global timer, two triple-timer counters.

#### I/O Peripherals and Interfaces:

- 10/100/1000 tri-speed Ethernet MAC peripherals GMII, RGMII, and SGMII interfaces.
- Two USB 2.0 OTG peripherals.
- Two full CAN 2.0B compliant CAN bus interfaces.
- Two SD/SDIO 2.0/MMC3.31 compliant controllers.
- Two full-duplex SPI ports with three peripheral chip selects.
- Two high-speed UARTs (up to 1 Mb/s).
- Two master and slave I2C interfaces.
- 8-Channel DMA Controller with scatter/gather capability.
- JTAG port for ARM debugging and FPGA programming.
- 12-bit ADC input.
- On-chip voltage and temperature sensing.

#### Programmable Logic:

- LVCMOS, LVDS, and SSTL signaling with 1.2V to 3.3V IO.
- Easily accessible from ARM cores through AXI bus (master or slave).
- Up to 125 programmable IO pins (Z-7020).
- Up to 85K programmable logics cells (Z-7020).
- Up to 560 KB distributed RAM (Z-7020).
- Up to 220 DSP slice and (Z-7020).

### 2.3.2 Epiphany Coprocessor

The Parallella-16 includes the E16G301 device with 16 CPU cores and the Parallella-64 includes the E64G401 device with 64 CPU cores. Both devices have the following basic features:

#### Epiphany Core (eCore):

- 32-bit dual-issue superscalar RISC architecture.
- Quad-bank 32KB local single cycle access memory.
- Floating point instruction set (IEEE754).
- 64-entry register file.
- Dual channel DMA engine.

- Two 32-bit timers.
- Nested interrupt controller.
- Memory protection unit.
- Debug unit.

#### Network-On-Chip (eMesh):

- Three separate networks:
  - rMesh for read transactions
  - xMesh for off-chip write transactions
  - cMesh for on-chip write transactions
- “API-less” network that processes regular load/store transactions.
- All transactions are complete and atomic 104 bit transactions (32-bit address, 64-bit data, and 8 control bits).
- Around robin arbitration at every mesh node.
- Mesh network extends off chip enabling glue-less multi-chip design.

#### Chip-To-Chip Links (eLink):

- North, east, west, south links for connecting to other Epiphany chips, FPGAs, or ASICs.
- Source synchronous LVDS links with transmit clock aligned in the middle of the data eye.
- Dual data rate communication (positive and negative edge transfers).
- Max transfer of 2 bytes transferred in and out simultaneously per link per clock cycle.
- Automatic bursting for sequential 64-bit write transactions.

### **2.3.3 SDRAM**

1GB 32-bit wide DDR3L SDRAM.

### **2.3.4 LED Indicators**

- A green LED controlled by the Zynq GPIO pin.
- A red LED controlled by the Epiphany flag pin.



- Two LEDs on the RJ45. The left LED indicates link speed. (amber=1Gb, green=100Mb, off=10Mb). The right indicates that there is activity on the port.

### 2.3.5 Serial Port

A three-pin header for 3.3V UART output from the Zynq.

### 2.3.6 Powering the Board

The Parallella should be powered through a stable 5V/2A power supply. Current consumption for the Parallella board can be as low as 0.3A but can reach 1.5A when fully loaded.

## 2.4 Parallella Start & Booting

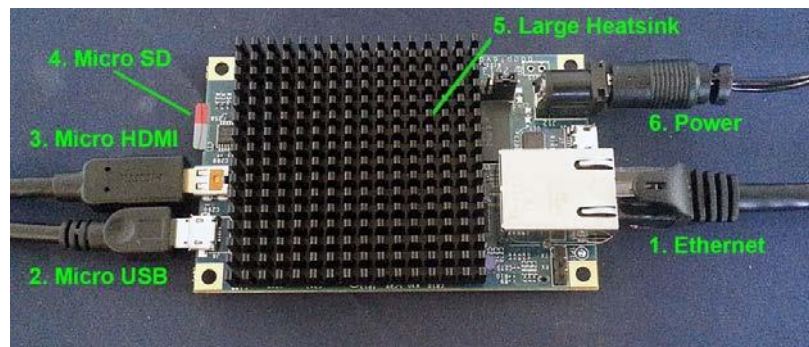


Figure 2-4: Parallella accessories

### Step1: Ensuring the required accessories

- A high quality 2000mA rated 5V DC power supply with 5.5mm OD / 2.1mm ID center positive polarity plug.
- An Ethernet cable.
- A fan.
- A micro HDMI to HDMI cable (not needed for headless option as in our project).
- A USB male Micro-B to female Standard-A cable (not needed for headless option as in our project).

### Step 2: Creating a bootable micro-SD card

- Using “SDFormatter” program to format SD card.

- Using “Win32DiskImager” program to boot Ubuntu image onto SD card
- Another way follows [1].

### **Step 3: Familiarizing with known issues**

- The board does get hot so we have to take precautions to cool the board properly. Before letting the board run for hours, we must ensure that the board doesn't overheat. (Preferably by using the ‘xtemp’ utility script exists on board)
- Boards used without a fan must be placed vertically.
- The Parallella is sensitive to static discharge and must be handled appropriately.

### **Step 4: Connect peripherals, fit the heat-sink and apply power**

- Connect the cables as indicated by #1, 4 in Figure (2-4)
- Attach a heatsink to the Zynq device using double face sticker shipped with board as indicated by #5 in Figure (2-4)
- Make sure a fan is directed at the board if required. A fan is required when using the small heatsink.
- Monitor the temperature using a utility such as xtemp, and keep the chip temp below 70 degrees Celsius.
- Apply power as indicated by #6 in Figure (2-4).

### **Step 5: Connect between PC and Board**

Option 1: Connecting using UART cable.

Option 2: Connecting by opening session with board using (ssh parallella@ip) command as we have used in our work.

## Step 6: Build and run a program

The system will boot and a login screen will appear.

Login with the username parallella and password parallella.

```
somaia@somaia-Lenovo:~$ ssh parallella@192.168.1.9
parallella@192.168.1.9's password:
Welcome to Linaro 14.04 (GNU/Linux 3.14.12-parallella-xilinx-g40a90c3 armv7l)

* Documentation: https://wiki.linaro.org/
Last login: Sat Mar 12 16:40:35 2016 from somaia-lenovo.zte.com.cn
```

Figure 2-5: Login in parallella board

## Chapter 3: Epiphany SDK

### 3.1 Introduction

#### 3.1.1 SDK (Software Development Kit) Overview

The Epiphany™ architecture defines a multicore, scalable, shared-memory computing fabric. It consists of a 2D array of mesh compute nodes connected by a low-latency mesh network-on-chip.

The Epiphany Software Development Kit (eSDK) is a state-of-the-art software development environment targeting the Epiphany multicore architecture. The eSDK is based on standard development tools including an optimizing C-compiler, functional simulator, debugger, and multicore integrated development environment (IDE). The eSDK enables out-of-the-box execution of applications written in regular ANSI-C and does not require any C-subset, language extensions, or SIMD style programming. The unparalleled energy efficiency of the Epiphany architecture and the ease of use and fine grain control of the eSDK offer developers best-in-class capabilities for the most demanding real-time applications. The Epiphany SDK framework is illustrated in Figure (3-1) and contains the following key components:

- Optimized ANSI-C compiler (based on gcc).
- Robust multicore Eclipse IDE (on selected platforms “doesn’t exist on our platform”).
- Multicore debugger (based on gdb).
- Multicore communication and hardware utility libraries.
- Fast functional simulator with instruction trace capability.

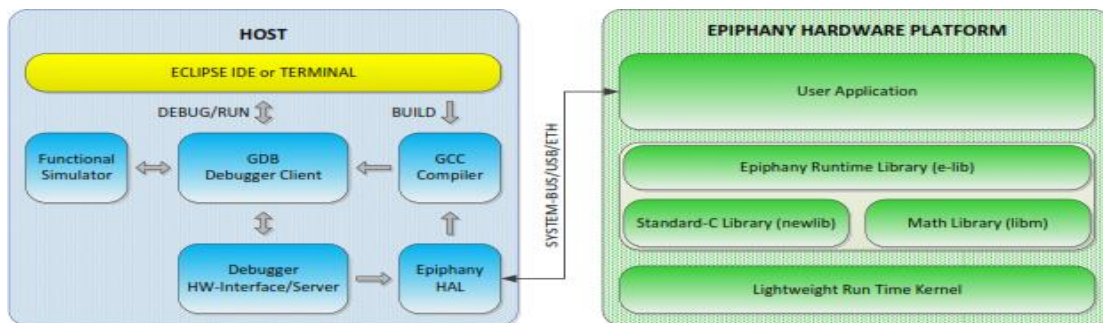


Figure 3-1: Epiphany SDK

### 3.1.2 Epiphany Memory Model

The Epiphany cores have access to two types of memory. Both types can be accessed directly [6].

#### ▪ Internal Memory

**Size:** 32KB (0x8000) per core

Location in address space:

- 0x00000000 - 0x00007fff when a core is referring to its own memory.
- 0xxx000000 - 0xxx07ffff when referring to the memory of any other core. The xxx indicate the core.

Terminology:

- Internal memory.
- eCore memory.
- SRAM or Static RAM. Not to be confused with Shared memory.

Usage:

- Program code, starting at lower addresses.
- Program data (global variables), starting at lower addresses after code.
- Stack (local variables), starting at 0x8000 expanding downwards.

#### ▪ External Memory

**Size:** 32 MB (0x02000000)

Location in address space:

- 0x8e000000 - 0x8fffffff

Terminology:

- External memory
- Shared memory
- DRAM or Dynamic RAM
- SDRAM or Shared DRAM

Usage:

- **Location:** 0x8e000000 - 0x8effffff

**Size:** 0x01000000 (16 MB)

**Contents:** newlib (the C library, with code, data, stack)

- **Location:** 0x8f000000 - 0x8fffffff

**Size:** 0x01000000 (16 MB)

**Contents:**

- **Location:** 0x8f000000 - 0x8f7fffff

**Size:** 0x00800000 (8 MB)

**Section label:** shared\_dram

**Contents:** used by the `e_shm_XXX` functions of the ESDK

**Extra info:** because of a bug, `malloc` returns addresses from this region which causes this region to be corrupted if one uses any C function that uses `malloc` internally.

- **Location:** 0x8f800000 - 0x8fffffff

**Size:** 0x00800000 (8 MB)

**Section label:** heap\_dram

**Contents:** is meant to be divided in 512KB for each core ( $16 * 512KB = 8MB$ ) and then used for `malloc` but this does **not** currently work. Instead `malloc` returns addresses from `shared_dram`

### 3.1.3 Epiphany Programming Framework

Each one of the Epiphany processor nodes can run independent programs. Figure (3-2) shows the general programming flow for the Epiphany architecture, highlighting the independent build of programs running on different cores and the use of a common loader to load the complete multicore program onto the chip.

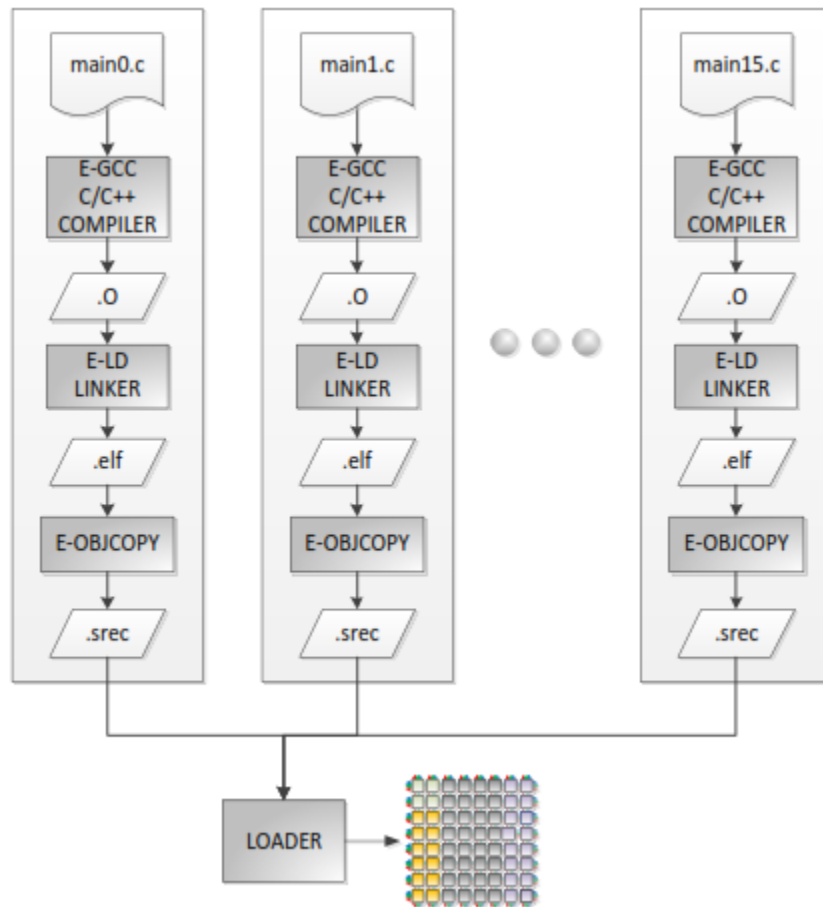


Figure 3-2: Epiphany program build flow

## 3.2 Building SDK

### 3.2.1 Building Steps

Step 1: Installing Prerequisites

(The following packages need to be installed in order to build the SDK)

```
Sudo apt-get install build-essential git bison flex libgmp3-dev
libncurses-dev libmpc-dev libmpfr-dev texinfo xzip lzip zip
gcc-arm-linux-gnueabi g++-arm-linux-gnueabi
```

## Step 2: Downloading the SDK

First, we should create a directory for our build environment, for example ``${HOME}/epiphany-sdk``. Next set an environment variable named `EPIPHANY_BUILD_HOME` which points to the root of your build tree.

```
mkdir epiphany-sdk
export EPIPHANY_BUILD_HOME=${HOME}/epiphany-sdk
```

### (Downloading the SDK Sources)

```
cd $EPIPHANY_BUILD_HOME
git clone https://github.com/adapteva/epiphany-sdk sdk
```

### (Downloading the SDK build Scripts)

```
mkdir buildroot && cd buildroot
git clone --branch 2016.3 https://github.com/adapteva/epiphany-sdk.git
sdk
```

## Step 3: Building the SDK

### (Building for Intel x86-64 (on x86-64))

```
./sdk/build-epiphany-sdk.sh -C -R -a x86_64
```

## Step 4: Adding the SDK to my Path

```
export PATH=/home/somaia/epiphany-sdk/esdk/tools/e-gnu/bin:$PATH
```

(Note: Building SDK is done completely over the Internet, so it needs stable internet also because the building lasts for 12 hours almost [5].)



### 3.2.2 Building check

To check that the SDK have been built successfully, we have run simple code of Hello-World using Compiler and Simulator of SDK.

1. We have downloaded text editor “gedit” for writing scripts.
2. Write simple code to print Hello-World.
3. To compile the code we have to change the path of code script to the path of “e-gnu” of SDK. To avoid changing path of every code to path of SDK we have edited “.bashrc” file with the path of SDK so it can see it always and we don’t need to change path again. Edit “.bachrc” with

```
export PATH=/home/somaia/epiphany-sdk/esdk/tools/e-gnu/bin:$PATH
export EPIPHANY_HOME=/home/somaia/epiphany-sdk/esdk
export EPIPHANY_HDF=/home/somaia/epiphany-sdk/
                                esdk.2015.1/bSPs/parallella_E16G3_1GB
```

4. Compile code by this command

```
$ e-gcc hello_world.c -o hello_world.elf
```

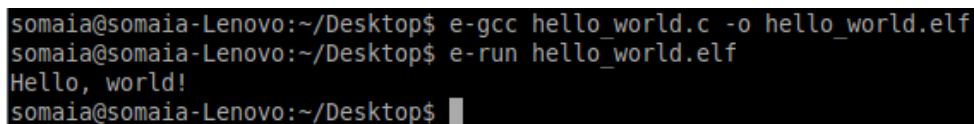
hello\_world.c: code file name

-o: option to generate output in the next file

hello\_world.elf: output file with elf extension for simulation

5. Simulate code by this command

```
$ e-run hello_world.elf
```



```
somaia@somaia-Lenovo:~/Desktop$ e-gcc hello_world.c -o hello_world.elf
somaia@somaia-Lenovo:~/Desktop$ e-run hello_world.elf
Hello, world!
somaia@somaia-Lenovo:~/Desktop$ █
```

Figure 3-3: Simulate hello-world simple code

### 3.3 Epiphany SDK Tool chain

#### 3.3.1 C/C++ Compiler (E-GCC)

The GCC compiler supports the following versions of C/C++:

- ISO/IEC 9899:1990 (C89)
- ISO/IEC 9899:1999 (C99)
- ISO/IEC 9899:2011 (C11) (*partial*)
- ISO/IEC 14882:1998 (C++98)
- ISO/IEC 14882:2011 (C++11) (*partial*)

To use the compiler to create an executable from a simple program source file without any optimization.

```
$ e-gcc hello_world.c -o hello_world.elf
```

The GCC compiler supports a wide range of options allowing for fine grain compilation process. Some are illustrated in Table (3-1) and some of optimization options are illustrated in Table (3-2).

Table 3-1: General Compiler Options

Option	Function
-c	Compile or assemble source code, but do not link
-o <i>file</i>	Place output in file <i>file</i> .
--version	Print the version number of the compiler
@ <i>file</i>	Read command-line options from file.
--help	Print (on the standard output) a description of the command line options understood by gcc.

Table 3-2: Optimization Options

Option	Function
-O0	Reduce compilation time and make debugging produce the expected results. This is the default.
-O1	'-O or -O1' turns on the following optimization flags: -fauto-inc-dec -fcprop-registers -fdce -fdefer-pop -fdelayed-branch -fdse -fguess-branch-probability -fif-conversion2 -fif-conversion -finline-small-functions -fipa-pure-const -fipa-reference -fmerge-constants -fsplit-wide-types -ftree-ccp -ftree-ch

	-ftree-copyrename -ftree-dce -ftree-dominator-opts -ftree-dse -ftree-fre -ftree-sra -ftree-ter -funit-at-a-time  ‘-O’ also turns on ‘-fomit-frame-pointer’
-O2	GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. It turns on all optimization flags in O1 and the following additional flags: -fthread-jumps -falign-functions -falign-jumps -falign-loops -falign-labels -fcaller-saves -fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fdelete-null-pointer-checks -fdevirtualize -fexpensive-optimizations -fgcse -fgcse-lm -finline-small-functions -findirect-inlining -fipa-sra -foptimize-sibling-calls -fpartial-inlining -fpeephole2 -fregmove -freorder-blocks -freorder-functions -frerun-cse-after-loop -fsched-interblock -fsched-spec -fschedule-insns -fschedule-insns2 -fstrict-aliasing -fstrict-overflow -ftree-switch-conversion -ftree-pre -ftree-rrp
-O3	Optimize yet more. -O3 turns on all optimizations specified by -O2 and also turns on: -finline-functions -funswitch-loops -fpredictive-commoning -fgcse-after-reload -ftree-vectorize -fipa-cp-clone

### 3.3.2 Linker (E-LD)

The Epiphany linker ‘e-ld’ combines a number of objects and archives, relocates their data and resolves symbol references.

To use the linker to create an elf executable from an object file using the default linker file for simulation using the Epiphany instruction set simulator.

```
$ e-ld my_object.o -o exec.elf
```

The link process is controlled by a linker script written in the GNU linker command language. The purpose of the linker script is to describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file. If you do not supply a linker file to ‘e-ld’, it will use a default linker file.

Executables compiled with the default linker will only execute correctly using the ‘e-run’ instruction set simulator and will not work correctly when loaded on specific hardware targets.

To correctly link for specific hardware targets, you should use the ‘-T’ option to specify one of the board specific linker files that come with the board support package (BSP) or your own custom linker file.

The linker description files that come with the different Epiphany Board Support Packages have a number of key words that allow fine grained management of code and data placement from within the C/C++ source code. The keywords gives the programmer and support libraries complete control of the placement of data and code within the memory system on a per-symbol, per-file, and per-object library. The keywords are derived from section names within the linker descriptor file and can be augmented by the user at his discretion. Table (3-3) shows configurations of the three basic linker descriptor files. The ‘legacy’ scenario is to be used for bringing up code quickly but will run slowly since all data and code is placed in external memory. The ‘fast’ scenario is used to place user code internally and standard library `code externally. The ‘internal’ scenario can be used for to effectively place all code and data in the local memory by default. The three descriptor files effectively determine the default placement of all sections and symbols within the objects. The user can override these settings on an individual basis from the C/C++ source code using the attributes defined in Table (3-4) to specify that certain variables and/or functions should be placed in specific program output sections. Note that with all of the predefined LDF’s, the heap is allocated externally. This means that use of stdio library will render the program very slow.

Table 3-3: Memory Management Scenarios

File	User Code & Data	Standard Library	Stack	Note
legacy.ldf	External SDRAM	External SDRAM	External SDRAM	Use to run any legacy code with up to 1MB of combined code and data.
fast.ldf	Internal SRAM	External SDRAM	Internal SRAM	Places all user code and static data in local memory, including the stack. Use to implement fast critical functions. It is the user's responsibility to ensure that the code fits within the local memory.
internal.ldf	Internal SRAM	Internal SRAM	Internal SRAM	Places all code and static data in local memory, including the stack. Use to implement fastest applications. It is the user's responsibility to ensure that the code fits within the local memory.

Table 3-4: Linker Sections

Section	User Controllable Sections
.text	Application code, read only
.data	Application data (global variables that are not constant)
.rodata	Application data, read only (constants, strings)
.bss	Static variables initialized to zero
.text_bank0	Starts at end of reserved section in bank0
.text_bank1	Starts at the beginning of bank1
.text_bank2	Starts at the beginning of bank2
.text_bank3	Starts at the beginning of bank3
.data_bank0	End of .text_bank0
.data_bank1	End of .text_bank1
.data_bank2	End of .text_bank2
.data_bank3	End of .text_bank3
.code_dram	Code section in external memory
.shared_dram	Data section in external memory
.heap_dram	Heap section in external memory

### 3.3.3 Instruction Set Simulator (E-Run)

The Epiphany Instruction Set Simulator (ISS) is an accurate and fast functional representation of the Epiphany Instruction Set Architecture. The simulator accurately models the instruction set and

register map of a single Epiphany core, but does not model pipeline behavior or any of the non-CPU hardware mechanisms such as the eMesh Network-On-Chip, DMA, or timers. The simulator runs in a host Linux environment, takes a binary ELF file as an input and supports standard I/O. To simplify program debugging and profiling, the simulator supports outputting program traces. Simulation of the execution of an Epiphany elf executable using the ISS within a Linux host platform is done by this command

```
$ e-run hello_world.elf
```

To get an instruction trace of the executed program, use the ‘-t’ option before the hello\_world.elf argument as follows:

```
$ e-run -t hello_world.elf
```

Table 3-5: Simulator Command Line Options

Option	Function
-t, --trace	Output simulated instruction trace to screen
--memory-region ADDRESS, SIZE	Defines a memory region as valid for simulator. Default is to allow 0x0 → 1MB
--help	Prints help

### 3.3.4 Debugger (E-GDB)

The Epiphany debugger (e-gdb) is based on the popular GNU GDB debugger. It allows to see what is going on inside a program while it executes. Some of the powerful debug features enabled by the debugger include:

- Interactive program load.
- Stopping program on specific conditions (usually a breakpoint placed in source code).
- Examine complete state of machine and program once program has stopped.
- Continuing program one instruction at a time or until the next stop condition is met.

The Epiphany debugger supports program debugging using the functional simulator as a target or the hardware platform as a target using the ‘e-server’. The only difference between the two modes of debugging is the argument specified with the ‘target’ command within the debugger client. The simulator only supports debugging programs running on a single Epiphany CPU core and is not multi-core aware.

To debug a simple “Hello World” program with the Epiphany instruction set simulator.

In a Linux shell, start an e-gdb session using your executable.

```
$ e-gdb hello_world.elf
```

Inside e-gdb, connect to the instruction set simulator debugging target.

```
(gdb) target sim
```

Load the program to the core memory.

```
(gdb) load
```

Place a breakpoint at the main function entry point.

```
(gdb) b main
```

Run the functional simulator.

```
(gdb) run
```

Continue program execution from breakpoint.

```
(gdb) c
```

Program then runs until completion and displays.

```
“Hello World!”
```

Exit debugger

```
(gdb) q
```

Debugging a program running on an Epiphany based hardware target.

Make sure that a connection has been established with the hardware using the e-server program:

```
$ e-server -hdf ${EPIPHANY_HOME}/bsps/emek3/emek3.xml -test-  
memory
```

In a Linux shell, start a e-gdb session using your executable (same as for the simulator).

```
$ e-gdb hello_world.elf
```

Inside e-gdb, connect to the TCP/IP socket connected to core that you want to debug.

```
(gdb) target remote:51000
```

Load the program the core memory.

```
(gdb) load
```

Place a breakpoint at the main function entry point.

```
(gdb) b main
```

Continue program execution from breakpoint.

```
(gdb) c
```

Program then runs until completion and displays.

```
"Hello World!"
```

Exit debugger

```
(gdb) q
```



Invoke the debugger by running the program ‘e-gdb’. Once started, ‘e-gdb’ reads commands from the terminal until you tell it to exit. You can also run ‘e-gdb’ with a variety of arguments and options, to specify more of your debugging environment at the outset.

The most common way to start ‘e-gdb’ is to simply specify the program as the only argument:

```
$ e-gdb program.elf
```

Table 3-6: Debugger Command Line Options

Option	Function
-x <i>file</i>	Execute gdb commands from file <i>file</i> .
-d <i>directory</i>	Add <i>directory</i> to the path to search for source files.
-quiet   q -silent	“Quiet”. Do not print the introductory and copyright messages.

### 3.3.5 Hardware Connection Server (E-SERVER)

The GDB client runs on the Linux host machine and communicates with the Epiphany GDB server using GDB's internal RSP (remote serial protocol) over TCP/IP ports. The e-server responds to the GDB client or Loader requests and controls the hardware or hardware emulation model. Each core in the system needs a separate GDB client and connects to the GDB server using a unique TCP/IP port. By default, cores connect to the e-server starting at port 51000. The e-server responds to the e-loader requests in the dedicated port.

An illustration of the GDB server/client operations is shown below in Figure (3-3):

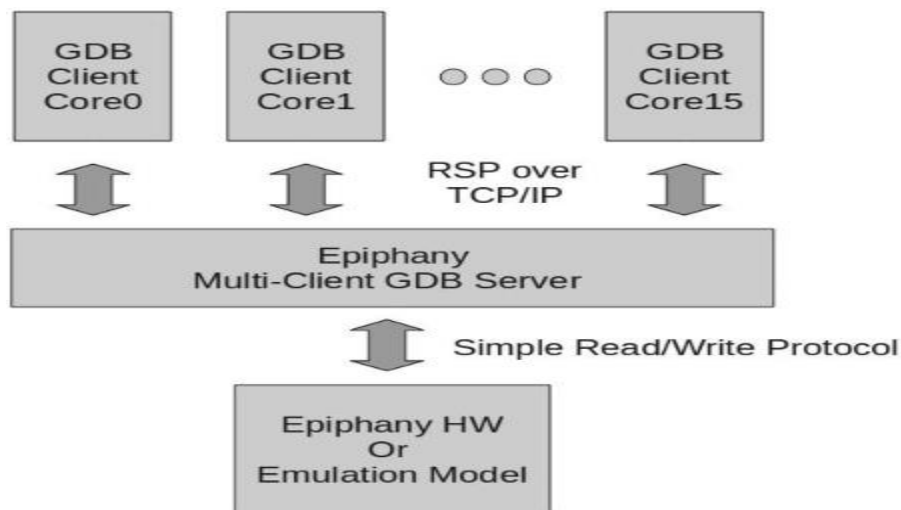


Figure 3-4: The eServer Client-Target Connection Concept

To start the Target Server, open a terminal window and type the following in the command line:

```
$ e-server -hdf  
${EPIPHANY_HOME}/bsps/zedboard/zed_E16G3_512mb.xml
```

The debug server now responds with status messages regarding the connection process and the results of the memory test, if performed. The output should be similar to the following:

```
Using the HDF file: zed_E16G3_512mb.xml  
Listening for RSP on port 51000  
Listening for RSP on port 51001  
Listening for RSP on port 51002  
Listening for RSP on port 51004  
Listening for RSP on port 51005  
:  
Listening for RSP on port 51006  
Listening for RSP on port 51007  
Listening for RSP on port 51010  
Listening for RSP on port 51009  
Listening for RSP on port 51011
```

Table 3-7: eServer Command line Options

Argument	Note
-hdf <i>file</i>	Mandatory argument, specifies the platform specific description file containing platform definitions, normally located in the esdk/bsps/{platform} directory.
--show-memory-map	Print out the supported memory map.
--test-memory	Test the memory before serving the debugger clients (*only for select platforms)

The run-time connection from a host to the epiphany target is performed via the eHAL library.

## 3.4 Epiphany SDK utilities (E-UTILS)

### 3.4.1 Introduction

The Epiphany SDK is provided with a group of command-line utility programs. These programs are used to perform Epiphany system related tasks during program development and debugging.

The e-utils programs include:

```
e-reset
```

```
e-loader
e-read
e-write
```

### 3.4.2 Reset Utility (E-RESET)

The Epiphany reset utility (e-reset) is used to reset the Epiphany subsystem, in case it gets stuck due to some unstable situation, or in order to bring it to a known state.

```
$ e-reset
```

### 3.4.3 Loader Utility (E-LOADER)

The Epiphany loader (e-loader) is responsible for loading programs onto the hardware platform. The input to the loader is a compiled and linked Epiphany program that was generated by e-gcc/e-ld. currently, the loader supports binary images formatted as a text file with a standard S-record (known as SREC) file format. This format is an ASCII hexadecimal ("hex") text encoding for binary data. The S-record is an output of the binary utility ‘e-objcopy’.

When loading a binary image on the chip there is a need to translate the internal core addresses to global space addresses. During compile time, the build tools do not know what core will be the target of the executable. This information is known only at load time. Thus, the insertion of the core ID data has to be done prior to sending the SREC file to the e-loader. When loading images of more than one core, each partial SREC has to be pre-processed separately.

```
e-loader [-s|--start] [-r|--reset] <e-program> [row col [rows cols]]
```

Table 3-8: Loader Command line Options

Option	Function
-r, --reset	Perform a full hardware reset of the Epiphany platform before loading the program.
-s, --start	With this option set, the loaded programs are started after they have finished loading on all cores in workgroup.
<e-program>	Path to the program image to load onto the core workgroup.
row, col	Absolute coordinates of first core in workgroup to be loaded. The default values are the platform’s first physical core.
rows, cols	Size of cores workgroup to be loaded. The default values are (1, 1).
h, --help	Display a help message.

After building an Epiphany elf program, translate from elf to S-record program load format:

```
$ e-objcopy --srec-forceS3 --output-target srec main1.elf main1.srec
```

Load program onto the target, on a 4\*4 block of cores starting at core 0x808 (32, 8) and start it immediately after:

```
$ e-loader --start main.srec 32 8 4 4
```

Or, perform a system reset and load the program onto a single core at the chip's origin. Then wait for host command to start the program.

```
$ e-loader --reset main.srec
```

### 3.4.4 Memory Read Utility (E-READ)

The Epiphany memory read utility (e-read) is used to read words from memory locations on the Epiphany chip(s) or the External Memory.

```
e-read [-v|-r] <row> [<col>] <address> [<num-words>]
```

Table 3-9: e-read Command line Options

Option	Function
<row>	Row coordinate of the target core. To read data from External Memory, enter -1.  The core coordinates are relative to the platform's chip bounding box. That is, all of the Epiphany chips are considered one workgroup, where the first core of the first chip is at coordinates (0,0).
[<col>]	Row coordinate of the target core. When reading from External Memory, this parameter is omitted.
<address>	The start address of the read data. Address is given as local space when reading from a core memory, or as an offset from the platform's External Memory base. Address should be in hexadecimal format and is rounded down to the word (4-bytes) alignment.
[<num-words>]	Number of words to read from the target, starting at <address>. If this parameter is omitted, a single word is read.
[-v]	Verbose mode - print more information.
[-r]	Raw mode - print only the memory contents.

### 3.4.5 Memory Write Utility (E-WRITE)

The Epiphany memory write utility (e-write) is used to write words to memory locations on the Epiphany chip(s) or the External Memory.

```
e-write [-v] <row> [<col>] <address> [<val0> <val1> ...]
```

Table 3-10: e-write Command line Options

Option	Function
<row>	Row coordinate of the target core. To read data from External Memory, enter -1.  The core coordinates are relative to the platform's chip bounding box. That is, all of the Epiphany chips are considered one Workgroup, where the first core of the first chip is at coordinates (0,0).
[<col>]	Row coordinate of the target core. When reading from External Memory, this parameter is omitted.
<address>	The start address of the read data. Address is given as local space when reading from a core memory, or as an offset from the platform's External Memory base. Address should be in hexadecimal format and is rounded down to the word (4-bytes) Alignment.
[<val0> <val1> ...]	Number of words to write to the target, starting at <address>. If  This parameter is omitted, the input is taken in interactive mode from the standard input, one word at a time, until an empty line is entered. Values are entered as 32-bit hexadecimal numbers.

### 3.5 Epiphany Hardware Utility Library (e-Lib)

The Epiphany Hardware Utility library provides functions for configuring and querying the Epiphany hardware resources. These routines automate many common programming tasks that are not provided by the C and C++ languages and are specific to the Epiphany architecture.

The master header file for the eLib, which includes all the per-family headers, is the "e-lib.h" header file. Include this header file at the beginning of a program that uses the eLib functions and objects.

```
#include "e-lib.h"
```

In order to use this library to build an Epiphany program, use the e-gcc compiler option -le-lib on the build command line.

### 3.6 Epiphany Host Library (eHAL)

The Epiphany Hardware Abstraction Layer (eHAL) library provides functionality for communicating with the Epiphany chip when the application runs on a host. The host can be a PC or an embedded processor. The communication is performed using memory writes to and reads from shared buffers that the applications on both sides should define. The library interface is defined in the e-hal.h header file.

In order to use this library in your application, the compiler and linker must be configured with the paths to the header file and the library binary. In your tools options use the following configurations:

```
$ gcc -I${EPIPHANY_HOME}/tools/host/include \  
      -L${EPIPHANY_HOME}/tools/host/lib -le-hal ...
```

# Chapter 4: Frame Structure

## 4.1 Introduction

The bits that are carried on the radio waves undergo a lot of processing before these are ready for transmission. The processing makes them resilient to the attenuation, and also packs them in a way to make them economical to transmit over the air. The processing starts right from the point where they leave the MAC layer so the first stage of processing is the Transport Channel Processing which is called the bit chain in Figure (4-1) and the bit sequence resulting from this stage is input to the next stage which is called the symbol chain as in Figure (4-2). Hence, we can say that the PUSCH coding processing is divided into 2 processing chains, the bit chain and the symbol chain. Here In the project we are working in the physical layer for the uplink shared data and we are interested in the symbol chain to be implemented.

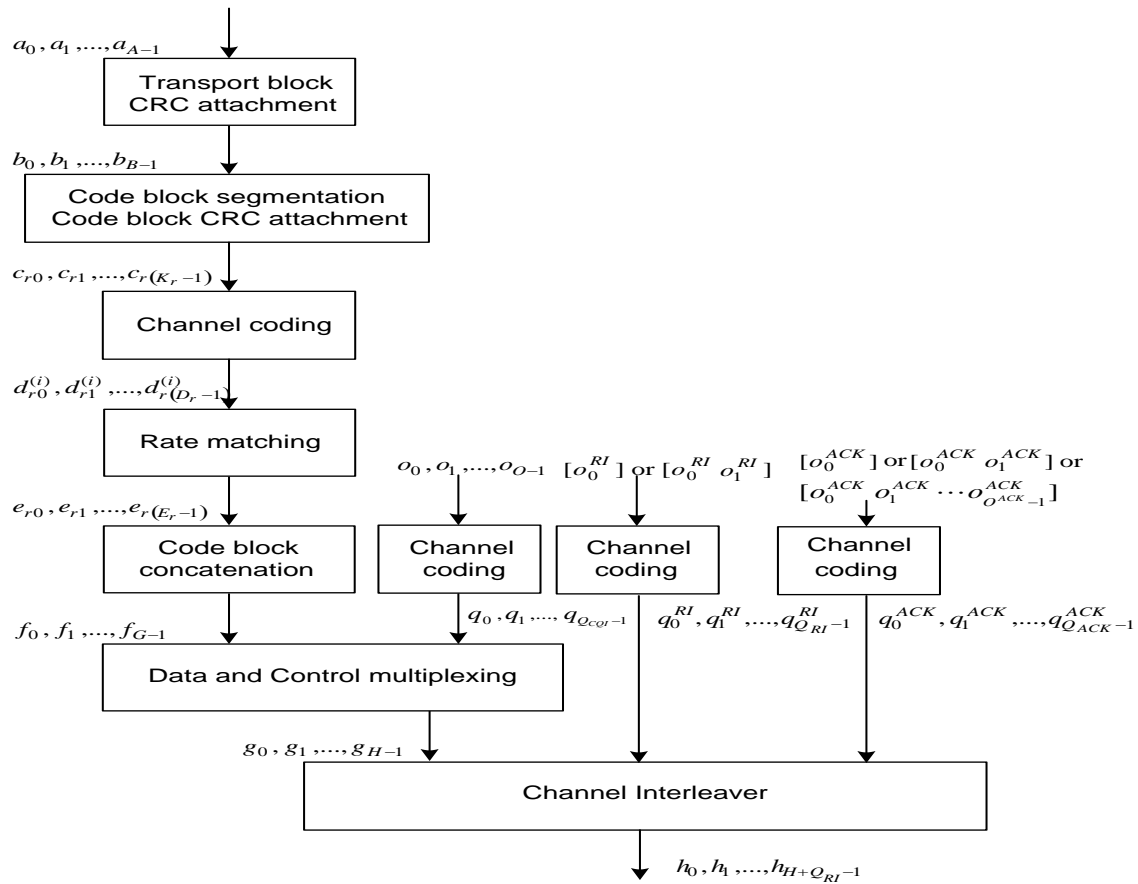


Figure 4-1: PUSCH Bit Processing Chain

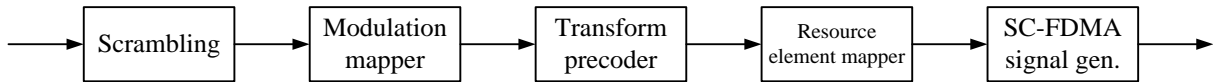


Figure 4-2: PUSCH Symbol Processing Chain

Let's now take a quick overview about the symbol processing chain blocks.

The baseband signal representing the physical uplink shared channel is defined in terms of the following steps: [10]

- Scrambling (change the input sequence by using a known random sequence)
- Modulation of scrambled bits to generate complex-valued symbols
- Transform Precoding to generate complex-valued symbols
- Mapping of complex-valued symbols to resource elements
- Generation of complex-valued time-domain SC-FDMA signal for each antenna port

But before we are talking about the symbol chain blocks in deep, their use and implementation ways, let's talk about the Frame Structure.

There are two types of frame structure in the LTE standard, Type 1 is FDD, it uses Frequency Division Duplexing (uplink and downlink are separated by frequency), and type 2 is TDD, it uses Time Division Duplexing (uplink and downlink are separated in time). There are more than one time unit for the frame structure as shown in Table (4-1).

Table 4-1: Frame units V.S. their times.

<b>UNIT</b>	<b>TIME (ms)</b>
Frame	10
Half Frame	5
Sub-frame	1
Slot	0.5



## 4.2 Frame Structure Type 1

Now, we are going to illustrate the FDD frame. Frame structure type 1 is applicable to both full duplex and half duplex FDD. Full duplex FDD means I can send and receive on different frequencies at the same time whereas half duplex means that UE can't receive while transmitting even if they are on different frequencies.

Each radio frame consists of 20 slots numbered from 0 to 19, each slot of length  $T_{\text{slot}} = 0.5$  ms. A subframe is defined as two adjacent slots, e.g. subframe number 5 consists of time slot number 10 and 11, to generalize we can say that sub-frame  $i$  consists of slots  $2i$  and  $2i+1$ .

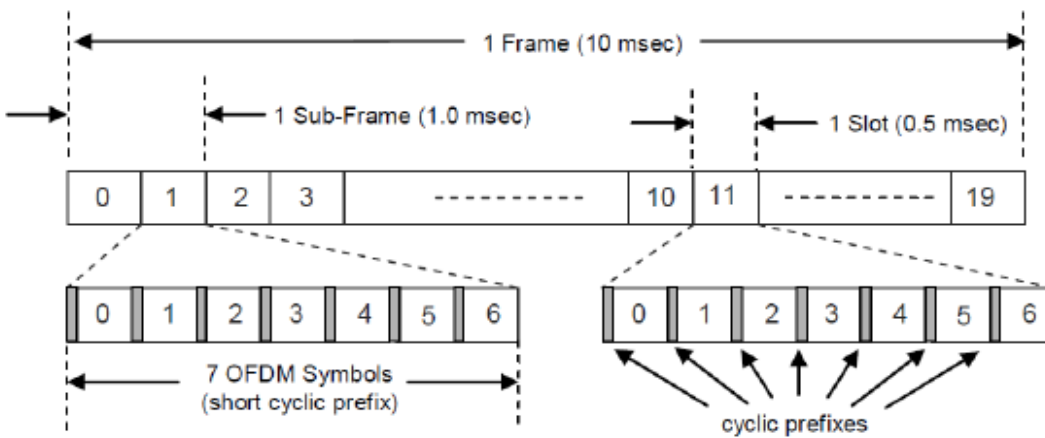


Figure 4-3: FDD Frame Structure Type 1

We can conclude from Figure (4-3), time duration for one radio frame is 10 ms, hence we can say that we have 100 radio frame per second. The number of samples in one frame is 307.2 K samples and also we can say that the number of samples per second is  $307.2\text{K} \times 100 = 30.72$  M samples. And as subframe is 1 ms, so we have 10 subframes per frame, and each subframe has two slots, so we have 20 slots per the whole frame, each slot has as shown 7 symbols in time and 12 subcarriers at frequency.

We can see also smaller structures within a symbol. At the beginning of the symbol we can see a very small span called 'Cyclic Prefix' and the remaining part is the real symbol data. There are two types of Cyclic Prefix. One is 'Normal Cyclic Prefix' and the other is 'Extended Cyclic Prefix' which is longer than the Normal Cyclic Prefix. These cyclic prefixes affect the number of symbols in one slot, when using 'Normal Cyclic Prefix' the number of symbols are 7 symbols while when

using 'Extended Cyclic Prefix' the number of symbols become 6 symbols (As the length of one slot is fixed and cannot be changed, the number of symbols that can be taken in a slot will be decreased. Hence we can have only 6 symbols if we use 'Extended Cyclic Prefix'). The first OFDM symbol in a slot is longer than the rest of OFDM symbols in the same slots. For short cyclic prefix (Normal) and long cyclic prefix (Extended), Table 4-2 specifies their lengths.

Table 4-2: Cyclic Prefixes Types V.S. Lengths

<b>Configuration</b>	<b>Cyclic prefix length</b>
Normal cyclic prefix	160 for $l = 0$ 144 for $l = 1, 2, \dots, 6$
Extended cyclic prefix	512 for $l = 0, 1, \dots, 5$

By observing the FDD frame, we notice that the time slot is the smallest unit that we can use, Right? By just observing we may say yes, but by knowledge we definitely will say NO. How? Let's get deeper into the time slot, and know more about it.

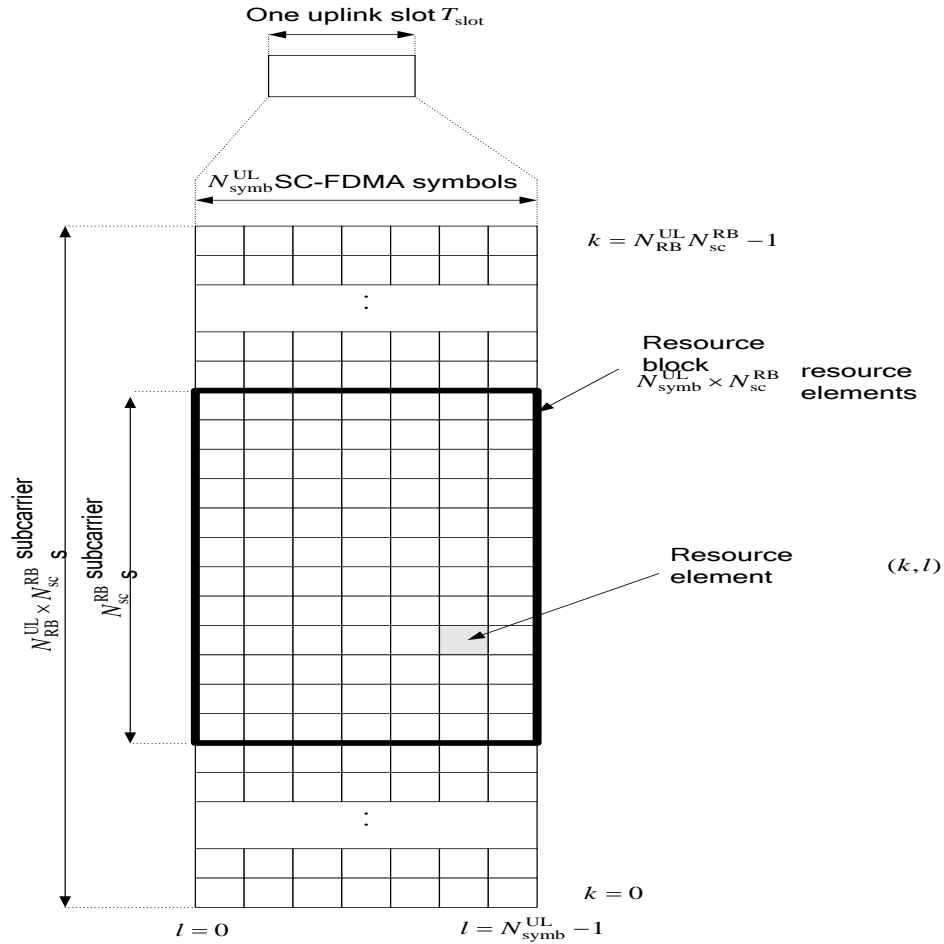


Figure 4-4: Uplink resource grid

We are right now inside the time slot. The time slot consists of a lot of elements, each is called Resource Block (RB) as shown in the bold box of the above figure. The RB is considered the smallest element that can be assigned for one user, but as shown, it's for sure not the smallest element in the time slot at all.

Each RB consists of 12 subcarriers in frequency and 7 symbols in time, 1 subcarrier x 1 symbol is called Resource Element (RE), which is the smallest part of the frame and contains a single complex value representing data from a physical channel or signal. The resource block is 180 kHz wide in frequency and 1 slot long in time. In frequency, these 180 kHz contains 12 subcarriers or in some cases can contains 24 subcarriers, so we can say, resource blocks are either 12 x 15 kHz subcarriers or 24 x 7.5 kHz subcarriers wide in frequency. But the mostly common used in channels and signals are 12 subcarriers per resource block.

The number of resource blocks and hence the subcarriers per frame depends on the bandwidth. The bandwidths defined by the standard are 1.4, 3, 5, 10, 15, and 20 MHz. Table 4-3 below shows how many subcarriers and resource blocks there are in each bandwidth for uplink and downlink.

Table 4-3: Subcarriers, resource blocks corresponding to the BW.

<b>Bandwidth (MHz)</b>	<b>Resource Blocks</b>	<b>Subcarriers (downlink)</b>	<b>Subcarriers (uplink)</b>
1.4	6	73	72
3	15	181	180
5	25	301	300
10	50	601	600
15	75	901	900
20	100	1201	1200

For downlink signals, the DC subcarrier is not transmitted, but is counted in the number of subcarriers. For uplink, the DC subcarrier does not exist because the entire spectrum is shifted down in frequency by half the subcarrier spacing and is symmetric about DC [12].

Another clear Figure (4-5) for FDD Frame of 1.4 MHz bandwidth and Normal CP for more illustration of the FDD frame structure, time slots and resource block and its content.

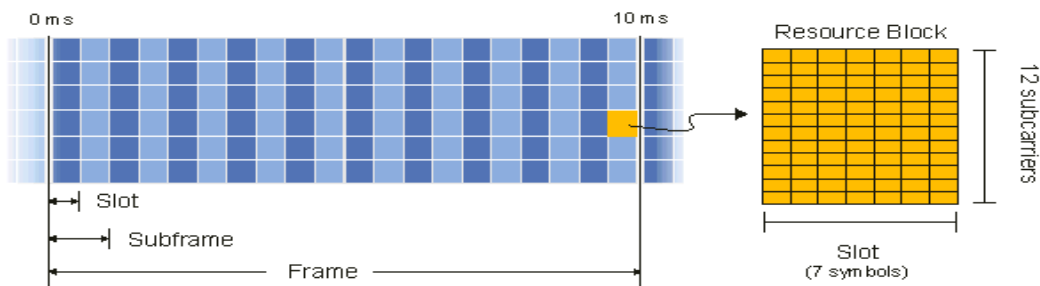


Figure 4-5: LTE FDD Frame of 1.4 MHz, Normal CP

### 4.3 Frame Structure Type 2

That was the frame structure type 1, now we are going to speak about frame structure type 2. Frame structure type 2 is applicable to TDD. In TDD mode, the uplink and downlink subframes are transmitted on the same frequency and are multiplexed in the time domain. The locations of the uplink, downlink, and special subframes are determined by the uplink-downlink configuration. The following is an illustration of a TDD frame with uplink-downlink configuration set to 2 and special subframe configuration set to 6.

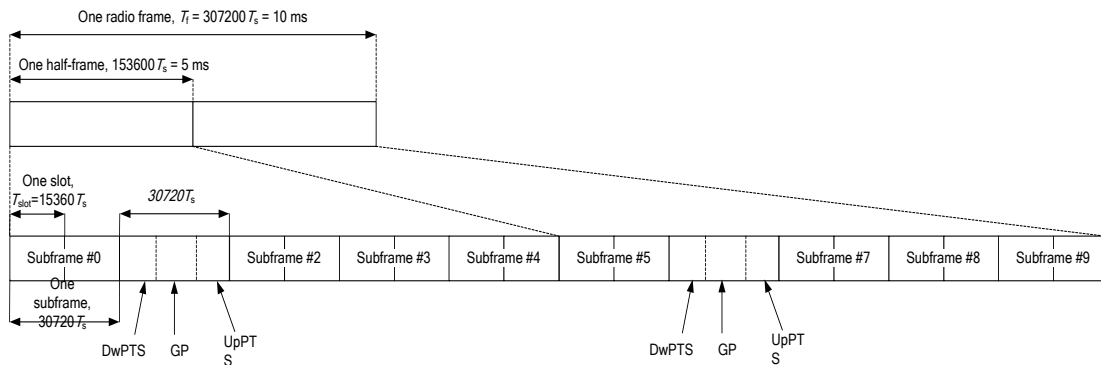


Figure 4-6: TDD Frame Structure type 2

Subframe number 1 and 6 contain three sections DwPTS, GP and UpPTS. These subframes are called Special subframes. They are used for switching from downlink to uplink. Subframes 0 and 5 and DwPTS are always reserved for downlink transmission. UpPTS and the subframe immediately following the special subframe are always reserved for uplink transmission. DwPTS is the Downlink Pilot Time Slot. DwPTS contains P-SS. PDSCH can also be transmitted during DwPTS when DwPTS is configured to be longer than a slot. UpPTS is the Uplink Pilot Time Slot. UpPTS can contain PRACH and SRS, but cannot contain or PUCCH or PUSCH. GP is a guard period between DwPTS and UpPTS. PRACH format 4 begins in the guard period. Otherwise, nothing else is transmitted during the guard period. The lengths of these three sections are determined by the special subframe configuration index (specified by the Dw/Gp/Up Len parameter). There are 9 possible configurations in the standard [10].

Now, we are going to talk about Uplink user transmission. Uplink user transmission consists of uplink user data (PUSCH), user control channels (PUCCH), random-access requests (PRACH), and sounding reference signals (SRS). FDD and TDD uplink transmissions have the same physical

channels and signals. The only difference is that TDD frames include a special subframe, part of which can be used for SRS and PRACH uplink transmissions. We will now illustrate the whole LTE uplink frame which contains the whole data, the shared one and the reference signals and whole control data. But before we illustrate that, the downlink frame will be illustrated which we can understand easily. Another reason for understanding the downlink frame is that we use an LTE open source to be reference for us, the one we've used is "OpenLTE". And we've made a survey before this choice about more than one open source for the LTE (this survey is shown in Table 4-4), then after this survey the "OpenLTE" is chosen to be our reference code. By the way, "OpenLTE" codes don't have a code for uplink frame and we need to do it whereas there is no reference, but a downlink frame structure code has been found in the reference code, so we needed to understand this code to make some code like it but for uplink as we need in the project, that's we can say the main reason for understanding the downlink frame structure first before moving into the uplink frame structure.

Table 4-4: Open Source LTE Survey

	<b>OPENLTE</b>	<b>LTE-SIM</b>	<b>SRS-LTE</b>
<b>3GPP LTE release Number</b>	Release 10	.....	LTE Release 8 compliant.
<b>Programming language</b>	C++	LTE-Sim has been written in C++, using the object-oriented paradigm	C
<b>Testing</b>	octave code is available for test and simulation of downlink transmit and receive functionality and uplink PRACH transmit and receive functionality	tested on - Linux i386 Ubuntu 12.04 with g++-4.6 - Linux amd64 Ubuntu 12.04 with g++-4.6 - Linux i386 Ubuntu 10.10 with g++-4.4 - Linux amd64 Ubuntu 10.10 with g++-4.4 - Mac OS X v10.6.8	UE receiver tested and verified with Amari soft LTE 100 eNodB and commercial LTE networks
<b>Implementation target</b>	E-NodB	B (eNB), Home eNB (HeNB)	SDR(UE), evolved Node B (eNB)
<b>The license</b>	License Affero General Public License	( GNU GENERAL PUBLIC LICENSE)	GPLv3
<b>Links we use</b>	[15] To download it... [7]	[17] [18]	[19]

## 4.4 Downlink Frame Structure

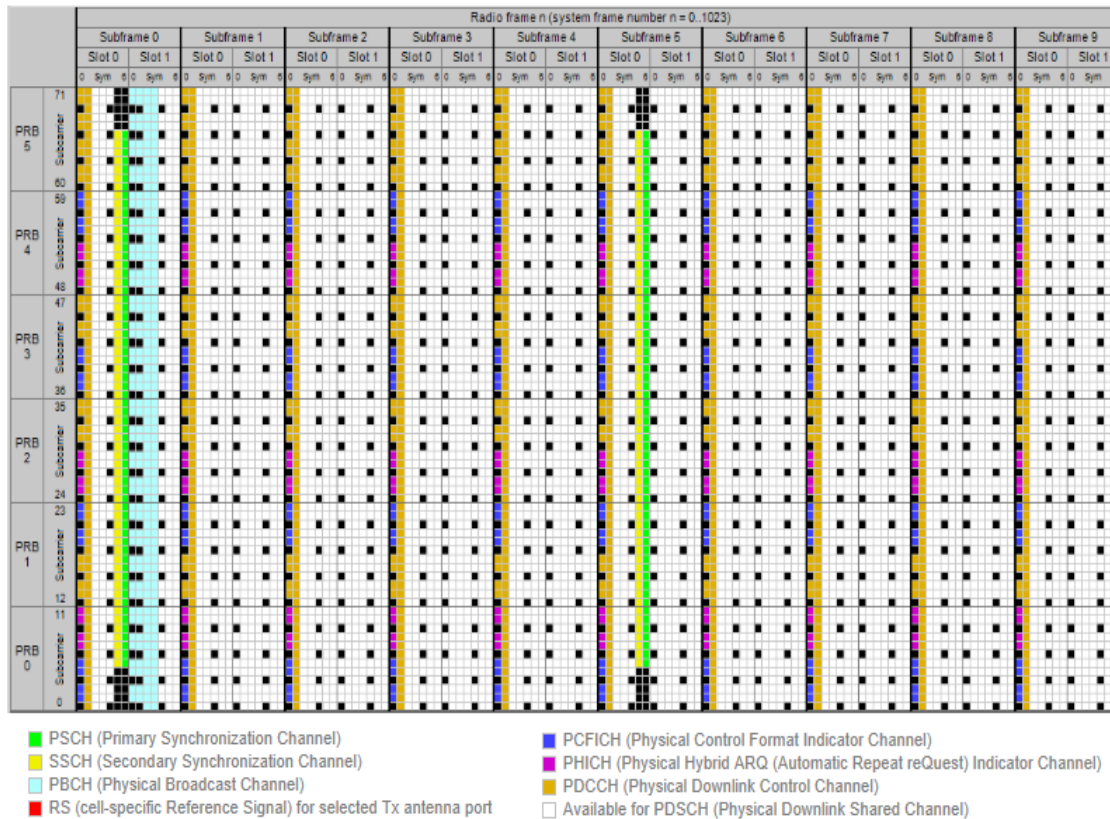


Figure 4-7: LTE resource grid for FDD

Figure (4-7) shows the overall subframe structure from "LTE Resource Grid".

We can see that the shared data in PDSCH is the available right area and the shared data is the one we are interested in to be implemented its chain, but in our project we refer to PUSCH not the downlink one. By the way, as shown in Figure (4-7) we have a lot of channels which is considered the control signals we have, each channel corresponds to its color which is considered its place in the resource grid.

We can see also black points in the grid, these are the reference signals. They are very important signals to be sent during the grid, some of its importance points is:

- 1- Synchronization
- 2- Channel estimation
- 3- Handover procedures, as within it we can know the power of mine and can be determined if handover is needed to be occur or not.



The rest of channels in our case or project, they are out of the project scope, it's just will be given to us to place it in its place in the subframe code.

## 4.5 Uplink Frame Structure

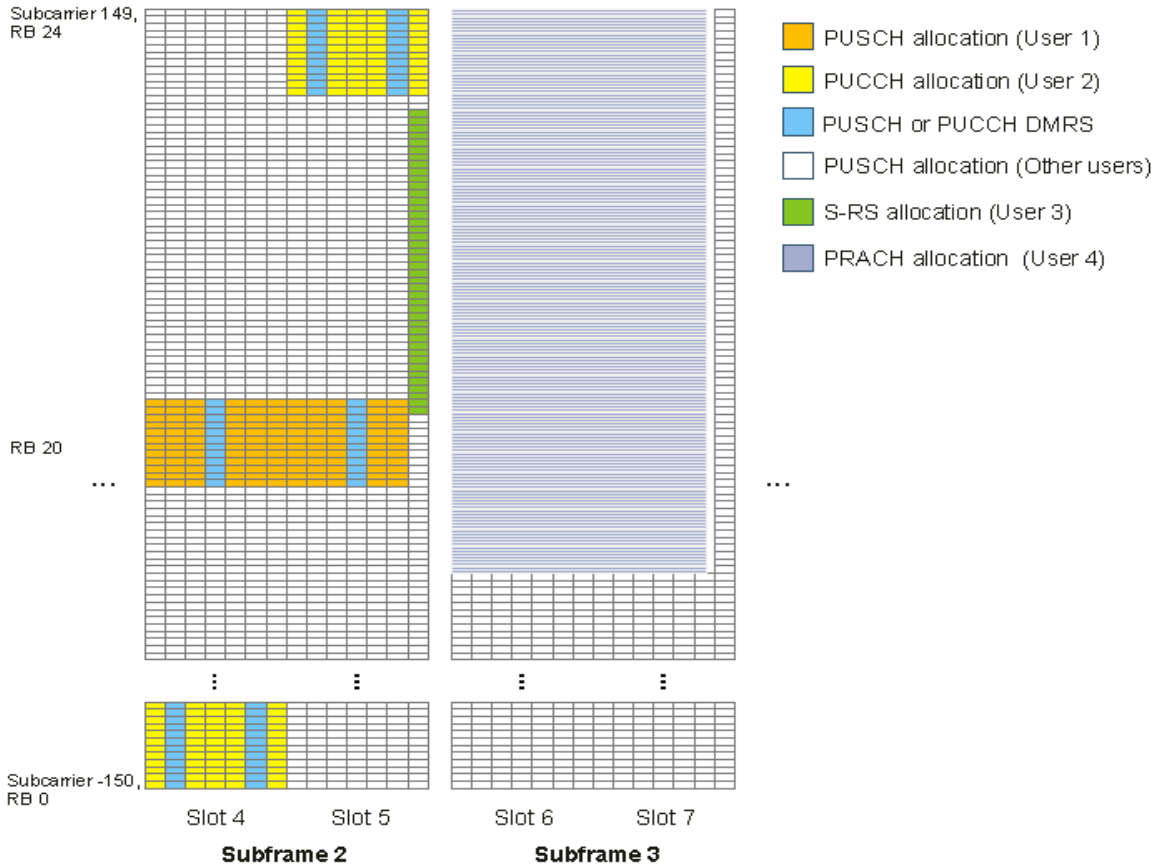


Figure 4-8: LTE Uplink Subframe 2-3 of 5 MHz, Normal CP

As we have seen in Figure (4-7) the downlink subframe structure, Figure (4-8) represents the uplink frame structure, actually they are only 2 subframes from it, subframe 2 and sub frame 3. But as before we have data signals represented in PUSCH (the channel we are interested in and going to implement its symbol chain) and have some control signals represented in the reference signals and the rest of channels except PUSCH. We are not interested in the control signals as it's out of our project scope, and we are interested in PUSCH. The benefit of this chapter is to know and understand the frame structure of both types specifically type 1 cause that's we need to

implement by the end of implementing our blocks of PUSCH and collect the rest of data in the subframe, that's actually what's has to be done.

## Chapter 5: Fixed Point Representation

### 5.1 Motivation

Our codes in the projects have been generated as implementing them firstly with MATLAB (may be this step has several versions) with this step we generate files similar to the test cases that we have. Then write the code in C and MATLAB. Then compare between both the generated files from MATLAB and others ones from c as shown in Figure (5-1).

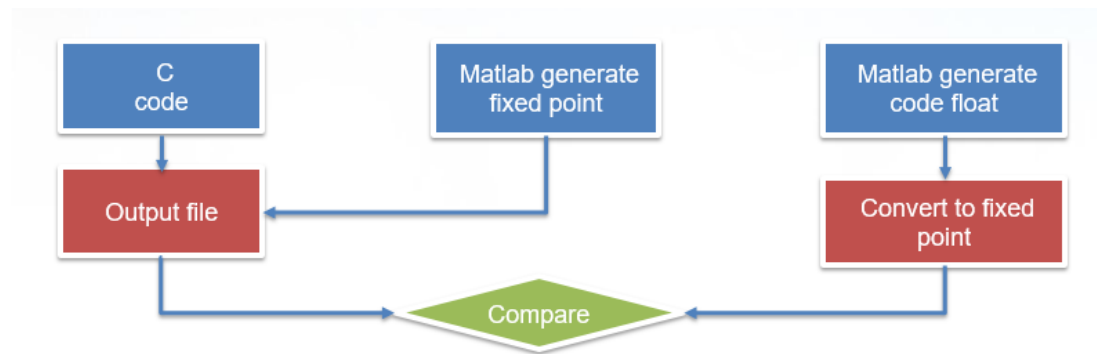


Figure 5-1: Progress of implement the codes of blocks to generate test cases

As the Figure (5-1) showed that we deal in general with fixed point .so let's know why and how we used fixed point.

In a digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of bits (1's and 0's). How hardware components or software functions interpret this sequence of 1's and 0's is defined by the data type. Binary numbers are represented as either fixed-point or floating-point data types. In order to implement an algorithm such as communication algorithms, the algorithm should be converted to the fixed-point domain and then it should be described with Hardware Description Language (HDL). In HDL coding process, it is necessary to indicate the size of the variables and registers. The registers should be large enough to represent the value of parameters with the desired precision [24]

To be accurately construct an algorithm, double or single precision floating-point data and coefficient values should be used. However there is significant processor overhead required to perform floating-point calculations resulting from the lack of hardware based floating-point

support. In some cases such as with lower powered embedded processors there is not even compiler support for double precision floating-point numbers. Floating-point overhead limits the effective iteration rate of an algorithm [25].

As we have in our platform Parallella which includes 16 superscalar floating point RISC CPUs (eCore), each one capable of two floating point operations per clock cycle and one integer calculation per clock cycle. So to build our algorithm that will consume huge power and processing time if we used floating point.

To improve mathematical throughput or increase the execution rate (i.e. increase the rate the algorithm could be repetitively run), calculations can be performed using fixed-point representations. Fixed-point representations require the programmer to create a virtual decimal place in between two bit locations for a given length of data (variable type).

Fixed-point data types can be either *signed* or *unsigned*. Signed binary fixed-point numbers are typically represented in one of different ways.

## 5.2 Ways of Fixed Point Representation

### 5.2.1 Sign/magnitude

- The leftmost bit represents the sign of the no.
- The remaining bits represent the binary equivalent of the magnitude of the no.
  - Advantages
- Simple and easy to understand
  - Disadvantages
- The no. zero can be represented in two ways: 00000000 and 10000000.
- The sign bit and the magnitude part have to be handled separately complicates the design of the circuit used for addition particularly.

### 5.2.2 One's complement

It's done by getting One's complement of number.

### 5.2.3 Two's complement

- Two's complement is the way every computer I know of chooses to represent integers (similar to the design of the memory).
- To get the two's complement negative notation of an integer, you write out the number in binary. You then invert the digits, and add one to the result.
- Two's complement is the most common representation of signed fixed-point numbers and is the only representation used by Fixed-Point Toolbox in MATLAB.

## 5.3 Q-Format number representation

As we saw fixed-point data type helps us to know what happens in the hardware. In the other words when an algorithm is represented in floating-point domain, all of the variables have 64 bits (in MATLAB programming). So all of the operations are done with large number of bits. We know that it is impossible to implement an algorithm with large number of flip flops. Because large number of flip flops need a larger area, and more power consumption. In order to solve this problem, the algorithm should be converted to the fixed-point domain. In the fixed-point domain a pair  $(W, F)$  is considered for each of the parameters in the algorithm, where  $W$  is the word length of the parameters and  $F$  is the fractional length of the parameters. It is obvious that larger  $W$  and  $F$  results in a better performance and lower bit error rate (BER) but the design needs a large silicon area. On the other hand, smaller  $W$  and  $F$  result in a larger BER but less area. So we should choose suitable values of  $(W, F)$  for each parameter in the algorithm. For this reason, a simulation should be run for the algorithm to get the *dynamic range* of the parameters. Simulation results indicate the dynamic range of the variables and the number of bits for  $W$  and  $F$ , which are used to represent the variables with the desired precision [24].

So from the previous section we see that  $W$  represents both QI (integer bits) and QF (floating bits). And this  $W$  usually corresponds to variable widths supported on a given processor. Typical word lengths would be {8, 16, 32} bits corresponding to {char, int, long int} C/C++ variable types commonly implemented in compilers for microcontrollers or DSPs.

For example, a binary representation of a generalized fixed-point number (either signed or unsigned) is shown below:

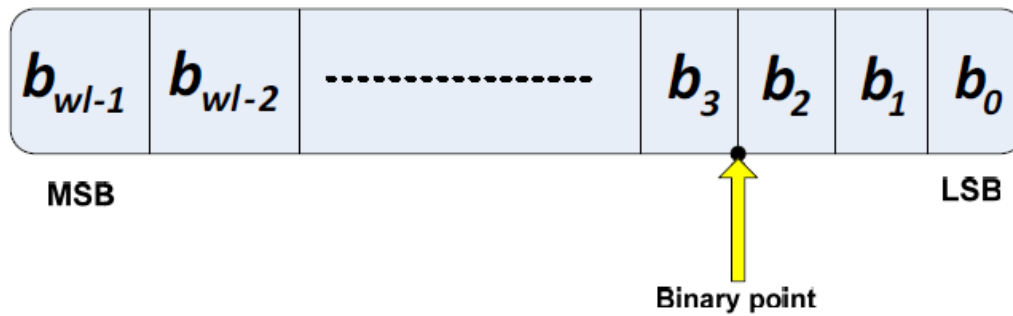


Figure 5-2: showing how the conversion of floating point to fixed point happens

Where:

$b_i$  Is the  $i$ th binary digit.

$wl$  Is the word length in bits.

$b_{wl-1}$  Is the location of the most significant, or highest, bit (MSB).

$b_0$  Is the location of the least significant, or lowest, bit (LSB).

*Fixed-Point Toolbox* provides fixed-point data types in MATLAB and enables algorithm development by providing fixed-point arithmetic like what we used in DFT block where we used `fi` to generate fixed point. Where `fi` takes parameters of (v,s,w,f) returns a fixed-point object with value v, Signed property value s, word length w, and fraction length f. Fraction length can be greater than word length or negative.

But also we can use steps of conversion floating point to fixed point

Consider a floating-point variable, x:

**Step 1:** Calculate  $y = x * 2^F$ , where  $F$  is the fractional length of the variable. Note that y is represented in decimal.

**Step 2:** Round the value of y to the nearest integer value. For example:

$$\text{round}(3.68) = 4, \text{round}(-1.7) = -2$$

**Step 3:** Convert y from decimal to binary representation and name the new variable Z.

**Step 4:** Now, we assume that  $Z$ , needs  $n$  bits to represent the value of  $y$  in binary. On the other hand, we obtain the values of  $W$  and  $F$ , from the simulation. So the value of  $W$  should be equal or larger than  $n$ . If Small value is chosen for  $W$ , we should truncate  $Z$ . If  $W$  is larger than  $n$ ,  $(W - n)$  zero-bits add to the leftmost of  $Z$  [24]. For example: truncate  $(3.68) = 3$ , round  $(-1.7) = -1$

So can verify this method by the MATLAB tool (fi function).

Note: in some cases instead of using  $y = x * 2^F$  we use  $y = x * 2^{F-1}$  when want to shift the range of signed integer number.

## 5.4 Converting Floating Point to Fixed Point Example

In our modulation code we needed to store the LUT (look up tables) of all modulation schemes (QPSK, 16QAM, 64QAM). So we found that the largest number we need to implement is  $\pm 7/\sqrt{42}$  so 2 bits for integer bits and if we used standard int 16 bit so  $Q=F=14$  for fractional bits. So this MATLAB code is LUT of QPSK we deal with real and complex separately.

```

Q=14; %2^Q
% table example
% QPSK
QPSK_I= round([1 ; 1 ; -1 ; -1] ./sqrt(2) * (2^Q) );
QPSK_Q= round([1 ; -1 ; 1 ; -1] ./sqrt(2) * (2^Q) );
%% store the converted value in 16 bit
QPSK_I = ((QPSK_I<0)*(2^16)+QPSK_I);
QPSK_Q = ((QPSK_Q<0)*(2^16)+QPSK_Q);

st_string = '';
%% output the QPSK LUT
st_string = [st_string, sprintf( 'uint16 QPSK[4*2]={\n'}];
for k = 1:length(QPSK_I)
st_string = [st_string, sprintf( '\t\t0x%04X, 0x%04X, \n',
QPSK_I(k),QPSK_Q(k))];
end
st_string = [st_string, sprintf( '};\n\n')];

%% saving the LUT to a file
fid = fopen('QPSK_LUT.txt','w','n');
fprintf(fid, '%s', st_string);
fclose(fid);

```

Notes:

In the all blocks we have to save after each block the same  $Q$  where:

–  $Q_m + Q_m \Rightarrow Q_m$

–  $Q_m \times Q_n \Rightarrow Q_{m+n}$

So to save the same  $Q_m$  we shifted the result  $n$  times to right usually we present the complex symbols as 16 bit real (MSB) and 16 bit real (LSB).Both of them as fixed points.



## Chapter 6: Scrambler

### 6.1 Introduction

In this chapter we are going to talk about the first block we implemented in the symbol chain, “Scrambler”. As we have said before that the scrambler input is the bits that get out from the last block in the bit chain “Interleaver”.

Before we talk about the block itself and its implementation way and the drawbacks that meet us in its implementation, let’s define the scrambler first and mention its use and importance first, then go to the implementation way and the rest of chain blocks.

What’s the scrambler? Simply it’ a block takes some input bits and its output is the same number of input bits but has another random values. But how is this made? Simply the scrambler block generates some random bits then it makes XOR operation between these random bits and the input sequence, so the output will be another sequence differs than the input one. The scrambler generated random bits is called pseudo random sequence.

Ok, now what is the importance of the scrambler block or why we use it? Actually there are more than one reason for that, but the most important reasons are security and data saving. How? Regarding the data saving, simply, if we have a long input bits or sequence of one’s or zero’s, and the channel was a fading channel and in this part of the channel the whole data can be lost, so I can’t ever know that there was here a long sequence of bits of the same value, therefore a great data will be lost, also a long sequence of ‘0’s or ‘1’s can degrade the timing/clock synchronizer performance or may even result in loss of synchronization, consequently, the importance of scrambler is appeared here, by xoring these long sequence of bits that have the same values, these long sequence will be disappeared and replaced by another random sequence based on the xored code with the input, so I can save the data from being lost and avoid long sequences of bits of the same value. Scrambler also introduces security (as part of an encryption procedure) to protect the data. After we do the scrambling operation, for sure at the receiver a reverse block to reverse the scrambler operation is needed to retrieve the original sent data, this block is called “Descrambler”.

Now after we know what the scrambler is and why the scrambler is being used, let's go on and talk about the implementation ways especially we code in C as there is no MATLAB code to be placed on the board, so it was a little bit weird and strange for us.

## 6.2 Implementation ways:

Pseudo-random sequences are defined by a length-31 Gold sequence. The output sequence  $c(n)$  of length  $M_{PN}$ , where  $n = 0, 1, \dots, M_{PN} - 1$ , is defined by [10]

$$\begin{aligned} c(n) &= (x_1(n + N_c) + x_2(n + N_c)) \bmod 2 \\ x_1(n + 31) &= (x_1(n + 3) + x_1(n)) \bmod 2 \\ x_2(n + 31) &= (x_2(n + 3) + x_2(n + 2) + x_2(n + 1) + x_2(n)) \bmod 2 \end{aligned}$$

What do these equations mean? And how we can generate our code from them? And what about the coding process of them? That's what we are going to explain now.

As we have known that we generate some bits and call them pseudo random sequence, this sequence represented in the first equation  $C(n)$ . This equation means that we will generate some other sequences to enable us generate this code, these sequences called  $X1(n)$ ,  $X2(n)$ . In the first equation  $N_c = 1600$ . What does it mean? It means that I will generate  $X1$  sequence and  $X2$  sequence for a lot of iterations but what I will take from them is after the generated sequence number 1600, after this number I will consider  $X1$  and  $X2$  sequences and use them to generate my required code. As we must know that any two bits if I added them then multiply them in modulus two as in the last equations that's equal to XORing these two bits with each other. The first equation means that we XOR the  $X1$  sequence with  $X2$  sequence and start from bit number 1600 in these two sequences to get the code wanted. But why  $N_c = 1600$  not any other number? What we conclude and got by searching and asking that the most fit and suitable number that can protect data and randomize the input and achieve the scrambler functionality efficiently is this number and this reason is for all the numbers that you can find in the standard and ask yourself why this number specifically that I must use! And also what we have found that these numbers came by try and error and choose the best fit number, and also not by try and error only, there are international meetings that's held between most countries representatives to argue, choose and assure on the

final chosen numbers. As we said also that the main reason for scrambling is changing the input sequence randomly to save and protect data, so by taking some bits after specific number like 1600 then start my code from this bit, provides more security for my data.

Now, what we need to get the code is getting X1 and X2 sequences and those can be got from the last two equations for them. Let's explain them,

$$x1(n + 31) = (x1(n + 3) + x1(n)) \bmod 2$$

X1 bits generation is starting from bit number 32. As  $n$  is starting from 0. This equation generates bit by bit in X1 sequence. The first 31 bits is given in the standard as an initialization for X1 then we get the rest of sequence.  $x_1(0) = 1, x_1(n) = 0, n = 1, 2, \dots, 30$ . The first bit  $x1(0) = 1$  and the rest 30 bit is equal zero's. To generate bit number 32 from the last equation  $x1(31)$  we xor bit number 4 with bit number 1 from the same x1 sequence  $x1(0) \text{ xor } x1(3)$ , and bit 33 is gotten from bits 2 and 5 and so on for the rest of sequence. From that we can conclude that each new bit in the sequence depends on the last bits of the same sequence so  $x1(n)$  depends on itself for the rest bit sequence.

Let's now move on to talk about the second sequence used to get the generated code.

$$x2(n + 31) = (x1(n + 3) + x1(n + 2) + x1(n + 1) + x1(n)) \bmod 2$$

This sequence is slightly different than the last one in its initialization. It's obvious from the equation that we do like the last one to get bit number 32 we xor between bits numbers 1, 2, 3 and 4. But for the first initialized 31 bits mentioned in the standard, it has a little difference that  $x1$ . We have the next two equations to get the first 31 bits.

$$c_{init} = n_{RNTI} \cdot 2^{14} + \lfloor n_s / 2 \rfloor \cdot 2^9 + N_{ID}^{cell}$$

$$c_{init} = \sum_{i=0}^{30} x_2(i) \cdot 2^i$$

The first 31 bit of x2 can be got from the last second equation. First we must get  $C_{init}$

And then get x2 from it.  $C_{init}$  In the last first equation comes from some parameters, these parameters comes from the e-NodeB to the UE used.  $N_{rnti}$  Represents the RNTI assigned to the UE, RNTI stands for Radio Network Temporary Identifier. RNTIs are used to differentiate/identify

a connected mode UE in the cell, a specific radio channel, a group of UEs in case of paging, a group of UEs for which power control is issued by the e-NodeB, system information transmitted for all the UEs by the e-NodeB etc...

There are several RNTI types in LTE such as SI-RNTI, P-RNTI, C-RNTI, Temporary C-RNTI, SPS-CRNTI, TPC-PUCCH-RNTI, TPC-PUSCH-RNTI, RA-RNTI, and M-RNTI. Each RNTI's usage, its value range etc. [14].

By the way, these parameters have range that it must not exceed, and by getting the needed configurations of these parameters, we can get  $C_{init}$  and then get the X2 initialization. But what does this equation mean?  $c_{init} = \sum_{i=0}^{30} x_2(i) \cdot 2^i$ .

Simply it means that  $C_{init}$  is an ordinary decimal number and to get X2 bits, you can convert  $C_{init}$  from decimal number to a binary number and take from the converted number the first 31 bits as an initialization for X2 to apply then X2 equation and get the rest of bits. That was about the Scrambler equations that will be applied inside the code.

Now, let's take a tour inside the written code and some of the drawbacks we've faced in this block. The code is just applying the previous explained equations, but by applying the test cases (it's an input file has some values and another output file with other values, for the code to be right, the output file from the code must be the same as the tested one when entered the given input file to the code) by the way, by applying these test cases, the output file was not as the tested one, so there is something wrong with the code and trying a lot by changing some parameters and values and make a lot trials to know where is the wrong in the code, by debugging the code, it's found that:

the input file is read and saved in the memory from right to left (Little Endian) and (X1 and X2) are saved from left to right (Big Endian), so a wrong bits is xored with each other, so for sure the output must be wrong and to discover that, it take a very long time from us.

After solving the last problem, the output is still wrong and there is no reason for that, by some trials to discover what's wrong, it's discovered that the input test file was wrong so also for sure the output must be wrong as we xor this input file with the generated code, and also this drawback takes much time to be solved.

And finally the code worked properly and the output test file from the code is as the same as the given one we have, to check out code as shown below.

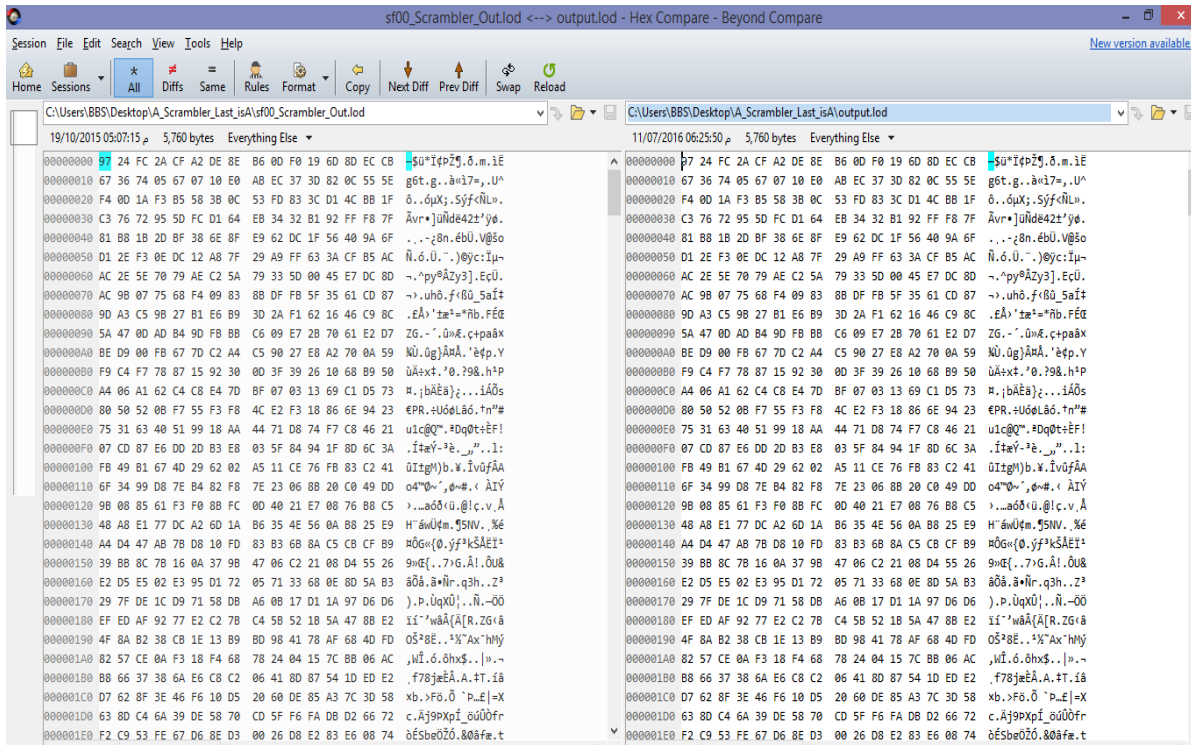


Figure 6-1: The output from Scrambler code versus test cases

In Figure (6-1), the output from the code and test cases are compared and both are the same. The program used to compare the files is called Beyond Compare.

Some of the most important things that must be taken in consideration is that we write a code for just a software:

- The codes must be modular. To say that a code or a component is modular, some specifications should be satisfied; the component should be implemented in separate .c/.h files. To link to the library, the testing (harness) code should include the header file.
- All the configurations and the memory POINTERS used should be in a private context (structure) this structure type is specific for this component. The component should be used through public APIs. In general 2 APIs should be used, one for configuration and one for processing. The arguments of configuration API is the component structure pointer and the current configuration. The input for the processing API should be only the pointer to the component context.

- The memory allocation and structure definition should be done in the upper testing code and by calling the component APIs, and passing pointers to the memory and the component structure, the component can register the memory and configurations in the component structure.

This modularization property is important to ease the components integration and dynamic and static structure manipulation, otherwise, conflicts and bugs are highly probable.

But for writing a code for a hardware system; we must take care of

- memory use in the code
- The time if it's critical
- Don't use malloc () or any other dynamic memory allocation as it's harmful in embedded systems because:
  - The memory is limited in embedded systems. (It is important that you do not suddenly find yourself out of memory).
  - Fragmentation - embedded systems can run for years which can cause a severe waste of memory due to fragmentation.
  - Not really required. Dynamic memory allocation allows you to reuse the same memory to do different things at different times. Embedded systems tend to do the same thing all the time (except at startup).
  - Speed. Dynamic memory allocation is either relatively slow (and gets slower as the memory gets fragmented) or is fairly wasteful (e.g. buddy system).
  - If you are going to use the same dynamic memory for different threads and interrupts then allocation/freeing routines need to perform locking which can cause problems servicing interrupts fast enough.
  - Dynamic memory allocation makes it very difficult to debug, especially with some of the limited/primitive debug tools available on embedded system. If you statically allocate stuff then you know where stuff is all the time which means it is much easier to inspect the state of something [26].

For the time it's critical in our case because we write for an LTE system which is the whole subframe takes 1 ms which is this block must be in micro seconds at least. For example, the process of generation the pseudo codes must be out the run time to not take time from the available 1 ms.

What's made in the code to optimize as most as possible is:

Regarding the memory, X1 and X2 were only single integer number not an array, and that's made by overwriting, generate new bit by the last operation and overwrite in same place of it, so some great memory has been saved.

Regarding the time, a way for saving and reducing the time is using a matrix, matrix to generate the pseudo code outside the run time by a quick way and also will be ready once needed for xoring with the input the thing that will reduce the code time a lot.

## Chapter 7: Modulation & Precoder

### 7.1 Introduction:

Modulation is the process of making messages (digital symbols or analogue signals) ready and suitable to be sent through the channel. There are many reasons why the modulation process is vital; to change the frequency of the (analogue) message, in order to match the channel's available frequency band or the antenna radiation range. In digital communication, it is quite necessary to make an analogue signal out of the digital symbols. Modulation also allows more than one user to simultaneously transmit through a shared channel. (Of course multiple access should be considered as a distinct concept from the modulation, but in a general picture, it is the result of using the modulation wisely). In the frequency domain, multiple adjacent tones or subcarriers are each independently modulated with complex data. Also the modulation is needed for improving the bandwidth, efficiency (where transmission of a plain digital signal gives you no opportunity for error correction, synchronization. Modulation along with coding solves the problem), noise immunity, antenna sizing and the Multiplexing (difficult to be achieved without modulation).

The modulation schemes used in the LTE standard include QPSK, 16QAM and 64QAM. Figure (7-1) shows the constellation diagrams of these three modulation schemes.

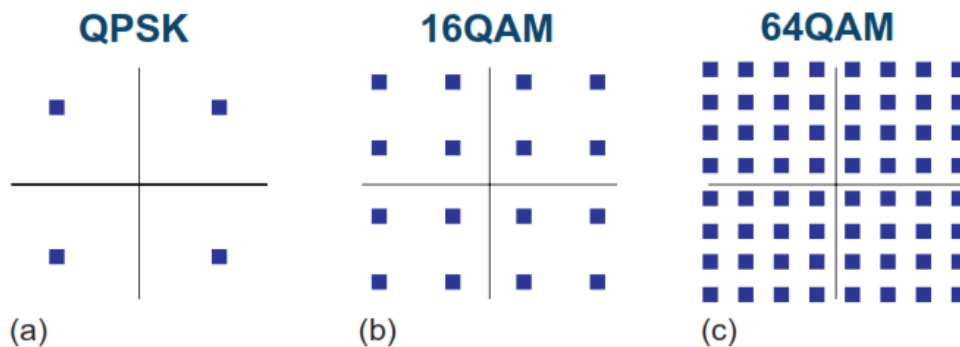


Figure 7-1: Signal constellations for: (a) QPSK; (b) 16QAM; (c) 64QAM.

In the case of QPSK modulation, each modulation symbol can have one of four different values, which are mapped to four different positions in the constellation diagram. QPSK needs 2 bits to encode each of its four different modulation symbols. The 16QAM modulation involves using 16



different signaling choices and thus utilizes 4 bits of information to encode each modulation symbol. The 64QAM modulation involves 64 different possible signaling values and thus requires 6 bits to represent a single modulation symbol.

Table 7-1: the number of bits corresponding to each modulation scheme

Modulation schemes	Bits per symbol
QPSK	2
16QAM	4
64QAM	6

After we've known about the modulation importance and definition, our target in the project is to achieve 64QAM while minimizing the processing power for LTE L1 processing in UEs.

The modulation mappings are applicable for the physical uplink shared channel, so our target to use the higher rate this mean to implement up the 3 modulation schemes. These are the tables of modulation mapper takes binary digits, 0 or 1, as input and produces complex-valued modulation symbols,  $x=I+jQ$ , as output.

### 7.1.1 QPSK

In case of QPSK modulation, pairs of bits,  $b(i),b(i+1)$  are mapped to complex-valued modulation symbols  $x=I+jQ$  according to Table 7-2 [10].

Table 7.2: QPSK modulation mapping

$b(i),b(i+1)$	$I$	$Q$
00	$1/\sqrt{2}$	$1/\sqrt{2}$
01	$1/\sqrt{2}$	$-1/\sqrt{2}$
10	$-1/\sqrt{2}$	$1/\sqrt{2}$
11	$-1/\sqrt{2}$	$-1/\sqrt{2}$

### 7.1.2 16QAM

In case of 16QAM modulation, quadruplets of bits,  $b(i),b(i+1),b(i+2),b(i+3)$  are mapped to complex-valued modulation symbols  $x=I+jQ$  according to Table 7-3 [10].

Table 7-3: 16QAM modulation mapping

$b(i), b(i+1), b(i+2), b(i+3)$	$I$	$Q$
0000	$1/\sqrt{10}$	$1/\sqrt{10}$
0001	$1/\sqrt{10}$	$3/\sqrt{10}$
0010	$3/\sqrt{10}$	$1/\sqrt{10}$
0011	$3/\sqrt{10}$	$3/\sqrt{10}$
0100	$1/\sqrt{10}$	$-1/\sqrt{10}$
0101	$1/\sqrt{10}$	$-3/\sqrt{10}$
0110	$3/\sqrt{10}$	$-1/\sqrt{10}$
0111	$3/\sqrt{10}$	$-3/\sqrt{10}$
1000	$-1/\sqrt{10}$	$1/\sqrt{10}$
1001	$-1/\sqrt{10}$	$3/\sqrt{10}$
1010	$-3/\sqrt{10}$	$1/\sqrt{10}$
1011	$-3/\sqrt{10}$	$3/\sqrt{10}$
1100	$-1/\sqrt{10}$	$-1/\sqrt{10}$
1101	$-1/\sqrt{10}$	$-3/\sqrt{10}$
1110	$-3/\sqrt{10}$	$-1/\sqrt{10}$
1111	$-3/\sqrt{10}$	$-3/\sqrt{10}$

### 7.1.3 64QAM

It's actually the same as 64 QAM, In case of 64QAM modulation, hexuplets of bits,  $b(i), b(i+1), b(i+2), b(i+3), b(i+4), b(i+5)$  are mapped to complex-valued modulation symbols  $x=I+jQ$  according to Table in the standard [1] section (7.1.4).

## 7.2 Implementation Ways

The block of scrambled bits resulting in a block of complex-valued symbols  $d(0), \dots, d(M_{\text{symb}} - 1)$  supporting (QPSK, 16QAM, 64QAM) [10]

LTE transmits data by dividing it into slower parallel paths that modulate multiple subcarriers in the assigned channel. The data is transmitted in segments of one symbol per segment over each subcarrier [11].

Hence, the code of modulation stored in it the LUT's of the 3 schemes modulation depending on the  $Q_m$  (modulation order) it decides which LUT will be accessed, and the size of the input of scrambler must be divisible by modulation order ( $Q_m$ ) as LTE standard shows.

Before we continue writing in modulation, let's stop here a little bit and talk about the LUT. It's an abbreviation of the (Look up Table), it's simply like an array, we save the data in it. Why we use it here? As we use a fixed point not floating point, so no float number is allowable to be used, hence, we can't use the last tables of the modulation schemes and as we've said we converted these floats number to a fixed number and save them in a new table has the new values, this table or array in the code is called LUT.

Back again to the modulation, the same rules of writing c code are applied here in modulation code. Where there are 2 mainly API one of them for configuration parameters which are modulation order and size of input file. And the second API is performed the modulation mapper function.

in our case the  $Q_m = 4$  so it's 16 QAM modulation order so each time read 4 bits from the input then accesses the 16QAM LUT for real part and complex part then concatenate these 2 fixed points numbers (real in MSB and imaginary in LSB) each of them stored in 16 bit. And that's how we get single complex symbol then repeat it till Length (Bytes) of input file divided by  $Q_m$ .

Then comparing the result with the test case either we have from Axxcelera with the one we have generated from MATLAB code. And finally the code worked properly and the output test file from the code is as the same as the given one we have, to check out code as shown below.

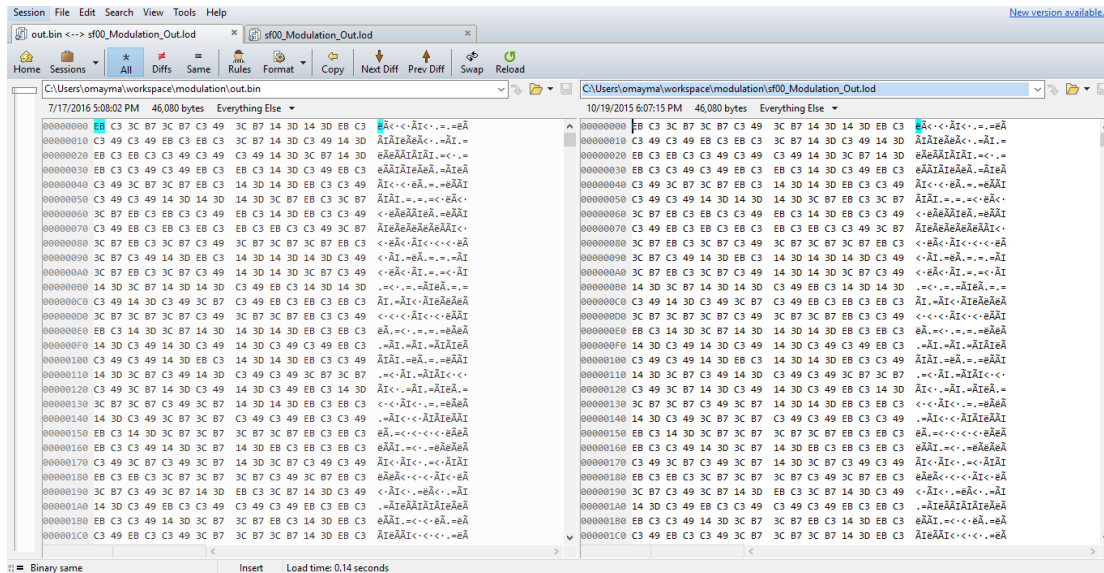


Figure 7-2: The output from Modulation code versus test cases

The case of  $Q_m=6$  (64QAM) has a condition in the code that the length ( Bytes)

should be dividable by 24 bits (3 bytes) as each loop generate 4 symbols from 3 bytes of the input .

In the platform, modulation is performed parallel in the multiple core and while the code depends mainly on access LUT so that make it more applicable with embedded system application.

### 7.3 Precoder

Precoding is a technique or process to distribute the incoming data to the antenna ports. It supports multi-layer transmission in multi-antenna wireless communications. it's for sure this process is not that simple to just say this part of data must go to antenna port 1 and the this part for antenna port 2 and so on, No it's not like that simple. In reality, the data from all the layers gets combined in a specific way and then those combined data gets distributed to each of the antenna port.

It's also can be used to be the pre-knowledge of the channel, the receiver is a simple detector, such as a matched filter, and does not have to know the channel side information. This technique will reduce the corrupted effect of the communication channel [30].

This was the importance and use of Precoder in a very simple way. Actually we are not interested to see or understand the complex equations and the ways the data are assigned to each antenna port

as it's considered the implementation way and we don't implement it, we have known about its use and importance and that's enough for our case, because it's used for the MIMO systems, the systems that have multi input and multi output each on an antenna port. And in our project we don't use MIMO Systems. Therefore, we don't have to implement the Precoder block.

# Chapter 8: DFT

## 8.1 Introduction

When transmitting data from the mobile terminal to the network, a power amplifier is required to boost the outgoing signal to a level high enough to be picked up by the network. The power amplifier is one of the biggest consumers of energy in a device and should thus be as power efficient as possible to increase the operation time of the device on a battery charge. But there is a problem for this transmission that, the PAPR are high so the data peaks will be clipped so we need to change the PA to solve this problem, but this solution will be very costly in the uplink side as we must in this case change each amplifier in all handsets which is not applicable as the handset will be very expensive so to solve this problem SC-FDMA is used instead of OFDMA in the uplink side, but in the downlink side, we still use OFDMA as changing the power amplifier of the E-NodeB will be only one time and not costly as changing the PA of each handset.

By applying SC-FDMA in the uplink side, DFT block is used before the IFFT block so we can reduce the high peaks of the data and protect it from clipping, that's why the DFT block is not exist in the downlink side and exists only in the uplink side.

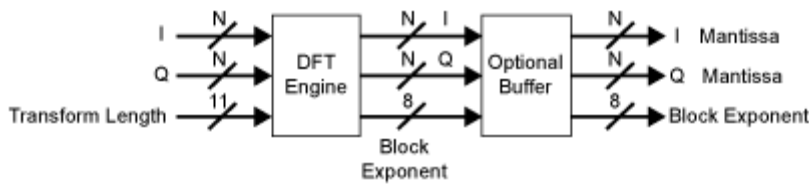


Figure 8-1: shows the DFT/IDFT reference design blocks.

## 8.2 Implementation way

### 8.2.1 Theory and Output Generated Signals

The block of complex-valued symbols  $d(0), \dots, d(M_{\text{symb}} - 1)$  is divided into  $M_{\text{symb}} / M_{\text{sc}}^{\text{PUSCH}}$  sets, each corresponding to one SC-FDMA symbol. Transform precoding shall be applied according to

$$z(l \cdot M_{sc}^{\text{PUSCH}} + k) = \frac{1}{\sqrt{M_{sc}^{\text{PUSCH}}}} \sum_{i=0}^{M_{sc}^{\text{PUSCH}}-1} d(l \cdot M_{sc}^{\text{PUSCH}} + i) e^{-j \frac{2\pi i k}{M_{sc}^{\text{PUSCH}}}}$$

$$k = 0, \dots, M_{sc}^{\text{PUSCH}} - 1$$

$$l = 0, \dots, M_{\text{ymb}} / M_{sc}^{\text{PUSCH}} - 1$$

Resulting in a block of complex-valued symbols  $z(0), \dots, z(M_{\text{ymb}} - 1)$ . The variable  $M_{sc}^{\text{PUSCH}} = M_{\text{RB}}^{\text{PUSCH}} \cdot N_{sc}^{\text{RB}}$ , where  $M_{\text{RB}}^{\text{PUSCH}}$  represents the bandwidth of the PUSCH in terms of resource blocks, and shall fulfil

$$M_{\text{RB}}^{\text{PUSCH}} = 2^{\alpha_2} \cdot 3^{\alpha_3} \cdot 5^{\alpha_5} \leq N_{\text{RB}}^{\text{UL}}$$

Where  $\alpha_2, \alpha_3, \alpha_5$  is a set of non-negative integers?

This the implementation way from the standard [10]. This way is the same function as we studied in the college which is described as the following equation

$$x(k) = \sum_{i=0}^{N-1} f_n(i) e^{-j \frac{2\pi k i}{N}}$$

Where it  $x(k)$  is sampled frequency domain signal which generated from a sampled time domain signal. Hence if we referred this equation to the standard equating  $X(k)$  will be equivalent to  $z(l \cdot M_{sc}^{\text{PUSCH}} + k)$ . and  $d$  is the input function (complex fixed point from modulation block as in our case precoder using single antenna )

We divide by square ( $M_{sc}^{\text{PUSCH}}$ ) as referred to standard for normalization.

But what's the ( $M_{sc}^{\text{PUSCH}}$ ) well, that's need to go back to chapter (4) .the e-NodB send configurations to each UE .These configurations include ( $M_{\text{RB}}^{\text{PUSCH}}$ ) which represents the PUSCH includes how many resource blocks and we have ( $N_{sc}^{\text{RB}}$ ) which represents the resource block includes how many frequency subcarrier and this value is 12 from PUSCH (chapter 4).

Therefore,  $M_{sc}^{\text{PUSCH}} = M_{\text{RB}}^{\text{PUSCH}} * 12$  so in our test case  $M_{\text{RB}}^{\text{PUSCH}} = 80$ ,  $M_{sc}^{\text{PUSCH}} = 960$

The DFT function is to distribute the complex symbols to the sub frame, the sub frame as we've said consists of 2 slots each of 7 symbols, we have to use 6 OFDM symbols of each slot, so we will use 12 OFDM symbols, and the rest 2 symbols (these 2 OFDM symbols are number 3 in 1<sup>st</sup> time slot and number 10 in the 2<sup>nd</sup> time slot in the sub frame symbols) will be used for inserting

the reference signals. Hence, simply we will generate  $M_{sc}^{PUSCH}$  DFT symbols per OFDM symbol (l) which equal 12 or as the equation from the standard to get (Lmax), therefore each L independently implemented  $M_{sc}^{PUSCH}$  DFT. so it helps to make them parallel operations on 12 core of epiphany platform. The implementation of LUT needed to have high rate that's make reduce the modules, multiplication and division operations as possible ...so we had to make a LUT with the worst case which is at  $M_{RB}^{PUSCH}=100$  (max) then  $M_{sc}^{PUSCH}$  will be 1200 hence the size of phase LUT will be  $2(2\text{bytes in int}16)*1200$  .then to access it at any other case we may use interpolation. But the problem of interpolation we made that was linear interpolation so that increased the BER OR to be more accurate increase the precision loss .to solve this problem we had to build exponential interpolation function which will makes another problem of the complexity of the code with embedded system and reduce the operation rate.

That's why we have implemented a special case of our test case and the phase LUT has size of 960 of type int 16 for each real and complex value of phase. Then in the code we simply implement the previous equation with index to LUT is  $((i*k) \% 960)$ .

Then multiply the input with value of phase and sum them over 960 then apply shift operation to keep the Q (bits of floating point constant=14). Applying the shifting operating outside the internal loop where the Q=28 (Q1=14 from input multiplied by Q2=14 from). Then we have to shift it 14 to right. But in our test case (from the company we have scale factor is 2).so the shifting will be 15 to right to make the file be applicable compared with the test case file.

Applying the shifting operating outside the internal loop is better from the point of view of precision loss. After that divide by square 960 to make it normalized .and finally constant the real and imaginary values as we did in modulation block.

Finally, we compared the generated file from C with the Test Cases and it showed as below.



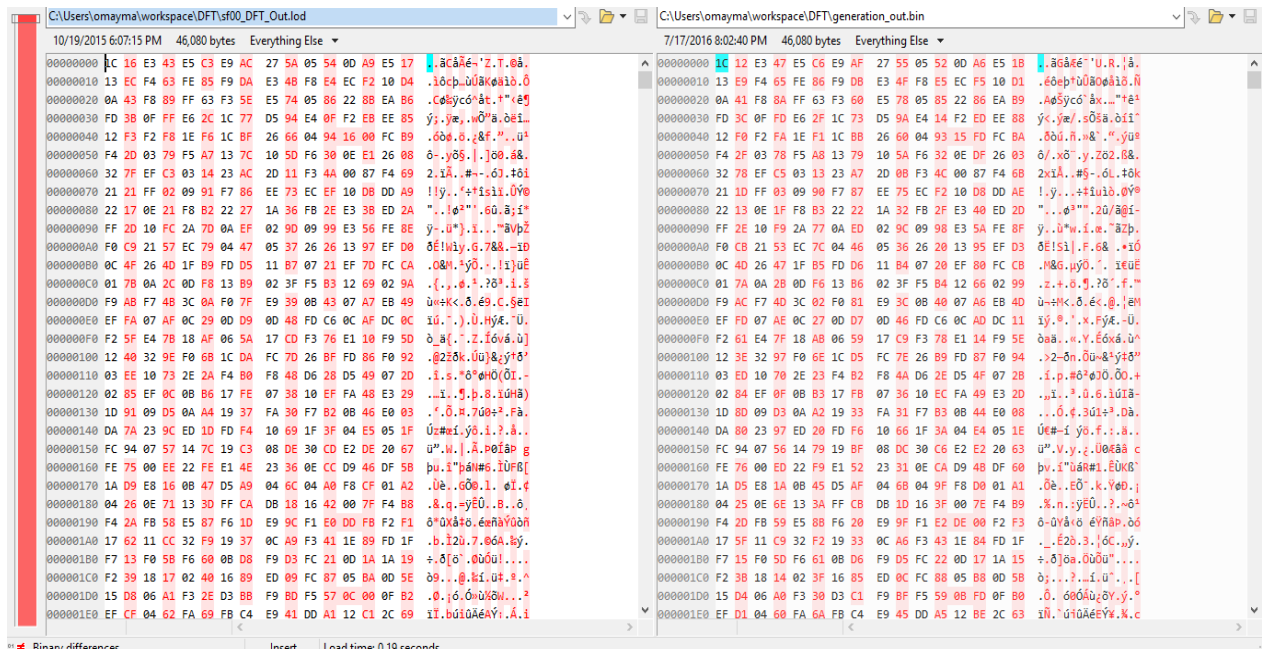


Figure 8-2: The output from DFT code versus test cases

As we saw the generated file not totally as the test cases not like modulation and scrambler. But it differs little bit in the least bits of real part (most 2 Bytes of int32 (complex value)) and it differs little bit in the least bits of imaginary part (least 2 Bytes of int32 (complex value)). That happens due to the arithmetic operation (summation, multiplication and division). As the test cases generated from floating point representation and finally convert then to fix. But the generated file is performed from the beginning with fixed point representation numbers.

Therefore, the generated code due to sifting operation (to keep Q) not 100% accurate but was approached to the right values. Let's try to measure this error (precision loss).

## 8.2.2 Error Measurement Ways

### 8.2.2.1 Comparing the Constellations

Firstly draw the constellations (using MATLAB) of the modulated signal as Figure (8-3).

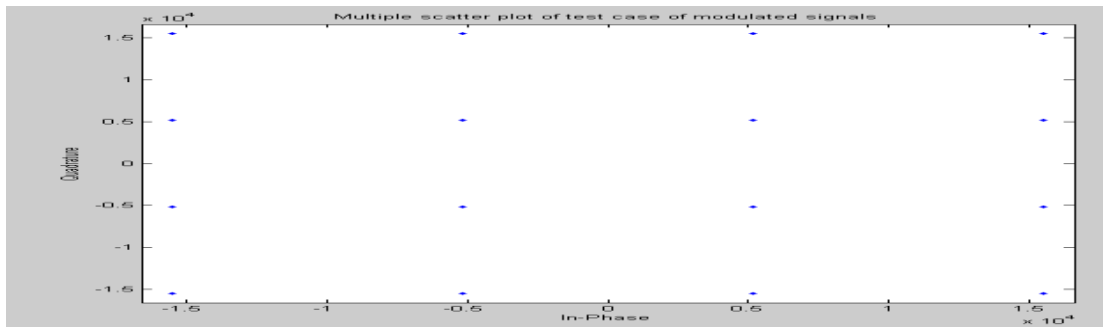


Figure 8-3: constellation plot of test case of modulated signals

Then draw the constellation (using MATLAB) of the signal after IFFT block of test cases of DFT Signals as Figure (8-4)

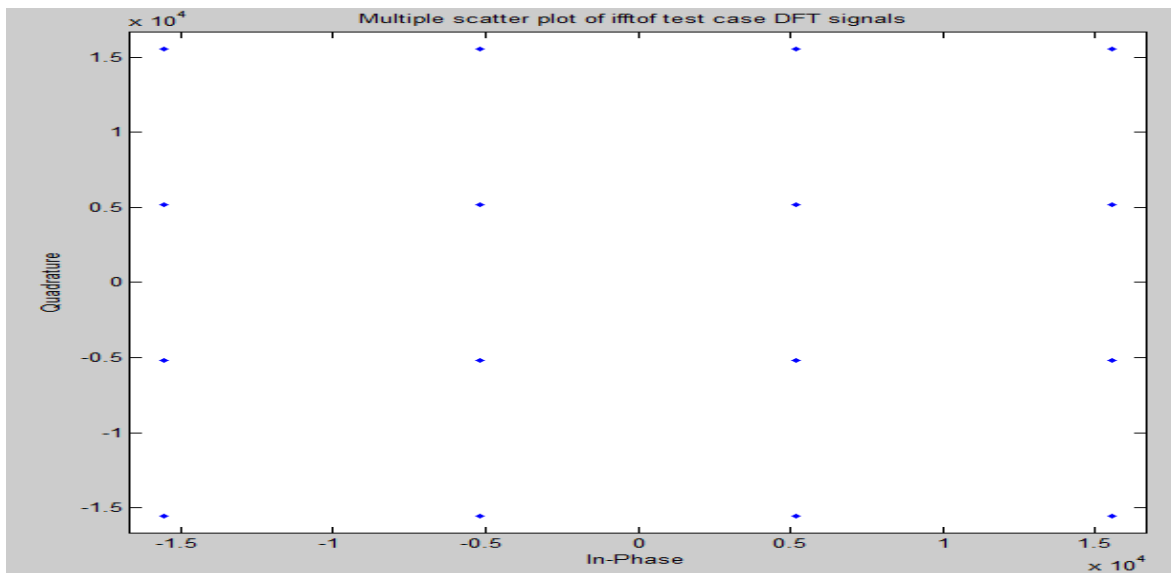


Figure 8-4: constellation plot of IFFT of test case DFT signals

we assume that if the error is zero so when we get the difference between two constellations it will be zero .hence BER(here is persession loss) propotional with the difference between the two previous figures as shown in Figure (8-5)

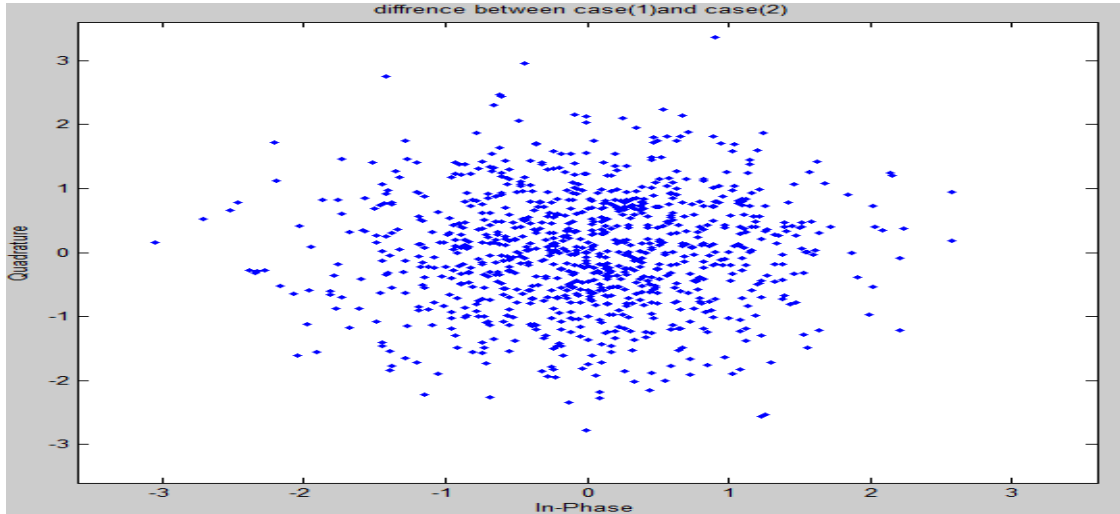


Figure 8-5: difference between case (1) and case (2)

As we see the max differences 3 which too low compared to the last 2 figures constellation was multiplied by  $10^4$  .then the error here is also low which is acceptable.

### 8.2.2.2 Calculating SNR

Measuring SNR of the DFT signal (single to noise ratio). That compares the level of a desired signal to the level of background noise. It is defined as the ratio of signal power to the noise power.

By applying these equations:

$$P_{signal}(dB) = 10\log_{10}(P_{signal}) \quad , \quad P_{error}(dB) = 10\log_{10}(P_{error})$$

$$SNR(dB) = 10\log_{10}\left(\frac{P_{signal}}{P_{error}}\right) = P_{signal}(dB) - P_{error}(dB)$$

The results are:

$$P_{signal}(dB)=118.8268 \quad , \quad P_{error}(dB)=54.1035$$

$$SNR(dB) =64.7233$$

### 8.2.2.3 Calculating EMV

Measuring the error by using EVM or The error vector magnitude (it's called receive constellation error or RCE) is a measure used to quantify the performance of a digital radio transmitter or receiver. A signal sent by an ideal transmitter or received by a receiver would have all constellation points precisely at the ideal locations, however various imperfections in the implementation [31].

Where can be calculated by these equations:

$$EVM(dB) = 10 \log_{10} \left( \frac{P_{error}}{P_{ref}} \right) \quad , \quad EVM(\%) = \left( \sqrt{\left( \frac{P_{error}}{P_{ref}} \right)} * 100 \right)$$

Where:

$P_{ref}$  and  $P_{error}$  Are RMS reference power vector and RMS error power vector, when applying these 2 equation on MATLAB the results are:

$$EVM(dB) = -64.7233 \quad , \quad EVM(\%) = 0.0581$$

## 8.3 IFFT

After the DFT block, the output of it is as we said has a low data peaks so it will not be clipped by the PA. Now it's the last step, sending the data over the air, and that's can be done by the IFFT block. It converts the frequency domain to time domain to be sent within the air. It has a higher number of input pins than the output of DFT because as we know if both DFT and IFFT has the same number of pins the effect of each other will be vanished. IFFT sizes are power of two, and its size is bigger than the DFT block which is a very good thing as the rest of pins we can put Zeros on them, so the overall data peaks will be decreased much more than the output of DFT which is what we want. The sizes of IFFT depends on the Bandwidth we have, as shown in Table (8-1).

Table 8.1: IFFT Sizes corresponding to each BW

<b>Bandwidth (MHz)</b>	<b>IFFT Size</b>
1.4	128
3	256
5	512
10	1024
15	1536
20	2048

DFT output of the data symbols is mapped to a subset of subcarriers, a process called Subcarrier mapping. The subcarrier mapping assigns DFT output complex values as the Amplitudes of some of the selected subcarriers. Subcarrier mapping can be classified into two types: localized mapping and distributed mapping. In localized mapping, the DFT outputs are mapped to a subset of consecutive sub-carriers thereby confining them to only a fraction of the system bandwidth. In distributed mapping, the DFT outputs of the input data are assigned to subcarriers over the entire bandwidth non-continuously, resulting in zero amplitude for the remaining subcarriers. A special case of distributed SC-FDMA is called interleaved SC-FDMA, where the occupied subcarriers are equally spaced over the entire bandwidth. Figure (10-1) is a general picture of localized and distributed mapping [32].

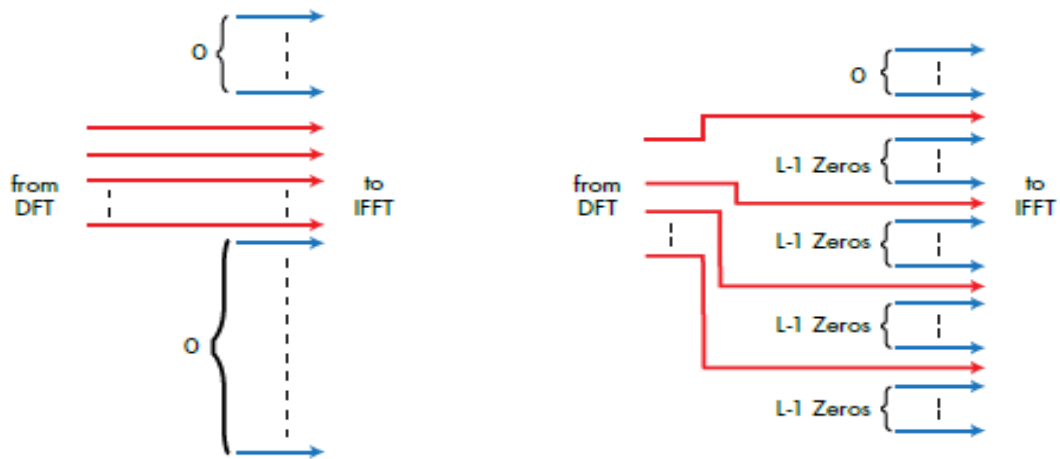


Figure 8-6: Localized mapping vs. Distributed mapping

That was the IFFT concept, we don't implement it in the project because, we were focusing on the project target which optimizing the time and power, so for the future work, it's recommended to do that block and complete the whole chain.

## Chapter 9: Hardware Prototype

### 9.1 Motivation

The demand for compute power is steadily rising. This also holds for embedded systems. Many-core processors provide a large computation power with less energy consumption compared to single core processors. We have choose parallella board to be our prototype because it introduces Epiphany system with has 16-core which will minimize power and it has low cost compared to other commercial many core boards.

### 9.2 FreeRTOS

#### 9.2.1 Background

FreeRTOS is a real-time operating system kernel for embedded devices that has been ported for more than 22 processor architectures [23] and recently to the Epiphany processor [22].

FreeRTOS supports many different compiler tool chains, and is designed to be small, and easy to use for real time embedded systems [21]. The kernel of FreeRTOS has pre-emptive, cooperative and hybrid configuration options and it is written mostly in C language. A Pre-emptive kernel [20] means that the OS can pre-empt (stop or pause) the currently scheduled task if a higher priority task is ready to run, while in a cooperative kernel [20] the running task is not allowed to be interrupted by other task until it yield or it finishes its execution. For better understanding of FreeRTOS the following parts are explained: Tasks in FreeRTOS: A task is a user-defined code with a given priority that performs a special function. In FreeRTOS tasks can have the following states:

**Ready:** When a new is task created, it will go directly to the ready list.

**Running:** The task is currently running.

**Blocked:** Task could be blocked due to accessing of a shared resource.

**Suspended:** All tasks except the running one will be suspended, the suspended tasks are out of scheduling and it needs to be resumed.

At the end of the tasks lifecycle the tasks can be deleted. A state diagram of FreeRTOS tasks are shown in Figure (9-1).

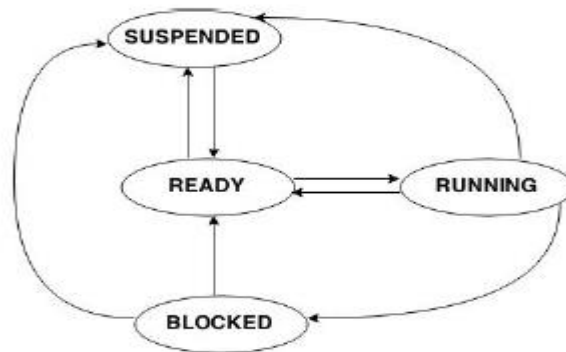


Figure 9-1: State diagram of FreeRTOS tasks [12]

**FreeRTOS Scheduler:** A scheduler works as a decision center that decides which task should run at a particular time. In the ready list the tasks are ordered according to their priority. The scheduler runs the highest priority task in the list, i.e. fixed priority scheduler. The timer interrupt makes the scheduler run periodically at every period tick.

**Communication:** In some systems, tasks need to communicate with each other. In FreeRTOS message queues are used to communicate between tasks.

**Resource Sharing:** Due to the fact that embedded systems have small recourses, away to synchronize the usage of shared resources is required. FreeRTOS use binary semaphore and mutexes to this purpose.

### 9.2.2 Problem formulation

In order to get the traversal time through the NoC, we need to calculate the time taken for reading and writing request between the cores. The question that should be answered is:

Can we find a proper response time analysis to get the worst case traversal time for resulting messages on the NoC?

Since main goal of our work is to bound the end-to-end delay for the communication between two cores we also need to find an analysis to calculate the worst case delay encountered during the



memory access on the tiles themselves. By looking at the hardware architecture and using formulas to describe the hardware's behavior, allows us to extract the end to end delay.

The question will be:

Can we calculate the end to end delay of the underlying architecture?

Because that we run one OS instance on each core, to enable communication between the different OS instances a message passing mechanism is needed. The next question will be:

How can we transport all messages on many-core processor by introducing a message passing mechanism?

In order to get no conflicts between cores, the need to guarantee mutual exclusion is required. The question that should be answered is: Can we find a suitable method for synchronizing memory accesses between cores?

### **9.2.3 Message**

A message is data, which needs to be sent from one core to another core. A Message is represented in flows. A flow is a data that is transferred between two cores, each flow has a specific size, a source and a destination addresses. A flow traverses between nodes via links until it reaches its destination. Since an instance of FreeRTOS is running on each core, thus messages are sent within a task. The message has a period and is sent periodically, which is done by the task. The maximum size of one message is a double word, which is equivalent to 64-bits. A message  $f$  is represented by the tuple  $\{s, dest, source, T\}$ , where  $s$  is the message size in flits,  $dest$ . And  $source$  are the destination and source node respectively and  $T$  is the period of the message.

### **9.2.4 Message-passing between Cores**

The cores communicate via message-passing. Since the cores cannot pass messages directly, instead they access other core's local memory. It is possible to get a global address of a memory location in another core's local memory [3]. Once a global address is known, it is always possible to pass messages via read and write transactions directly to the local memory of that core. For achieving message passing, suitable APIs has been developed. These APIs can access cores local memory explicitly. The user can use those APIs to communicate between the cores through assigning the destination core ID without having knowledge about the underlying operation.

Before starting using APIs for message passing, it is important to create a message box to store messages in each core. The following features should be considered in the message box:

1. It is important to keep the size of the message box small, because the Epiphany cores have a small amount of memory, which is 32 KB for both data and code.
2. The message box should be able to save messages from more than one task of the cores.
3. To get faster communication, accessing the message box must have a high performance.

#### 9.2.4.1 FreeRTOS Queues

Since each core executes an instance of FreeRTOS, it is possible to use FreeRTOS queues as a message box. They can be used to send messages between tasks. They use a FIFO (First in First Out) buffer and the data can be sent to the back or the front of the queue. A task can be blocked if it is attempting to read from an empty queue or to write to a full queue to avoid CPU time consumption [20]. The problem with this method is that the queue takes too much memory space, because it will require creating a queue for each core. This means that each queue will need an allocated memory for it, and as the Epiphany processor has 16 cores, this means that each core will have 16 queues, one queue for each core, and it will require to allocate memory for the entire 16 queue. In Addition, messages cannot be stored directly in the queues, it will need a variable to store they arrived message, and then pass this variable to the queue. The equations (1), (2), (3) and (4) have been used to find a total size that is required for the data structures inside each core's local memory.

$$Qs=Nm*S \qquad \text{Equation (1) Queue size}$$

Where:

Qs: Queue size

Nm: Number of messages per core

S: Size of one message

Equation (1) calculates the size of the queue that is used to store messages from each core by multiplying the number of messages with the size of one message.

$$S=Ds+Ms \qquad \text{Equation (2) Size of one message}$$

Where:

Ds: Data size

Ms: Mutex size

Equation (2) calculates the size of one message by adding the data size with the mutex size. The data size is specified by a data type of message data, i.e. integer, float or double.

The mutex size is a size of integer.

$$Y = Qs \quad \text{Equation (3) size of variables to store messages}$$

Where:

Y: The size of variables that store messages before sending it to the queues

Equation (3) calculates the size of the variable that is used to store the message to pass it to the queue. The size of variables is the same as the queue size.

$$Ts = Qs * Nt * Nc + Y \quad \text{Equation (4) Total size of FreeRTOS Queue}$$

Where:

Ts: Total size

Nt: Number of tasks per core (each core can run more than one task)

Nc: Number of cores

This equation calculates the total size of the data structure by using the results from the previous equations.

#### **9.2.4.2 Creating the Message Box**

Before passing messages between cores it is necessary to initialize a message box. An API (create\_box ()) has been developed for this purpose. The API initializes the message box as follows:

1. Allocate a memory for the message box.
2. Initialize reads and writes mutex.

3. Lock a read and unlocks a write mutex to make sure that the message box has got a message.

#### **9.2.4.3 Mutual Exclusion**

The Epiphany processor uses mutexes to guarantee mutual exclusion while accessing shared resources. Read and write mutexes have been added to the message box structure to ensure mutual exclusion between cores during write and read transactions. In the many-core processor each core has a read and write mutexes for locking of a shared resource. Once the mutex is locked, no other core can access the resource of that core until the mutex is unlocked. All cores can lock and unlock mutexes across core boundaries.

#### **9.2.5 Simulation**

The simulation is yielding the same behavior as experienced on the Epiphany processor NoC including: network topology, routing mechanism, and wormhole switching. In addition, it also contains parts of Epiphany processor such as crossbar, mutex, system cycle counter, read and write channels.

We use FreeRTOS API's and methods in SDK simulator:

1- At the beginning we make "Hello world "using FreeRTOS on one core to be enable next to test communication blocks codes

Steps:

1.1- We have download FreeRTOS "source codes & Demo projects for different platforms.

1.2 -Study careful Demo project of parallella to follow it in making our code.

1.3- Study FreeRTOS manual to understand configuration of functions in demo project.

1.4 -Study Makefile (Makefiles are a simple way to organize code compilation) basics.

1.5 -Implement our first Demo which just hello world code.

2- Simulate modulation block code using FreeRTOS:

As mention before in chapter 2 that core internal memory size is only 32 kB so when we tested at first modulation block, more problem about memory size appear so we need to deal with simulation debugger to follow problems and solve it.

3- Simulate all blocks at first before testing on platform using FreeRTOS.

## 9.2.6 Hardware

We don't need to use FreeRTOS on hardware as we discover good optimization way better than massaging passing using FreeRTOS.

## 9.3 ARM Cross Compiler

To test ARM processor of board we have compiled simple hello-world code with ARM cross compiler to generate binary file suitable to run on parallella board.

First we have installed the Cross Compilers, utilities.

Install the GCC, G++ cross compiler support programs by:

```
$ sudo apt-get install libc6-armel-cross libc6-dev-armel-cross  
$ sudo apt-get install binutils-arm-linux-gnueabi  
$ sudo apt-get install libncurses5-dev
```

Using an Acqua board:

```
$ sudo apt-get install gcc-arm-linux-gnueabi  
$ sudo apt-get install g++-arm-linux-gnueabi
```

Create simple code of hello-world then compile it by:

```
$ arm-linux-gnueabi-gcc hello.c -o hello
```

Then copy the executable binary to board by:

```
$ scp hello parallella@ip:~/
```

Run the executable binary on board:

```
./hello
```

```
somaia@somaia-Lenovo:~$ arm-linux-gnueabi-gcc hello.c -o hello
somaia@somaia-Lenovo:~$ scp hello parallella@192.168.1.9:~/
parallella@192.168.1.9's password:
hello
somaia@somaia-Lenovo:~$ ssh parallella@192.168.1.9
parallella@192.168.1.9's password:
Welcome to Linaro 14.04 (GNU/Linux 3.14.12-parallella-xilinx-g40a90c3 armv7l)

* Documentation: https://wiki.linaro.org/
Last login: Sat Mar 12 16:40:35 2016 from somaia-lenovo.zte.com.cn
parallella@parallella:~$ ls
Parallella epiphany-examples hello parallella-examples-old tests
parallella@parallella:~$ ./hello
Hello world !
parallella@parallella:~$ █
```

Figure 9-2: Hello-world with ARM cross compiler

## 9.4 Hello world on epiphany core

### 9.4.1 Introduction

Here we test every core on parallella platform and make sure that every core work Properly, therefore we create a simple application which is run on each core of the epiphany device in turn and performs a simple hello world type application.

Achieving this make us ensure that we know deeply how to:

- 1- Use of the epiphany hardware abstraction layer to initialize, reset, open and load the target Epiphany device
- 2- Use of basic commands of the epiphany hardware utility library(e-Lib).
- 3- Improve a build script to create the host (ARM processor) & target (Epiphany co-processors). ELF executables
- 4- Improve a run script to execute the application on the host (ARM processor) and target (Epiphany co-processors).

### 9.4.2 Implementation

First step it to create two directories within our chosen development directory, these directories should be called Debug and Source, Within the Source directory we will store the application source codes for the host and target (epiphany). The Debug folder is where the build process will store the executable files.

The application will use the shared DRAM memory to communicate between the host and target application. At a predefined memory location known to both applications, the target application will write a string using the `sprint` function while the host application will look for, retrieve and print out that message.

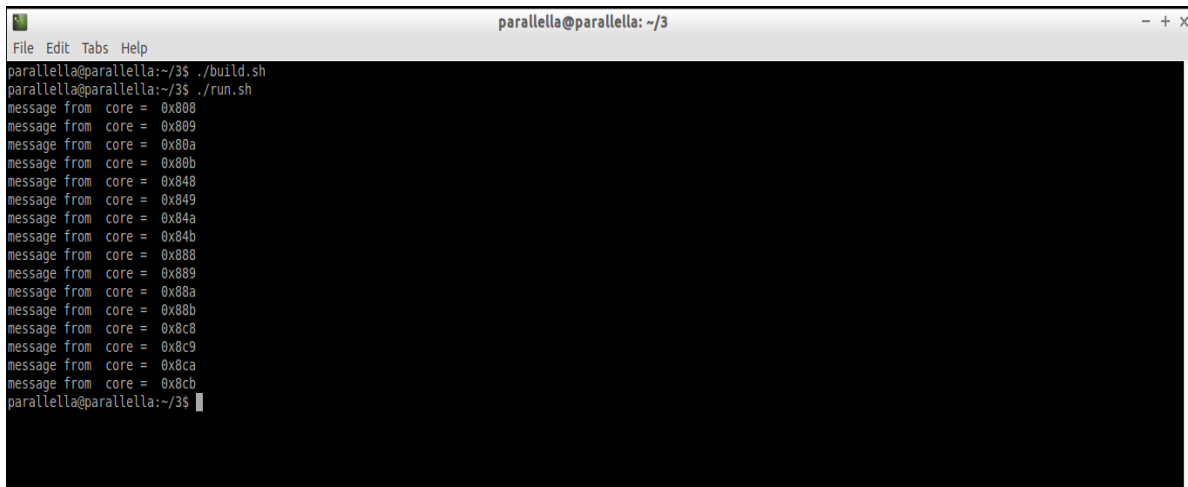
The main file for this demonstration is going to be the host application this application will:

1. System initialization: using the function `e_init ()`
2. Reset the system: using the function `e_reset_system ()`
3. Get the platform information: using the function `e_get_platform_info ()` which defines core configuration, number of devices within the system and number of external memories.
4. Allocate new location of the shared memory used for communication.
5. Address each core in turn and retrieve its printed string. To do this the host must:
  1. Define work group to be a single core: using the function `e_open ()`.
  2. Reset the workgroup: using the function `e_reset_group ()`
  3. Load the `s` record into the working group: using the function `e_load ()`, when doing this it will also check for the success of the operation.
  4. Wait for the target to finish execute its application
  5. Read the shared memory location: using the function `e_read()` to obtain the message from the target
  6. Get the message from Arm as output.
6. Having finally completed addressing each core the host application will then close and tidy up the epiphany and allocated memory before exiting

By comparison the epiphany target application is much simpler, it just gets the core ID for the core it is currently running upon, formats this into a string and writes it into the shared memory location.

### 9.4.3 Results:

Figure (9-3) below appears that every core work properly.



```
parallella@parallella: ~/3$ ./build.sh
parallella@parallella:~/3$ ./run.sh
message from core = 0x808
message from core = 0x809
message from core = 0x80a
message from core = 0x80b
message from core = 0x848
message from core = 0x849
message from core = 0x84a
message from core = 0x84b
message from core = 0x888
message from core = 0x889
message from core = 0x88a
message from core = 0x88b
message from core = 0x8c8
message from core = 0x8c9
message from core = 0x8ca
message from core = 0x8cb
parallella@parallella:~/3$
```

Figure 9-3: The way to implement code on single core on parallella

## 9.5 Test Functionality of Communication Blocks

### 9.5.1 Introduction

Here we start implementing every block code on single core on platform by using test cases input and compare output with full file of output test cases.

### 9.5.2 Modulation block

We insert modulation code as application on epiphany core in the same way like hello world application on previous section.

We face more problems related to memory size as input size is very large is about 5760 bytes and output size 4680 byte which very large compared to internal memory size of core so we directed to shared memory we store input on it on host program, and read it from epiphany program the epiphany write output again on shared memory as shown on Figure (9-4)



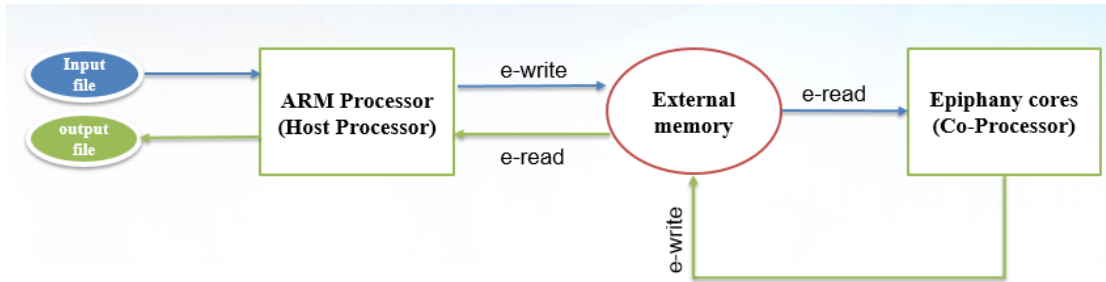


Figure 9-4: The way to implement code on single core on parallella

Then after sure that all test cases pass and achieve functionality of code we measure time is equal = 0.002783 sec as shown on Figure (9-5)

```

parallella@parallella:~/Modulation_Done$ ./build.sh
parallella@parallella:~/Modulation_Done$ ./run.sh
modulation output : "ebc33cb7"
Modulation Time : "0.002783"sec
parallella@parallella:~/Modulation_Done$ █
  
```

Figure 9-5: First modulation output and time consumption.

### 9.5.3 Scrambler & DFT Blocks

The same way we implement Modulation block we implement DFT and Scrambler blocks and their results shown on two figures below, but as shown in Figure (9-4) the DFT block take more time.

```

parallella@parallella:~/Scramblers$ ./build.sh
parallella@parallella:~/Scramblers$ ./run.sh
Scrambler time: "0.030653"sec
scrambler output is = 97
parallella@parallella:~/Scramblers$ █
  
```

Figure 9-6: First scrambler output and time consumption.

```

parallella@parallella:~/DFT_Block$ ./build.sh
parallella@parallella:~/DFT_Block$ ./run.sh
Dft time: "26.903342"sec
DFT output= 1c12e347
parallella@parallella:~/DFT_Block$ █
  
```

Figure 9-7: First DFT output and time consumption.

## 9.6 Optimization Levels

### 9.6.1 Introduction

We have to optimize in three factors memory, time and power to achieve our project target.

### 9.6.2 Running each Block on Single Core

In this phase we run each block on single core after each other as at first scrambler run on single core then deliver output to modulation which run and wait input on another core and finally modulation deliver its output to DFT which also run and wait input on third core.

In this stage we test that combination between blocks work successfully, the result of first output and time taken of each block and overall time taken for three cores shown in Figure (9-8)

```
parallella@parallella:~/all_blocks$ ./build.sh
parallella@parallella:~/all_blocks$ ./run.sh
Scrambler time:"0.031593"sec
Modulation time:"0.010857"sec
Dft      time:"26.871104"sec
all_code time:"26.913866"sec
scrambler output is  = 97
Modulation output is = ebc33cb7
DFT output is       = 1c12e347
parallella@parallella:~/all_blocks$
```

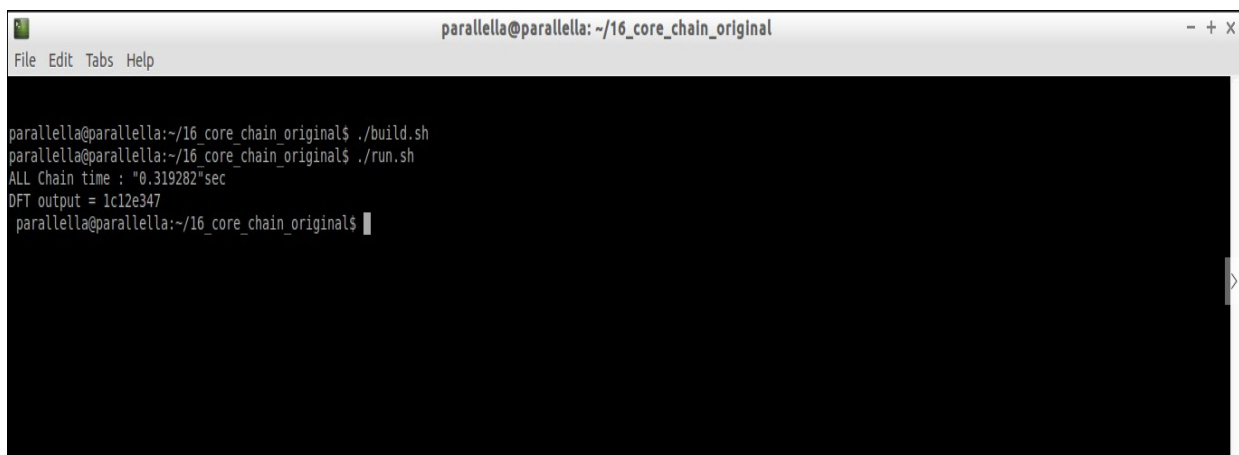
Figure 9-8: First output and time for each block and overall time of application.

### 9.6.3 Running all Blocks on 12-core in parallel

Here the final optimization level we have done, steps we did in this level:

- 1- Make only one binary for three blocks to avoid adding loading time in power measurement.
- 2- Divide input to 12 part to insert every part on different core.
- 3- Load the same binary on 12 core, but each core takes different input and result different output depend on core ID.

This way decrease time more than 100% as it decreased it from 28 msec to 319 msec as shown in Figure (9-9)

A terminal window titled "parallella@parallella: ~/16\_core\_chain\_original" with a menu bar containing "File", "Edit", "Tabs", and "Help". The terminal output shows the following commands and results:

```
parallella@parallella:~/16_core_chain_original$ ./build.sh
parallella@parallella:~/16_core_chain_original$ ./run.sh
ALL Chain time : "0.319282"sec
DFT output = 1c12e347
parallella@parallella:~/16_core_chain_original$
```

Figure 9-9: First DFT output and overall time of application

## Conclusion

In this thesis we presented the implementation of LTE UE-L1 on many-core general-purpose processors(GPP) Epiphany NoC system represented in Parallella platform, it was a challenge to achieve low power while achieving low time at the same time, but we achieved great results as final time for the implementation is 332 m sec which is good time compared to the PHY team in Axxcellera Company when measure the UL channel, it took 400msec before optimization, so 332 m sec is normal and this will be decreased when optimizing the code and using upgraded version of our Parallella with more numbers of cores as what will be described in future work section in this thesis and as we mentioned in introduction chapter the power of all epiphany 16 core system is less than 2 watt which is much less than DSP power.

The Project flow is summarized in 6 main stages which are:

- 1- Write c code for communication blocks (Scrambler-Modulation-DFT).
- 2- Using SDK simulator to test codes.
- 3- Prepare hardware environment to load codes on cores of Parallella.
- 4- Implement codes of blocks on hardware and test them with full test cases files.
- 5- Measure time for each block on platform.
- 6- Optimization level for code and the way of hardware implementation to decrease time.

We faced a lot of problems most of them were related to:

- 1- Parallella Memory (internal and external).
- 2- Tradeoff between power consumption and Time of processing as power consumption may be increased when use all cores (as one core take 25 mw which for sure less than 12 cores) but the time is decreased in this case, so it is a tradeoff.

The memory problems were solved by dividing the input file on 12 cores as described on chapter 9 section 5. And regarding the tradeoff problem between time and power consumption we have some recommendation described in future work chapter.

## Future Work

Most of the future work that can be done is in the Optimization level which is divided into two ways.

First way: Optimization in software codes by

1. Using Butterfly algorithm in implementing DFT block which will decrease time.
2. Using exponential interpolation in DFT implementation to generate the max size of phase Look up Table.
3. Using a matrix to generate the pseudo code outside the run time by a quick way and it also will be ready once needed for XORing with the input the thing that will reduce the code time a lot.

Second way: Optimization by Epiphany System

1. Using all 16-core of epiphany system instead of 12-core only which will decrease over all time.
2. Using “Embedded “version of Parallella which has 64-core where using all of them in parallel to implement all blocks will decrease time a lot.

The rest of future work that we recommend to be continued is implementing the IFFT block and place it with all codes on Parallella and trying to optimize in it regarding the time, also to implement and form the whole subframe as a software code, then place it on the board and try to optimize as much as possible to reach the 1ms required time of sub-frame.

## References

- [1] <http://www.parallella.org/create-sdcard/>
- [2] “Parallella-1.x Reference Manual”
- [3] Adapteva Inc.,”Epiphany SDK Reference”, REV 5.13.09.10, 2008-2013
- [4] Adapteva Inc.,”Epiphany Architecture Reference”, REV 14.03.11, 2008-2013
- [5] <https://github.com/adapteva/epiphany-sdk/wiki/Building-the-SDK>
- [6] <https://github.com/coduin/epiphany-bsp/wiki/Memory-on-the-parallella>
- [7] “FreeRTOS - A FREE RTOS for small real time embedded systems,” 2003-2005
- [8] Nahro Nadir, Omar Jamal, “Communication mechanism among instances of many-core real time system,” Malardalen University, 2015
- [9] [http://www.acmesystems.it/arm9\\_toolchain](http://www.acmesystems.it/arm9_toolchain)
- [10] 3GPP TS 36.211 v9.1.0, “Evolved Universal Terrestrial Radio Access (E-UTRA); Physical Channels and Modulation,” Release 9
- [11] [http://www.sharetechnote.com/html/FrameStructure\\_DL.html](http://www.sharetechnote.com/html/FrameStructure_DL.html)
- [12] [http://rfmw.em.keysight.com/wireless/helpfiles/89600B/webhelp/subsystems/lte/content/lte\\_overview.htm](http://rfmw.em.keysight.com/wireless/helpfiles/89600B/webhelp/subsystems/lte/content/lte_overview.htm)
- [13] <http://3gpphelp.blogspot.com/2011/11/physical-layer-bit-processing.html>
- [14] <http://howtostuffworks.blogspot.com/2014/06/rntis-in-lte.html>
- [15] <https://en.wikipedia.org/wiki/OpenLTE>
- [16] <http://sourceforge.net/projects/openlte/>
- [17] <https://code.google.com/p/lte-sim/issues/list>
- [18] <https://www.google.com/webhp?sourceid=chromeinstant&ion=1&espv=2&icq=UTF-8#q=%2F%2Ftelematics.poliba.it%2F%2FLTE-Sim>
- [19] <https://github.com/srsLTE/srsLTE>
- [20] Giorgio C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling
- [21] FreeRTOS <http://www.freertos.org/>. Last access 15, May, 2015.
- [22] <http://forums.parallella.org/viewtopic.php?f=28&t=1948>, (FreeRTOS running on Epiphany core) Last access 15, May, 2015.
- [23] <http://www.freertos.org/a00090.html>, (FreeRTOS Ports) Last access 15, May, 2015.
- [24] Floating-point to Fixed-point conversion

- [25] Erick L. Oberstar, "Fixed-Point Representation & Fractional Math ," 2004-2007 Oberstar Consulting, Revision 1.2, Released August 30, 2007
- [26] <https://www.quora.com/Why-is-malloc-harmful-in-embedded-systems>
- [27] <http://www.radio-electronics.com/info/cellularcomms/lte-long-term-evolution/lte-ofdm-ofdma-scdma.php>
- [28] John Wiley and Sons, Ltd, "Understanding LTE with MATLAB®: From Mathematical Modeling to Simulation and Prototyping," First Edition, 2014
- [29] <http://electronicdesign.com/4g/introduction-lte-advanced-real-4g>
- [30] <https://en.wikipedia.org/wiki/Precoding>
- [31] [https://en.wikipedia.org/wiki/Error\\_vector\\_magnitude](https://en.wikipedia.org/wiki/Error_vector_magnitude)
- [32] Rev A, "SC-FDMA Single Carrier FDMA in LTE", November 2009
- [33] <https://gigaom.com/2012/11/28/volte-calls-consumer-twice-the-power-of-2g-voice-calls/>
- [34] <https://gigaom.com/2012/02/17/why-lte-sucks-your-battery-that-is/>

## **Appendix I      Supported Tutorials**

1- Embedded for linux course By Dr.Ahmed ElArabawy

<http://linux4embeddedsystems.com/courses/course/index.php?categoryid=2>

2- Rohde & Schwarz LTE Basics Webinar - Part 01

<https://www.youtube.com/watch?v=VMcnDNZm5jY&list=PL65F3CFB5FF6EB6AC>



## Appendix II

### ▪ ARM Code

```
*****

/*
main.c

Copyright (C) 2016 Team (3).
Contributed by Somia, Dina, Omayma, Basma

This program is the ARM part of Scrambler, Modulation and DFT running on
parallella platform

July, 3 2016

*/

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <stdint.h>
#include <e-hal.h>
#include <time.h>

#define ALLOC_MEM 200000

uint32_t      scram_in   [1440];
uint32_t      mod_out    [2*5760];
uint32_t      DFT_out    [2*5760];
uint8_t       scram_out  [5760];

FILE *fo;
FILE *foo;
FILE *fooo;
FILE *foooo;
Int DFT_finish, DFT_finish2, DFT_finish3, DFT_finish4, DFT_finish5, DFT_finish6,
DFT_finish7, DFT_finish8, DFT_finish9, DFT_finish10, DFT_finish11, DFT_finish12;
int src[1];

int main()
{
    double diff ;
    struct timespec start_1, end_1;
        clock_t      begin, end ;
        double      time_spent ;
        unsigned    row_loop, col_loop;
        e_platform_t epiphany;
        e_epiphany_t dev;
        e_mem_t      memory;
        int          rc;
        int          i;

```

```

src[0] = 0x0;

/* *** Reading input file of Scrambler *** */
fo =
fopen("/home/parallella/16_core_chain/Src/sf00_Interleaver_Out_m2lob.lod" ,
"r");
        if(fo == NULL)
        {
            printf("can't open file");
        }

        else
        {
            fread(scram_in, sizeof(int32_t),1440, fo);
        }

fclose(fo);

/* initialization platform and reset system */
e_init(NULL);
e_reset_system();
e_get_platform_info(&epiphany);

/* start calculate processing time */
// begin= clock();

/* Allocate buffer in shard ram */
rc = e_alloc(&memory, 0x01800000,ALLOC_MEM);
if (rc != E_OK)
{
    return EXIT_FAILURE;
}

/* initialize allocated buffer with zeros */
for ( i=0; i< ALLOC_MEM; i++)
{
    e_write(&memory,0,0,i,src,1) ;
}

/* Write input buffer into memory */
e_write(&memory,0,0,0x00000000,scram_in,sizeof(scram_in)) ;

/* creating workgroup of 3 cores */
row_loop=0;
col_loop=0;

e_open(&dev,row_loop,col_loop,3,4);
e_reset_group(&dev);

/* load progrms on cores */

e_load_group("full_chain.srec", &dev,0,0,3,4, E_TRUE);

/* Calculating Epiphany Time */

```

```

int start,end1 ;
    while(1)
{
    e_read(&memory,0,0,0x000019738, &start,sizeof(int));
if(start==1) break;
}
begin= clock();
clock_gettime(CLOCK_REALTIME,&start_1);

    while(1)
{
    e_read(&memory,0,0,0x000019738, &end1,sizeof(int));
if(end1==2) break;
}
end =clock();
clock_gettime(CLOCK_REALTIME,&end_1);
diff=(1000000000L*(end_1.tv_sec - start_1.tv_sec)+ end_1.tv_nsec -
start_1.tv_nsec);
printf("time of clock %f \n",diff*0.000000001);

/* check for DFT Ending */
    while(1)
    {
        e_read(&memory,0,0,0x00019708, &DFT_finish, sizeof(int));
        e_read(&memory,0,0,0x0001970c, &DFT_finish2, sizeof(int));
        e_read(&memory,0,0,0x00019710, &DFT_finish3, sizeof(int));
        e_read(&memory,0,0,0x00019714, &DFT_finish4, sizeof(int));
        e_read(&memory,0,0,0x00019718, &DFT_finish5, sizeof(int));
        e_read(&memory,0,0,0x0001971c, &DFT_finish6, sizeof(int));
        e_read(&memory,0,0,0x00019720, &DFT_finish7, sizeof(int));
        e_read(&memory,0,0,0x00019724, &DFT_finish8, sizeof(int));
        e_read(&memory,0,0,0x00019728, &DFT_finish9, sizeof(int));
        e_read(&memory,0,0,0x0001972c, &DFT_finish10, sizeof(int));
        e_read(&memory,0,0,0x00019730, &DFT_finish11, sizeof(int));
        e_read(&memory,0,0,0x00019734, &DFT_finish12, sizeof(int));

        if((DFT_finish == 1)&&(DFT_finish2 == 1)&& (DFT_finish3 == 1)&&
(DFT_finish4 == 1)&&(DFT_finish5 == 1)&&(DFT_finish6 == 1)&&(DFT_finish7 ==
1)&&(DFT_finish8 == 1)&&(DFT_finish9 == 1)&&(DFT_finish10 ==
1)&&(DFT_finish11 == 1)&&(DFT_finish12 == 1))
        {

break;
        }
    }

/* Ending calculate time */
    //end = clock();
    time_spent = (double)(end -begin)/CLOCKS_PER_SEC ;
    printf("\n%f\n",time_spent);

```

```

/* Reading Scrambler , Modulation and DFT output */
//      e_read(&memory,0,0,0x00001700 ,scram_out, sizeof(scram_out));
//      e_read(&memory,0,0,0x00002e00 ,mod_out,  sizeof( mod_out));
//      e_read(&memory,0,0,0x0000e280 ,DFT_out,  sizeof( DFT_out));

/* Printing first element of output */
//  fprintf(stderr,"scrambler output is  = %x \n", scram_out[0]);
//  fprintf(stderr,"Modulation output is  = %x \n",mod_out[0]);
//  fprintf(stderr,"DFT output is  = %x \n",      DFT_out[0]);

/* Writing output into file */
//      foo=fopen("/home/parallella/16_core_chain/modulation_out.lod","wb");
//      fwrite( mod_out,sizeof(uint32_t),11520 , foo);

      fooc=fopen("/home/parallella/16_core_chain_original/DFT_out.lod","wb");
      fwrite( DFT_out,sizeof(uint32_t),11520 , fooc);
//      foocoo=fopen("/home/parallella/16_core_chain/scram_out.lod","wb");
//      fwrite( scram_out,sizeof(uint8_t),5760 , foocoo);

/* close and free work group */
e_close(&dev);
e_free(&memory);
e_finalize();

return 0;
}

```

\*\*\*\*\*

## ▪ Epiphany Code

\*\*\*\*\*

```

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <time.h>

#include "e_lib.h"

#include "lte_scrambler.h"
#include "lte_modulation.h"
#include "lte_dft.h"
#include "common_functions.h"

#define maxSize 480

/* Flags addresses */
#define address_scrambler_flag 0x8f819700
#define address_modulation_flag 0x8f819704
#define address_dft_flag      0x8f819708

```

```

#define address_dft_flag2      0x8f81970c
#define address_dft_flag3      0x8f819710
#define address_dft_flag4      0x8f819714
#define address_dft_flag5      0x8f819718
#define address_dft_flag6      0x8f81971c
#define address_dft_flag7      0x8f819720
#define address_dft_flag8      0x8f819724
#define address_dft_flag9      0x8f819728
#define address_dft_flag10     0x8f81972c
#define address_dft_flag11     0x8f819730
#define address_dft_flag12     0x8f819734
#define start_end              0x8f819738

/* Flags Declaration */
int *flag_scram = (unsigned int *) address_scrambler_flag ;
int *flag_mod = (unsigned int *) address_modulation_flag ;
int *flag_dft = (unsigned int *) address_dft_flag ;
int *flag_dft2 = (unsigned int *) address_dft_flag2 ;
int *flag_dft3 = (unsigned int *) address_dft_flag3 ;
int *flag_dft4 = (unsigned int *) address_dft_flag4 ;
int *flag_dft5 = (unsigned int *) address_dft_flag5 ;
int *flag_dft6 = (unsigned int *) address_dft_flag6 ;
int *flag_dft7 = (unsigned int *) address_dft_flag7 ;
int *flag_dft8 = (unsigned int *) address_dft_flag8 ;
int *flag_dft9 = (unsigned int *) address_dft_flag9 ;
int *flag_dft10 = (unsigned int *) address_dft_flag10 ;
int *flag_dft11 = (unsigned int *) address_dft_flag11 ;
int *flag_dft12 = (unsigned int *) address_dft_flag12 ;

int *clock1 = (unsigned int *) start_end ;

int z1=0;
int z2=0;

typedef struct intermediate_buffs_s {

    uint32_t in_scram[maxSize / 4];
    uint8_t in_mod[maxSize];
    uint32_t in_dft[maxSize * 2];
    uint32_t in_ifft[maxSize * 2];

} intermediate_buffs ;

uint32_t x1;
uint32_t x2;
int32_t address,address1,out_address_mod,out_address_scram,out_address_dft
;
int num ;

int main() {
// *clock1=1;
    unsigned row ,col ;
    e_coreid_t coreid;
    e_group_config_t remote ;

```

```

        intermediate_buffs    intr_buffs ;
pusch_scramb                scramb_obj ;
pusch_mod                   mod_obj;
pusch_dft                   dft_obj;

        e_emem_config_t      emem;
        int                  i,l;
        int                  scrambler_finish;
        int                  modulation_finish;
        int                  dft_finish;
/*****initializations
*****/
        uint8_t N_rnti = 142, Ns = 0, N_id_cell = 168;
        uint8_t Qm = 4;
        uint32_t length = maxSize * 8;
        uint8_t N_rb= 80;
        uint8_t L_max=maxSize * 2/(12*N_rb); //mod_symboles_num= max-size*2

/***** some parameters
*****/
        int32_t coff_out_size;
        if (Qm==2)
            coff_out_size=4*maxSize;
        else if (Qm==4)
            coff_out_size=2*maxSize;
        else if (Qm==6)
            coff_out_size=(maxSize*4)/3;

/*****Scrambler Function
*****/

        coreid = e_get_coreid();

        if(coreid == 0x808)    {address=0x8f800000 ; num=1;
out_address_scram= 0x01801700; out_address_mod=0x01802e00 ;
out_address_dft=0x0180e280 ;}
        else if(coreid == 0x809) {address=0x8f8001e0 ; num=2;
out_address_scram= 0x018018e0; out_address_mod=0x01803d00 ;
out_address_dft=0x0180f180 ;}
        else if(coreid == 0x80a) {address=0x8f8003c0 ; num=3;
out_address_scram= 0x01801ac0; out_address_mod=0x01804c00 ;
out_address_dft=0x01810080 ;}
        else if(coreid == 0x80b) {address=0x8f8005a0 ; num=4;
out_address_scram= 0x01801ca0; out_address_mod=0x01805b00 ;
out_address_dft=0x01810f80 ;}
        else if(coreid == 0x848) {address=0x8f800780 ; num=5;
out_address_scram= 0x01801e80; out_address_mod=0x01806a00 ;
out_address_dft=0x01811e80 ;}
        else if(coreid == 0x849) {address=0x8f800960 ; num=6;
out_address_scram= 0x01802026; out_address_mod=0x01807900 ;
out_address_dft=0x01812d80 ;}
        else if(coreid == 0x84a) {address=0x8f800b40 ; num=7;
out_address_scram= 0x01802240; out_address_mod=0x01808800 ;
out_address_dft=0x01813c80 ;}

```

```

        else if(coreid == 0x84b) {address=0x8f800d20 ; num=8;
out_address_scramb= 0x01802420; out_address_mod=0x01809700 ;
out_address_dft=0x01814b80 ;}
        else if(coreid == 0x888) {address=0x8f800f00 ; num=9;
out_address_scramb= 0x01802600; out_address_mod=0x0180a600 ;
out_address_dft=0x01815a80 ;}
        else if(coreid == 0x889) {address=0x8f8010e0 ; num=10;
out_address_scramb= 0x018027e0; out_address_mod=0x0180b500 ;
out_address_dft=0x01816980 ;}
        else if(coreid == 0x88a) {address=0x8f8012c0 ; num=11;
out_address_scramb= 0x018029c0; out_address_mod=0x0180c400 ;
out_address_dft=0x01817880 ;}
        else if(coreid == 0x88b) {address=0x8f8014a0 ; num=12;
out_address_scramb= 0x01802ba0; out_address_mod=0x0180d300 ;
out_address_dft=0x01818780 ;}

    for(i = 0; i < maxSize/4; i++)
    {
        intr_buffs.in_scramb[i] = ((uint32_t*)address)[i];
    }

    pusch_scramb_init(&scramb_obj, intr_buffs.in_scramb
,intr_buffs.in_mod);

    pusch_scramb_param_init(&scramb_obj, N_rnti, Ns, N_id_cell,num);

    pusch_scramb_start(&scramb_obj, maxSize);

    big_little_sweeping(scramb_obj.in, maxSize);

    output_converting_to_8bit(&scramb_obj, maxSize);

//    e_write(&emem,intr_buffs.in_mod
,0,0,(void*)out_address_scramb,sizeof(intr_buffs.in_mod ));

/*****Modulation Function
*****/

    pusch_mod_init(&mod_obj, intr_buffs.in_mod, intr_buffs.in_dft);

    pusch_mod_param(&mod_obj, Qm, length);

    pusch_mod_start(&mod_obj);

//
e_write(&emem,intr_buffs.in_dft,0,0,(void*)out_address_mod,sizeof(intr_buffs.
in_dft));

/*****DFT Function
*****/

*flag_dft = 0;
*flag_dft2 = 0;

```

```

*clock1=1;
    pusch_dft_init (&dft_obj,intr_buffs.in_dft , intr_buffs.in_ifft ,
N_rb);
    pusch_dft_start (&dft_obj, 0);

e_write(&emem,intr_buffs.in_ifft,0,0,(void*)out_address_dft,sizeof(intr_buffs
.in_ifft ));

    if(coreid == 0x808){*flag_dft=1;}
else if(coreid == 0x809){*flag_dft2=1;}
else if(coreid == 0x80a){*flag_dft3=1;}
else if(coreid == 0x80b){*flag_dft4=1;}
else if(coreid == 0x848){*flag_dft5=1;}
else if(coreid == 0x849){*flag_dft6=1;}
else if(coreid == 0x84a){*flag_dft7=1;}
else if(coreid == 0x84b){*flag_dft8=1;}
else if(coreid == 0x888){*flag_dft9=1;}
else if(coreid == 0x889){*flag_dft10=1;}
else if(coreid == 0x88a){*flag_dft11=1;}
else if(coreid == 0x88b){*flag_dft12=1;}

        *clock1=2;

/***** The End
*****/
        return EXIT_SUCCESS;
}

```

\*\*\*\*\*

- **Symbol chain Blocks Codes**

**Scrambler codes:**

```

*****

/*scrambler.h

*

* Created on: ??p/??p/???

* Author: BBS

*/

#ifndef SCRAMBLER_H_
#define SCRAMBLER_H_

```



```

#include <stdint.h>

typedef struct pusch_scrambler_structure {
    uint8_t N_rnti ;
    uint8_t Ns ;
    uint8_t N_id_cell ;
    uint32_t x1 ;
    uint32_t x2 ;
    int32_t *in;
    int8_t *out;
} pusch_scramb;

void pusch_scramb_init(pusch_scramb *scrambler_obj , uint32_t *in , uint8_t
*out);

void pusch_scramb_param_init(pusch_scramb *scrambler_obj ,uint8_t N_rnti ,
uint8_t Ns , uint8_t N_id_cell, int num ) ;

void pusch_scramb_start(pusch_scramb *scrambler_obj , uint32_t size) ;

void output_converting_to_8bit(pusch_scramb *scrambler_obj , uint32_t size);

void big_little_sweeping(int32_t *x, int32_t size);

#endif /* SCRAMBLER_H_ */
*****

*****

#include <stdint.h>

#include <stdio.h>

#include "lte_scrambler.h"

int32_t i , n ;

/*****i
*****/

/***** this function is for initialization, we insert the input and
output arrays
* to the pointers inside the pusch_scrambler_structure
*****/

void pusch_scramb_init(pusch_scramb *scrambler_obj , uint32_t *in , uint8_t
*out){

    scrambler_obj->in = in ;

```

```

    scrambler_obj->out = out ;
}

/*****

**** just insert the initialization of the parameters to the scrambler
structure *****/

void pusch_scramb_param_init(pusch_scramb *scrambler_obj,
    uint8_t N_rnti, uint8_t Ns, uint8_t N_id_cell, int num) {

    scrambler_obj->N_id_cell = N_id_cell;

    scrambler_obj->N_rnti = N_rnti;

    scrambler_obj->Ns = Ns;

    scrambler_obj->x1 = 0x21a127a;

    //scrambler_obj->x2 = 0x6f587d5c ;

/*****generate the first 1600 bits of scrambler*****/
    for (int i = 0; i < 50; i++) {
        for (int i = 0; i < 32; i++) {

            x2 <<= 1;

            x2 |= ((x2 & (0x80000000)) ^ ((x2 & (0x80000000 >> 1)) <<
1)
                ^ ((x2 & (0x80000000 >> 2)) << 2)
                ^ ((x2 & (0x80000000 >> 3)) << 3)) >> 31;
        }
    }

/*****
/** the code is distributed over 12 cores, so to choose the the codes
that will run on each
* core, this function does that ( 5760 / 12 = 480 )
* 5760 is the file input bytes , num represents the core number
* at num=0, no need for this function, only the first 480 bytes will
be taken
* at num =1, the 2nd 480 pseudo code will be taken and so on *****/
*/

    for (n = 0; n < ((num - 1) * 120); n++) {

        for (i = 0; i < 32; i++) {

            scrambler_obj->x1 <<= 1;

            scrambler_obj->x2 <<= 1;

```

```

scrambler_obj->x1 |= ((scrambler_obj->x1 & (0x80000000))
    ^ ((scrambler_obj->x1 & (0x80000000 >> 3)) << 3)) >> 31;

scrambler_obj->x2 |= ((scrambler_obj->x2 & (0x80000000))
    ^ ((scrambler_obj->x2 & (0x80000000 >> 1)) << 1)
    ^ ((scrambler_obj->x2 & (0x80000000 >> 2)) << 2)
    ^ ((scrambler_obj->x2 & (0x80000000 >> 3)) << 3)) >> 31;
}
}
}

/*****/
/*****/

void pusch_scramb_start(pusch_scramb *scrambler_obj, uint32_t maxSize) {

    /**** generating the output ( xoring the code with the input )
    * an within generate each pseduo code that i will xor with *****/

    /*****/

    for (n = 0; n < (maxSize / 4); n++) {

        scrambler_obj->in[n] = scrambler_obj->in[n] ^ scrambler_obj->x1
            ^ scrambler_obj->x2;

        for ( i = 0; i < 32; i++) {

            scrambler_obj->x1 <<= 1;

            scrambler_obj->x2 <<= 1;

            scrambler_obj->x1 |= ((scrambler_obj->x1 & (0x80000000))
                ^ ((scrambler_obj->x1 & (0x80000000 >> 3)) << 3)) >> 31;

```

```

scrambler_obj->x2 |= ((scrambler_obj->x2 & (0x80000000))
^ ((scrambler_obj->x2 & (0x80000000 >> 1)) << 1)
^ ((scrambler_obj->x2 & (0x80000000 >> 2)) << 2)
^ ((scrambler_obj->x2 & (0x80000000 >> 3)) << 3)) >> 31;
}
}
}

/*****
/** the modulation input is 8 bit and scrambler output is 32 bit so this
function does this conversion***/
*****/

void output_converting_to_8bit(pusch_scramb *scrambler_obj , uint32_t size){
    int k = 0 ;
        for (n = 0; n < size-1 ; n+=4) {

/*scrambler_obj->in[i] |= ((temp & 0xFF000000) >> 24)
| ((temp & 0x00FF0000) >> 8) |
((temp & 0x0000FF00) << 8)
| ((temp & 0x000000FF) << 24);*/

scrambler_obj->out[n] = scrambler_obj->in[k] & (0x000000FF);
scrambler_obj->out[n + 1] = (scrambler_obj->in[k] & (0x0000FF00)) >> 8;
scrambler_obj->out[n + 2] = (scrambler_obj->in[k] & (0x00FF0000)) >> 16;
scrambler_obj->out[n + 3] = (scrambler_obj->in[k] & (0xFF000000)) >> 24;

        k++;
    }
}

```

## Modulation Codes

```
*****

#ifndef LTE_MODULATION_H_INCLUDED
#define LTE_MODULATION_H_INCLUDED

#include <stdint.h>

typedef struct pusch_mod_struct
{
    uint8_t mod_type;
    uint32_t length;
    int8_t *in;
    int32_t *out;
} pusch_mod;

void pusch_mod_init (pusch_mod *mod_obj, int8_t *input, int32_t *output);
void pusch_mod_param (pusch_mod *mod_obj, uint8_t Qm, uint32_t length);
void pusch_mod_start (pusch_mod *mod_obj);

#endif // LTE_MODULATION_H_INCLUDED

*****

*****

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include "lte_modulation.h"
//LUT the most segnificant 2 byts >> real part of symbol
//LUT the least segnificant 2 byts >> imaginary part of symbol
//LUT of QPSK scheme
static const int32_t QPSK[] = { 0x2D412D41, 0x2D41D2BF, 0xD2BF2D41,
0xD2BFD2BF };
//LUT of 16QAM scheme
static const int32_t QAM16[] = { 0x143D143D, 0x143D3CB7, 0x3CB7143D,
0x3CB73CB7,
0x143DEBC3, 0x143DC349, 0x3CB7EBC3, 0x3CB7C349, 0xEBC3143D,
0xEBC33CB7,
0xC349143D, 0xC3493CB7, 0xEBC3EBC3, 0xEBC3C349, 0xC349EBC3,
0xC349C349 };
//LUT of 64QAM scheme
static const int32_t QAM64[] = { 0x1DA01DA0, 0x1DA009E0, 0x09E01DA0,
0x09E009E0,
0x1DA03161, 0x1DA04521, 0x09E03161, 0x09E04521, 0x31611DA0,
0x316109E0,
0x45211DA0, 0x452109E0, 0x31613161, 0x31614521, 0x45213161,
0x45214521,
0x1DA0E260, 0x1DA0F620, 0x09E0E260, 0x09E0F620, 0x1DA0CE9F,
0x1DA0BADF,
0x09E0CE9F, 0x09E0BADF, 0x3161E260, 0x3161F620, 0x4521E260,
0x4521F620,
```

```

    0x3161CE9F, 0x3161BADF, 0x4521CE9F, 0x4521BADF, 0xE2601DA0,
0xE26009E0,
    0xF6201DA0, 0xF62009E0, 0xE2603161, 0xE2604521, 0xF6203161,
0xF6204521,
    0xCE9F1DA0, 0xCE9F09E0, 0xBADF1DA0, 0xBADF09E0, 0xCE9F3161,
0xCE9F4521,
    0xBADF3161, 0xBADF4521, 0xE260E260, 0xE260F620, 0xF620E260,
0xF620F620,
    0xE260CE9F, 0xE260BADF, 0xF620CE9F, 0xF620BADF, 0xCE9FE260,
0xCE9FF620,
    0xBADFE260, 0xBADFF620, 0xCE9FCE9F, 0xCE9FBADF, 0xBADFCE9F,
0xBADFBADF };

/*Qm(modulation order): 2,4,6 */
/*length number of inputs in bits */

/* input is an array .. the array name is a pointer to the array ..
so we put the address of the first element of the array in
the pointer inside the pusch_mod structure to use "in" as my pointer to the
input array */

void pusch_mod_init(pusch_mod *mod_obj, int8_t *input, int32_t *output)

{
    mod_obj->in = input;
    mod_obj->out = output;
}
//mapping Qm and length of input array
void pusch_mod_param(pusch_mod *mod_obj, uint8_t Qm, uint32_t length) {
    mod_obj->mod_type = Qm;
    mod_obj->length = length;
}
//the main function of modulation
void pusch_mod_start(pusch_mod *mod_obj) {
    int i, k, index, j = 0;
    int8_t x, y, z; //temporary variable
    //make sure that Qm is 2 or 4 or 6
    if ((mod_obj->mod_type == 2) || (mod_obj->mod_type == 4)
        || (mod_obj->mod_type == 6)) {
        /////////////////////////////////////////////////////////////////// Qm=2 ==>> QPSK
        ///////////////////////////////////////////////////////////////////
        if (mod_obj->mod_type == 2) {
            for (k = 0; (mod_obj->in != '\0') && j < (mod_obj->length /
2); k++) // loop on input bytes
            {
                index = (mod_obj->in[k] & (0xC0)); //casting 1st
symbol per byte
                index = index >> 6; //shift to (LSB)to convert it to
index(number) entry of look up table
                mod_obj->out[j] = QPSK[index]; //get the complex
fixed symbol from QPSK LUT
                j += 1;

                index = (mod_obj->in[k] & (0x30)); //casting 2nd
symbol per byte

```

```

        index = index >> 4; //shift to (LSB)to convert it to
index(number) entry of look up table
        mod_obj->out[j] = QPSK[index]; //get the complex
fixed symbol from QPSK LUT
        j += 1;

        index = (mod_obj->in[k] & (0x0C)); //casting 3rd
symbol per byte
        index = index >> 2; //shift to (LSB)to convert it to
index(number) entry of look up table
        mod_obj->out[j] = QPSK[index]; //get the complex
fixed symbol from QPSK LUT
        j += 1;

        index = (mod_obj->in[k] & (0x03)); //casting 4th
symbol per byte
        index = index >> 0; //shift to (LSB)to convert it to
index(number) entry of look up table
        mod_obj->out[j] = QPSK[index]; //get the complex
fixed symbol from QPSK LUT
        j += 1;
    }
    mod_obj->out[j] = '\\0';
    return;
}

////////////////////////////////////// Qm=4 ==>> 16QAM
//////////////////////////////////////
    else if (mod_obj->mod_type == 4) {
        for (k = 0; (mod_obj->in != '\\0') && j < (mod_obj->length /
4); k++) // loop on input bytes
        {
            index = (mod_obj->in[k] & (0xF0)); //casting 1st
symbol per byte
            index = index >> 4; //shift to (LSB)to convert it to
index(number) entrie of look up table
            mod_obj->out[j] = QAM16[index]; //get the complex
fixed symbol from 16QAM LUT
            j += 1;

            index = (mod_obj->in[k] & (0x0F)); //casting 2nd
symbol per byte
            index = index >> 0; //shift to (LSB)to convert it to
index(number) entry of look up table
            mod_obj->out[j] = QAM16[index]; //get the complex
fixed symbol from 16QAM LUT
            j += 1;
        }
        mod_obj->out[j] = '\\0';
        return;
    }
}

```

```

////////////////////////////////////// Qm=6 ==> 64QAM
//////////////////////////////////////
//////////////////////////////////////we will deal with 4 bytes for each iteration
else {
    if (((mod_obj->length) % 8) != 0)
        || ((mod_obj->length) % 6) != 0) // %8 not
wasted bits & %6 cause of modulation
    {
        return;
    } else {
        k = 0;
        for (i = 0; k < ((mod_obj->length) / 6); i += 3) {
            //take 1st 6 bits from the 1st byte to get the
1st symbol
            index = (mod_obj->in[i] & 0xFC);
            index = index >> 2; // //shift to (LSB)to
convert it to index(number) entry of look up table
            mod_obj->out[k] = QAM64[index]; //get the
complex fixed symbol from 64QAM LUT

            k += 1;
            //////////////////////////////////take the rest 2 bits from the 1st byte and
1st 4 bits from the 2nd byte//////////

            x = mod_obj->in[i] << 6; // //shift the 6
bits that we took before .to get the last 2 bits
            y = mod_obj->in[i + 1] >> 2; // getting the rest
4 bits from the 2nd byte

            z = x | y; // OR to get the 6 bits together
            index = (z & 0xFC); //take 2nd (6 bits) to get
the 2nd 64QAM symbol

            index = index >> 2; // //shift to (LSB)to
convert it to index(number) entry of look up table
            mod_obj->out[k] = QAM64[index]; //get the
complex fixed symbol from 64QAM LUT

            k += 1;

            //////////////////////////////////take the rest 4 bits from the 2st byte and
1st 2 bits from the 3rd byte//////////

            x = mod_obj->in[i + 1] << 4; // //shift the 4
bits that we took before .to get the last 4 bits
            y = mod_obj->in[i + 2] >> 4; // getting the rest
4 bits from the 3rd byte

            z = x | y; // OQ to get the 6 bits together
            index = (z & 0xFC); //take 3rd (6 bits) to get
the 3rd 64QAM symbol

            index = index >> 2; // //shift to (LSB)to convert
it to index(number) entry of look up table
            mod_obj->out[k] = QAM64[index]; //get the
complex fixed symbol from 64QAM LUT

            k += 1;
            //////////////////////////////////take the rest 6 bits from the 3rd
byte//////////

```



```

        x = mod_obj->in[i + 2] << 2;////shift the 2
bits that we took before .to get the last 6 bits
        index = (x & 0xFC); //take 4th (6 bits) to
get the 3rd 64QAM symbol
        index = index >> 2; //shift to (LSB)to convert
it to index(number) entry of look up table
        mod_obj->out[k] = QAM64[index];//get the
complex fixed symbol from 64QAM LUT

        k += 1;
    } // each loop has 3 bytes &generate 4 symbols
    mod_obj->out[k] = '\0';
    return;
}
}
}
else {
    return;
}
}

```

\*\*\*\*\*

## DFT Codes

\*\*\*\*\*

```

/*
 * DFT1.h
 *
 * Created on: Apr 29, 2016
 * Author: omayma
 */

#ifndef DFT1_H_
#define DFT1_H_
typedef struct pusch_DFT_struct
{
    uint16_t M_PUSH; //number rb *12
    uint8_t l;
    int32_t *in ;
    int32_t *out ;
} pusch_dft;

void pusch_dft_init (pusch_dft *dft_obj, int32_t *in, int32_t *out, uint8_t
N_rb);
void pusch_dft_start (pusch_dft *dft_obj, uint8_t l);
int16_t pusch_dft_interplation (int16_t y1 ,int16_t y2 ,int8_t x1,int8_t x2
,float i );
void big_little_swapping(int32_t *x ,uint32_t size);

```

```

#endif /* DFT1_H_ */

*****

*****

/*
=====
Name      : DFT1.c
Author    : omayma
Version   :
Copyright : Your copyright notice
Description : Hello World in C, Ansi-style
=====
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include "lte_dft.h"

///// LUT of real part of exponential (phase shift) with size 960 /////
static const int16_t exp_DFT_I[]={
    0x4000 ,0x4000 ,0x3FFF ,0x3FFD ,0x3FFA ,0x3FF7 ,0x3FF3 ,0x3FEF
,0x3FEA ,0x3FE4 ,0x3FDD ,0x3FD6 ,0x3FCD ,0x3FC5 ,0x3FBB ,0x3FB1 ,0x3FA6
,0x3F9B ,0x3F8E ,0x3F81 ,0x3F74 ,0x3F65 ,0x3F56 ,0x3F47 ,0x3F36 ,0x3F25
,0x3F13 ,0x3F01 ,0x3EEE ,0x3EDA ,
    0x3EC5 ,0x3EB0 ,0x3E9A ,0x3E83 ,0x3E6C ,0x3E54 ,0x3E3B ,0x3E22
,0x3E08 ,0x3DED ,0x3DD2 ,0x3DB6 ,0x3D99 ,0x3D7B ,0x3D5D ,0x3D3F ,0x3D1F
,0x3CFF ,0x3CDE ,0x3CBD ,0x3C9B ,0x3C78 ,0x3C54 ,0x3C30 ,0x3C0B ,0x3BE6
,0x3BC0 ,0x3B99 ,0x3B72 ,0x3B4A ,
    0x3B21 ,0x3AF7 ,0x3ACD ,0x3AA3 ,0x3A78 ,0x3A4C ,0x3A1F ,0x39F2
,0x39C4 ,0x3995 ,0x3966 ,0x3937 ,0x3906 ,0x38D5 ,0x38A4 ,0x3871 ,0x383F
,0x380B ,0x37D7 ,0x37A2 ,0x376D ,0x3737 ,0x3701 ,0x36C9 ,0x3692 ,0x3659
,0x3620 ,0x35E7 ,0x35AD ,0x3572 ,
    0x3537 ,0x34FB ,0x34BE ,0x3481 ,0x3444 ,0x3406 ,0x33C7 ,0x3388
,0x3348 ,0x3307 ,0x32C6 ,0x3285 ,0x3243 ,0x3200 ,0x31BD ,0x3179 ,0x3135
,0x30F0 ,0x30AA ,0x3065 ,0x301E ,0x2FD7 ,0x2F90 ,0x2F48 ,0x2EFF ,0x2EB6
,0x2E6D ,0x2E22 ,0x2DD8 ,0x2D8D ,
    0x2D41 ,0x2CF5 ,0x2CA9 ,0x2C5C ,0x2C0E ,0x2BC0 ,0x2B71 ,0x2B22
,0x2AD3 ,0x2A83 ,0x2A33 ,0x29E2 ,0x2991 ,0x293F ,0x28ED ,0x289A ,0x2847
,0x27F3 ,0x279F ,0x274B ,0x26F6 ,0x26A1 ,0x264B ,0x25F5 ,0x259E ,0x2547
,0x24F0 ,0x2498 ,0x2440 ,0x23E7 ,
    0x238E ,0x2335 ,0x22DB ,0x2281 ,0x2227 ,0x21CC ,0x2171 ,0x2115
,0x20B9 ,0x205D ,0x2000 ,0x1FA3 ,0x1F46 ,0x1EE8 ,0x1E8A ,0x1E2B ,0x1DCD
,0x1D6E ,0x1D0E ,0x1CAE ,0x1C4E ,0x1BEE ,0x1B8D ,0x1B2D ,0x1ACB ,0x1A6A
,0x1A08 ,0x19A6 ,0x1943 ,0x18E1 ,
    0x187E ,0x181B ,0x17B7 ,0x1753 ,0x16F0 ,0x168B ,0x1627 ,0x15C2
,0x155D ,0x14F8 ,0x1492 ,0x142D ,0x13C7 ,0x1361 ,0x12FB ,0x1294 ,0x122D
,0x11C6 ,0x115F ,0x10F8 ,0x1090 ,0x1029 ,0x0FC1 ,0x0F59 ,0x0EF1 ,0x0E88
,0x0E20 ,0x0DB7 ,0x0D4E ,0x0CE5 ,
    0x0C7C ,0x0C13 ,0x0BAA ,0x0B40 ,0x0AD7 ,0x0A6D ,0x0A03 ,0x0999
,0x092F ,0x08C5 ,0x085B ,0x07F0 ,0x0786 ,0x071B ,0x06B1 ,0x0646 ,0x05DB

```

,0x0570 ,0x0505 ,0x049B ,0x0430 ,0x03C5 ,0x0359 ,0x02EE ,0x0283 ,0x0218  
,0x01AD ,0x0142 ,0x00D6 ,0x006B ,  
0x0000 ,0xFF95 ,0xFF2A ,0xFEBE ,0xFE53 ,0xFDE8 ,0xFD7D ,0xFD12  
,0xFCA7 ,0xFC3B ,0xFBD0 ,0xFB65 ,0xFAFB ,0xFA90 ,0xFA25 ,0xF9BA ,0xF94F  
,0xF8E5 ,0xF87A ,0xF810 ,0xF7A5 ,0xF73B ,0xF6D1 ,0xF667 ,0xF5FD ,0xF593  
,0xF529 ,0xF4C0 ,0xF456 ,0xF3ED ,  
0xF384 ,0xF31B ,0xF2B2 ,0xF249 ,0xF1E0 ,0xF178 ,0xF10F ,0xF0A7  
,0xF03F ,0xEFD7 ,0xEF70 ,0xEF08 ,0xEEA1 ,0xEE3A ,0xEDD3 ,0xED6C ,0xED05  
,0xEC9F ,0xEC39 ,0xEBD3 ,0xEB6E ,0xEB08 ,0xEAA3 ,0xEA3E ,0xE9D9 ,0xE975  
,0xE910 ,0xE8AD ,0xE849 ,0xE7E5 ,  
0xE782 ,0xE71F ,0xE6BD ,0xE65A ,0xE5F8 ,0xE596 ,0xE535 ,0xE4D3  
,0xE473 ,0xE412 ,0xE3B2 ,0xE352 ,0xE2F2 ,0xE292 ,0xE233 ,0xE1D5 ,0xE176  
,0xE118 ,0xE0BA ,0xE05D ,0xE000 ,0xDFA3 ,0xDF47 ,0xDEEB ,0xDE8F ,0xDE34  
,0xDDD9 ,0xDD7F ,0xDD25 ,0xDCCB ,  
0xDC72 ,0xDC19 ,0xDBC0 ,0xDB68 ,0xDB10 ,0xDAB9 ,0xDA62 ,0xDA0B  
,0xD9B5 ,0xD95F ,0xD90A ,0xD8B5 ,0xD861 ,0xD80D ,0xD7B9 ,0xD766 ,0xD713  
,0xD6C1 ,0xD66F ,0xD61E ,0xD5CD ,0xD57D ,0xD52D ,0xD4DE ,0xD48F ,0xD440  
,0xD3F2 ,0xD3A4 ,0xD357 ,0xD30B ,  
0xD2BF ,0xD273 ,0xD228 ,0xD1DE ,0xD193 ,0xD14A ,0xD101 ,0xD0B8  
,0xD070 ,0xD029 ,0xCFE2 ,0xCF9B ,0xCF56 ,0xCF10 ,0xCECB ,0xCE87 ,0xCE43  
,0xCE00 ,0xCDBD ,0xCD7B ,0xCD3A ,0xCCF9 ,0xCCB8 ,0xCC78 ,0xCC39 ,0xCBFA  
,0xCBBC ,0xCB7F ,0xCB42 ,0xCB05 ,  
0xCAC9 ,0xCA8E ,0xCA53 ,0xCA19 ,0xC9E0 ,0xC9A7 ,0xC96E ,0xC937  
,0xC8FF ,0xC8C9 ,0xC893 ,0xC85E ,0xC829 ,0xC7F5 ,0xC7C1 ,0xC78F ,0xC75C  
,0xC72B ,0xC6FA ,0xC6C9 ,0xC69A ,0xC66B ,0xC63C ,0xC60E ,0xC5E1 ,0xC5B4  
,0xC588 ,0xC55D ,0xC533 ,0xC509 ,  
0xC4DF ,0xC4B6 ,0xC48E ,0xC467 ,0xC440 ,0xC41A ,0xC3F5 ,0xC3D0  
,0xC3AC ,0xC388 ,0xC365 ,0xC343 ,0xC322 ,0xC301 ,0xC2E1 ,0xC2C1 ,0xC2A3  
,0xC285 ,0xC267 ,0xC24A ,0xC22E ,0xC213 ,0xC1F8 ,0xC1DE ,0xC1C5 ,0xC1AC  
,0xC194 ,0xC17D ,0xC166 ,0xC150 ,  
0xC13B ,0xC126 ,0xC112 ,0xC0FF ,0xC0ED ,0xC0DB ,0xC0CA ,0xC0B9  
,0xC0AA ,0xC09B ,0xC08C ,0xC07F ,0xC072 ,0xC065 ,0xC05A ,0xC04F ,0xC045  
,0xC03B ,0xC033 ,0xC02A ,0xC023 ,0xC01C ,0xC016 ,0xC011 ,0xC00D ,0xC009  
,0xC006 ,0xC003 ,0xC001 ,0xC000 ,  
0xC000 ,0xC000 ,0xC001 ,0xC003 ,0xC006 ,0xC009 ,0xC00D ,0xC011  
,0xC016 ,0xC01C ,0xC023 ,0xC02A ,0xC033 ,0xC03B ,0xC045 ,0xC04F ,0xC05A  
,0xC065 ,0xC072 ,0xC07F ,0xC08C ,0xC09B ,0xC0AA ,0xC0B9 ,0xC0CA ,0xC0DB  
,0xC0ED ,0xC0FF ,0xC112 ,0xC126 ,  
0xC13B ,0xC150 ,0xC166 ,0xC17D ,0xC194 ,0xC1AC ,0xC1C5 ,0xC1DE  
,0xC1F8 ,0xC213 ,0xC22E ,0xC24A ,0xC267 ,0xC285 ,0xC2A3 ,0xC2C1 ,0xC2E1  
,0xC301 ,0xC322 ,0xC343 ,0xC365 ,0xC388 ,0xC3AC ,0xC3D0 ,0xC3F5 ,0xC41A  
,0xC440 ,0xC467 ,0xC48E ,0xC4B6 ,  
0xC4DF ,0xC509 ,0xC533 ,0xC55D ,0xC588 ,0xC5B4 ,0xC5E1 ,0xC60E  
,0xC63C ,0xC66B ,0xC69A ,0xC6C9 ,0xC6FA ,0xC72B ,0xC75C ,0xC78F ,0xC7C1  
,0xC7F5 ,0xC829 ,0xC85E ,0xC893 ,0xC8C9 ,0xC8FF ,0xC937 ,0xC96E ,0xC9A7  
,0xC9E0 ,0xCA19 ,0xCA53 ,0xCA8E ,  
0xCAC9 ,0xCB05 ,0xCB42 ,0xCB7F ,0xCBBC ,0xCBFA ,0xCC39 ,0xCC78  
,0xCCB8 ,0xCCF9 ,0xCD3A ,0xCD7B ,0xCDBD ,0xCE00 ,0xCE43 ,0xCE87 ,0xCECB  
,0xCF10 ,0xCF56 ,0xCF9B ,0xCFE2 ,0xD029 ,0xD070 ,0xD0B8 ,0xD101 ,0xD14A  
,0xD193 ,0xD1DE ,0xD228 ,0xD273 ,  
0xD2BF ,0xD30B ,0xD357 ,0xD3A4 ,0xD3F2 ,0xD440 ,0xD48F ,0xD4DE  
,0xD52D ,0xD57D ,0xD5CD ,0xD61E ,0xD66F ,0xD6C1 ,0xD713 ,0xD766 ,0xD7B9  
,0xD80D ,0xD861 ,0xD8B5 ,0xD90A ,0xD95F ,0xD9B5 ,0xDA0B ,0xDA62 ,0xDAB9  
,0xDB10 ,0xDB68 ,0xDBC0 ,0xDC19 ,  
0xDC72 ,0xDCCB ,0xDD25 ,0xDD7F ,0xDDD9 ,0xDE34 ,0xDE8F ,0xDEEB  
,0xDF47 ,0xDFA3 ,0xE000 ,0xE05D ,0xE0BA ,0xE118 ,0xE176 ,0xE1D5 ,0xE233

```

,0xE292 ,0xE2F2 ,0xE352 ,0xE3B2 ,0xE412 ,0xE473 ,0xE4D3 ,0xE535 ,0xE596
,0xE5F8 ,0xE65A ,0xE6BD ,0xE71F ,
    0xE782 ,0xE7E5 ,0xE849 ,0xE8AD ,0xE910 ,0xE975 ,0xE9D9 ,0xEA3E
,0xEAA3 ,0xEB08 ,0xEB6E ,0xEBD3 ,0xEC39 ,0xEC9F ,0xED05 ,0xED6C ,0xEDD3
,0xEE3A ,0xEEA1 ,0xEF08 ,0xEF70 ,0xEFD7 ,0xF03F ,0xF0A7 ,0xF10F ,0xF178
,0xF1E0 ,0xF249 ,0xF2B2 ,0xF31B ,
    0xF384 ,0xF3ED ,0xF456 ,0xF4C0 ,0xF529 ,0xF593 ,0xF5FD ,0xF667
,0xF6D1 ,0xF73B ,0xF7A5 ,0xF810 ,0xF87A ,0xF8E5 ,0xF94F ,0xF9BA ,0xFA25
,0xFA90 ,0FAFB ,0xFB65 ,0FBD0 ,0FC3B ,0FCA7 ,0FD12 ,0FD7D ,0FDE8
,0xFE53 ,0FEBE ,0FF2A ,0FF95 ,
    0x0000 ,0x006B ,0x00D6 ,0x0142 ,0x01AD ,0x0218 ,0x0283 ,0x02EE
,0x0359 ,0x03C5 ,0x0430 ,0x049B ,0x0505 ,0x0570 ,0x05DB ,0x0646 ,0x06B1
,0x071B ,0x0786 ,0x07F0 ,0x085B ,0x08C5 ,0x092F ,0x0999 ,0x0A03 ,0x0A6D
,0x0AD7 ,0x0B40 ,0x0BAA ,0x0C13 ,
    0x0C7C ,0x0CE5 ,0x0D4E ,0x0DB7 ,0x0E20 ,0x0E88 ,0x0EF1 ,0x0F59
,0x0FC1 ,0x1029 ,0x1090 ,0x10F8 ,0x115F ,0x11C6 ,0x122D ,0x1294 ,0x12FB
,0x1361 ,0x13C7 ,0x142D ,0x1492 ,0x14F8 ,0x155D ,0x15C2 ,0x1627 ,0x168B
,0x16F0 ,0x1753 ,0x17B7 ,0x181B ,
    0x187E ,0x18E1 ,0x1943 ,0x19A6 ,0x1A08 ,0x1A6A ,0x1ACB ,0x1B2D
,0x1B8D ,0x1BEE ,0x1C4E ,0x1CAE ,0x1D0E ,0x1D6E ,0x1DCD ,0x1E2B ,0x1E8A
,0x1EE8 ,0x1F46 ,0x1FA3 ,0x2000 ,0x205D ,0x20B9 ,0x2115 ,0x2171 ,0x21CC
,0x2227 ,0x2281 ,0x22DB ,0x2335 ,
    0x238E ,0x23E7 ,0x2440 ,0x2498 ,0x24F0 ,0x2547 ,0x259E ,0x25F5
,0x264B ,0x26A1 ,0x26F6 ,0x274B ,0x279F ,0x27F3 ,0x2847 ,0x289A ,0x28ED
,0x293F ,0x2991 ,0x29E2 ,0x2A33 ,0x2A83 ,0x2AD3 ,0x2B22 ,0x2B71 ,0x2BC0
,0x2C0E ,0x2C5C ,0x2CA9 ,0x2CF5 ,
    0x2D41 ,0x2D8D ,0x2DD8 ,0x2E22 ,0x2E6D ,0x2EB6 ,0x2EFF ,0x2F48
,0x2F90 ,0x2FD7 ,0x301E ,0x3065 ,0x30AA ,0x30F0 ,0x3135 ,0x3179 ,0x31BD
,0x3200 ,0x3243 ,0x3285 ,0x32C6 ,0x3307 ,0x3348 ,0x3388 ,0x33C7 ,0x3406
,0x3444 ,0x3481 ,0x34BE ,0x34FB ,
    0x3537 ,0x3572 ,0x35AD ,0x35E7 ,0x3620 ,0x3659 ,0x3692 ,0x36C9
,0x3701 ,0x3737 ,0x376D ,0x37A2 ,0x37D7 ,0x380B ,0x383F ,0x3871 ,0x38A4
,0x38D5 ,0x3906 ,0x3937 ,0x3966 ,0x3995 ,0x39C4 ,0x39F2 ,0x3A1F ,0x3A4C
,0x3A78 ,0x3AA3 ,0x3ACD ,0x3AF7 ,
    0x3B21 ,0x3B4A ,0x3B72 ,0x3B99 ,0x3BC0 ,0x3BE6 ,0x3C0B ,0x3C30
,0x3C54 ,0x3C78 ,0x3C9B ,0x3CBD ,0x3CDE ,0x3CFF ,0x3D1F ,0x3D3F ,0x3D5D
,0x3D7B ,0x3D99 ,0x3DB6 ,0x3DD2 ,0x3DED ,0x3E08 ,0x3E22 ,0x3E3B ,0x3E54
,0x3E6C ,0x3E83 ,0x3E9A ,0x3EB0 ,
    0x3EC5 ,0x3EDA ,0x3EEE ,0x3F01 ,0x3F13 ,0x3F25 ,0x3F36 ,0x3F47
,0x3F56 ,0x3F65 ,0x3F74 ,0x3F81 ,0x3F8E ,0x3F9B ,0x3FA6 ,0x3FB1 ,0x3FBB
,0x3FC5 ,0x3FCD ,0x3FD6 ,0x3FDD ,0x3FE4 ,0x3FEA ,0x3FEF ,0x3FF3 ,0x3FF7
,0x3FFA ,0x3FFD ,0x3FFF ,0x4000 ,
};

```

```

//// LUT of imaginary part of exponential (phase shift) with size 960 ////

```

```

static const int16_t exp_DFT_Q[]={
    0x0000 ,0xFF95 ,0xFF2A ,0xFEBA ,0xFE53 ,0xFDE8 ,0xFD7D ,
0xFD12 ,0FCA7 ,0FC3B ,0FBD0 ,0FB65 ,0FAFB ,0FA90 ,0FA25 ,
0xF9BA ,0xF94F ,0xF8E5 ,0xF87A ,0xF810 ,0xF7A5 ,0xF73B ,0xF6D1 ,
0xF667 ,0xF5FD ,0xF593 ,0xF529 ,0xF4C0 ,0xF456 ,0xF3ED ,
    0xF384 ,0xF31B ,0xF2B2 ,0xF249 ,0xF1E0 ,0xF178 ,0xF10F ,
0xF0A7 ,0xF03F ,0xEFD7 ,0xEF70 ,0xEF08 ,0xEEA1 ,0xEE3A ,0xEDD3 ,
0xED6C ,0xED05 ,0xEC9F ,0xEC39 ,0xEBD3 ,0xEB6E ,0xEB08 ,0xEAA3 ,
0xEA3E ,0xE9D9 ,0xE975 ,0xE910 ,0xE8AD ,0xE849 ,0xE7E5 ,
    0xE782 ,0xE71F ,0xE6BD ,0xE65A ,0xE5F8 ,0xE596 ,0xE535 ,
0xE4D3 ,0xE473 ,0xE412 ,0xE3B2 ,0xE352 ,0xE2F2 ,0xE292 ,0xE233 ,

```

0xE1D5 , 0xE176 , 0xE118 , 0xE0BA , 0xE05D , 0xE000 , 0xDFA3 , 0xDF47 ,  
0xDEEB , 0xDE8F , 0xDE34 , 0xDDD9 , 0xDD7F , 0xDD25 , 0xDCCB ,  
0xDC72 , 0xDC19 , 0xDBC0 , 0xDB68 , 0xDB10 , 0xDAB9 , 0xDA62 ,  
0xDA0B , 0xD9B5 , 0xD95F , 0xD90A , 0xD8B5 , 0xD861 , 0xD80D , 0xD7B9 ,  
0xD766 , 0xD713 , 0xD6C1 , 0xD66F , 0xD61E , 0xD5CD , 0xD57D , 0xD52D ,  
0xD4DE , 0xD48F , 0xD440 , 0xD3F2 , 0xD3A4 , 0xD357 , 0xD30B ,  
0xD2BF , 0xD273 , 0xD228 , 0xD1DE , 0xD193 , 0xD14A , 0xD101 ,  
0xD0B8 , 0xD070 , 0xD029 , 0xCFE2 , 0xCF9B , 0xCF56 , 0xCF10 , 0xCECB ,  
0xCE87 , 0xCE43 , 0xCE00 , 0xCDBD , 0xCD7B , 0xCD3A , 0xCCF9 , 0xCCB8 ,  
0xCC78 , 0xCC39 , 0xCBFA , 0xCBBC , 0xCB7F , 0xCB42 , 0xCB05 ,  
0xCAC9 , 0xCA8E , 0xCA53 , 0xCA19 , 0xC9E0 , 0xC9A7 , 0xC96E ,  
0xC937 , 0xC8FF , 0xC8C9 , 0xC893 , 0xC85E , 0xC829 , 0xC7F5 , 0xC7C1 ,  
0xC78F , 0xC75C , 0xC72B , 0xC6FA , 0xC6C9 , 0xC69A , 0xC66B , 0xC63C ,  
0xC60E , 0xC5E1 , 0xC5B4 , 0xC588 , 0xC55D , 0xC533 , 0xC509 ,  
0xC4DF , 0xC4B6 , 0xC48E , 0xC467 , 0xC440 , 0xC41A , 0xC3F5 ,  
0xC3D0 , 0xC3AC , 0xC388 , 0xC365 , 0xC343 , 0xC322 , 0xC301 , 0xC2E1 ,  
0xC2C1 , 0xC2A3 , 0xC285 , 0xC267 , 0xC24A , 0xC22E , 0xC213 , 0xC1F8 ,  
0xC1DE , 0xC1C5 , 0xC1AC , 0xC194 , 0xC17D , 0xC166 , 0xC150 ,  
0xC13B , 0xC126 , 0xC112 , 0xC0FF , 0xC0ED , 0xC0DB , 0xC0CA ,  
0xC0B9 , 0xC0AA , 0xC09B , 0xC08C , 0xC07F , 0xC072 , 0xC065 , 0xC05A ,  
0xC04F , 0xC045 , 0xC03B , 0xC033 , 0xC02A , 0xC023 , 0xC01C , 0xC016 ,  
0xC011 , 0xC00D , 0xC009 , 0xC006 , 0xC003 , 0xC001 , 0xC000 ,  
0xC000 , 0xC000 , 0xC001 , 0xC003 , 0xC006 , 0xC009 , 0xC00D ,  
0xC011 , 0xC016 , 0xC01C , 0xC023 , 0xC02A , 0xC033 , 0xC03B , 0xC045 ,  
0xC04F , 0xC05A , 0xC065 , 0xC072 , 0xC07F , 0xC08C , 0xC09B , 0xC0AA ,  
0xC0B9 , 0xC0CA , 0xC0DB , 0xC0ED , 0xC0FF , 0xC112 , 0xC126 ,  
0xC13B , 0xC150 , 0xC166 , 0xC17D , 0xC194 , 0xC1AC , 0xC1C5 ,  
0xC1DE , 0xC1F8 , 0xC213 , 0xC22E , 0xC24A , 0xC267 , 0xC285 , 0xC2A3 ,  
0xC2C1 , 0xC2E1 , 0xC301 , 0xC322 , 0xC343 , 0xC365 , 0xC388 , 0xC3AC ,  
0xC3D0 , 0xC3F5 , 0xC41A , 0xC440 , 0xC467 , 0xC48E , 0xC4B6 ,  
0xC4DF , 0xC509 , 0xC533 , 0xC55D , 0xC588 , 0xC5B4 , 0xC5E1 ,  
0xC60E , 0xC63C , 0xC66B , 0xC69A , 0xC6C9 , 0xC6FA , 0xC72B , 0xC75C ,  
0xC78F , 0xC7C1 , 0xC7F5 , 0xC829 , 0xC85E , 0xC893 , 0xC8C9 , 0xC8FF ,  
0xC937 , 0xC96E , 0xC9A7 , 0xC9E0 , 0xCA19 , 0xCA53 , 0xCA8E ,  
0xCAC9 , 0xCB05 , 0xCB42 , 0xCB7F , 0xCBBC , 0xCBFA , 0xCC39 ,  
0xCC78 , 0xCCB8 , 0xCCF9 , 0xCD3A , 0xCD7B , 0xCDBD , 0xCE00 , 0xCE43 ,  
0xCE87 , 0xCECB , 0xCF10 , 0xCF56 , 0xCF9B , 0xCFE2 , 0xD029 , 0xD070 ,  
0xD0B8 , 0xD101 , 0xD14A , 0xD193 , 0xD1DE , 0xD228 , 0xD273 ,  
0xD2BF , 0xD30B , 0xD357 , 0xD3A4 , 0xD3F2 , 0xD440 , 0xD48F ,  
0xD4DE , 0xD52D , 0xD57D , 0xD5CD , 0xD61E , 0xD66F , 0xD6C1 , 0xD713 ,  
0xD766 , 0xD7B9 , 0xD80D , 0xD861 , 0xD8B5 , 0xD90A , 0xD95F , 0xD9B5 ,  
0xDA0B , 0xDA62 , 0xDAB9 , 0xDB10 , 0xDB68 , 0xDBC0 , 0xDC19 ,  
0xDC72 , 0xDCCB , 0xDD25 , 0xDD7F , 0xDDD9 , 0xDE34 , 0xDE8F ,  
0xDEEB , 0xDF47 , 0xDFA3 , 0xE000 , 0xE05D , 0xE0BA , 0xE118 , 0xE176 ,  
0xE1D5 , 0xE233 , 0xE292 , 0xE2F2 , 0xE352 , 0xE3B2 , 0xE412 , 0xE473 ,  
0xE4D3 , 0xE535 , 0xE596 , 0xE5F8 , 0xE65A , 0xE6BD , 0xE71F ,  
0xE782 , 0xE7E5 , 0xE849 , 0xE8AD , 0xE910 , 0xE975 , 0xE9D9 ,  
0xEA3E , 0xEAA3 , 0xEB08 , 0xEB6E , 0EBD3 , 0xEC39 , 0xEC9F , 0xED05 ,  
0xED6C , 0xEDD3 , 0xEE3A , 0xEEA1 , 0xEF08 , 0xEF70 , 0xEFD7 , 0xF03F ,  
0xF0A7 , 0xF10F , 0xF178 , 0xF1E0 , 0xF249 , 0xF2B2 , 0xF31B ,  
0xF384 , 0xF3ED , 0xF456 , 0xF4C0 , 0xF529 , 0xF593 , 0xF5FD ,  
0xF667 , 0xF6D1 , 0xF73B , 0xF7A5 , 0xF810 , 0xF87A , 0xF8E5 , 0xF94F ,  
0xF9BA , 0xFA25 , 0xFA90 , 0FAFB , 0xFB65 , 0FBFD0 , 0xFC3B , 0xFCA7 ,  
0xFD12 , 0xFD7D , 0xFDE8 , 0xFE53 , 0FEBE , 0xFF2A , 0xFF95 ,  
0x0000 , 0x006B , 0x00D6 , 0x0142 , 0x01AD , 0x0218 , 0x0283 ,  
0x02EE , 0x0359 , 0x03C5 , 0x0430 , 0x049B , 0x0505 , 0x0570 , 0x05DB ,

0x0646 , 0x06B1 , 0x071B , 0x0786 , 0x07F0 , 0x085B , 0x08C5 , 0x092F ,  
0x0999 , 0x0A03 , 0x0A6D , 0x0AD7 , 0x0B40 , 0x0BAA , 0x0C13 ,  
0x0C7C , 0x0CE5 , 0x0D4E , 0x0DB7 , 0x0E20 , 0x0E88 , 0x0EF1 ,  
0x0F59 , 0x0FC1 , 0x1029 , 0x1090 , 0x10F8 , 0x115F , 0x11C6 , 0x122D ,  
0x1294 , 0x12FB , 0x1361 , 0x13C7 , 0x142D , 0x1492 , 0x14F8 , 0x155D ,  
0x15C2 , 0x1627 , 0x168B , 0x16F0 , 0x1753 , 0x17B7 , 0x181B ,  
0x187E , 0x18E1 , 0x1943 , 0x19A6 , 0x1A08 , 0x1A6A , 0x1ACB ,  
0x1B2D , 0x1B8D , 0x1BEE , 0x1C4E , 0x1CAE , 0x1D0E , 0x1D6E , 0x1DCD ,  
0x1E2B , 0x1E8A , 0x1EE8 , 0x1F46 , 0x1FA3 , 0x2000 , 0x205D , 0x20B9 ,  
0x2115 , 0x2171 , 0x21CC , 0x2227 , 0x2281 , 0x22DB , 0x2335 ,  
0x238E , 0x23E7 , 0x2440 , 0x2498 , 0x24F0 , 0x2547 , 0x259E ,  
0x25F5 , 0x264B , 0x26A1 , 0x26F6 , 0x274B , 0x279F , 0x27F3 , 0x2847 ,  
0x289A , 0x28ED , 0x293F , 0x2991 , 0x29E2 , 0x2A33 , 0x2A83 , 0x2AD3 ,  
0x2B22 , 0x2B71 , 0x2BC0 , 0x2C0E , 0x2C5C , 0x2CA9 , 0x2CF5 ,  
0x2D41 , 0x2D8D , 0x2DD8 , 0x2E22 , 0x2E6D , 0x2EB6 , 0x2EFF ,  
0x2F48 , 0x2F90 , 0x2FD7 , 0x301E , 0x3065 , 0x30AA , 0x30F0 , 0x3135 ,  
0x3179 , 0x31BD , 0x3200 , 0x3243 , 0x3285 , 0x32C6 , 0x3307 , 0x3348 ,  
0x3388 , 0x33C7 , 0x3406 , 0x3444 , 0x3481 , 0x34BE , 0x34FB ,  
0x3537 , 0x3572 , 0x35AD , 0x35E7 , 0x3620 , 0x3659 , 0x3692 ,  
0x36C9 , 0x3701 , 0x3737 , 0x376D , 0x37A2 , 0x37D7 , 0x380B , 0x383F ,  
0x3871 , 0x38A4 , 0x38D5 , 0x3906 , 0x3937 , 0x3966 , 0x3995 , 0x39C4 ,  
0x39F2 , 0x3A1F , 0x3A4C , 0x3A78 , 0x3AA3 , 0x3ACD , 0x3AF7 ,  
0x3B21 , 0x3B4A , 0x3B72 , 0x3B99 , 0x3BC0 , 0x3BE6 , 0x3C0B ,  
0x3C30 , 0x3C54 , 0x3C78 , 0x3C9B , 0x3CBD , 0x3CDE , 0x3CFF , 0x3D1F ,  
0x3D3F , 0x3D5D , 0x3D7B , 0x3D99 , 0x3DB6 , 0x3DD2 , 0x3DED , 0x3E08 ,  
0x3E22 , 0x3E3B , 0x3E54 , 0x3E6C , 0x3E83 , 0x3E9A , 0x3EB0 ,  
0x3EC5 , 0x3EDA , 0x3EEE , 0x3F01 , 0x3F13 , 0x3F25 , 0x3F36 ,  
0x3F47 , 0x3F56 , 0x3F65 , 0x3F74 , 0x3F81 , 0x3F8E , 0x3F9B , 0x3FA6 ,  
0x3FB1 , 0x3FBB , 0x3FC5 , 0x3FCd , 0x3FD6 , 0x3FDD , 0x3FE4 , 0x3FEA ,  
0x3FEF , 0x3FF3 , 0x3FF7 , 0x3FFA , 0x3FFD , 0x3FFF , 0x4000 ,  
0x4000 , 0x4000 , 0x3FFF , 0x3FFD , 0x3FFA , 0x3FF7 , 0x3FF3 ,  
0x3FEF , 0x3FEA , 0x3FE4 , 0x3FDD , 0x3FD6 , 0x3FCD , 0x3FC5 , 0x3FBB ,  
0x3FB1 , 0x3FA6 , 0x3F9B , 0x3F8E , 0x3F81 , 0x3F74 , 0x3F65 , 0x3F56 ,  
0x3F47 , 0x3F36 , 0x3F25 , 0x3F13 , 0x3F01 , 0x3EEE , 0x3EDA ,  
0x3EC5 , 0x3EB0 , 0x3E9A , 0x3E83 , 0x3E6C , 0x3E54 , 0x3E3B ,  
0x3E22 , 0x3E08 , 0x3DED , 0x3DD2 , 0x3DB6 , 0x3D99 , 0x3D7B , 0x3D5D ,  
0x3D3F , 0x3D1F , 0x3CFF , 0x3CDE , 0x3CBD , 0x3C9B , 0x3C78 , 0x3C54 ,  
0x3C30 , 0x3C0B , 0x3BE6 , 0x3BC0 , 0x3B99 , 0x3B72 , 0x3B4A ,  
0x3B21 , 0x3AF7 , 0x3ACD , 0x3AA3 , 0x3A78 , 0x3A4C , 0x3A1F ,  
0x39F2 , 0x39C4 , 0x3995 , 0x3966 , 0x3937 , 0x3906 , 0x38D5 , 0x38A4 ,  
0x3871 , 0x383F , 0x380B , 0x37D7 , 0x37A2 , 0x376D , 0x3737 , 0x3701 ,  
0x36C9 , 0x3692 , 0x3659 , 0x3620 , 0x35E7 , 0x35AD , 0x3572 ,  
0x3537 , 0x34FB , 0x34BE , 0x3481 , 0x3444 , 0x3406 , 0x33C7 ,  
0x3388 , 0x3348 , 0x3307 , 0x32C6 , 0x3285 , 0x3243 , 0x3200 , 0x31BD ,  
0x3179 , 0x3135 , 0x30F0 , 0x30AA , 0x3065 , 0x301E , 0x2FD7 , 0x2F90 ,  
0x2F48 , 0x2EFF , 0x2EB6 , 0x2E6D , 0x2E22 , 0x2DD8 , 0x2D8D ,  
0x2D41 , 0x2CF5 , 0x2CA9 , 0x2C5C , 0x2C0E , 0x2BC0 , 0x2B71 ,  
0x2B22 , 0x2AD3 , 0x2A83 , 0x2A33 , 0x29E2 , 0x2991 , 0x293F , 0x28ED ,  
0x289A , 0x2847 , 0x27F3 , 0x279F , 0x274B , 0x26F6 , 0x26A1 , 0x264B ,  
0x25F5 , 0x259E , 0x2547 , 0x24F0 , 0x2498 , 0x2440 , 0x23E7 ,  
0x238E , 0x2335 , 0x22DB , 0x2281 , 0x2227 , 0x21CC , 0x2171 ,  
0x2115 , 0x20B9 , 0x205D , 0x2000 , 0x1FA3 , 0x1F46 , 0x1EE8 , 0x1E8A ,  
0x1E2B , 0x1DCD , 0x1D6E , 0x1D0E , 0x1CAE , 0x1C4E , 0x1BEE , 0x1B8D ,  
0x1B2D , 0x1ACB , 0x1A6A , 0x1A08 , 0x19A6 , 0x1943 , 0x18E1 ,  
0x187E , 0x181B , 0x17B7 , 0x1753 , 0x16F0 , 0x168B , 0x1627 ,  
0x15C2 , 0x155D , 0x14F8 , 0x1492 , 0x142D , 0x13C7 , 0x1361 , 0x12FB ,

```

0x1294 , 0x122D , 0x11C6 , 0x115F , 0x10F8 , 0x1090 , 0x1029 , 0x0FC1 ,
0x0F59 , 0x0EF1 , 0x0E88 , 0x0E20 , 0x0DB7 , 0x0D4E , 0x0CE5 ,
    0x0C7C , 0x0C13 , 0x0BAA , 0x0B40 , 0x0AD7 , 0x0A6D , 0x0A03 ,
0x0999 , 0x092F , 0x08C5 , 0x085B , 0x07F0 , 0x0786 , 0x071B , 0x06B1 ,
0x0646 , 0x05DB , 0x0570 , 0x0505 , 0x049B , 0x0430 , 0x03C5 , 0x0359 ,
0x02EE , 0x0283 , 0x0218 , 0x01AD , 0x0142 , 0x00D6 , 0x006B ,
};

```

```

//initial function to map to the dft_obj structure
//mapping the input , output and number of resource blocks to get M_PUSH
void pusch_dft_init (pusch_dft *dft_obj, int32_t *input, int32_t *output ,
uint8_t N_rb)
{
    dft_obj->in = input;
    dft_obj->out = output;
    dft_obj -> M_PUSH = N_rb*12;
}

```

```

//the essential function of building DFT
// take the arguments of the DFT structure and number of OFDM symbol (l)

```

```

void pusch_dft_start (pusch_dft *dft_obj, uint8_t l)
{
    int32_t sumreal , sumimag;
    int16_t exp_real,exp_imag,temp_I,temp_Q;
    uint16_t k,temp,init_indx_out,i,init_indx_in;
    //the bits of summation must be enough for the over flow which is
    calculated by log2(N)where N :size of internal loop
    //here the over flow bits here is log2(960)>> 10 bits so the all bits
    16+16+10=42 bit
    //that mean using int 64
    //from matlab the most number of bits of effective sum are in 36 bit
    here in this cale because the numbers is low
    int64_t sumreal2 , sumimag2;

    dft_obj -> l = l; // ofdm symbol
    //update the index of output from DFT
    //by using l which is here 12 .one for each M_PUSH
    init_indx_out=(dft_obj -> l)*(dft_obj -> M_PUSH) ; // start of output
    index per L (ofdm symbol)

    for (k = 0; k < dft_obj -> M_PUSH; k++) /* For each output element */
    {
        // initial sum =0
        sumreal2 = 0;
        sumimag2 = 0;
        //update the part of input array that will be udes to get DFT
        //by using l which is here 12 .one for each M_PUSH
        init_indx_in=(dft_obj -> l)*(dft_obj -> M_PUSH) ; // start of
        input index per L (ofdm symbol)
    }
}

```

```

//internal loop of summation the multiplication
    for (i = 0; i < dft_obj -> M_PUSH; i++,init_indx_in++) /* For
each input element */
    {
        //the index of real and imaginary phase LUT's
        temp= ((i)*k)%960); // after the M_push the phase start
new cycle

//get the real and imaginary values from LUT's
        exp_real=exp_DFT_I[temp];
        exp_imag=exp_DFT_Q[temp];

        temp_I =((dft_obj ->in[init_indx_in])>>16);
        temp_Q= ((dft_obj ->in[init_indx_in])&0x0000FFFF);

        sumreal = (((int32_t)temp_I * exp_real) - ((int32_t)temp_Q
* exp_imag)) ; //both exp and input are complex
        sumimag = (((int32_t)temp_Q * exp_real) + ((int32_t)temp_I
* exp_imag)) ;

        //the summation on the internal loop
        sumreal2 = (sumreal+sumreal2);
        sumimag2 = (sumimag+sumimag2);

    }

    // save the Q-format (fixed point representation =14)
    // (Q_of input=14)*(Q_of exp =14 ) =Q=28
    // after that scaling by divide on 2 to match test cases so
(shift to right 1)
    // so shifting 15 came from Q_exp=14 and 1 for dividing by 2(
scaling factors)
    //////////////////////////////////////
    /// the real part of input normalized (/sqrt(M_push) )
    sumreal2 = ((sumreal2)>>15)/31;
    // Q = 14 , Bmx = 26
    sumreal2 =sumreal2&0x000000000000FFFF;
    /// the imaginary part of input normalized (/sqrt(M_push) )
    sumimag2 = ((sumimag2)>>15)/31;
    sumimag2 =sumimag2&0x000000000000FFFF;

    //generate the output symbol by concatenate the real(MSB)
&real(LSB)
    dft_obj ->out[init_indx_out] = (sumreal2)<<16 |((sumimag2)
&0xFFFF);

    init_indx_out+=1;
}

```



```

}

//linear interpolation function
//it will be used if LUT with size 1200
int16_t pusch_dft_interplation (int16_t y1 ,int16_t y2 ,int8_t x1,int8_t x2
,float i )

{
    int16_t z;

    z = y1 + ((y2 - y1) *((i - x1)) / (x2 - x1));

    return z;
}

// swapping endianness function
void big_little_swapping(int32_t *x ,uint32_t size)

{
    int k;

    for (k=0;k < size ;k++)
    {
        x[k]= ((x[k]>>24)&0xff) | // move byte 3 to byte 0
              ((x[k]<<8)&0xff0000) | // move byte 1 to byte 2
              ((x[k]>>8)&0xff00) | // move byte 2 to byte 1
              ((x[k]<<24)&0xff000000); // byte 0 to byte 3
    }

    return;
}

```

\*\*\*\*\*

## Appendix III Budget

<b>Component</b>	<b>Quantity</b>	<b>Source</b>	<b>Price</b>
<b>Parallella Board</b>	2	Amazon	3000 LE
<b>3.3V Fr232RL FTDI Usb to Serial Adapter Module</b>	1	Amazon	200 LE
<b>Ethernet cable</b>	1	RAM	30 LE
<b>SD Card</b>	2	Elbostan Mall	120 LE
<b>Total</b>			<b>3350 LE</b>