Mo, Ganyong; Castells-Rufas, David, dir. FPGA implementation of bluetooth low energy physical layer with OpenCL. 2022. 82 pag. (1170 Màster Universitari en Enginyeria de Telecomunicació / Telecommunication Engineering)

A Thesis for the

Master in Telecommunication Engineering

FPGA Implementation of Bluetooth Low Energy Physical Layer with OpenCL

by

Ganyong Mo
Ganyong.Mo@autonoma.cat

Supervisor: David Castells Rufas
David.Castells@uab.cat

Department of Computer Architecture and Operating Systems

Escola Tècnica Superior d’Enginyeria (ETSE)
Universitat Autònoma de Barcelona (UABs)

June, 2022

El sotasignant, David Castells Rufas, Professor de l’Escola Tècnica Superior d’Enginyeria (ETSE) de la Universitat Autònoma de Barcelona (UAB),

CERTIFICA:

Que el projecte presentat en aquesta memòria de Treball Final de Master ha estat realitzat sota la seva direcció per l’alumne Ganyong Mo.

I, perquè consti a tots els efectes, signa el present certificat.

Bellaterra, 30 de Juny de 2022

Signatura: David Castells Rufas

Resum:

Aquesta dissertació presenta principalment el disseny de processament digital de senyals (DSP) entre la transmissió en Capa Física de Bluetooth de Baixa Energia (BLE PHY), i la seva implementació en dispositius Field Programmable Gate Array (FPGA) utilitzant Open Computing Language (OpenCL).

Durant el disseny de DSP, es basa en l'arquitectura en fase / quadratura-fase (IQ) per construir els processos de modulació i demodulació del senyal mitjançant l'ús d'un esquema de modelador de senyal anomenat Gaussian Frequency-Shift Keying (GFSK), en la comunicació de curt abast que presenta un fort rendiment anti-interferència. Pel que fa a l'OpenCL, és un dels mètodes de síntesi d'alt nivell (HLS) per al disseny de FPGA. No només compta amb una alta productivitat, sinó que també pot realitzar una alta eficiència operativa per FPGA mitjançant l'ús d'arquitectura de programació paral·lela. A més, aquí invoca una plataforma remota anomenada Intel DevCloud per controlar el FPGA per verificar el programa, faria que el disseny fos més còmode i econòmic.

Paraules clau: BLE PHY, GFSK, FPGA, OpenCL, Intel DevCloud, etc.

Resumen:

Esta disertación presenta principalmente el diseño de Procesamiento Digital de Señales (DSP) entre la transmisión en Bluetooth Low Energy Physical Layer (BLE PHY), y su implementación en Field Programmable Gate Array (FPGA) con Open Computing Language (OpenCL).

Durante el diseño de DSP, se basa en la arquitectura In-Phase/Quadrature-Phase (IQ) para construir los procesos de modulación y demodulación de la señal mediante la utilización de un esquema de modelador de señal llamado Gaussian Frequency-Shift Keying (GFSK), en la comunicación de corto alcance presenta un fuerte rendimiento anti-interferencia. Con respecto al OpenCL, es uno de los métodos de síntesis de alto nivel (HLS) para el diseño de FPGA. No solo presenta una alta productividad, sino que también puede lograr una alta eficiencia operativa para FPGA mediante el uso de la arquitectura de programación paralela. Además, aquí invoca una plataforma remota llamada Intel DevCloud para controlar la FPGA para verificar el programa, lo que haría que el diseño fuera más conveniente y económico.

Palabras clave: BLE PHY, GFSK, FPGA, OpenCL, Intel DevCloud, etc.

Summary:

This dissertation is primarily presenting the design of Digital Signal Processing (DSP) between the transmission in Bluetooth Low Energy Physical Layer (BLE PHY), and its implementation in a Field Programmable Gate Array (FPGA) device with Open Computing Language (OpenCL).

During the design of DSP, it bases on the In-Phase/Quadrature-Phase (IQ) architecture to construct the modulation and demodulation processes of signal by utilizing a signal shaper scheme called Gaussian Frequency-Shift Keying (GFSK), in the short-rang communication it features strong anti-interference performance. Regarding with the OpenCL, it's one of High-Level Synthesis (HLS) methods for FPGAs design. It not only features high productive, but also can realize high operational efficiency for FPGA by using parallel programming architecture. Moreover, here invokes a remote platform called Intel DevCloud to control the FPGA for verifying the program, it would make the design more convenient and economic.

Key words: BLE PHY, GFSK, FPGA, OpenCL, Intel DevCloud, etc.

ACKNOWLEDGMENTS

After worked for three years, it’s so amazing for me that I could back to campus again for further studying. Therefore, thanks for Universitat Autònoma de Barcelona (UAB) gave me this opportunity two years ago, even if we were encountering the grave global pandemic, she also offered me the best studying environment in this duration.

I'm grateful to meet many professional, knowledgeable, and adorable tutors and professors within the campus. Especially my supervisor David, who is a patient teacher as well. He guided me to learn much specialized knowledge, and shared many novel, helpful ideas with me. Meanwhile my views were further broadened during this study, thanks for him sincerely.

I'm deeply appreciative for my parents, my sister, my uncle, and his family, meanwhile, I really miss my grandmother. They always are the most significant people in my life, I’m able to hold on all the time that is because they are always understanding, concerning, supporting and encouraging for me, I want to express my genuine gratitude for my family once more.

Thanks for Jingming, his wisdom always inspired me at key moments, whether working or studying.

Thanks for Marc, he gave me a lot of help when I need something in this project research.

Thanks for my roommates, Sorayda, Leonador, Yuanjiao, Di, Merceders and Haotian, they were accompanying with me when I was studying in Barcelona. They made me felt happiness and warmth as well, though I was far away from my motherland and family.

Thanks for my old friends, Sipan, Sihao, Zhengbiao, Yijie, Minqing, Jinyuan, Xiuhui, Xin, Fanglai ..., they always stood with me and gave me courage to face when I stuck in serious troubles or was at rock bottom every time.

Finally, thanks for all of authors who gave me helpful references for finishing this design and dissertation, and thanks all the people I’ve met ever.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATION

Adaptive Logic Modules	*ALMs*
Amazon Elastic Compute Cloud	*Amazon EC2*
Amazon Web Services	*AWS*
Analog to Digital	*AD*
Binary None Zero	*BNZ*
Bit Error Rate	*BER*
BLE Physical Layer	*BLE PHY*
Bluetooth Low Energy	*BLE*
Central Processing Unit	*CPU*
Complex Programmable Logic Device	*CPLD*
Configurable Logic Block	*CLB*
Cyclic Redundancy Check	*CRC*
Digital Signal Processing	*DSP*
Digital Signal Processors	*DSPs*
Digital to Analog	*DA*
Dynamic Random Access Memory	*DRAM*
Field Programmable Gate Array	*FPGA*
Finite Impulse Response	FIR
First-In, First-Out	*FIFO*
Frequency Hopping	*FH*
Frequency Shift Keying	*FSK*
Gaussian Frequency-Shift Keying	*GFSK*
Gaussian Minimum Shift Keying	*GMSK*

Graphic Processing Unit	*GPU*
Hardware Description Language	*HDL*
High-Level Synthesis	*HLS*
High-Performance Computing	*HPC*
Industrial, Scientific, Medical	*ISM*
Infinite Impulse Response	*IIR*
In-Phase/Quadrature-Phase	*I/Q*
Inter Symbol Interference	*ISI*
Logic Array Block	*LAB*
Look-Up Table	*LUT*
Microcontroller	*uC*
Microprocessor	*uP*
Multiple Instruction Stream Multiple Data Stream	*MIMD*
Multiple Instruction Stream Single Data Stream	*MISD*
Non-Return Zero	*NRZ*
Open Computing Language	*OpenCL*
Operations Per Cycle	*OPC*
Oversample Rate	*OSR*
Programmable Logic Device	*PLD*
Protocol Data Unit	*PDU*
Radio Frequency	*RF*
Random-Access Memories	*RAMs*
Register Transfer Level	*RTL*
Sample per Symbol	*sps*
Single Instruction Stream Multiple Data Stream	*SIMD*

Single Instruction Stream Single Data Stream	*SISD*
Software Define Radio	*SDR*
VHSIC HDL	*VHDL*

1 INTRODUCTION

The communication between the Bluetooth Low Energy (BLE) is working on the 2.4GHz radio wave band, which is licence-free and mainly provide for the fields of Industrial, Scientific, Medical (ISM), so that we can utilize this band for the data transmission easily. Regards with this project, the first we will describe the short-range wireless communication architecture in order to figure out what we mainly need to do.

1.1 Communication Construction

The Figure 1.1 is showing the basic architecture of signal communication, there are three parts, which consists of the Radio Frequency (RF) Transceiver for receiving and transmitting data, Analog to Digital (AD) / Digital to Analog (DA) Transform for changing signals, and the part of Digital Signal Processing (DSP) [1]. However, for this thesis, here is mainly presenting how to design an DSP, where the signal is received from the BLE Physical Layer (BLE PHY), and complete it in Field Programmable Gate Array (FPGA) by using Open Computing Language (OpenCL). Where OpenCL is a framework that includes a method to describe the accelerator (FPGAs, Graphic Processing Units (GPUs)) behaviour by using C/C++, and in the case of FPGA, the C/C++ kernels can be converted into Hardware Description Language (HDL) via a process of High-Level Synthesis (HLS) methods. Comparing to the traditional methods such as VHSIC HDL (VHDL) and Verilog-HDL, the OpenCL has better portability and is easily able to realize parallel programming, hence it features the characteristics of high productive and high-performance processing, moreover, it can increase the interoperability between different platforms as well. Before to detail the design, the next section is introducing basic background and approach.

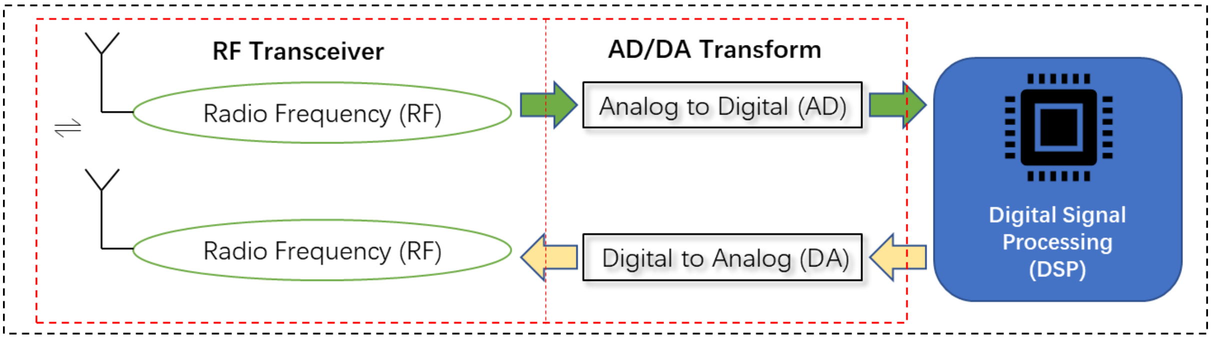


Figure 1.1 Architecture of wireless communication system

1.2 Background and Approach

A hands-free mobile headset was the first Bluetooth product to be made available to consumers, and it was not until 2001 that the first Bluetooth-enabled mobile phone entered the market. Meanwhile, the first notebook with integrated Bluetooth was introduced by IBM in the same year as well [1]. On the other hand, with the rapidly development of portable or mobile devices, within the limit power of battery, the power-efficiency performance and transmission rate have been taken more and more attentions by consumers and developers. Therefore, in the foundation of classical Bluetooth, it was extended to the novel version 4.0 as calling BLE by Nokia in 2006, which has more great performances such as low power consumption, less cost and so forth [2]. Since then, it became one of fundamental technology in each mobile machine which consist of mobile phones, headset, computers and even automobile and so forth, it gave us more and more convenient in everyday life, and we can see and use it all around us during the short-range transmission. With the benefits of BLE, it was shipped approximately four billion Bluetooth devices by 2019, and even if due to the pandemic, in 2020 the shipped was almost same as the last year. Obviously, there is no doubt the Bluetooth is already widespread adoption for the short-rang communication applications [2, 3].

Nowadays, more and more designers or developers are taking focuses on the capabilities of programmability, re-use (re-programmability and component recycling) for the electronic circuits and systems, which allow them to design systems with a relative short time. For the digital systems design, there are a lot of compliant approaches and devices for these features. For example, the complementation with software programs written, we can base on Microprocessor (uP), Microcontroller (uC), Digital Signal Processors (DSPs). Or the realization with a hardware-configuring, we can through Programmable Logic Device (PLD), which includes the Simple (SPLD), Complex (CPLD), and the FPGA. All of these options for designing a digital system is given by Figure 1.2 [4]. For this project, because the performances of energy efficiency and high computing are taken more attentions by us, and comparing all of processors and accelerators, in most cases, the FPGA is more suitable for these essential requirements. Where one important technical specification of FPGA is that the number of gates, which is associating with the capability of computing and processing. And this index was increasing from the 10 thousand level in 1980s, up to 50 million level in 2010s [5], hence we can see that the complex-computing ability of FPGA was becoming stronger step by step. Moreover, there are more than 40 companies have been engaging to this industry so far, such as Xilinx, Altera (now it belongs to Intel), Lattice and so on. Besides, with the rapid development of internet and communication in recent decade, and considering the cost and convenience for program developing, testing and debugging, the vendor Intel launched a platform called DevCloud, which bases on the idea of edge inference solutions for learning, prototyping, testing, and running the workload by using the latest hardware or software sets of Intel. In this platform, it includes a serial of hardware such as Central Processing Units (CPUs), GPUs, FPGAs. Specially, there is an environment we desired in this remote platform, that is OpenCL for FPGA development.

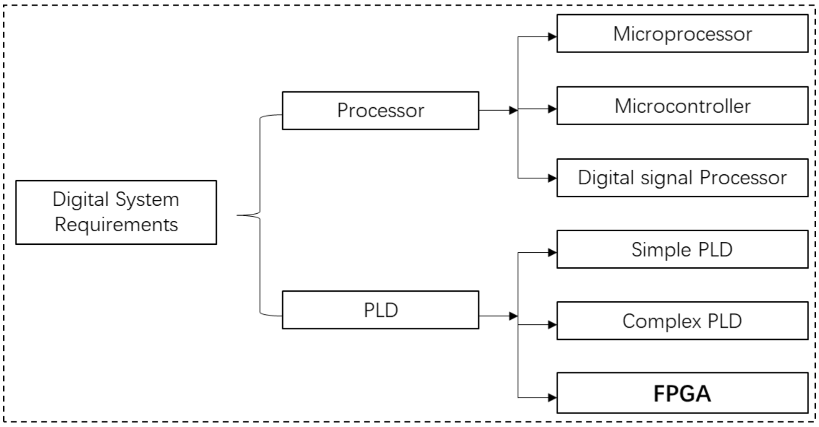


Figure 1.2 The options for digital system design

About the OpenCL, different from others language, but it has non-overlook relationship with others as well. Because it bases on C99, C++14 and C++17, but can write program for executing the across heterogeneous platforms that combination of CPUs, GPUs, FPGAs, DSPs and other processor or accelerators. OpenCL is an Open standard source by the non-profit technology consortium Khronos Group, which promoter members consist of AMD, Nvidia, ARM, Intel, Qualcomm, Apple, Huawei and so forth, so, there are many development sources we can utilize easily and conveniently, the perspective is very bright in the future. In this case, in 2008, the initial iteration of OpenCL instructions was released, which was considering the developers of program don’t have to learn exclusive or multiple languages when work with different devices. In other words, make a common interface, in order to apply a same program to the cross platforms. Besides the advantage of portability, there is a very powerful and attractive characteristic difference with others languages as well, that is offering us the model of parallel programming [6], which can make the program processing more efficient. Now the latest version was released in 2020 and has an instruction called OpenCL Specification version 3.0.

Among this design, we accept a signal architecture named In-Phase/Quadrature-Phase (I/Q), which accomplished by the pulse shaping technology of Gaussian Frequency-Shift Keying (GFSK) modulation. The approach of GFSK has many advantages, such as power spectrum, constant amplitude envelope, narrow spectrum, great performance in anti-interference, low-cost and so forth, most of these are very desirable in the wireless communication systems [7, 8]. As a result, the approach of GFSK is very appropriate for processing the signal between the transmission of BLE PHY, even said exclusive method. Besides, take the processing efficiency into account, here invokes the method OpenCL which can deal well with the parallel programming and then improve the operational rate of hardware in the aspect of software. As for the FPGA platform, the Intel DevCloud was accepted, where there are several types of FPGA and support to run the OpenCL program, so that we can select one and control online. Obviously, it absolutely is a lowest-cost and minimum-time approach for us to verify and debug the programs what we designed.

1.3 Dissertation Outline

During this project, we would work around the objective, so this thesis is briefly structured as following:

The second chapter will explain the primary theories about the BLE PHY, GFSK modulation and demodulation, FPGA, the remote platform Intel DevCloud and the OpenCL. Where it will depict these via a several of aspects such as architectures, algorithms, common usages and so forth.

About the chapter 3, according to the previous chapter, it will illustrate and shortly analyze the simulating processes, and graphically show the results of GFSK modulation and demodulation via software Simulink, let’s clearly perceive what GFSK actually modulates and demodulates.

The chapter 4 dedicates to show how to realize the GFSK modulation and demodulation in OpenCL, which composes of host part and kernel part. It will construct a whole project relying on the C standard libraries and assembling the specific usages from OpenCL. Besides, it will illustrate the methods and results of compiling, executing and debugging with FPGA in Intel DevCloud. Then we will accomplish the comparison and analysis between the results of Simulink simulated and FPGA executed, and the verification with real BLE data stream from theories to practice, all of these are interpreted in here.

The final chapter, it will make a conclusion for this work, and look forward some perspectives.

2 THEORIES

The preceding chapter has explicated the objective, and if we want to reach this destination, we need to figure out the essential theories what we will utilize. In this case, here includes some architectures, concepts, algorithms, methods and so forth. Therefore, in this chapter, we will explain the knowledge of BLE PHY, the modulation and demodulation of GFSK, and the algorithms among these designs. Besides, there are FPGA principles, the primary design concepts of OpenCL and some usages of remote platform as well.

2.1 BLE

Nowadays, the application of Bluetooth is pervasive, especially after the launched of BLE. In this case, this section will interpret the protocol architecture of BLE and the principle of BLE PHY. The first thing, let's see what is the most attractive characteristic of BLE.

2.1.1 Why BLE

BLE is one of short-rang wireless communication protocol, besides, there are others well performance of wireless protocols such as Wi-Fi and Zigbee. Even if BLE can be a fundamental function or part in a large amount of mobile or portable devices, the Wi-Fi is also enjoyed by us almost in daily life and everywhere. Meanwhile, the Wi-Fi and Zigbee are working in the free-licence band of ISM as well, why we select the BLE rather than Wi-Fi or Zigbee?

Table 2.1 Characteristics comparing of BLE, Zigbee and Wi-Fi

Class **Characteristics**	**BLE**	**Zigbee**	**Wi-Fi**
IEEE Specification	802.15.1	802.15.4	802.11a/b/g
Frequency Band	2.4GHz	2.4GHz, 868/915MHz	2.4GHz/5GHz
Modulation	GFSK	DSSS	OFDM, DSSS, CCK
Range	< 100m	10~100m	50m
Network Topologies	Point to Point, Start, Mesh	Star, Mesh	Star
Data Rate	1Mb/s, 2Mb/s	250Kb/s	11Mb/s, 54Mb/s, 150Mb/s +
Peak Current Consumption	<15mA, ~5.5 mA in latest devices	19mA Rx, 35mA Tx	60mA Rx, 200mA Tx
Standby Current	< 2 uA	~5 uA	<100uA

The answer is among in the Table 2.1, which is about the characteristic comparison of BLE, Zigbee and Wi-Fi [9-11]. Obviously, we can see the *Peak Current Consumption* and *Standby Current* of BLE, that is lowest among these three protocols. In this case, it is a critical factor we selected BLE, and can be more prospective in industrial communication or other short-rang wireless communication fields because of the lower consumption. At the same time, that is why it can be a fundamental part in all of mobile or portable devices as well, though people not always use it in the day life comparing with Wi-Fi.

2.1.2 BLE PHY

However, there are many protocol stacks among the BLE, mainly consists of Application, Host and Controller. The Figure 2.1 shows the exactly position of BLE PHY [12], we can see that the physical layer is in the Controller, which in the lowest layer of the stack. Besides, the PYH is able to transmission the data over the air via radio waves, that is also itself responsibility.

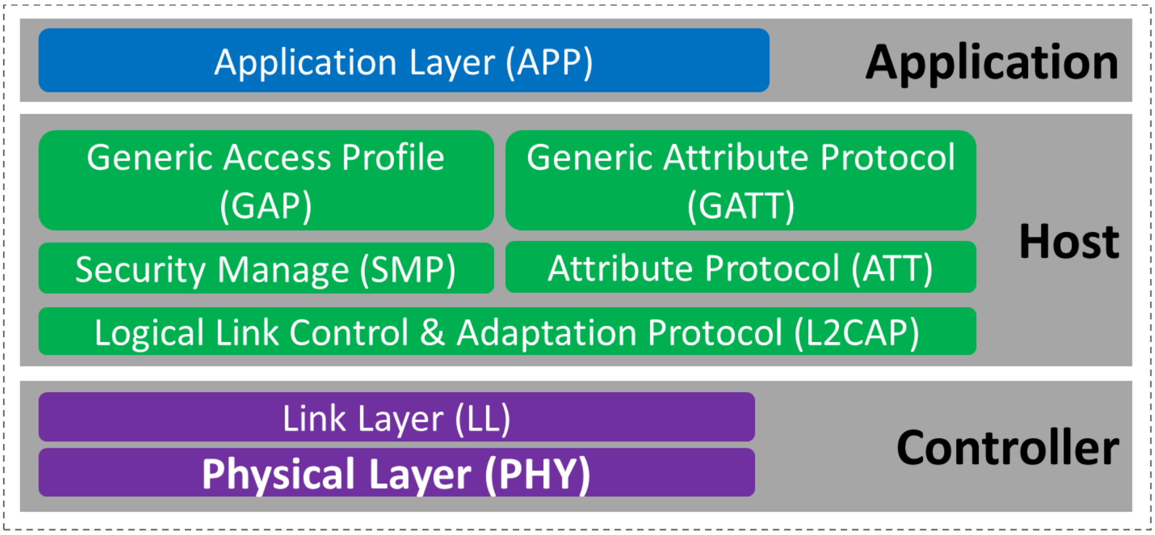


Figure 2.1 Protocol stacks of BLE

As mention previous, BLE PHY will transmission the data through the 2.4GHz frequency band via the radio wave. Meanwhile, the frequency band is separated into 40 channels basing on 2MHz interval from 2400.0MHz to 2483.5MHz, which starting at 2402MHz, and it is shown in the Figure 2.2 [9, 12]. We can see that there are two modes of channel called Data and Advertising respectively, where the 37, 38, 39 are referred to the advertising channels for broadcasting, connection establishing and device discovering. On the other hand, the rest channels are data channels and can be used for bidirectional communication and adaptive Frequency Hopping (FH).

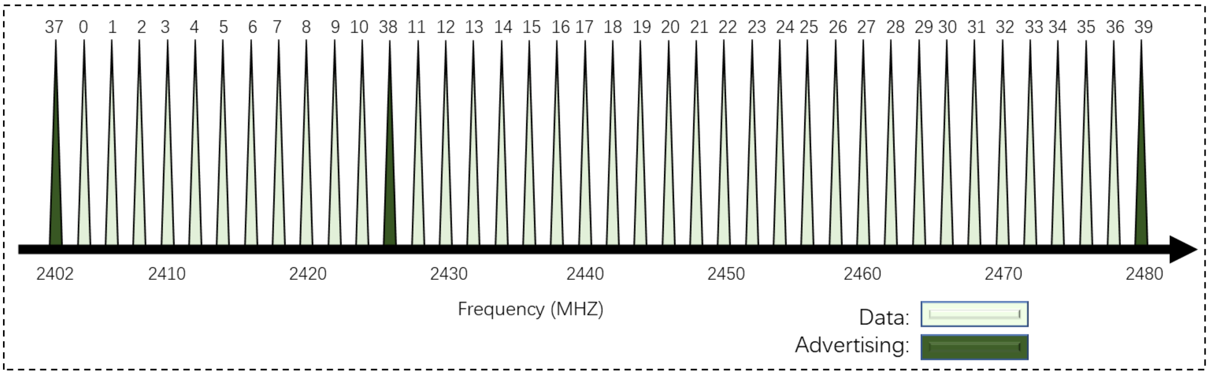


Figure 2.2 BLE PHY frequency bands

In 2.4GHz band, it can transfer information by Bluetooth, Wi-Fi, Zigbee, NFC and so forth, then the overcrowding, congestion are common problems, even will cause the signal interference and fading. In this case, BLE uses the method FH to suppress these issues. According to a pseudorandom sequence of the channel index, the RF signals are quickly shifted to different channels in a system, and that is the principle of FH. [13], incidentally, it is applied in data channels.

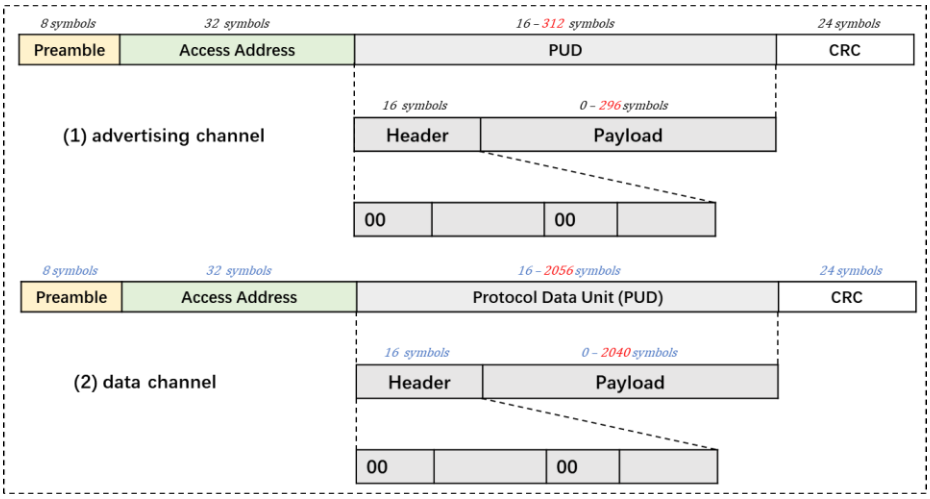


Figure 2.3 BLE packet format (1) advertising channel (2) data channel

But what kinds of information it will send and receive? Therefore, there is the BLE PHY packet format, as showing in the Figure 2.3. Where mainly consists of the preamble, access address, Protocol Data Unit (PDU) and Cyclic Redundancy Check (CRC) [14]. Clearly, they are all symbols which means 1 bit per symbol.

Where the preamble and access address have special sequence format separately, and the detail is described in Table 2.2 [15]. We can see that the preamble depends on the access address so that both of them together can be used for distinguishing a data packet whether belongs to BLE. Besides, the capability of transferring symbols via data channel is larger than the advertising channel. For some applications such as the Bluetooth Beacon, it can broadcast information via the advertising channel, and the advantage is that the FH method is not to be necessary for it. Incidentally, there are iBeacon, Eddystones Beacon, and AltBeacon in real life normally.

Table 2.2 The preamble and access address format in advertising channel.

Preamble	‘10101010’ - if LSB of access address is 0
	‘01010101’ - if LSB of access address is 1
Access Address	‘0x8E89BED6’ in hex ‘1000 1110 1000 1001 1011 1110 1101 0110’ in binary

However, if we want to transfer a large data through beacon, it can be accepted the method of two-time sending for transmitting extension data. That is the first time through the advertising channel to broadcast a message which it will send a big data, when the receiver responded, the transmitter will do the second sending with big data via the data channel, but all of above perhaps can be accomplished in the 5.0 version of Bluetooth.

The data can be enveloped into radio transmissions through a lot of methods, such as the values of amplitude, phase, or frequency of a wave. For BLE, it’s a special scheme calls GFSK, which is refined from the Frequency Shift Keying (FSK) where information is formed by shifting the frequency according to the deviation, and the critical part is adopted a Gaussian filter. Therefore, the symbols are presenting as zero ‘0’ and one ‘1’ in binary, where ‘0’ is coded to negative frequency deviation, in contrast, the symbol of ‘1’ is coded to positive deviation. Finally, we will get the I/Q basebands and through the modulation scheme of BLE. So, the next is going to describe it in detail, including the demodulation as well.

2.2 Gaussian Frequency-Shift Keying

In general, there are two signal modes of transmission which are passband and baseband, the difference is that the passband transmission will add a carrier frequency basing on the I/Q basebands. For example, there is a waveform with carrier frequency f, and then associates with the deviation of basebands which implies ‘0’ and ‘1’, it will generate two new waveforms with frequencies f_1 and f_2 which also imply the symbol ‘0’ and ‘1’, the waveform is shown in Figure 2.4. This operation could be realized by the hardware called Software Define Radio (SDR). Hence, we should get the baseband signal firstly, which refers to the deviation of symbols in frequency.

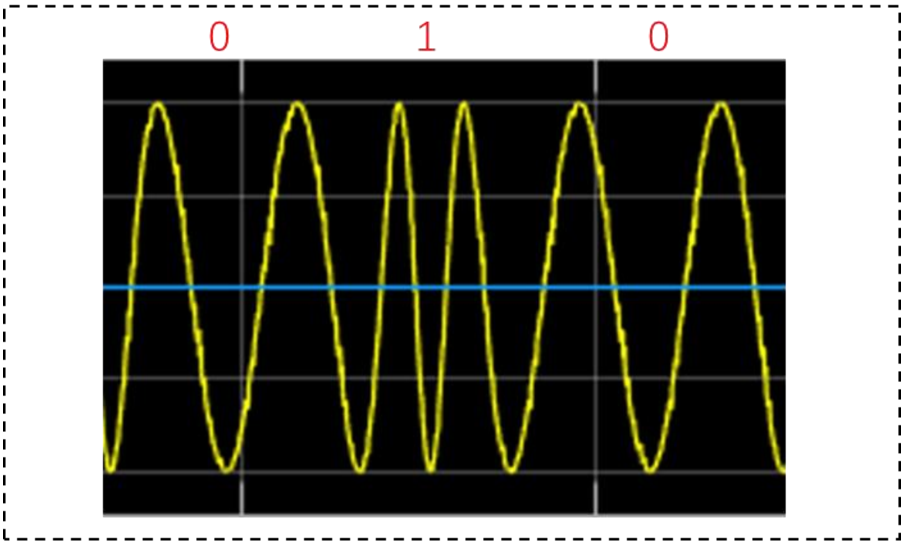


Figure 2.4 The example of passband represented

About the frequency modulation/demodulation, the FSK is widely used because it has better noise suppression in comparing with the others modes such as amplitude and phase, and the demodulation is less complicated for it. Therefore, GFSK is the extension of FSK which is applying a gaussian filter before the frequency modulation procedure. Actually, as one of low-pass filters, the gaussian filter can avoid the high frequencies because of the switching, so that it can make the signal spectral bandwidth to be narrow, which is able to decrease the adjacent channels' interference. Here, it will begin from the modulation to deeply explain the architecture and algorithm of GFSK.

2.2.1 Modulation

As mentioned earlier, the BLE will sent the symbols with '0' and '1' in binary, but it's necessary to envelop these into IQ phases instead of sending directly, because it can strength the capability of anti-interference when the system is working at relatively low data rate. The Figure 2.5 is showing the block diagram of the GFSK modulation, which mainly includes 6 steps.

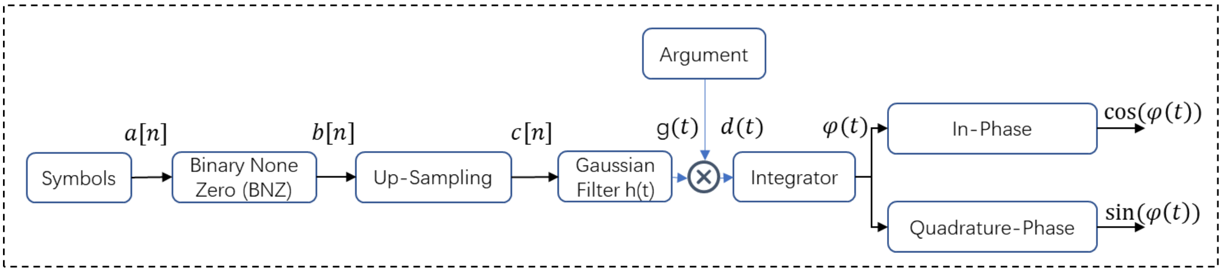


Figure 2.5 Architecture of GFSK modulation

2.2.1.1 Binary None Zero (BNZ)

In the BLE PHY, it supports 1M PHY and 2M PHY, where 1M PHY means the 1-megabit PHY, and the 2M PHY is 2-megabit PHY which introduced in Bluetooth 5.0 as an optional scheme. Normally, both of them refer to the bit rate which this PHY is capable of, and it's a trade-off scheme that the higher bit rate means potentially higher Bit Error Rate (BER).

Commonly, the BLE PHY will select the bit rate of 2 Megabit per second, and the symbol is the return-zero sequence, thus the first step is converting symbols to Non-Return Zero (NRZ) binary signal, where the new symbol '-1' instead of the original symbol '0', and symbol '+1' instead of '1'. The meaning of this data transformation is that it will be able to construct the deviation sequence based on each symbol afterward, and the algorithm as following the equation (2-1) [7]. We assume the original sequence as *a[n]*, the result as *b[n]* after processed, and the letter *k* is representing the index of symbol.

$$b(k) = 2 \times a(k) - 1 \tag{2-1}$$

2.2.1.2 Up-Sampling

Because BNZ sequence still is integer, and gaussian filter will transform these into float so that can realize the pulse shaping. However, it can't be computed for getting a correct consequence due to just a single symbol. In this case, we have to enlarge the single symbol so that can be processed with Gaussian method, that is the meaning of Up-sampling. Therefore, for this project, the output-data rate can reach 13 MHz if the input-data rate is 1 MHz because it must generate 13 samples of output per each input data period [1, 7], the reason will be indicated later, and the example we can see the Figure 2.6: the example of Up Sampling. So, the formula as showing in equation (2-2), which we assume the output sequence as *c[n]*.

$$c[n] = b\left[\left\lfloor \frac{n}{13} \right\rfloor\right] \tag{2-2}$$

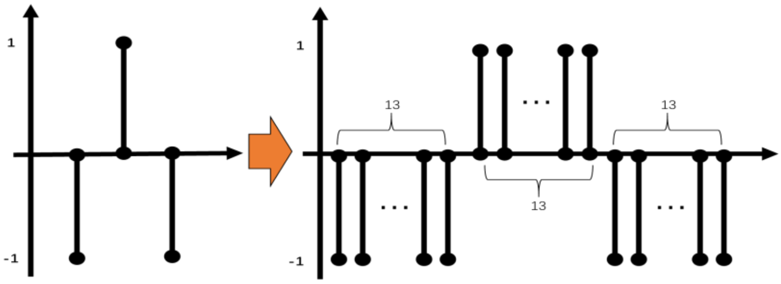


Figure 2.6 The example of Up Sampling

2.2.1.3 Gaussian Filter

As for the Gaussian filter, it's about the low-pass filter and mainly aims two goals [16]. Firstly, it can pre-filter the baseband waveform via smoothing the edge of input data, thus it can narrow the data spectrum of transmitting. Secondly, it can eliminate the glitches brought on by the data-level converter circuit.

To the beginning, we need to initiate the Gaussian pulse-shaping filter [17]. Then the transfer function of Gaussian low-pass filter which is shown in equation (2-3).

$$H(f) = e^{-(\alpha f)^2} \tag{2-3}$$

Where the argument of α is associating with B, which is -3 dB bandwidth of the baseband. And it is commonly expressed as a normalized product (BT_s) of -3 dB bandwidth and symbol time, T_s is representing the period, so the expression can be given by equation (2-4).

$$\alpha = \frac{\sqrt{ln\ (2)}}{\sqrt{2}} \frac{T_s}{BT_s} \tag{2-4}$$

With the α increases, the spectral occupancy of the Gaussian filter is reducing, and the impulse response spreads over adjacent symbols, then it leads to increase the Inter Symbol Interference (ISI) at the receiver. Therefore, in the continuous-time domain, the impulse response of Gaussian filter is represented in equation (2-5). In order to fit with standard deviation $\sigma_h = \alpha/(\sqrt{2}\pi)$, which is the canonical form of a zero-mean Gaussian random variable, And then rearranged it and got the equation (2-6).

$$h(t) = \frac{\sqrt{\pi}}{\alpha} e^{-(\frac{\pi}{\alpha}t)^2} \tag{2-5}$$

$$h(t) = \frac{1}{\sqrt{2\pi}(\frac{\alpha}{\sqrt{2}\pi})} e^{-\frac{t^2}{2\times(\frac{\alpha}{\sqrt{2}\pi})^2)}} \tag{2-6}$$

On the other hand, in the discrete-time domain, assume the Oversample Rate as OSR, then let $t_0 = T_s / OSR$ be an integer oversample of the symbol duration and $t = kt_0$. As mentioned before, k refers to the sample index. As a result, the expression of discrete-time impulse response is depicted in equation (2-7). And the equation (2-8) is describing the result that dropping the dependence on t_0 after invoked the equation (2-4) into (2-7).

$$h(kt_0) = \frac{\sqrt{\pi}}{\alpha} e^{-(\frac{\pi}{\alpha}kt_0)^2} \tag{2-7}$$

$$h[k] = \frac{\sqrt{2\pi}}{\sqrt{ln\ (2)}} \frac{BT_s}{T_s} e^{-(\frac{\sqrt{2\pi}}{\sqrt{ln\ (2)}} BT_s \frac{k}{OSR})^2} \tag{2-8}$$

Moreover, the peak of the impulse frequency response can explicit as equation (2-9) according to the first argument of equation (2-8).

$$h_{max} = \frac{\sqrt{\pi}}{\alpha} = \frac{\sqrt{2\pi}}{\sqrt{ln\ (2)}} \frac{BT_s}{T_s} \tag{2-9}$$

For Bluetooth, when assume $BT_s = 0.5$, $T_s = 1us$, and $OSR = 8$, we can get the value of h_{max} = 1.5054 MHz, and the case of $BT_s = 0.5$ is also called Gaussian Minimum Shift Keying (GMSK) [19, 20], which is a special scheme of GFSK.

Besides, the Gaussian filter is a 39 tabs FIR filter with the 39 coefficients generated by using the Marcum's function [1, 18], that is the length of sample. However, here we accept 28 tabs because it's more appropriate in this project than 39 tabs according to the Up-Sampling. The 28 tabs are stored as a Look-Up Table (LUT), and the shape is shown in Figure 2.7, which after the calculation of above formulas.

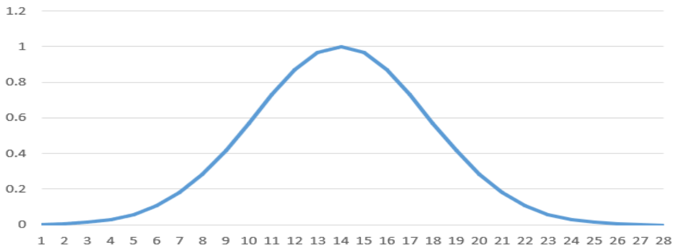


Figure 2.7 The shape of the Gaussian Filter initialization

When the initialization of Gaussian Filter is accomplished, the next is to do the convolution between the initial value of Gaussian filter and the input data, finally output the signal with time domain. The expression as showing equation (2-10) [20], where '$*$' refers to the convolution operator. However, in order to describe it clearly, and facilitate to realize, we can transform the expression from time domain to discrete domain which is resulted as the equation (2-11).

$$g(t) = c(t) * h(t) \tag{2-10}$$

$$g[k] = \sum_{i=-13}^{13} c[k+i] \cdot g[i] \tag{2-11}$$

2.2.1.4 Argument

This part is mainly adjusting some arguments between gaussian filter and integrator, and the constant value was determined as ((*Gaussian Length* / 5) $\times \pi$) finally, where the *Gaussian length* is depended on the tabs of Gaussian filter we accepted.

2.2.1.5 Integrator

After the signal go through the gaussian filter, the output indicates the instantaneous frequency deviation $\Delta f(n)$, and then it will be integrated as the instantaneous phase shift $\Delta\varphi(n)$ by using an Infinite Impulse Response (IIR) digital integrator, which has a transfer function as equation (2-12) [1].

$$H(z) = \frac{1}{1 - z^{-1}} \tag{2-12}$$

For GFSK modulation, we can get the equation (2-13) of the relationship between input and output according to the transfer function, which is in discrete domain. And the flow graph is shown in Figure 2.8.

$$\varphi[k] = \varphi[k-1] + d[k] \tag{2-13}$$

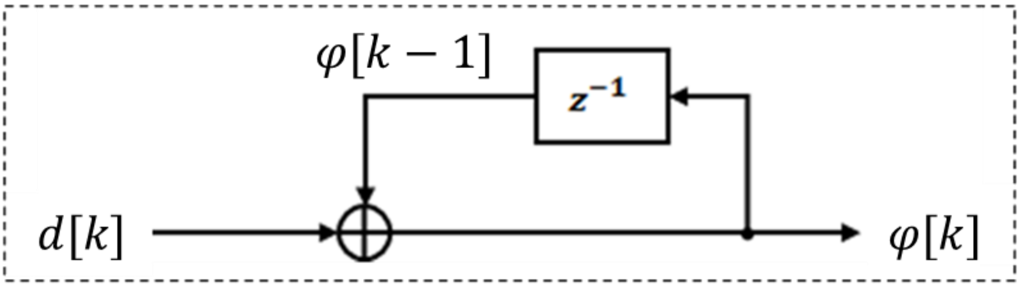


Figure 2.8 Implementation of the integrator in discrete domain

2.2.1.6 I/Q Basebands

Now, we have got the instantaneous phase deviation, and then call it to the sine and cosine wave generator respectively, hence to get the baseband signals of I/Q.

Moreover, the extensional step is that the I/Q signals can be associated with a carrier frequency separately, thus to get the passband signals, and sent it over air via radio wave. However, the passbands created will be dealt well with the hardware such as the radio frequency transmitter. Figure 2.9 is showing the architecture of passband transforming, where f_c is a carrier frequency. And the computing procedure is given by the equation (2-14), so the finally signal will send out by transmitter that is described in equation (2-15). Of course, before sending, there maybe also go through a filter which depends on the requirements.

$$S_{GFSK}(t) = cos\big(\varphi(t)\big)\,cos(w_c t) - sin\big(\varphi(t)\big)\,sin(w_c t) \tag{2-14}$$

$$S_{GFSK}(t) = cos\big(w_c t + \varphi(t)\big) \tag{2-15}$$

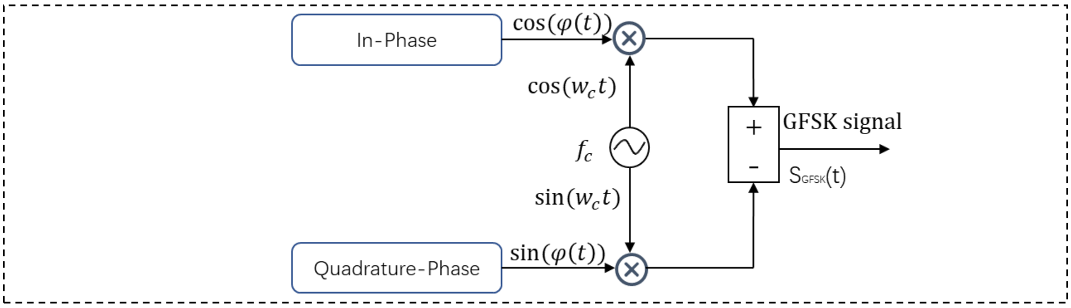


Figure 2.9 Architecture of GFSK modulation passband

2.2.2 Demodulation

Now we have seen what information format transmitted after the modulation in DSP, that is the waveforms which associated with the cosine and sine, and also called the In-phase and quadrature-phase separately. Both of these waveforms are basing on the phase deviation which computed by integrator block. However, all of above steps are the scope of transmitter, and also about the construction procedure of baseband signal in DSP by GFSK modulation, hence, the hardware of radio frequency transmitter can send the passband signal according to the baseband signal.

In the further, when radio frequency receiver gets the passband signal of GFSK modulation from the air, the first thing is compositing the waveforms of corresponding carrier frequency respectively, for recovering the signal from the passband to the baseband. And then go through a low-pass filter to smooth the waveform, the processing flow of this part is described in Figure 2.10. However, all of above can process in the hardware, for this thesis, we just need to discuss the later part that has been got the smooth baseband waveforms. Thus, about the architecture of GFSK demodulation is illustrated in the Figure 2.11.

We can see that there are 5 computing blocks which comprises of Arctangent, Unwrap, Derivative, Coder and Down-sampling, these seem to be a little simple than the block steps of modulation.

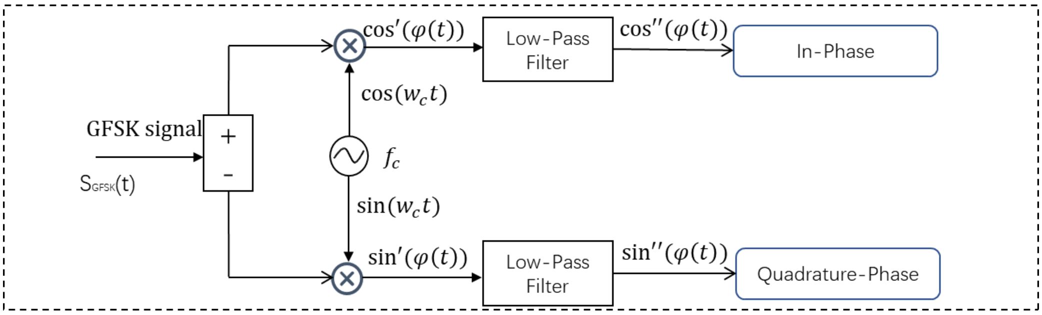


Figure 2.10 Architecture of GFSK modulation passband

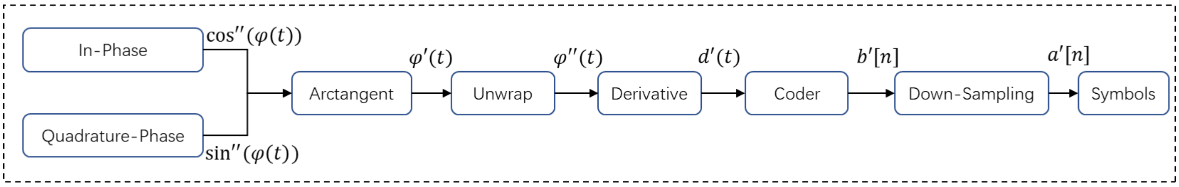


Figure 2.11 Architecture of GFSK demodulation

2.2.2.1 Arctangent

The first step of demodulation in DSP we need to do the computation between the sine and cosine by the function arctangent. The expression is given by the equation (2-16).

$$\varphi'(t) = arctan\,(\frac{sin''(\varphi(t))}{cos''(\varphi(t))}) \tag{2-16}$$

Whereas, because of the limitation of arctangent function, the values are limited in $[-\pi, \pi]$, or $[-\frac{\pi}{2}, \frac{\pi}{2}]$, which depend on the arctangent function in the real program, anyway, the next step called unwrapping is necessary.

2.2.2.2 Unwrap

The meaning of unwrapping is releasing the limitation because of arctangent, and then the results is corresponding with the output of integrator in modulation. We can realize it by judging several conditions we set when we do the program, and the detail will be indicated later.

2.2.2.3 Derivative

As mentioned early, the integrator is responsible for transforming the deviation from the frequency aspect to the phase aspect in the modulation. So, it’s clearly for this part that is recovering the frequency deviation from the phase aspect. we can express the procedure of derivative in discrete domain as the equation (2-17).

$$d'[k] = \varphi''[k] - \varphi''[k-1] \tag{2-17}$$

2.2.2.4 Coder

We've got the frequency deviations, then, in this part, is coding the values of frequency deviation to the symbols in binary. We can set two conditions to judge these and output the result, the detail will be described in later chapter.

2.2.2.5 Down-Sampling

As so far, the information is still in the up-sampling range, and that is not our expectation of demodulation. Besides, we knew the BLE packet has a relative fixed pattern as Figure 2.3, particularly the preamble and access address in Table 2.2, they are able to realize the signal synchronization between the transmitter and receiver of BLE in time domain. In the case of the down sampling, the better scheme is to sample the symbols from the preamble of a BLE data packet, because it can make the processing of down sampling to be more effective. However, here we firstly discuss how to accomplish the process of down sampling, as for the preamble and the access address detecting will be realized in the further work. In this case, assuming we got an exact BLE data packet which has went through the preamble detected, and in order to recover it to the original sequence, we need the down-sampling processing, which the basic concept is sampling one symbol per 13 symbols. On the other hand, during this procedure, we could select the mean value within a sequence of a serial of sequential and same symbols. This method can reduce the possibility of the bit error sampled, because the symbols would be more stable which is around the middle of this sequence, in contrast, it's easily occurring some bits error in the near of the both sides.

2.3 FPGA

During the parts of above, we've relatively understood the primary structure of BLE PHY and the principles of GFSK modulation/demodulation, then, according to these concepts we will realize all the function blocks of DSP in the FPGA platform. Before the practice, we need to learn the basic knowledge about the FPGA as well.

The FPGA belongs to the programmable logic device, and as a type of PLD, compares with the other class PLD such as CPLD and SPLD, they all are programmed by using HDLs such as Verilog HDL or VHDL, but in the aspect of logic gate arrays, a CPLD consists of a few thousand logic gates, whereas a FPGA can reach million level. Which means FPGAs can deal well with more complex computing or threads processing than CPLDs and SPLDs. The higher complexity will be increasing the cost, therefore, according to the design of requirement, CPLDs are more appropriate for the less complex applications, and the FPGAs are in contrast.

Besides, for the complex application, there are CPUs and GPUs as well, why we select the FPGA as a realization platform? Among the several advantages of FPGA, such as flexibility, energy efficiency, custom instructions, rich I/O and so forth, the most we heed to is the energy efficiency because of the characteristic low energy of BLE. In CMOS era, there is a formula for trading off the performance of energy efficiency that is indicated in equation (2-18) [21].

$$G = \frac{OPC \times f_{clk}}{P} \tag{2-18}$$

Where the *OPC* means Operations Per Cycle, f_{clk} is the operational frequency, and P is the power. In this case, at a same cycle, we can implement many parallel blocks to increase *OPC*, thus improve the performance of energy efficiency. Besides, about the *P*, in FPGA, it can communicate data through register or (small) on-chip memories in a same chip between the different devices so that it could reduce *P*. In contrast, for CPUs and GPUs, they tend to use external Dynamic Random Access Memory (DRAM), which has a higher power demand [22]. As a result, it’s better scheme to utilize the platform FPGA for this project.

There are a lot of vendors for the FPGA industrial files, nevertheless, among these various FPGAs, the basic structure, the design flow and the hardware algorithm are almost same, so this part primary indicates the concepts of FPGA through above aspects.

2.3.1 FPGA Architecture

Normally, programmable logic element, programmable I/O element, and programmable interconnect element are the three fundamental parts of an FPGA. A programmable I/O element can increase the peripheral devices where a programmable logic element indicates a logic function, and a programmable interconnect element can connect various blocks [5]. In the modern heterogeneous FPGA structure, there are also embedded memories such as Random-Access Memories (RAMs), DSP units and other hard blocks, which can improve the performance of computing capability. All of these blocks are interconnected by using bit-level programmable routing [5], that is the programmable interconnect element. Figure 2.12 shows the structure of an island-style FPGA [23, 24, 25]. Among academic and industrial FPGAs, this architecture is the one that is most frequently used. Since the logic blocks in this form resemble islands in a sea of programmable routing connection elements, it is obvious why this structure is called island style.

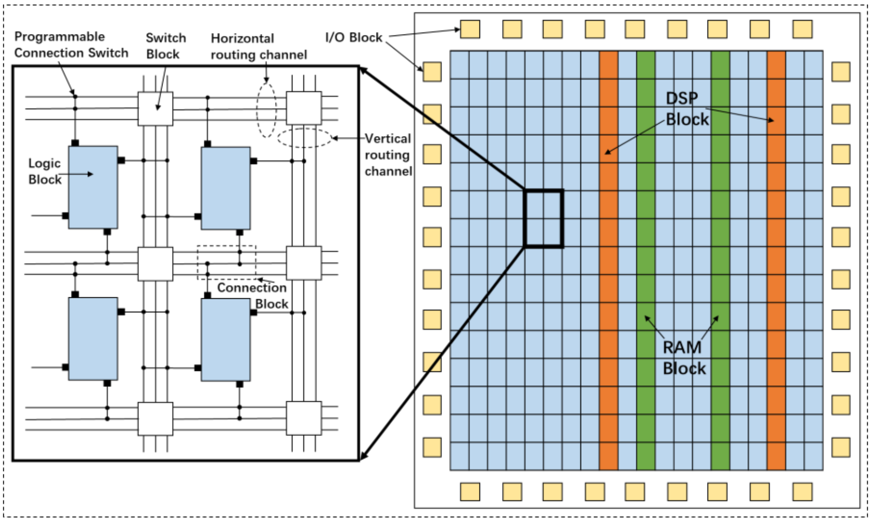


Figure 2.12 The island-style structure of FPGA

In this architecture, the programmable logic element is corresponding to the logic block. However, logic block is named differently among various FPGA venders. For example, it’s called Configurable Logic Block (CLB) in Xilinx FPGA, otherwise it’s Logic Array Block (LAB) in Altera [5]. The connection block contains the programmable connection switch, the vertical routing channel, and the horizontal routing channel, while the switch block contains the programmable interconnect element. Finally, the I/O blocks are connected to the programmable I/O element.

2.3.2 FPGA Design Flow and Design tools

The traditional design flow of FPGA is based on the HDL such as the Verilog-HDL and VHDL, it’s universal and typical method for the most FPGA development companies. In other word, no matter what HDL method we selected, take consider of the given specifications and constraints, then describe source codes, draw the circuit diagrams, and set the parameters. The Figure 2.13 is indicating the traditional method for FPGA design in HDL [5, 25].

So, obviously, in the traditional method with HDL, the first thing is describing the source code in Register Transfer Level (RTL) description, then go through the logic synthesis, technology mapping, place and route, generating a bitstream and configuration data, finally, programmed and executed on an FPGA. Besides of design procedure, there are also the verification in order to check whether the design corresponding to the requirements or constrains, which roughly include the RTL verification, timing analysing, and prototyping.

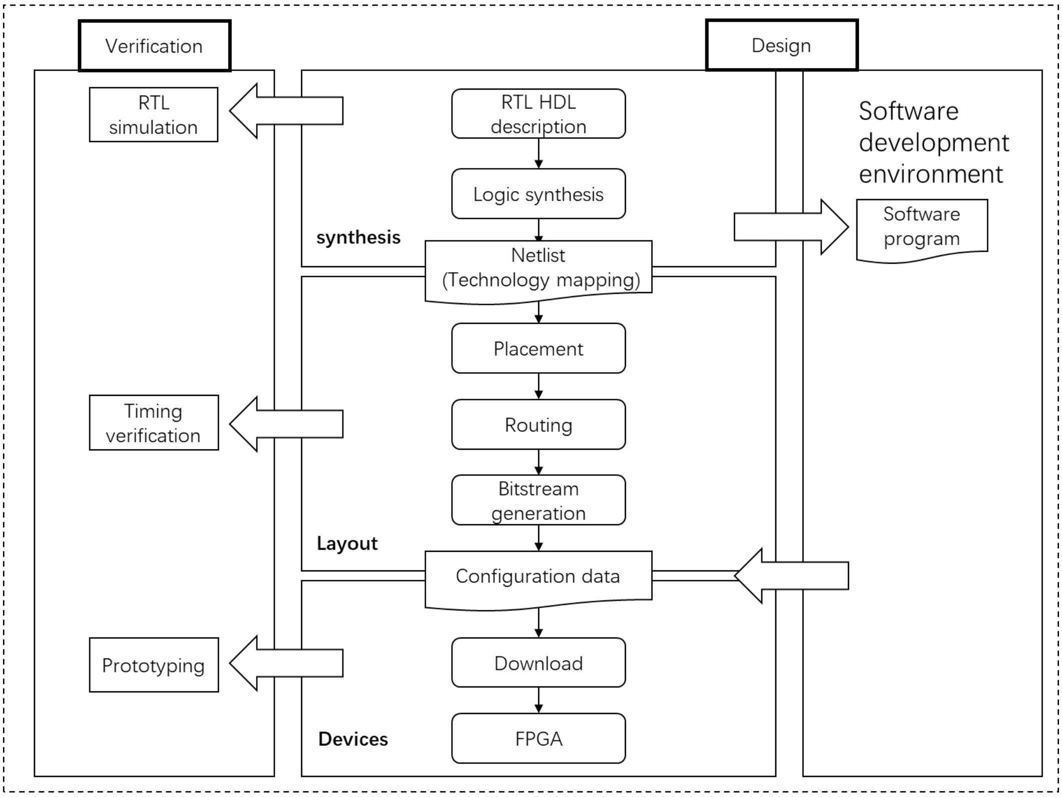


Figure 2.13 Design flow of FPGA in HDL

2.3.3 Hardware Algorithms

To achieve a high performance, it's not enough just to complete with hardware, especially, for the computing capability of FPGA, it's determined by utilizing efficient hardware algorithm for the target application. There are two common and useful schemes of hardware algorithms, that is pipelining and parallel processing.

2.3.3.1 Pipelining

The pipeline is a method for streamlining numerous processing iterations, and the processing principle is presented as the Figure 2.14 [5]. In the non-pipelined situation of (1), the processing *iteration 2* is done sequentially after the accomplishment of *iteration 1*. In contrast, the (2) is illustrating the pipeline processing, where we separate a unit processing iteration into n stages with uniform proportion. In this case, here $n = 5$, and the second processing iteration will start with done the first stage of the first iteration, the third will start with done the first stage of the second processing iteration, and so on. Finally, it can be accomplished 6 processing iterations with pipelining method during the time of 2 processing iterations in the non-pipeline situation. Obviously, the speed of calculation in FPGA is skyrocketing through the pipeline scheme.

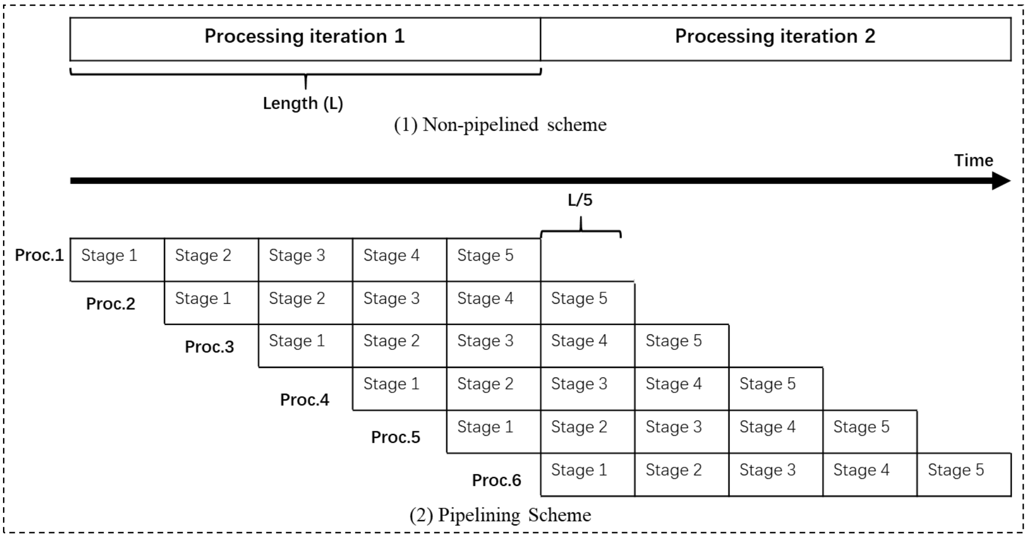


Figure 2.14 The comparison of schemes between (1) non-pipelined and (2) pipelining

2.3.3.2 Parallel Processing

The difference between the parallel processing and pipeline scheme is that the principle. For the pipelining case in Figure 2.14, it’s considered as parallel processing of five processing iteration when the 5th stage of processing *iteration 1* is started. However, the processing structure of parallelism mainly have three schemes, which are Single Instruction Stream Multiple Data Stream (SIMD), Multiple Instruction Stream Single Data Stream (MISD), Multiple Instruction Stream Multiple Data Stream (MIMD). The corresponding architectures of above are presented in the Figure 2.15. In addition, the structure of Single Instruction Stream Single Data Stream (SISD) is just displaying the basic components what it’s comprised of, it does not have the behaviours of parallel processing. Moreover, there all are divided into two parts which are control unit and processing unit. For the SIMD, it’s an architecture making realize the data parallelism, and is applicable the situation of processing numerous data synchronously with a single sequence of instructions. Nevertheless, in the MISD, each CU provides different function for the PU even if they are sequential executed, so that it’s also called functional parallelism. Compare with the SIMD and MISD, the MIMD is combination of the both different characteristics that has the performances of data parallelism and functional parallelism.

Besides, there are others schemes of hardware algorithms for increasing the computing or processing capability as well, such as systolic algorithm, stream processing and so forth. Here would not to extend these details, because the methods of pipeline and parallel processing are more typical and pervasive among all the hardware algorithms, specially, for this thesis we will mostly apply the parallel architecture.

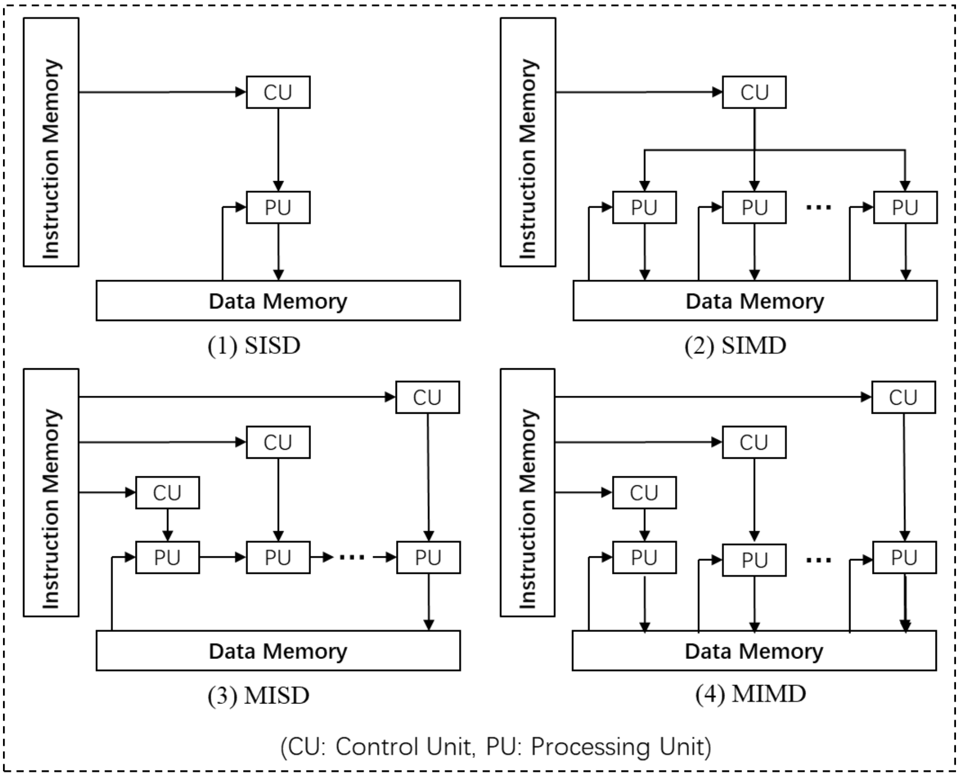


Figure 2.15 The structures of parallel processing

2.3.4 The Platform of Intel DevCloud

As so far, we have introduced some primary basic knowledge about FPGA and OpenCL, it's clear that we need to combinate the hardware and software to realize the object. On the one hand, when we are just developing about the program by OpenCL, we can build the hardware via the FPGA development board to verify the program, but it would cost a lot of money for it. Besides, in order to verify and debug the program, we perhaps need to follow again and over again the rest traditional design method except the construction of RTL description in HDL. On the other hand, if we just want to verify and debug the program, for a deep-learning programmer, it's better that not need to construct any hardware in real and then complete them, so that some remote platforms of FPGA were appearing these years.

The main FPGA vendor such as Xilinx, Altera, have been launched its cloud platform respectively. For the Xilinx, it was launched the relevant remote devices in the Amazon Web Services (AWS), which cooperating with the enterprise Amazon. In the AWS, the Amazon Elastic Compute Cloud (Amazon EC2) F1 instances features FPGAs of Xilinx that can be programmed to build hardware accelerations for applications. Additionally, for the Altera, the correlative devices have launched in the cloud platform called Intel DevCloud. This platform has several series of FPGAs supporting the OpenCL, and perhaps it’s benefited from the mother enterprise Intel, there are not only FPGAs, but also others computing devices such as CPUs, GPUs, and others accelerators. Besides, as mentioned before, the DevCloud supports the OpenCL development for FPGAs. Thus, as a result, we selected the latter platform as a primary scheme to verify and debug our program in OpenCL, and the environment configuring we can check in [33, 34, 35].

2.4 OpenCL

After via the learning with basic architecture of FPGA, the typical design flow in HDL, and some hardware algorithms of thread processing, we would have roughly understood the correlative knowledge in traditional design of FPGA from hardware to the software. On the other hand, along with the complexity of system design increasing, the capabilities of computing, productivity, efficiency and so forth, which are taken more and more attentions by developers and vendors nowadays. Additionally, it can reach higher performance by using manual optimization in the traditional design methods such as Verilog-HDL or VHDL, but this method would be time consuming and might produce human errors. In these cases, a novel method calls High-Level Synthesis (HLS) or behavioural synthesis was appeared [4, 26].

An algorithm is used as input in the automated design process known as HLS to generate the digital hardware necessary to carry out the required function. [27]. In general, A high-level programming language is used to create the control algorithms, such as C/C++ or variants (System C, OpenCL framework, etc.), additionally, the automated tool offers a description of the RTL hardware. For the most programmer whom is used to high-level language, to design a whole regular DSP will be felt a little bit complex by using the HDL method. In addition, the high-level language is a well-known tool. Therefore, the HLS method gradually become an alternative scheme to implement an DSP design in FPGA.

Normally, a C/C++ code just associates with software program, this might result in the synthesized hardware module performing poorly. The worst scenario is also that it might not succeed in being synthesized into hardware. Considering the hardware produced by HLS, we need carefully characterize the source code. In general, we can use variables, operators, substitutions, control statements (if, for, while, etc.), and function calls in C/C++ code to signal the target's behaviour. In this instance, an HLS situation can make an array into a memory, a function into a hardware module, and a variable into a register, all of which are connected to HDL. Additionally, a state machine would be implemented for the control, which consists of sequential executions, branches, loops, and function cells. While the majority of HLS tools have some common limitations, such as disabling recursive calls and disabling dynamic pointer [5]. Where the both of restrictions are because that would correspond to the dynamically instantiated at run time in hardware module, that is exceeding the scope of concept within present digital circuits. As a whole, a source code would be created using a high-level language like C or C++, and from there, an RTL description would be produced using HLS or behaviour synthesis, finally, the rest steps will almost follow the traditional method which starts from RTL description as mentioned proceeding.

Bases on above thoughts, for this project we invoke OpenCL to construct a source code. For the portability in OpenCL, means that we can build by C/C++, Java, Python, and then can apply the program for the FPGAs, CPUs, GPUs, thus it resembles a common interface for the most popular program languages and the target devices. Besides, the parallel programming which can obviously increase the computing capability as explained in the section 2.3.3 hardware algorithm before. In this case, the next will introduce the OpenCL via the aspects of itself architecture, primary elements in detail.

2.4.1 Structure of OpenCL

For the parallel programming, there are two parts we need to distinguish that is host application and the kernel. In other word, where the host application is associating with the control unit, and the kernel corresponds to the processing unit. So, the source code will be normally divided into two parts, host code and kernel code.

2.4.2 Host code

The good news is that we can use high-level language to generate source code in the host code section, the bad news is that we need to comprehend the six peculiar elements, which composes platform, device, context, program, kernel, command queue. Except the platform, the others' relationship can be described in the graph as Figure 2.16 [6].

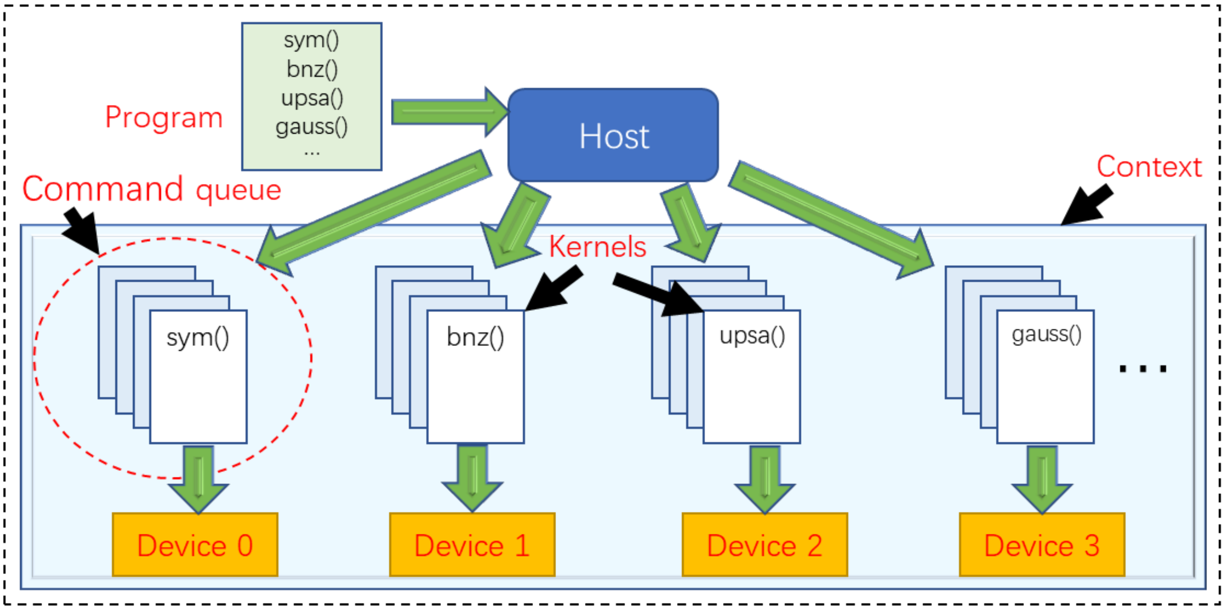


Figure 2.16 The relationship of the main elements in host code

We assume there are several functions, `sym()`, `bnz()`, `upsa()`, `gauss()` and so forth, each function can be explained by a kernel independently, thus the kernel is container of functions. Furthermore, all of these functions are contained in the program, so we can say that program is the container of kernels. On the other hand, the devices, command queues are included in the context, thus the context is responsible for identifying a set of devices, and making it possible to create command queues. Where for the former, it can receive task or function from the host. And when identify by context, it not means all possible devices, but only the devices which are selected to work together. However, for the latter, the command queues can transmission information between host code and kernel code, which is similar as a bridge between the both. While the program is configured in host code, at the same time all the necessary kernels are set as well. On the other hand, we define the command queues and in order to generate it in the context. Therefore, when the host code is executed, the context identifies a serial of devices, then the host sends the task or functions that as kernels to the devices through the command queues within the context according the program configuration.

2.4.2.1 Platform

It’s a little pity there is no any description of platform in the relationship of the main elements in host code as showing in Figure 2.16, but here will compensate this part as detail as possible.

Because OpenCL can apply for the different platforms, such as FPGA, CPU, GPU, and each platform have many vendors and different classes. In this case, we need to distinguish what platform the code will be applied for. For example, it's one situation we have known the information of platform so we can directly configure it to the host code. But if we have not any information about the platform yet, such as the seller, who want to sell the OpenCL application. Thus, in order to make it possible no matter what situations, the data structure of `cl_platform_id` by providing from OpenCL specification guide can deal well with it [6, 28, 29].

There are usually three steps to finish a platform configuration in code, that is firstly allocating memory for one or more `cl_platform_id` structures, and then utilizing the function called `clGetPlatformIDs()` to initialize these structures. The entire arguments are shown below.

```
cl_int  clGetPlatformIDs (cl_unit         num_entries,
                          cl_platform_id* platform,
                          cl_unit*        num_platform)
                          num_platform)
```

Finally, if we want to discover the version of OpenCL a platform supports or who made it, we can use the second function called `clGetPlatformInfo()`, and the overall arguments are given below.

```
cl_int  clGetPlatformInfo (cl_platform_id      platform,
                           cl_platform_info    para_name,
                           size_t              param_value_size,
                           void*               param_value,
                           size_t *            param_value_size_ret)
```

Where the second argument is `para_name` can be used one of the values within the Table 2.3, besides, all of them return the char-type array. However, in the version 3.0 has added some new parameters, such as *CL_PLATFORM_NUMERIC_VERSION* and so forth [30], here will not to extend the detail. And the third parameter is *param_value_size* which indicates how many bytes we want to store.

Table 2.3 OpenCL Platform Information Parameters

Parament name	**Description**
CL_PLATFORM_NAME	Return the platform-corresponding name
CL_PLATFORM_VENDOR	Return the vendor that matches the platform.
CL_PLATFORM_VERSION	Return the highest OpenCL version that the platform will support.
CL_PLATFORM_PROFILE	Identify if the platform supports the complete OpenCL standard (FULL_PROFILE) or the embedded standard (EMBEDDED_PROFILE)
CL_PLATFORM_EXTENSIONS	Return a list of the platform's supported extensions.

2.4.2.2 Device

About the others part we have roughly understood during the preceding explanation, so the next is about how to realize these in code. Now we start from the device function. Similarly, devices are designated by the data structure `cl_device_id` in code. Comparing with the platform configuration, it also will experience two steps in the device aspect, which corresponds two functions we will utilize.

The first function is about creating device structures and that is the `clGetDeviceIDs()`. It populating a `cl_device_id` array with structure corresponding to OpenCL devices, and the whole arguments in this function are represented as below.

```
cl_int clGetDeviceIDs (cl_platform_id     platform,
                       cl_device_type     device_type,
                       cl_uint            num_entries,
                       cl_device_id*      devices,
                       cl_uint *          num_devices)
```

Where the first argument indicates the `cl_platform_id` structure representing the platform we want. The second indicates a device type, which can be set by using the values which is given by the Table 2.4.

Table 2.4 OpenCL Device Types

Device type	**Description**
CL_DEVICE_TYPE_ALL	Means every piece of device could connect to the platform
CL_DEVICE_TYPE_CUSTOM	Specialized devices that don't support all required OpenCL functionality
CL_DEVICE_TYPE_DEFAULT	Means that they are not special devices but rather those connected to the platform's default kind.
CL_DEVICE_TYPE_CPU	Means the host processor
CL_DEVICE_TYPE_GPU	Means a device containing a graphics processor unit (GPU)
CL_DEVICE_TYPE_ACCELERATOR	Means an external device used to accelerate computation

As for the second function, it can access devices and obtain device information, that is `clGetDeviceInfo()`, and the full arguments are represented as below.

```
cl_int clGetDeviceInfo (cl_device_id          device,
                        cl_device_info        param_name,
                        size_t                param_value_size,
                        void*                 param_value,
                        size_t*               param_value_size_ret)
```

Obviously, these arguments are very similar with platform's, and the difference is transforming the first two data structure to the `cl_device_id` and `cl_device_info` respectively.

2.4.2.3 Context

Meanwhile, the data structure of `cl_context` is indicating the OpenCL context, we can utilize the functions of `clCreateContext()` or `clCreateContextFromType()` to create it. Their whole arguments are shown below respectively.

```
cl_context clCreateContext (const cl_context_properties*   properties,
                            cl_uint                        num_devices,
                            const  cl_device_id*           device,
                            void (CL_CALLBACK*            notify_func) (…)
                            void*                          user_data,
                            cl_int*                        error)
```

```
cl_context clCreateContextFromType (const cl_context_properties*  properties,
                                    cl_device_type             device_type,
                                    void (CL_CALLBACK*     notify_func) (…)
                                    void*                   user_data,
                                    cl_int*                       error)
```

The first function will establish a context by explicitly identifying devices, which is the primary distinction between the two. The second function, in comparison, creates a context that includes the devices of a specific type which available within the Table 2.4. On the other hand, we also need to distinguish the `cl_context_properties` pointer and `void` pointer which are both included. The *properties* pointer must designate an array of names and values which is terminated with 0. However, the *void* pointer can point to any data we want. Besides, both arguments can be set as *NULL*. About the `callback` function as an argument within both functions, this may be used if an error happens while the context is in working.

Moreover, after create contexts by using the one of above functions, we could get context information through the function `clGetContextInfo`, which is similar with the function of `clGetPlatformInfo` and `clGetDeviceInfo` in the platform and devices separately, and here will not elaborate it.

2.4.2.4 Program

The devices can receive the tasks and functions from the host, and then the program is responsible for storing the tasks and functions what will be sent by the host. Thus, in a word, the program is a container of all functions. In OpenCL, the data structure of `cl_program` is representing a program. This section will be divided into three steps to explain the program how to be configurated in code, that is creating a program, building a program and obtaining the information of program.

For the first step, OpenCL provides two methods to create a new program, their functions are represented as `clCreateProgramWithSource()` and `clCreateProgramWithBinary()`. Both them can transform the code into a `cl_program`, but neither accepts filenames or file handles. Therefore, before calling one of the functions, we must read the content of the file into a buffer when the kernel code is contained in a file. However, there are some differences between the both functions, and the primary is the method of reading data. For the `clCreateProgramWithSource()`, it expects the buffers to hold text-based code, and all the arguments are given below.

```
clCreateProgramWithSource (cl_context          context,
                           cl_uint             src_num,
                           const  char**       src_strings,
                           const  size_t*      src_sizes,
                           cl_int*             err_code)
```

The function of `clCreateProgramWithBinary()` is similar with the previous, but it reads bytes from binary file rather than reading strings from text file. Then, all the arguments of itself are shown below.

```
clCreateProgramWithBinary (cl_context                  context,
                           cl_uint                     num_devices,
                           const cl_device_id*         devices,
                           const size_t*               bin_sizes,
                           const unsigned char**       bins,
                           cl_int*                     bin_status,
                           cl_int*                     err_code)
```

we need to note that in the third argument these devices must be contained within the `cl_context` which provided by the first argument.

As for the second step to build a program, we would use the function like `clBuildProgram()`. Although there are different types of compiler for different vendors, one crucial common provision is that every compiler must be accessible through `clBuilProgram()`. So, this function is responsible for compiling and linking a `cl_program` for devices associated with the platform. The full arguments are represented bellow.

```
clBuilProgram (cl_program          program,
               cl_uint             num_devices,
               const cl_device_id* devices,
               const char*         options,
               void (CL_CALLBACK*  notify_func) (…),
               void*               user_data)
```

Finally, by using the function of `clGetProgramInfo()` or `clGetProgramBuildInfo()` to access information related to program when it has created and compiled. The first function gives details about the program's data structures, including its context and target devices, so it's similar with the preceding functions of `clGetContextInfo()` and `clGetDeviceInfo()`. The second function includes instructions on how to create a program, which is crucial because it's the only method to learn what occurred during the creation of the program. As a result, the first is easier to test the return value of `clBuildProgram`, however, if we want to find out the reason when it built fail, we need to call the second function, that is `clGetProgramBuildInfo()`.

2.4.2.5 Kernel

As mentioned early, the kernel is a container of function, so this part is describing how to package the function in kernels. Each kernel can be represented by the data structure of `cl_kernel`, so, then the first thing is to create kernels by using the function `clCreateKernel()` or `clCreateKernelsInProgram()`.

For the former function it's to construct a single kernel, which requires the name known of function from which the kernel is to be create. So, the arguments what are contained in this function just like below.

```
clCreateKernel (cl_program          program,
                const char*         kernel_name,
                cl_int*             error)
```

If there are multiple kernels, we just need to duplicate the function of `clCreateKernel()`.

However, it's easier to using the latter function `clCreateKernelsInProgram()` to work on it, because it can produce a kernel for each function in the program. In this case, the entire arguments are represented below.

```
clCreateKernelsInProgram (cl_program      program,
                          cl_uint         num_kernels,
                          cl_kernel*      kernels,
                          cl_uint*        num_kernel_ret)
```

where the new `cl_kernels` are placed in the kernels array, and the *num_kernel_ret* identifies the number of available kernels. Via invoking this function twice we can determine the capacity of memory has been allocated, and after that, put the kernels in memory.

The second step is obtaining the kernel information and just like the step of program, context, and device. Therefore we can utilize the function `clGetKernelInfo()` to get the information about which function it represents and which program it belongs to.

2.4.2.6 Command queue

As same as the briefly explained at the beginning, the command queue resembles a bridge between the host and device. In this case, we don't need to identify a target device when we create a kernel, whereas within the procedure of creating command queue. That means when we dispatch kernels to the queue, they will be sent to the devices automatically according to the queues. Normally, besides the work mode of kernel execution which can dispatch the kernel to a command queue, there are also three modes for the data transmission between the host and devices, that is writing data from host to a device, reading data from a device to host, and copying data between devices. Among these data-transfer modes, only one direction can be realized at the same time, that means when the command moves from the host to device, the device couldn't send any command to the host.

According to the proceeding experience, we can easily derive the function of creating a command queue, which would start with the data structure `clEnqueue`, and that is representing the command queue. Therefore, there is the function called `clCreateCommandQueue()` which is only one way to create a new queue, the entire arguments are following below.

```
clCreateCommandQueue (cl_context                       context,
                      cl_device_id                     device,
                      cl_command_queue_properties      properties,
                      scl_int*                         error)
```

Where the third arguments must be selected one enumerated type from the Table 2.5.

Table 2.5 OpenCl command queue properties

Parameter	**Description**
CL_QUEUE_PROFILING_ENABLE	Enables event profiling
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE	Enables out-of-order execution of queue commands

When the queue executes a command, the former argument lets us know that we might get timing events. The latter parameter relates to how the device handles queue items. Generally, command queues follow the First-In, First-Out (FIFO) principle by default, which means the first kernel to be executed will be the one that is displayed to a command queue.

Comparing with the above subsections, in the command queue part there is not the step of obtaining information, instead, the second step is generating the enqueues of kernel execution commands. So, the function called `clEnqueueTask()` is utilized which sends a kernel execution command to a device via a command queue. The entire arguments are following below.

```
clEnqueueTask (cl_command_queue     queue,
               cl_kernel            kernel,
               cl_uint              num_events,
               const  cl_event*     wait_list,s
               cl_event*            event)
```

When we call this function, the device would carry out the kernel function while it processed the command, so we didn't need to invoke any additional routines to do so. In other word, the kernel execution command is sent to the command queue when we call this function.

As for now, the primary six elements of host code have been introduced, and there are others elements are contained in the host codes, such as the memory allocating, buffer objects, others command queues (buffer reading and buffer writing), the resource allocated releasing (kernel, program, context, queue) and so forth, these will be explained with real codes in the later chapters, so here not deploys these so detail any more. Therefore, we can clearly find that is for constructing a serial of conditions about the data transmission in host code, and the goal is to communicate with the kernel code. Obviously, about the computing part is described in the kernel code.

2.4.3 Kernel code

During the host code, we can utilize the standard C libraries to construct it, the difference is that it would be associated with some special expressions of OpenCL in some cases. However, within the kernel code, there are more constrains than the host code when we program in the standard C language, particularly some data structures. For instance, the global variable must be a pointer, and the constant value can’t be defined within a function. Meanwhile, some difference concepts and structures are applied for this part, which are given below.

- Each kernel declaration must begin with the string `__kernel`
- Every kernel function must return `void`.
- Some platforms won’t allow kernel compilation without arguments or attributions.

- It can enable some extensions such as channel, and so forth.

2.4.3.1 Architecture of kernel cost

Because the kernel machine will communicate with the host machine, it must comprise of reading data from and writing data to the host separately. Of course, the data processing or computing must be contained in here. So above three parts are the primary components in kernel code, and the detail of associating with this project would be discussed later. However, there are others crucial principles we would to have a basic understanding, that is extension concept such as the channel, the kernel attribution such as autorun, and the method of parallel program such as the pragma of unrolling. All of above will explain roughly here, because it will be explained more clearly when associating with the practice description.

2.4.3.2 Extension of channel

There are many extensions we can utilized for the OpenCL kernel code, such as some function about the math, geometric, image processing and so forth, we can reference the OpenCL extension specifications [31]. However, in the kernel code it has a normal format to invoke the extension what we want, that is

```
#pragma OPENCL EXTENSION extension_name: behaviour
#pragma OPENCL EXTENSION all: behaviour
```

Obviously, the former is invoking a specified extension and the latter can invoke all the extensions together. Where the `#pragma OPENCL EXTENSION` directly controls the behaviour of the OpenCL compiler with respect to extensions, and the behaviour contains *enable* and *disable*. In general, the *extension_name* can be described as the *cl_<vendor_name>_<name>*. In this case, for this project, here we would use the extension of channel, which can be described as below.

```
#pragma OPENCL EXTENSION cl_intel_channels: enable
```

Where “*intel*” is indicating a vendor, “*channel*” is a specified extension.

The basic concept of channel extension just likes that provides temporary memory or buffer for storing the data or information, so that kernel can read data from and write data to these temporary memories or buffers separately, and realized the data transmission between the kernels. In some cases, we need to process data through a serial of steps, and there will be several kernels to realize the computing part, therefore the extensional structure channel would be a simple method to deal well with the communication between kernels. In a word, it can synchronize kernels and transmit data between them with great efficiency and low latency.

2.4.3.3 Autorun attribution

Similarly, there are a lot of attributions we can define from the correlative specifications of OpenCL kernel [31], and the structure can be represented as `__attribution__((qualifier_name))`. For this project, we would utilize the autorun attribution, so the command line can be described as `__attribution__((autorun))`, which will be in the position before the kernel function.

The basic concept of autorun is that the kernel function will be launched before `main()` function beginning, this is an idea for realizing the parallel programming. In this case, within the function of kernel autorun, it will be used a `while` loop which has an infinite execution cycle.

2.4.3.4 Unrolling pragma

As mentioned at last section, In the autorun kernels, there will be a `while` loop. If there are some simple computing processes in while loop, such as addition, subtraction, multiplication, division and so forth, it doesn’t matter. But if it is about iteration such as `for` loop within the `while` loop in the autorun kernel, and we also want to make it parallel, then we maybe need to take a consider of the unrolling pragma, which is a crucial concept in the OpenCL kernel code. The command line can be represented as `#pragma unroll`, and in the position before the `for` loop. By the way, it’s worth to note that the unrolling pragma just for the *for* loop, can’t apply for `while` loop.

On the other hand, loop unrolling entails duplicating a loop body and lowering a loop's number of trips. In other words, it can decrease or completely do away with the FPGA's loop control overhead. Additionally, if there are no loop-carried dependencies and the offline compiler is able to run loop iterations in parallel, it can also minimize latency [32].

3 SIMULATION

Relying on the above concepts and methods, this chapter will explain some simulations in MATLAB Simulink. Here we would separate two primary parts to illustrate the GFSK modulation and demodulation.

3.1 GFSK modulation with Simulink

About the GFSK modulation with Simulink, there are mainly six steps, which includes generating symbols in binary, transforming the symbols to the non-return zero, going through gaussian filter, multiplying argument, integrating, and finally acquiring the IQ basebands by invoking the waveform shaper of cosine and sine respectively. Therefore, the block flow with Simulink is represented in Figure 3.1. Comparing with the architecture of GFSK modulation in Figure 2.5, we notice here the up-sampling block is missing. Actually, this function has been comprised within the Gaussian filter processing.

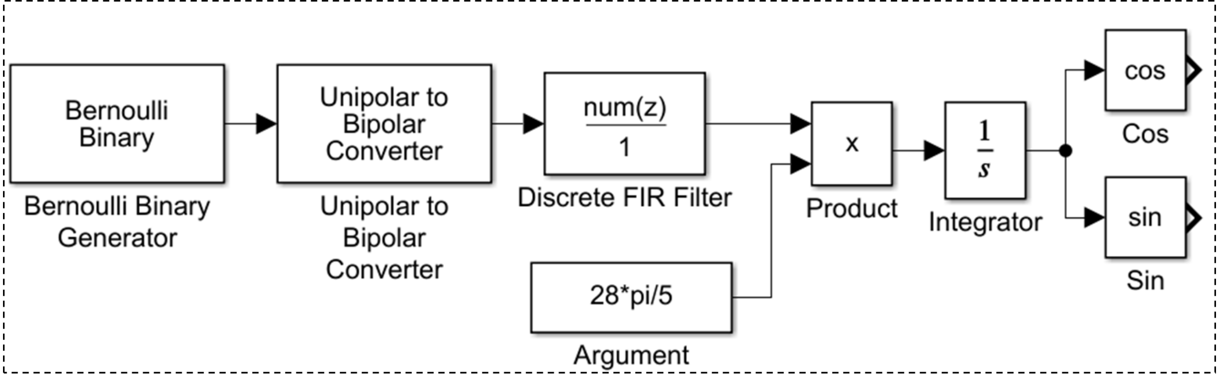
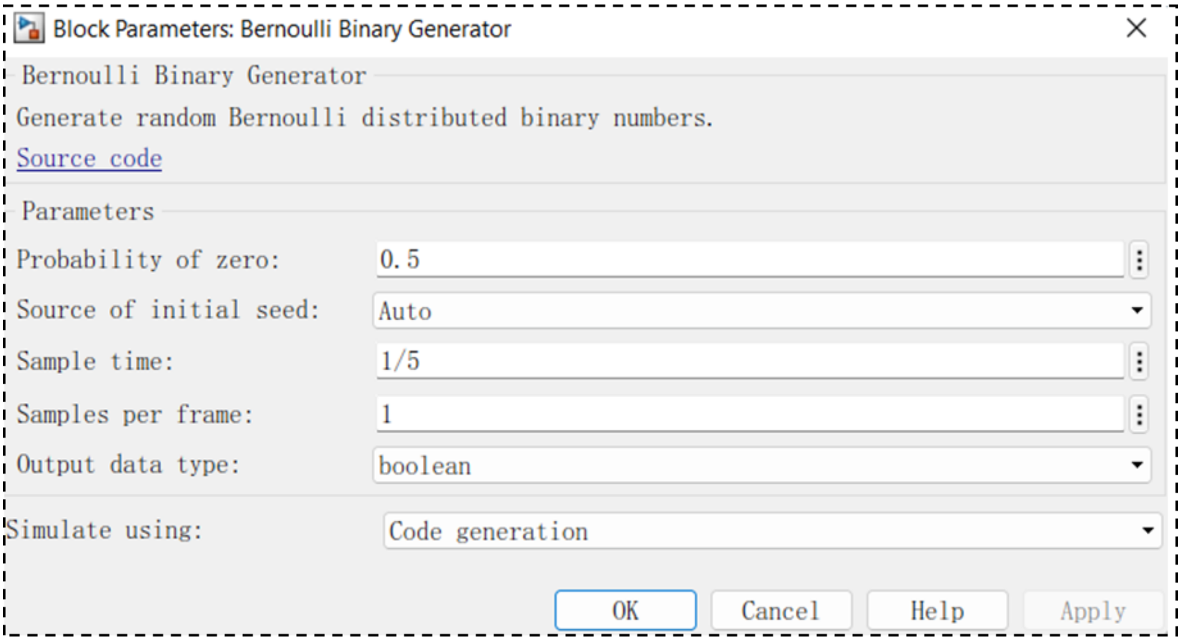


Figure 3.1 GFSK modulation in Simulink

3.1.1 Bernoulli Binary Generator

The first step is generating the symbols in binary, we can call the block **Bernoulli Binary Generator** in Simulink, which can generate random binary numbers using a Bernoulli distribution. The detail of parameter configurated is represented in Figure 3.2. Where the argument of *sample time* means how many symbols will be generated per second, here we want to graph it clearly so that setting this argument as 1/5, and it will generate 5 symbols per second, there will be 10 second shown in the analyser blocks, so total 50 symbols will be processing. The result is graphed in Figure 3.3, where one point refers one symbol.



Block Parameters: Bernoulli Binary Generator

Bernoulli Binary Generator

Generate random Bernoulli distributed binary numbers.

Source code

Parameters

Probability of zero: 0.5

Source of initial seed: Auto

Sample time: 1/5

Samples per frame: 1

Output data type: boolean

Simulate using: Code generation

Figure 3.2 Parameter configured of Bernoulli Binary Generator (Simulink)

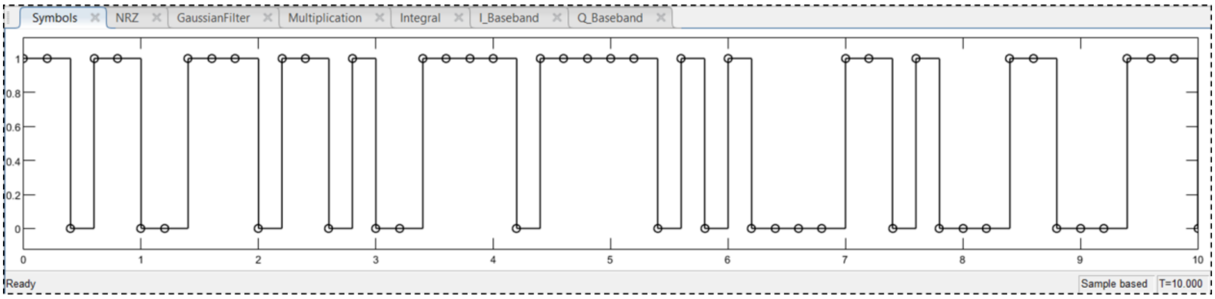


Figure 3.3 Output of Bernoulli Binary Generator (Simulink)

3.1.2 Unipolar to Bipolar Converter

This part is responsibility for transforming the symbols from binary to the sequence of NRZ, it’s easy understand and then we invoke the block called **Unipolar to Bipolar Converter**, which maps the unipolar input signal to bipolar output signal. The result is representing in Figure 3.4.

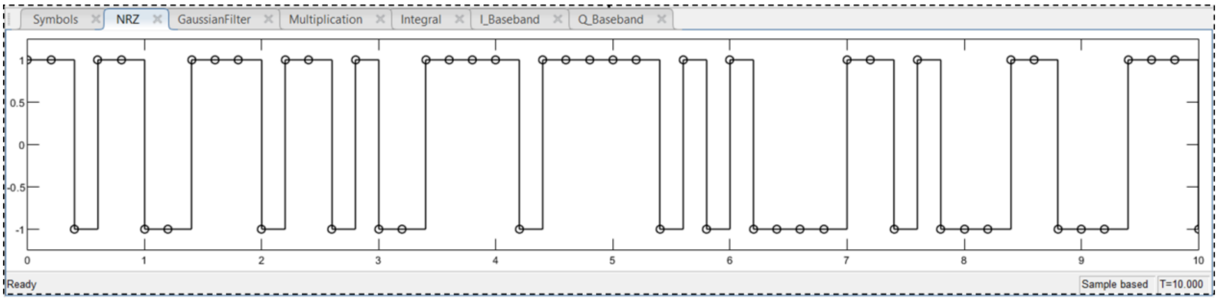
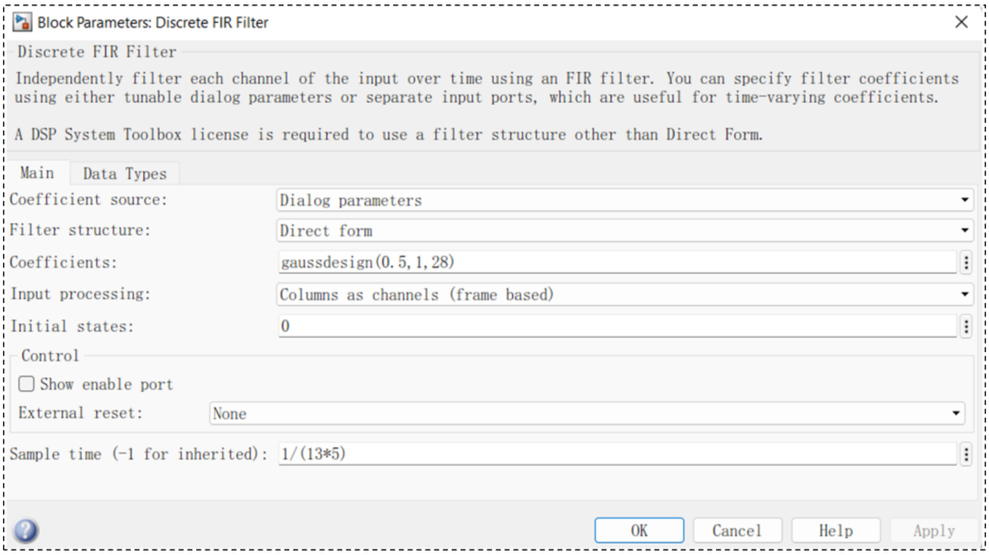


Figure 3.4 Output of Unipolar to Bipolar Converter (Simulink)

3.1.3 Gaussian filter in Simulink

How to construct a Gaussian Filter is a most crucial section in the GFSK modulation. In the Simulink, it has a readied block called **GMSK Modulation Baseband**, which can help us to get the baseband directly through setting several parameters, obviously, it is a shortcut to realize a communication system. However, for this project, one main objective is to understand how to construct a Gaussian Filter and others computing parts within the GFSK modulation, thus we turned down such shortcut temporarily. Because the Gaussian filter is one of low-pass filters, we can invoke a block called Finite Impulse Response (FIR) filter to build a low-pass filter. And then configure the parameters in FIR filter to realize a Gaussian filter, the detail is shown in the Figure 3.5.



Block Parameters: Discrete FIR Filter

Discrete FIR Filter

Independently filter each channel of the input over time using an FIR filter. You can specify filter coefficients using either tunable dialog parameters or separate input ports, which are useful for time-varying coefficients.

A DSP System Toolbox license is required to use a filter structure other than Direct Form.

Coefficient source: Dialog parameters

Filter structure: Direct form

Coefficients: gaussdesign(0.5,1,28)

Input processing: Columns as channels (frame based)

Initial states: 0

Control

☐ Show enable port

External reset: None

Sample time (-1 for inherited): 1/(13*5)

Figure 3.5 Parameter configured of FIR filter (Simulink)

Where there are two primary parameters that we have to set well in order to realize the function of Gaussian Filter. The first one is *coefficients*, we need to invoke a function called `gaussdesign(bt, span, sps)` [36], this equals the initialization of Gaussian filter. Among the three parameters, *bt* is corresponding to the BT_s which is the product of -3dB bandwidth and period as mentioned in equation (2-4). Besides, the parameter *span* indicates the number of symbols, and the *sps* refers to Sample per Symbol. Here the `gaussdesign (0.5,1,28)` means that BT_s=0.5, and 1 symbol was sampled 28 times. The number 28 is also associating with the argument of *Gaussian Length* as mentioned in 2.2.1.4, then the result is corresponding the Figure 2.7. On the other hand, the parameter *sample time* refers the up-sampling procedure. Because there were generating 5 symbols per second that we've set in the generator, here we need to multiply the number 5 within the denominator in addition. In these cases, we can obtain the result as Figure 3.6. We can see that points are increased, which implies it has been went through the up-sampling procedure successfully.

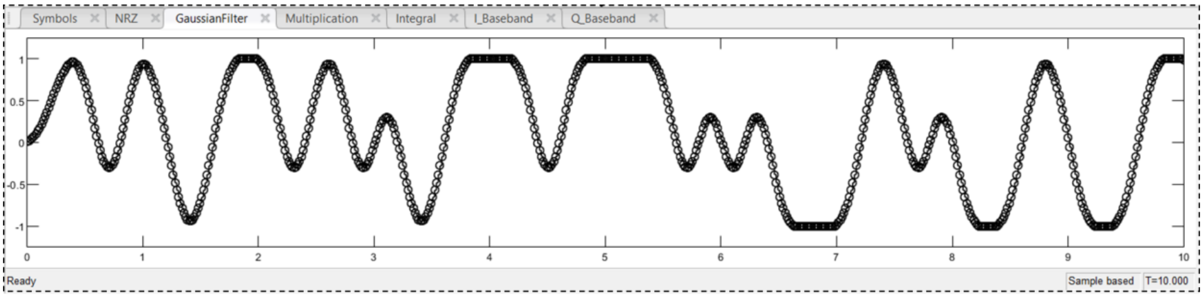


Figure 3.6 Output of gaussian filter (Simulink)

3.1.4 Argument Multiplying

This step just processes a multiplication with a constant *((Gaussian Length / 5)* $\times$ *π)*, in order to calibrate the later I/Q waveform. The result is illustrated in Figure 3.7.

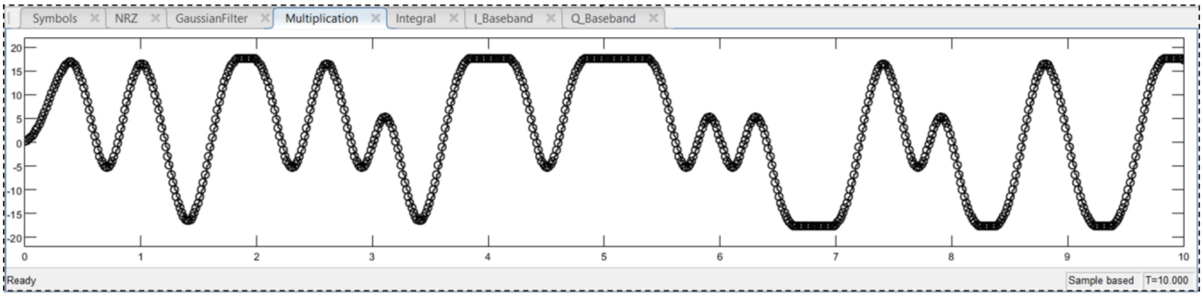


Figure 3.7 Output of argument multiplying (Simulink)

3.1.5 Integrator in Simulink

Because the output of Gaussian filter refers to a frequency deviation, and the I/Q basebands are associating with the phase deviation. In this case, we need an **Integrator** to transform the signal from frequency deviation to the phase deviation, then the output is displayed in Figure 3.8.

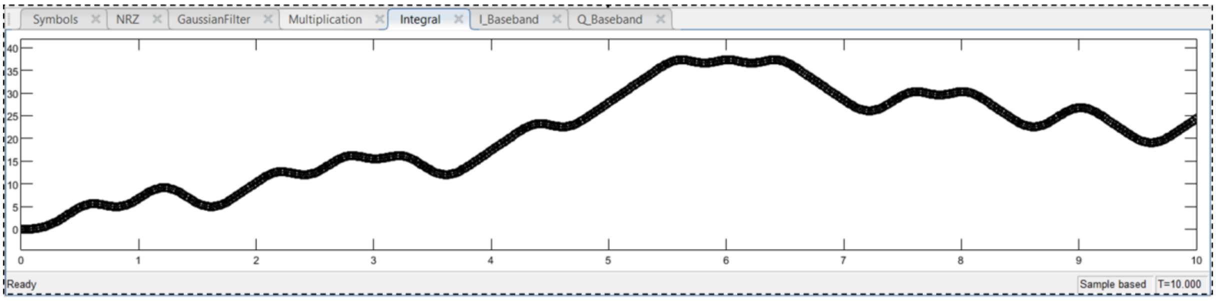


Figure 3.8 Output of integrator (Simulink)

3.1.6 I/Q Basebands in Simulink

After we got the phase deviations from the integrator, the next is sending the signal to the waveform generator of *cosine* and *sine*, finally obtain the I/Q basebands. The waveforms are shown in Figure 3.9 and Figure 3.10 respectively. We can see the difference of both waveforms is 90-degrees phase, and that is the meaning of I/Q modulation as well.

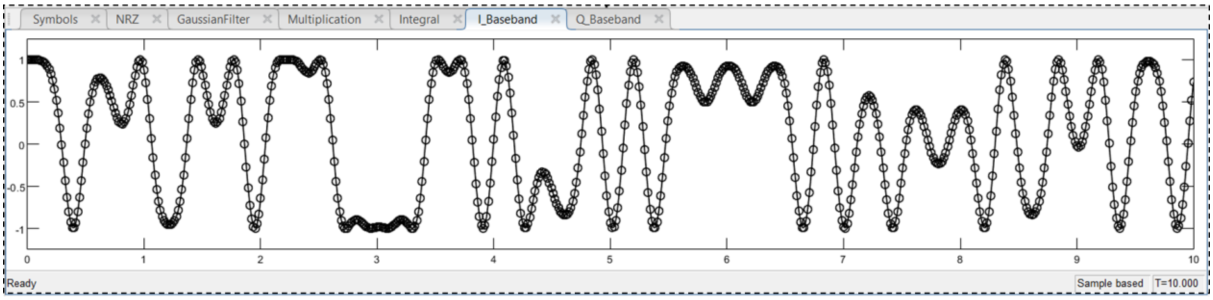


Figure 3.9 Waveform of In-phase baseband (Simulink)

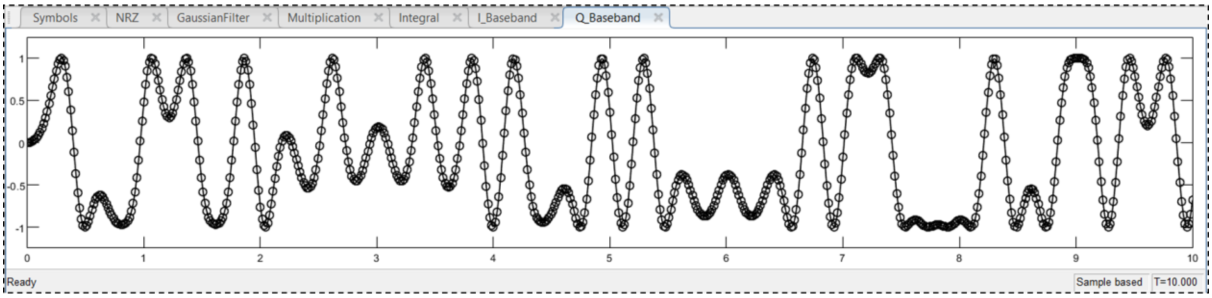


Figure 3.10 Waveform of Quadrature-phase baseband (Simulink)

3.2 GFSK demodulation in Simulink

I think we've almost understood the procedures of GFSK modulation, sequentially, the demodulation is converting the procedures of modulation in contrast. During Simulink, there are several blocks we can invoke and to realize the symbols recovering. The Figure 3.11 is associating with the GFSK demodulation with Simulink, where comprises of the algorithm of arctangent, the waveform unwrapping, derivative, coder and finally converting the symbols from bipolar to unipolar.

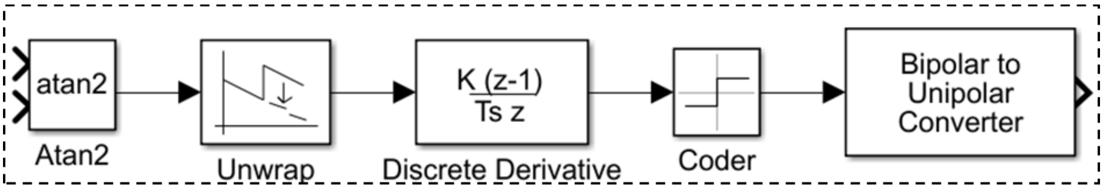


Figure 3.11 GFSK demodulation in Simulink

When we got the I/Q basebands from modulation, we need to utilize a trigonometric function to convert the signal from I/Q baseband to the phase deviation, that is arctangent, and the corresponding block in Simulink is **Atan2**, therefore we can acquire the output as representing in Figure 3.12.

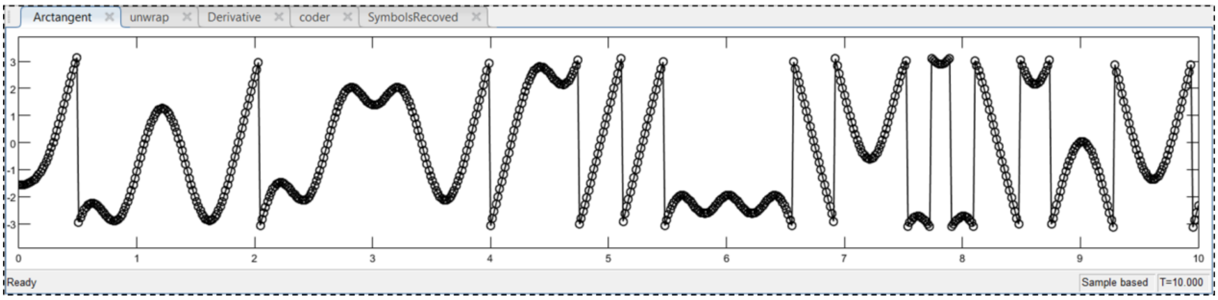


Figure 3.12 Result of Arctangent (Simulink)

Obviously, there is a limitation around $[-\pi, \pi]$, and resembles to cut the waveform when it reaches the boundary. In this case, we could eliminate the limitation by the block **Unwrap**, here we can see the results in Figure 3.13.

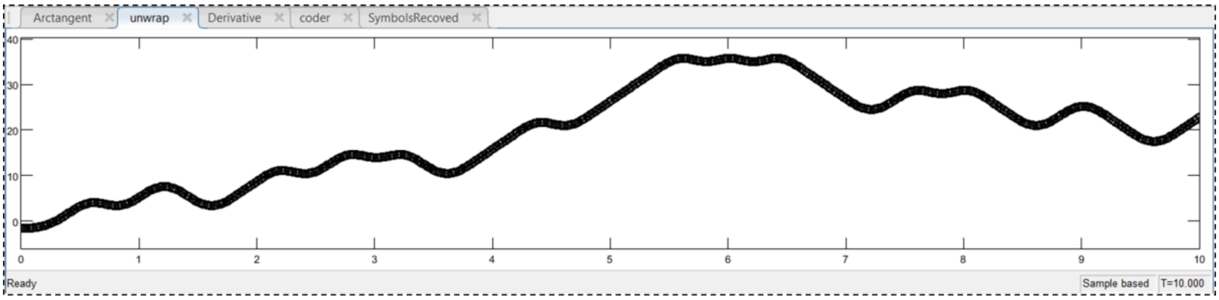


Figure 3.13 Result of Unwarp (Simulink)

We are definitely not strange with this waveform, because it almost same as the Figure 3.8 in previous subsection. Exactly, here the start and end points are difference with preceding, but the whole shape is almost same, perhaps the reason is due to a little delay and then appearing a little bit error, for now, it’s still acceptable.

Sequentially, the next is to convert the phase deviation to the frequency deviation, here we can utilize the **Discrete Derivative** block, then Figure 3.14 illustrates the outcome.

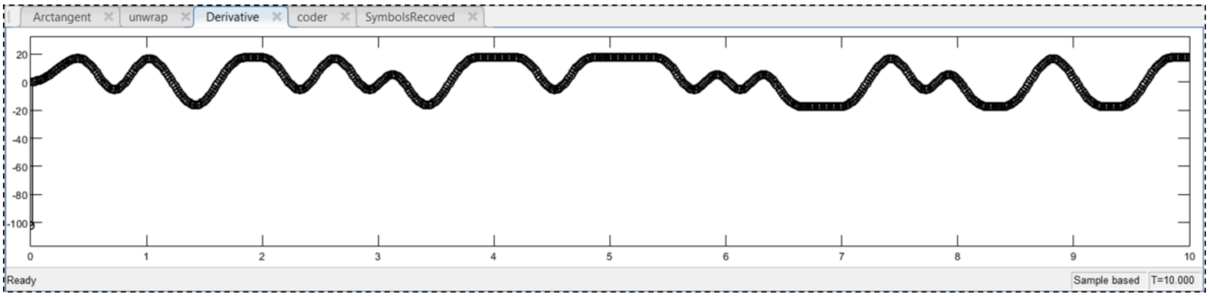


Figure 3.14 Result of discrete derivative (Simulink)

The beginning bit is error due to a little delay, consequently, the waveform is not looking very good. However, the rest shape is almost similar with Figure 3.7, so, we can use a quantizer to code the deviations, then the result is illustrated in Figure 3.15.

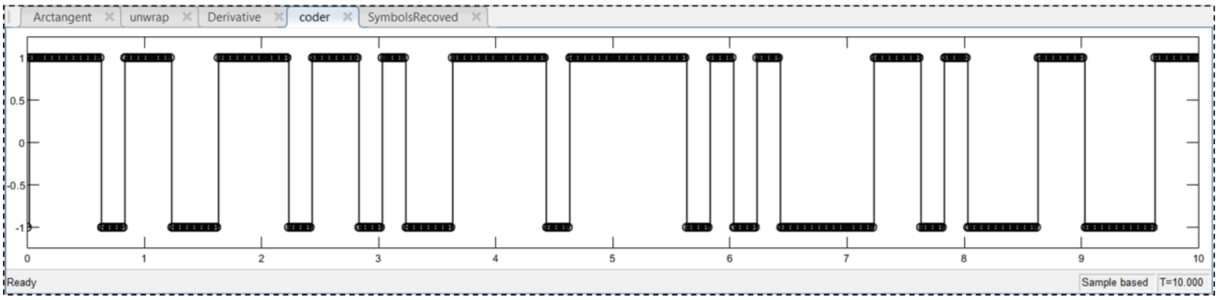


Figure 3.15 Result of Coder (Simulink)

Distinctly, it's NRZ sequence, so the next is turning it to the format of binary through the block of **Bipolar to Unipolar Converter**. Thus, the Figure 3.16 is describing the consequence.

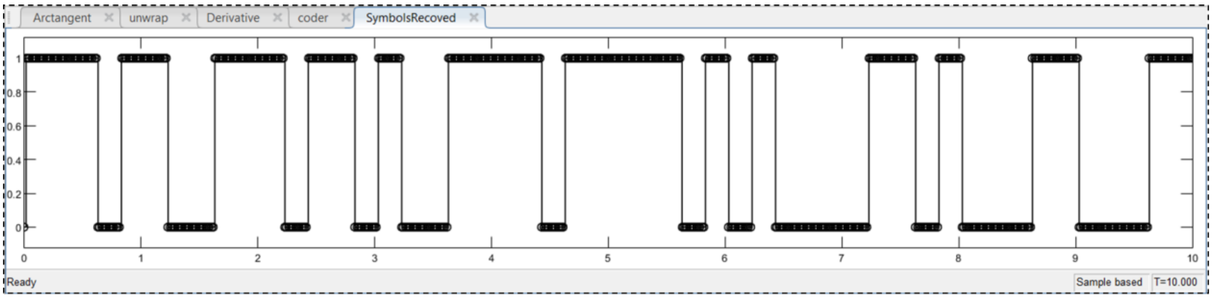


Figure 3.16 Output of Bipolar to Unipolar converter (Simulink)

Comparing with the Figure 3.3 which refers to original symbol sequence, the shape is almost recovered. But for now, the waveform still in the up-sampling scope. In this case, for recovering the symbol sequence totally, we also need to finish the last step called down sampling. It will be explained in the next chapter with OpenCL.

4 REALIZATION WITH OPENCL IN FPGA

During the chapter 2, we’ve already explained the architecture and the basic elements for constructing an OpenCL frame. Where it is separated as two parts, host code and kernel code, and it mainly has six elements within host code, that is platform, device, context, program, kernel, command queue. Moreover, as well as there are memory object and releasing resource allocated etc., which are just briefly mentioned during the theorical chapter. In addition, within the kernel code, we’ve illustrated some characteristics and novel concepts we are not met in the standard C libraries.

Along the objective, basing on the knowledge of OpenCL and GFSK, in this chapter we will explain the procedures of application from the theory of GFSK to the practice of OpenCL, and associate with the real codes for elaborating the architectures in host part and kernel part for this project. Besides, we will compile, execute and debug this program in the remote platform called Intel DevCloud, that would be interesting.

4.1 Host Part

As mentioned previously, the host code can program by C/C++, thus the suffix of file can be (.cpp), and the grammar or syntax of programming almost can inherit from the standard C/C++. Therefore, in general we need to invoke a serial of head files at the beginning. As showing below, there lists some head files what we need to associate with.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <string.h>
#include "CL/opencl.h"
#include "aocl_utils.h"
```

Where the first six head files belong to standard C/C++ libraries, they have defined a lot of function prototypes and macros such as the description of input and output, file streaming, etc., so that compiler is able to understand what their parameters are, and the meaning of return values, computing actions, and so forth. The final two are associated with the OpenCL, which also indicate a number of functions and macros. But they are the specifying elements and interfaces of OpenCL, such as the basic combination about platform, device, etc., we can invoke these within the host code when we need.

The next we need to declare platform by `cl_platform_id`, define device by `cl_device_id,` and so forth, the total six basic elements and a *status* are described below.

```
static cl_platform_id platform = NULL;
static cl_device_id device = NULL;
static cl_context context = NULL;
static cl_program program = NULL;
static cl_kernel in_kernel = NULL;
static cl_kernel out_kernel = NULL;
static cl_command_queue fpga_to_host_queue = NULL;
static cl_command_queue host_to_fpga_queue = NULL;
static cl_int status = 0;
```

Where the specifier "*static*" indicates to declare a variable or parameter in global address, and we also can see that there define two different kernels and two different command queues separately. As the syntax said, these are associating with the kernel part, which can realize the communication between the host code and kernel code. Moreover, except the final one, all of these are given an initial value called *NULL*, which equals to zero actually, because it's depended on the data structure.

Then, we are going to create the six basic elements, which are contained in a function call `bool init()`. We've noticed that this function is declared by `bool` data structure, which means the return value would be in Boolean. Because during this function, we need to judge whether all the elements were created successfully, and then return a Boolean value indicates yes or no. For creating the structure of platform, we can describe it in OpenCL as showing below.

```
cl_uint num_platforms;
status = clGetPlatformIDs(1, &platform, &num_platforms);
if (platform == NULL)
{
      printf("ERROR: Unable to find Intel(R) FPGA OpenCL platform.\n");
      return false;
}
```

Where the syntax `clGetPlatformIDs` is corresponding with explained in the section 2.4.2.1. Besides, the subfunction "`if`" refers to make a condition to know whether this structure is created successfully, and return a Boolean value, that is why we declared the function `init()` in the data structure `bool`. As for the rest elements, the code architecture for creating themselves structure are similar with the platform, here will not display these one by one, but we still need to take a notice about the kernel and command queue which are going to be created two different structures for associating with the kernel part.

Moreover, because we created the structure of context, program, kernel and command queue, these will be allocated to buffers. And considering the finite resource in FPGA, another function about resource releasing that we need to create for these elements, that is *void* `cleanup()`. and the code lines are shown below.

```
if(in_kernel)            clReleaseKernel(in_kernel);
if(out_kernel)           clReleaseKernel(out_kernel);
if(host_to_fpga_queue)  clReleaseCommandQueue(host_to_fpga_queue);
if(fpga_to_host_queue)  clReleaseCommandQueue(fpga_to_host_queue);
if(program)              clReleaseProgram(program);
if(context)              clReleaseContext(context);
if(input_buf)            clReleaseMemObject(input_buf);
if(output_buf)           clReleaseMemObject(output_buf);
```

Where the last two lines are about freeing the buffers of input and output, which can communicate data with kernel part.

There are also some functions we need to create, for example, it can read data from and write data to an external file separately, which are able to be described by normal C code, thus here will not explain too much.

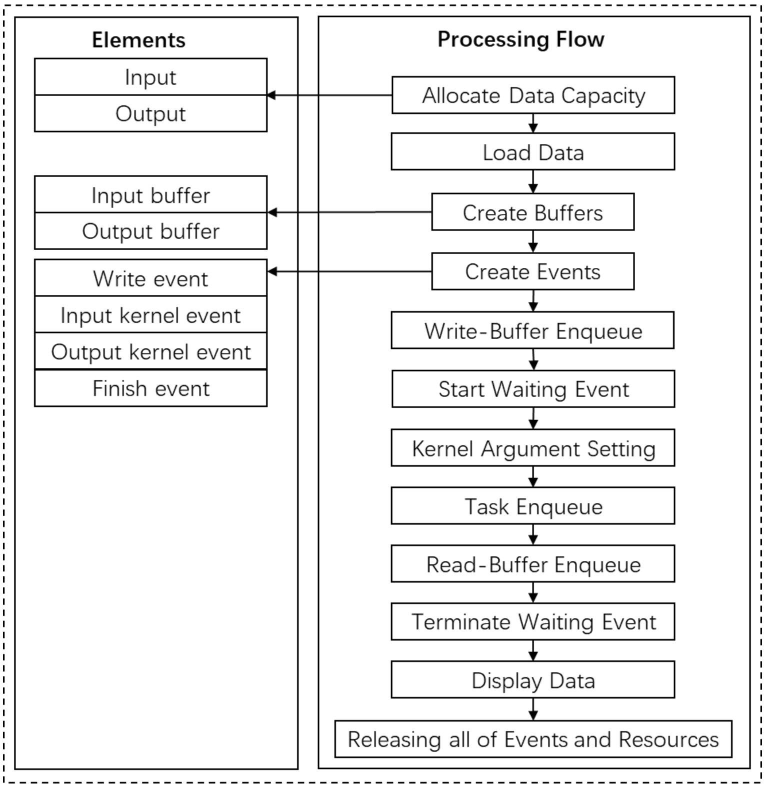


Figure 4.1 The processing flow of main() function in host part

Eventually, we turn to the `main()` function after finished the configuration of initialization functions. During the `main()` function, the procedure is illustrated in Figure 4.1.

We can see that it's beginning from the data capacity allocating, we can invoke a syntax called *malloc* to realize it. In addition, notice that it is necessary to allocate two different data address space, named `input` and `output` for storing data which preparing to send to and receive from kernel part respectively. The exact code is shown below.

```
unsigned int *input = (unsigned int *)malloc(dat_size*sizeof(int));
unsigned int *output = (unsigned int *)malloc(dat_size_out*sizeof(int));
```

Where the `dat_size` and `dat_size_out` refer to the size of sending data and receiving data separately.

After we finished the allocations of data space, the second step is loading the data what we want to send to kernel part. Here it accepts an approach that reading from a file, the prerequisite is that before the `main()` function we've declared the function of reading data from an external file. Then transferring the data to the data space named `input`, which allocated in the first step, and the code is shown below.

```
for (int i =0; i < dat_size; ++i)
{
      Readfile(input_ch);
      input[i] = input_ch.read();
}
```

The next step is allocating the buffer object which can involve any data type but not image. And this also need to create two different buffer objects by command `clCreatBuffer()`. The code is represented below.

```
input_buf = clCreateBuffer(context, CL_MEM_READ_ONLY, dat_size *size-
                                                        of(unsigned int), NULL, &status);
output_buf = clCreateBuffer(context, CL_MEM_WRITE_ONLY, dat_size_out * size-
                                                        of(unsigned int), NULL, &status);
```

Obviously, the difference of argument setting is only the second parameter, the `input_buf` is specified to read only, and the `output_buf` is in contrast.

When we finish the buffer object created, some events and enqueues would be created for executing, communicating with kernel part. So, there are the events called `write_event`, `finish_event`, `in_kernel_event`, and `out_kernel_event`. Where the `finish_event` will judge when the kernel part starting and finishing. Then, to create an enqueue for writing data to buffer object and the code is displayed below.

```
status = clEnqueueWriteBuffer(host_to_fpga_queue, input_buf, CL_FALSE, 0,
                 dat_size * sizeof(unsigned int), input, 0, NULL, &write_event);
if(status!=CL_SUCCESS) printf("Failed to transfer input\n");
```

we can see that arguments are associating with command queue of from host to fpga, and the *input_buffer* object, data size, data space, and the *write_event*, which have been defined before. And then judge whether make it enqueue successfully through setting a condition by "`if`".

Now the data will turn to the kernel part. At the beginning, here we call a function named `clWaitForEvents()` to interpret the work of kernel part is starting, we need to wait it accomplished. Therefore, the `finish_event` is invoked for this function as shown below.

```
clWaitForEvents(0, &finish_event);
```

Where the argument *0* indicates the event of kernel part is starting.

However, we don't know which kernel of kernel part will receive the data from host or write result to host, so we need to set some arguments for kernels in host part to recognize the corresponding kernels of kernel part. In this case, the function `clSetKernlArg()` is invoked here. Moreover, we need to configure the data size and buffers in input kernel and output kernel respectively, so there are 4 command lines for these, which are shown as below.

```
status = clSetKernelArg(in_kernel, argi++, sizeof(cl_mem), &input_buf);
status = clSetKernelArg(in_kernel, argi++, sizeof(cl_int), (void *)
                                                              &dat_size)
status = clSetKernelArg(out_kernel, argi++, sizeof(cl_mem), &output_buf);
status = clSetKernelArg(out_kernel, argi++, sizeof(cl_int), (void *)
                                                        &dat_size_out);
```

where the `in_kernel` and `out_kernel` are separately corresponding to the kernels in kernel part for receiving data from host and writing data to host, that is we need to declare both during the host part.

Once we know which kernels in kernel part are docking with the kernels of host part, then the host part can arrange the tasks for kernel part, and the function `clEnqueueTask()` would be invoked. The code lines are represented just like below.

```
status = clEnqueueTask(host_to_fpga_queue, in_kernel, 1, &write_event,
                                                  &in_kernel_event);
status = clEnqueueTask(fpga_to_host_queue, out_kernel, 0, NULL,
                                                  &out_kernel_event);
```

where there are also two parts according to the command queue. The former indicates the specified kernel of kernel part receives the data from the kernel of host part, and the latter refers to write.

While the kernel part finishes computing, and then returns the consequence to the host part, the data will be stored in the buffer called `out_buf` in host part. Therefore, if we want to display it, we need to transfer the data from buffer to the data space, that is why we called the function named `clEnqueueReadBuffer()`, which creates an enqueue for reading data from a buffer. The correlative code is represented as below.

```
status = clEnqueueReadBuffer(fpga_to_host_queue, output_buf, CL_FALSE, 0,
            dat_size_out*sizeof(unsigned int), output, 1, &out_kernel_event,
                                                            &finish_event);
```

Obviously, it's in contrast with the enqueue called `clEnqueueWriteBuffer()`. And then we enable the event called `finish_event` to indicate that the waiting event is terminal, the code as showing below.

```
clWaitForEvents(1, &finish_event);
```

Ultimately, we call a *for* loop to display the result from kernel part, and then release all of events and resources.

4.2 Kernel Part

As explained in the last section, while the data has been sent to the input buffer object in host part, the corresponding kernel in kernel part will read the data from host, then, the work of kernel part is beginning, until returning the results to the host by a specific kernel. Besides, through the analysis before, the host part mainly was constructed a serial of conditions for communicating with kernel part, as for the data computing and processing, they are almost within kernel part. Therefore, the GFSK modulation and demodulation will be realized in kernel part, and here will explain how to construct them via associating with relevant theories and simulation.

Firstly, for this project, the procedure of GFSK modulation and demodulation is illustrated in Figure 4.2. Clearly, there are mainly 4 parts within this processing flow, which comprises of the host, global memory, processing units, and channels extension. Where the host refers to the host part, the rest belong to kernel part. Moreover, during the processing units, it's both containing the modulation and demodulation, because here we want to verify the feasibility and validity of algorithms and concepts, and shrink the time cost by the way. In addition, notice that only the processing units of `Symbols` and `Output` can communicate with the host, obviously, they are docking with the kernels of host part, and corresponding to the `in_kernel` and `out_kernel` as mention in the last section.

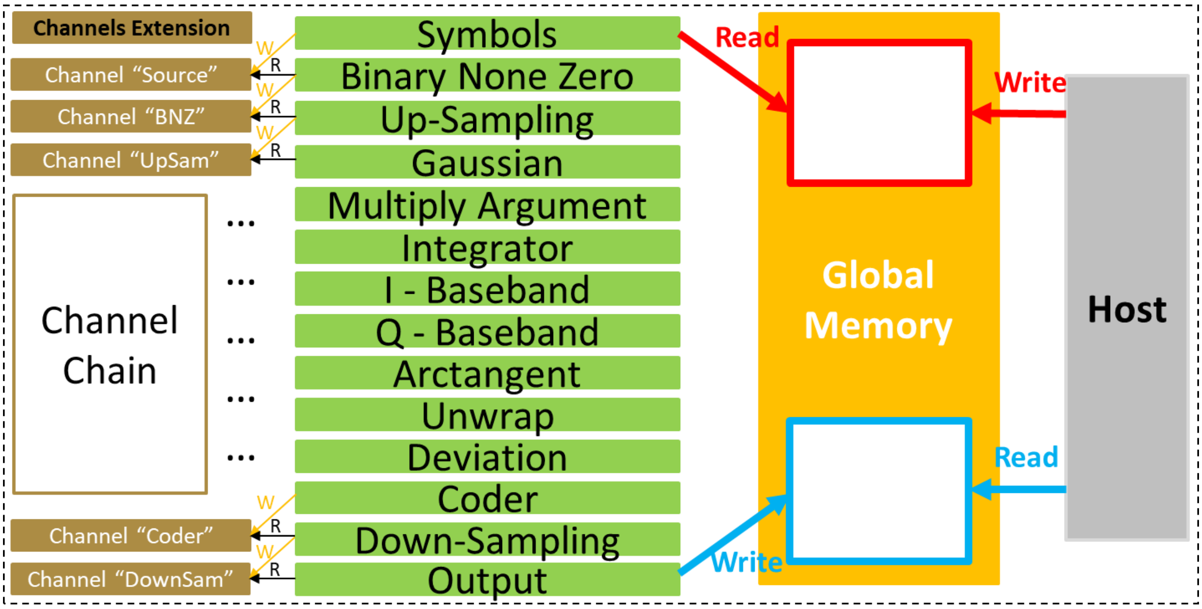


Figure 4.2 Processing Flow of GFSK Modulation and Demodulation in Kernel Part

Now, let’s turning to the construction of kernel code. For this project, the basic architecture in program as showing below.

```
attribute__((max_global_work_dim(0)))
kernel void Symbols (__global memory)
{
      for () {Read data from host machine and write to the channel}
}

attribute__((max_global_work_dim(0)))
attribute__((autorun))
kernel void BinaryNoneZero()
{
      while() {  Read data from the last channel.
      Computing, which almost similar to C/C++.
      Write data to the next channel. }
}
…
…
attribute__((max_global_work_dim(0)))
kernel void Output (__global memory)
{
      for () {read data from the last channel and write to the host machine}
}
```

For the first kernel and the final kernel, they are both invoking the `for` loop. The former one is for reading data from global memory and writing to a new channel, the latter one is for reading data from the last channel and writing to the global memory.

Except the above two kernels, the others are invoking the *while* loop, which has an infinite feature. The difference is that these kernels are specifically declared an *autorun* attribution, which means they don't need to receive the orders from host machine and then are able to process data automatically. In other word, they can be initialized before executing the `main()` function in host part, so that realized the concept of parallel programming. Additionally, they transfer the data through the channels, which is an extension of OpenCL, and the invoking method has already represented before.

Besides, all of the kernels are given an attribution called `max_global_work_dim(0)`. On the one hand, it can instruct the compiler to ignore logic which generates and dispatches global, local, and group IDs into the compiled kernel [32]. On the other hand, it's because we have invoked the `autorun` attribution, we have to call it as well.

During simulation by Simulink, we've almost realized a whole modulation and demodulation, and there are many modules what we can call for the design in simulation, even if we need to configure some parameters within the blocks. However, that is not our primary purpose in this project, we aim to understand the algorithms each block, and realize them via a serial of computing processes with OpenCL. Therefore, here we dedicate to the construction of algorithms in Kernel Part. So far, we have already had a clearly conception about a whole structure of kernel part, in this case, the next we will separate two subsections to respectively illustrate the GFSK modulation and demodulation in OpenCL, especially the realization of Gaussian filter.

4.2.1 GFSK modulation in OpenCL

At the beginning, we need to call the pragma of channels, and declare a lot of corresponding channels [37], a part of code is illustrated as below.

```
#pragma OPENCL EXTENSION cl_intel_channels : enable
channel int Source_ch;
channel float Gaussian_ch;
```

Here, the first line indicates enabling the channel extension, then the last two lines declare the corresponding channels for storing data temporally, they have different data structure that depend on the type of transferring data. Similarly, the rest channels are declared by following these formats.

After finished the declaration of channels, we can call these channels within kernels. According to the architecture of kernel part, firstly, it should read data from host part, then write to channel. The relevant code is given by below.

```
attribute__((max_global_work_dim(0)))
kernel void Symbols(__global unsigned int* inputdata, unsigned int dat_size)
{
	for (int i=0; i < dat_size; i++)
	write_channel_intel(Source_ch, inputdata[i]);
}
```

We can see that it defines a pointer in global type, that means the kernel will read data from a global memory through a pointer. Following the declaration in the host part before, the host will send data to the global memory of kernel part, thus here the kernel can read data from global memory directly as long as configured the correlative arguments. Then, during the kernel function, it calls a `for` loop and writes data to the first channel. By the way, the `Symbols` kernel is corresponding to the block of **Symbols** in Theories chapter, the block of **Bernoulli Binary Generator** in Simulink, and the kernel named `in_kernel` in host part. Successively, it is going to experience a number of autorunning kernels, which are emphases for this section.

Be associated with the preceding basic architecture of program, we need to configure the attribution of *autorun* and *(max_global_work_dim(0))* for each kernels during the processes of autorunning, besides, call *while* loop as well. Therefore, for the first autorunning kernel named `BinaryNoneZero`, is correspond to the block of **Binary None Zero (BNZ)** in Theories chapter, and the block of **Unipolar to Bipolar Converter** in Simulink. Clearly, the function is to transform the symbol type from binary to NRZ, the code within `while` loop is illustrated below.

```
int v = read_channel_intel(Source_ch);
if (v == 0)		v = -1;
write_channel_intel(BinaryNoneZero_ch, v);
```

It's easily understood. First line depicts to define a variable (actually it refers to an array) which reading data from the first channel. Then the second line indicates to process the variable, that is setting a condition by invoking "`if`" syntax, to transform the symbols from "0" to "-1". The final line depicts to write the result to a new channel.

According to the Figure 2.5, the next is up sampling, so, the code in the `while` loop of the kernel `Upsample` is shown below.

```
int v = read_channel_intel(BinaryNoneZero_ch);
for (int i=0; i<13; i++)
	write_channel_intel(Upsample_ch, v);
```

Similarly, the first thing is declaring a variable which reading data from the last channel. However, here is some differences, for the `for` loop, it doesn't call the pragma unrolling, and arrange the behaviour of writing data to channel within here. The reason is because we need to interpolate the symbols basing on the original data structure, that is duplicating 13 times for each symbol, and then write to channel by the way.

Eventually, we are going to the most crucial part during a whole GFSK modulation and demodulation, that is construction of Gaussian filter. The relevant algorithms and processes are explained explicitly during the equations from (2-3) to (2-11), hence we can associate with these equations to build a Gaussian filter in OpenCL. There are two steps, the first is initializing a gaussian filter, the second is done the convolution according to the initial values of Gaussian and the input value. Incidentally, the initialization of gaussian filter must be prior to the `while` loop invoked, because we just need to execute it once. Then, let's check the code of Gaussian initialization as below.

```
1      float gaussian_values[GAUSSIAN_LEN];
2      float hmax = 1.5E6;
3      float ts = 1E-6;
4      float OSR = 8;
5      float acum = 0;
6      for (int i=0; i < GAUSSIAN_LEN; i++)
7      {
8            int k = i - GAUSSIAN_LEN/2;
9            float v = (hmax*ts*k/OSR);
10           gaussian_values[i]=  exp(-(v*v));
11           acum += gaussian_values[i];
12     }
13     for ( int i=0; i < GAUSSIAN_LEN; i++)
14           gaussian_values[i] = gaussian_values[i] / acum;
15     float samples[GAUSSIAN_LEN];
```

To the beginning, it defines an array space for gaussian value at the first line, and the length is associating with the the number of sampling per symbol, which corresponding to the parameter *sps* in FIR filter of Simulink, and here is equal to 28 as well. And then the lines between 2nd and 4th are declaring the values of h_{max}, T_s, *OSR*, which are according to the description part that locating after the equation (2-9), in conclusion, these parameters are suitable for Bluetooth. Sequentially, it calls `for` loop twice. During the first loop, a half of gaussian length is a reference value, in order to obtain the discrete sequence *k*. In this case, this structure will be symmetrical about Y orientation in the coordinate system. Then according to the equation (2-8), substituting the values of h_{max}, T_s, *OSR* to this equation and acquiring the array of initial gaussian values, as illustrating within the 8th to 10th line, and the structure resembles the Figure 2.7. Finally, the 11th line in the first `for` loop indicates to accumulate all the values, which is for the computing in the second `for` loop. Therefore, within the second `for` loop, we need to divide each initial gaussian value by the result of accumulation, and that's the consequence of gaussian initialization what we want. The last line declares an array named *samples* for storing the data which reading from channel. Then we turn to the computing part in the `while` loop, as showing below.

```
#pragma unroll
for (int i=0; i < GAUSSIAN_LEN-1; i++)
      samples[i] = samples[i+1];
samples[GAUSSIAN_LEN-1] = (float) read_channel_intel(Upsample_ch);

float acum_ga = 0;
#pragma unroll
for (int i=0; i< GAUSSIAN_LEN; i++)
      acum_ga += samples[i] * gaussian_values[i];

write_channel_intel(Gaussian_ch, acum_ga);
```

Here it calls the `for` loop twice as well, but the deference is they need to invoke the pragma unrolling, because they are both in `while` loop. Besides, different with previous behaviour of reading data from the last channel, it needs to shift one bit to left firstly by a `for` loop, because the data type needs to show as an array. For the second `for` loop, that is the convolution computing step, according to the equation (2-11) we can realize it in here. Finally, to write the results to a new channel.

As so far, we've finished the construction of Gaussian filter and output the result to the channel. And then the rest computing units in modulation is along to the architecture of autorun kernel. Hence, the computing segment codes are depicted respectively as below.

```
Multiple Kernel:        float v = read_channel_intel(Gaussian_ch);
                        float cv = v*(GAUSSIAN_LEN/5)*pi;
                        write_channel_intel(Multiple_ch, cv);
Integrator Kernel:      cv +=  read_channel_intel(Multiple_ch);
                  write_channel_intel(Integral_I_ch, cv/(GAUSSIAN_LEN-1));
                  write_channel_intel(Integral_Q_ch, cv/(GAUSSIAN_LEN-1));
I_Baseband Kernel:      float v = read_channel_intel(Integral_I_ch);
                        float c_i = cos(v);
                        write_channel_intel(I_Baseband_ch, c_i);
Q_Baseband Kernel:      float v = read_channel_intel(Integral_Q_ch);
                        float c_q = sin(v);
                        write_channel_intel(Q_Baseband_ch, c_q);
```

Notice that when we finished the integral processing, we need to send the same result to difference channels so that the kernels of `I_Baseband` and `Q_Baseband` can separately read it successfully. Otherwise it will report an error by compiler, that it’s not allowed to read data from a same channel by different kernels.

4.2.2 GFSK demodulation in OpenCL

As for the kernel part of GFSK demodulation, it is similar to the modulation. Now we are going to follow the sequential steps according to the Figure 2.11 and Figure 4.2, hence there are `Arctangent` kernel, `Unwrap` kernel, `Derivative` kernel, `Coder` kernel, `DownSamples` kernel and `Output` kernel here. Therefore, according to the architecture of autorunning kernel and the algorithm of arctangent, the computing part in `while` loop of *Arctangent* kernel is shown in below.

```
float v1 = read_channel_intel(I_Baseband_ch);
float v2 = read_channel_intel(Q_Baseband_ch);
float cv = atan(v2/v1);
write_channel_intel(Arctangent_ch, cv);
```

we can see that here can read data from different channels separately within a kernel, but different kernels cannot read from a same channel.

Then the `Unwrap` kernel will deal with data in array mode, so it need a similar structure which in the *Gaussian filter* kernel, for reading data from channel. In this case, the code in the *while* loop is shown below.

```
#pragma unroll
for (int i=0; i < GAUSSIAN_LEN-1; i++)
      samples[i] = samples[i+1];
samples[GAUSSIAN_LEN-1] = read_channel_intel(Arctangent_ch);
float acum = 0;
#pragma unroll
for (int i=0; i< GAUSSIAN_LEN; i++)
{
      if (((samples[i] - samples[i-1])<pi/2) && ((samples[i] –
                                                samples[i-1])>-pi/2))
            acum = samples[i];

      for (int j=1; j< GAUSSIAN_LEN; j++)
      {
             if (((samples[i] - samples[i-1]) <-(2*j-1)*pi/2) &&
                        ((samples[i] – samples[i-1]) >-(2*j+1)*pi/2))
            {
                  samples[i] += j*pi;
                  acum = samples[i];
            }
            else if (((samples[i] - samples[i-1]) >(2*j-1)*pi/2) &&
                        ((samples[i] - samples[i-1]) <(2*j+1)*pi/2))
            {
                  samples[i] -= j*pi;
                  acum = samples[i];
            }
      }
}
```

The crucial part is second `for` loop, which are the processing of unwrapping, the basic concept is divided two parts. The first is that it doesn't need to do the processing of unwrapping before the first time of reached limitation. And the second step is starting to unwrap when the data over the first limitations due to the function arctangent. During the third *for* loop, there are two conditions to solve the wrapping data. The first is for the process when data more than 0, and the second is for the data less than 0. As showing in Figure 3.11, the split point is π and $-\pi$ in Simulink, however, during OpenCL, the result is limited between $[-\frac{\pi}{2},\frac{\pi}{2}]$, so, the $\pm(2j-1)\pi/2$ and $\pm(2j+1)\pi/2$ are selected.

In the same way, the `Derivative` kernel will process data in array type, and according to the equation (2-17) we can write the code within `while` loop as showing below.

```
#pragma unroll
for (int i=0; i < GAUSSIAN_LEN-1; i++)
      samples[i] = samples[i+1];
samples[GAUSSIAN_LEN-1] = read_channel_intel(Unwrap_ch);
float acum = 0;
#pragma unroll
for (int i=0; i<GAUSSIAN_LEN; i++)
      acum=samples[i]-samples[i-1];
write_channel_intel(Derivative_ch, acum);
```

By the way, the bold part is about the computing process.

Regarding with the `Coder` kernel and `Down-sampling` kernel, because they are processing the data in a variable type, then they call the normal method of reading data from channel, and the segment code of them are respectively represented below.

`Coder` kernel:

```
float v = read_channel_intel(Derivative_ch);
unsigned int acum=0;
if (v>0)    acum=1;
if (v<0)    acum=0;
write_channel_intel(Coder_ch, acum);
```

`Down-sampling` kernel:

```
unsigned int v = read_channel_intel(Coder_ch);
for (int i=7; i< 1; i+=13)
      write_channel_intel(dsam_ch, v);
```

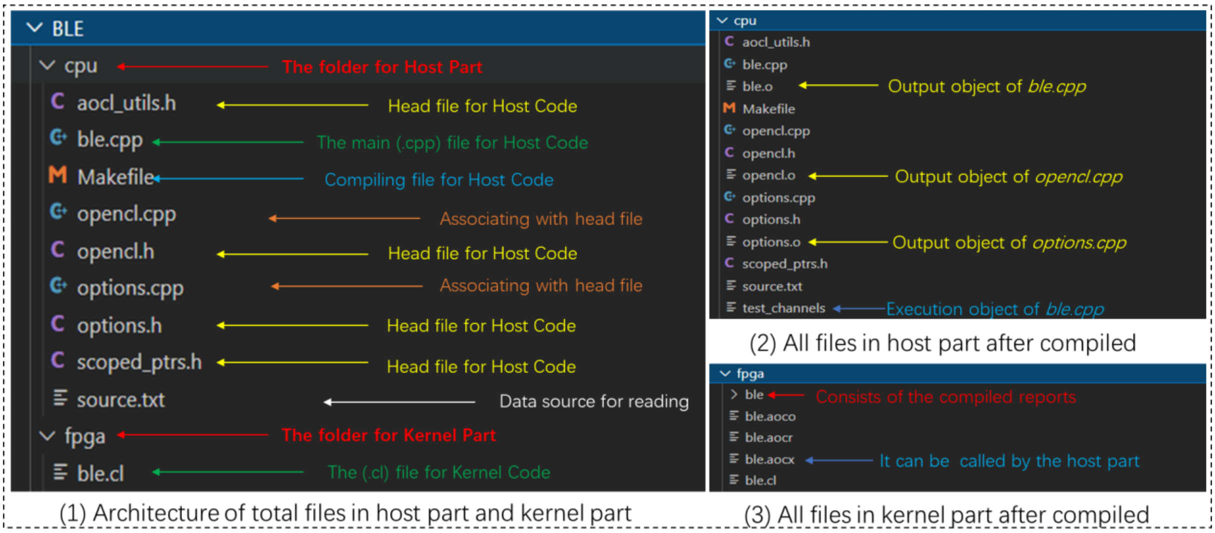
similarly, the computing pieces are in bold separately.

Finally, the last kernel is named `Output` kernel, which is responsible for writing the final result of kernel part to the global memory, so that the host can read this data directly. And the code is shown below.

```
attribute__((max_global_work_dim(0)))
kernel void Output(__global unsigned int* outputdata, unsigned int dat_size)
{
      for (int i=0; i<dat_size; i++)
      outputdata[i] = read_channel_intel(dsam_ch);
}
```

4.3 Compiling, Executing and Debugging with FPGA in Intel DevCloud Plaform

Because the program of OpenCL can be built with C/C++, we can compile and execute this file by the gcc/g++ tool in a program environment. Moreover, for more convenient during debugging, I built a compiling file called MakeFile. It can compile the main(.cpp) file with a single word "*make*" in the host part, instead of a normal method. About the kernel part, there are a serial of exclusive usages for operating. Especially, for compiling, we can call the usage "`aoc <file_name.cl>`" [39], where the file suffix (.cl) means this file is OpenCL kernel code.



(1) Architecture of total files in host part and kernel part

(2) All files in host part after compiled

(3) All files in kernel part after compiled

Figure 4.3 The constructures of host part and kernel part before and after compiled

As a whole, the architecture of total files in host part and kernel part are shown the Figure 4.3 (1). Where it divides two folders named *cpu* and *fpga* and they are corresponding to the host part and kernel part respectively. During the directory *cpu*, it comprises of main (.cpp) file, compiling files or making file, the data source file, a number of head files, and some (.cpp) files for corresponding to the head files. On the other hand, within the *fpga* folder, it only contains a (.cl) file for kernel code.

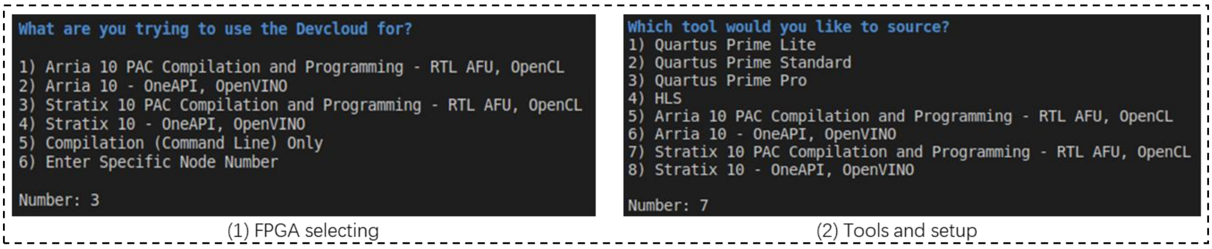
4.3.1 Compiling and executing the codes

Therefore, basing on the knowledge in the section of 2.3.4 about the environment configuration within Intel DevCloud, we create a new folder called *BLE* in this remote platform, and then paste the folders *cpu* and *fpga* to there from the folder *BLE* in local directory. After that, we need to access some scripts to login the compute node for compiling and executing the relevant programs, the detail is explained in Table 4.1.

Table 4.1 Some usages for programs compiling and executing in Devcloud

Step	**Detail**
1	`ssh devcloud` Login to the platform of Intel DevCloud
2	`source /data/intel_fpga/devcloudLoginToolSetup.sh` Source the script of FPGA
3	`devcloud_login` or `devcloud_login -I <device_name>` Select a FPGA device
4	`source /data/intel_fpga/devcloudLoginToolSetup.sh` Source the script of FPGA again
5	`tools_setup` Select a corresponding tools and setup.
6	`aoc <file_name>.cl` Compile the specified OpenCL kernel file (.cl).
7	`make` Compile the specified host main file (.cpp) and generate an execution file.
8	`./<execution_file>` Execute the execution file.
9	`qstatus` Check the status of current computing node or job
10	`qdel <job_name>` Halt a specified computing node or job
11	`aocl diagnose` Query a list of installed devices on our machine
12	`aocl program <device_name> <kernel_file_name.aocx>` Program an FPGA device offline or without a host

Where the steps from the 1st to the 8th are the normal verifying procedure for the program project of OpenCL in Devcloud. Noticed that we have to source the script of FPGA for twice, the 2nd and 3rd step are for selecting an FPGA, which is shown in Figure 4.4 (1), and the console of terminal will enter a compute node of FPGA. Then the 4th and 5th step are the way to speed up user engagement when they want to source the environment variable settings for a tool and when they are interactively logged into a compute node [35]. Besides, the selection must be corresponded with the FPGA what we've selected in the last step, which is shown in Figure 4.4 (2). Obviously, we made the option of FPGA called Stratix 10 in DevCloud. In contrast to Arria 10, which balances performance with low power requirements, it focuses on high performance solutions. Successively, the 6th step is executed in the directory of *fpga*, then generated several files and a folder named *ble*, which are shown in Figure 4.3 (3). Within the folder *ble*, we can find a lot of compiled reports, especially the synthesis result of this OpenCL design, which is shown in Figure 4.5. Regarding with this flow, firstly it tells us the synthesis was done successfully. Then it reveals some information about the device, Adaptive Logic Modules (ALMs) , Registers, DSPs, and Memories which have been used by this design, and these messages can also reflect the data-processing capability of FPGA. And the 7th and 8th step are executed within the folder of *cpu*, then all the output documents are illustrated in Figure 4.3 (2). Moreover, the rest of steps are the auxiliary usages for the platform-environment debugging.



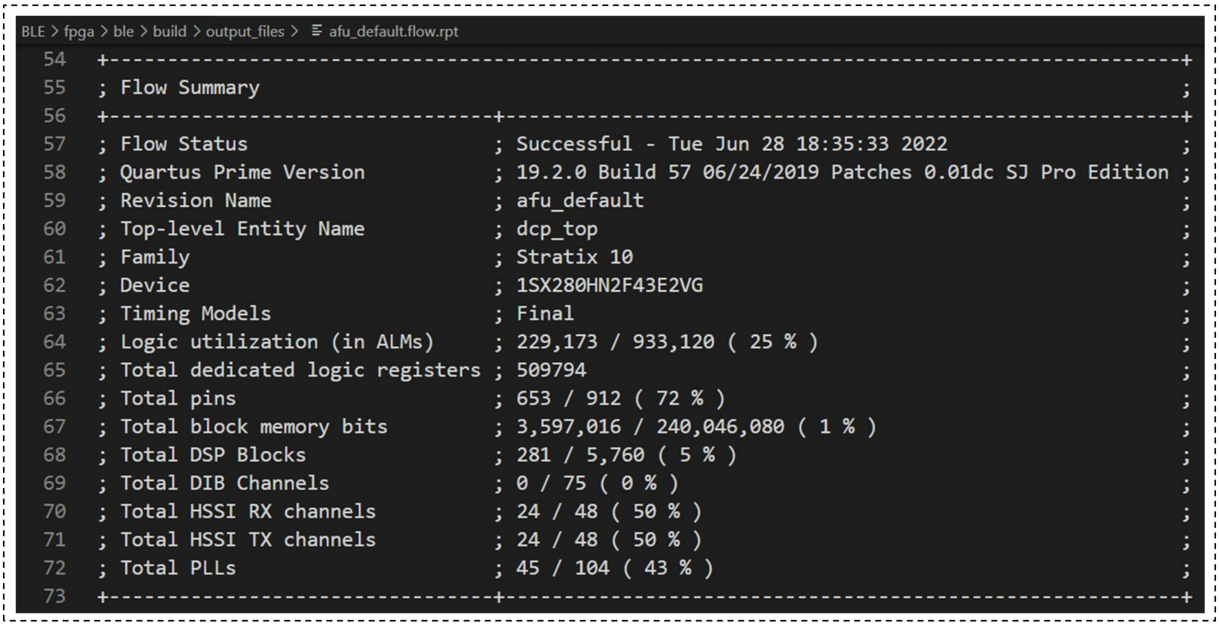
```
What are you trying to use the Devcloud for?

1) Arria 10 PAC Compilation and Programming - RTL AFU, OpenCL
2) Arria 10 - OneAPI, OpenVINO
3) Stratix 10 PAC Compilation and Programming - RTL AFU, OpenCL
4) Stratix 10 - OneAPI, OpenVINO
5) Compilation (Command Line) Only
6) Enter Specific Node Number

Number: 3
```

(1) FPGA selecting

```
Which tool would you like to source?
1) Quartus Prime Lite
2) Quartus Prime Standard
3) Quartus Prime Pro
4) HLS
5) Arria 10 PAC Compilation and Programming - RTL AFU, OpenCL
6) Arria 10 - OneAPI, OpenVINO
7) Stratix 10 PAC Compilation and Programming - RTL AFU, OpenCL
8) Stratix 10 - OneAPI, OpenVINO

Number: 7
```

(2) Tools and setup

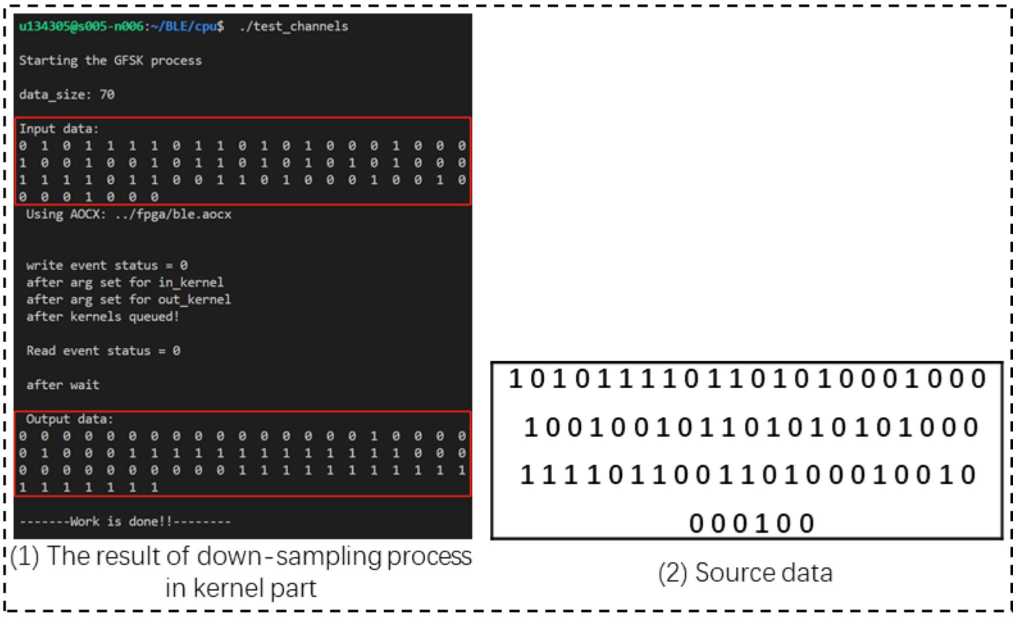
Figure 4.4 Configuration of FPGA environment in DevCloud



BLE > fpga > ble > build > output_files > afu_default.flow.rpt

```
+-----------------------------------------------------------------------------------+
; Flow Summary                                                                      ;
+--------------------------------+--------------------------------------------------+
; Flow Status                    ; Successful - Tue Jun 28 18:35:33 2022            ;
; Quartus Prime Version          ; 19.2.0 Build 57 06/24/2019 Patches 0.01dc SJ Pro Edition ;
; Revision Name                  ; afu_default                                      ;
; Top-level Entity Name          ; dcp_top                                          ;
; Family                         ; Stratix 10                                       ;
; Device                         ; 1SX280HN2F43E2VG                                 ;
; Timing Models                  ; Final                                            ;
; Logic utilization (in ALMs)    ; 229,173 / 933,120 ( 25 % )                       ;
; Total dedicated logic registers ; 509794                                          ;
; Total pins                     ; 653 / 912 ( 72 % )                               ;
; Total block memory bits        ; 3,597,016 / 240,046,080 ( 1 % )                  ;
; Total DSP Blocks               ; 281 / 5,760 ( 5 % )                              ;
; Total DIB Channels             ; 0 / 75 ( 0 % )                                   ;
; Total HSSI RX channels         ; 24 / 48 ( 50 % )                                 ;
; Total HSSI TX channels         ; 24 / 48 ( 50 % )                                 ;
; Total PLLs                     ; 45 / 104 ( 43 % )                                ;
+--------------------------------+--------------------------------------------------+
```

Figure 4.5 Synthesis result of OpenCL design



```
u134305@s005-n006:~/BLE/cpu$  ./test_channels

Starting the GFSK process

data_size: 70

Input data:
0  1  0  1  1  1  1  0  1  1  0  1  0  1  0  0  0  1  0  0  0
1  0  0  1  0  0  1  0  1  1  0  1  0  1  0  1  0  1  0  0  0
1  1  1  1  0  1  1  0  0  1  1  0  1  0  0  0  1  0  0  1  0
0  0  0  1  0  0  0
 Using AOCX: ../fpga/ble.aocx


 write event status = 0
 after arg set for in_kernel
 after arg set for out_kernel
 after kernels queued!

 Read event status = 0

 after wait

 Output data:
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0
0  1  0  0  0  1  1  1  1  1  1  1  1  1  1  1  1  1  0  0  0
0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1

-------Work is done!!--------
```

(1) The result of down-sampling process in kernel part

```
1 0 1 0 1 1 1 1 0 1 1 0 1 0 1 0 0 0 1 0 0 0
 1 0 0 1 0 0 1 0 1 1 0 1 0 1 0 1 0 1 0 0 0
 1 1 1 1 0 1 1 0 0 1 1 0 1 0 0 0 1 0 0 1 0
               0 0 0 1 0 0
```

(2) Source data

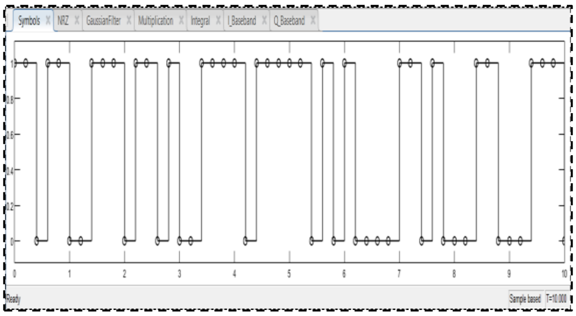
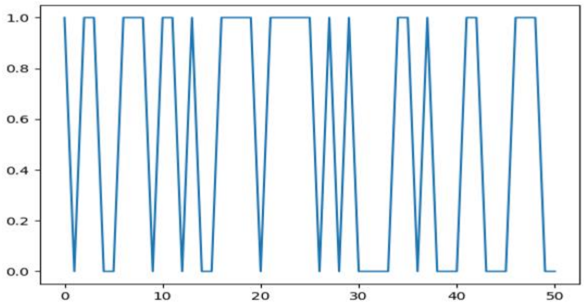
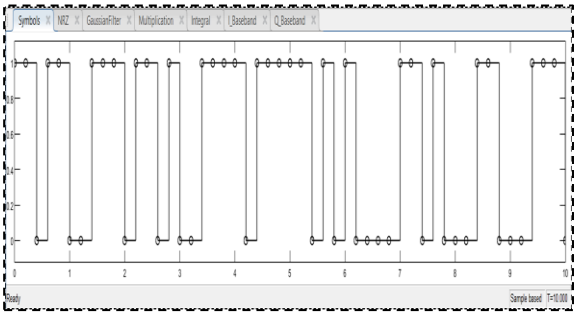
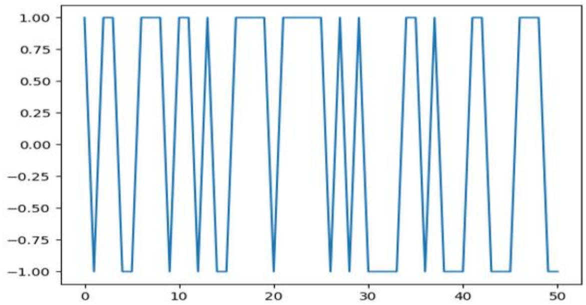
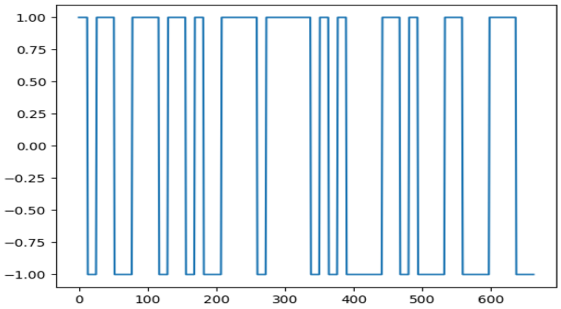
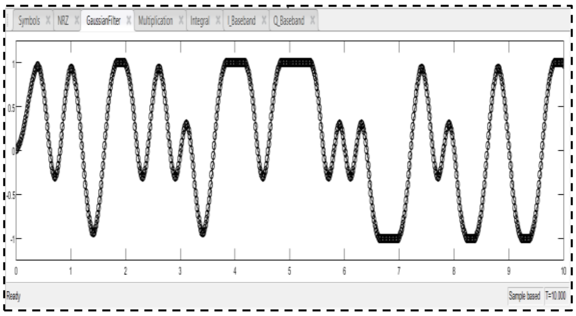
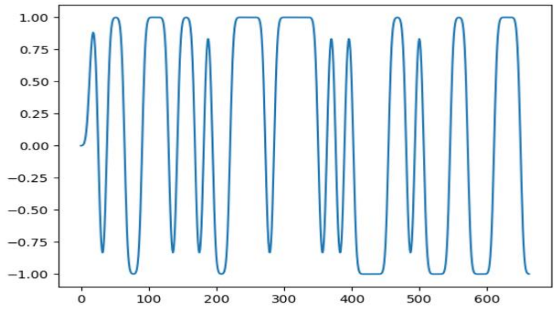
Figure 4.6 Result of execution with down sampling in kernel part

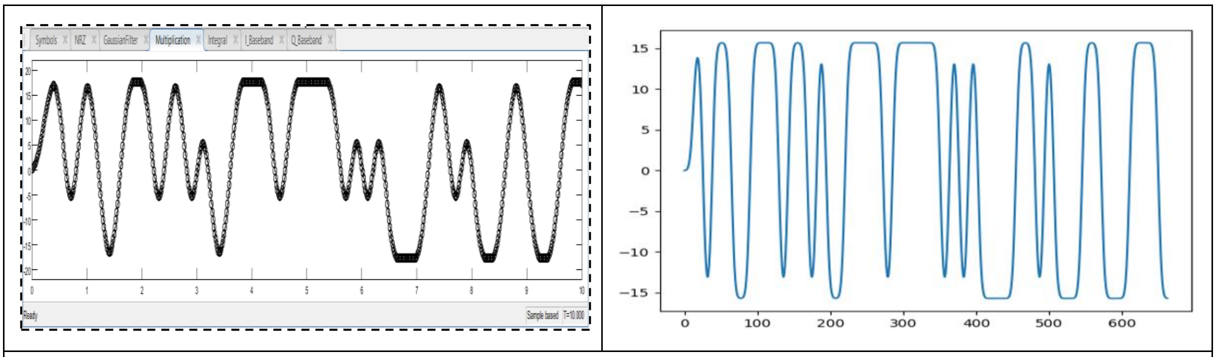
Finally, after went through the 8th step in Table 4.1, we can check the results in Figure 4.6 (1). Where the *Input data* refers to the result of `Symbols` kernel, and comparing with the random source data, which is given in the Figure 4.6 (2), this sequence has shifted 1 bit to left in a whole. Besides, the *Output data* corresponds to `Output` kernel, which is transferring the final processed result from the `Down-sampling` kernel. We can see that the input and output are totally different, this result is not our expected, in this case, we turn to debug the codes.

4.3.2 Debugging the codes

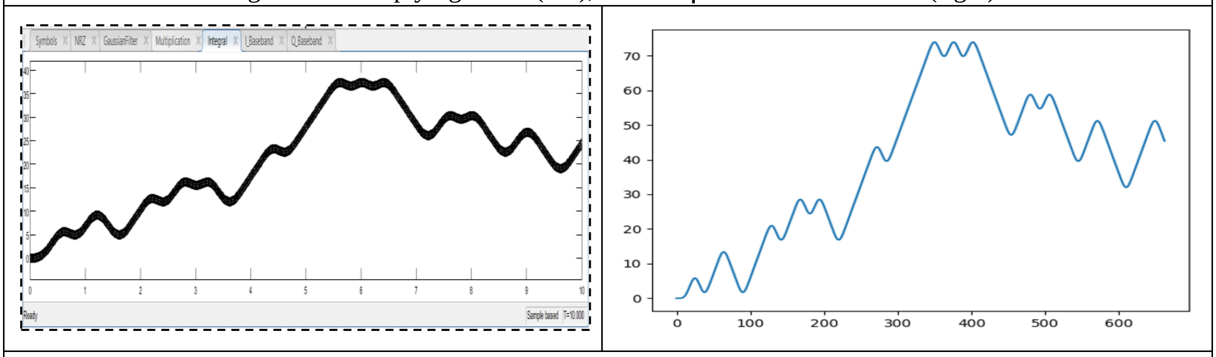
In order to show where problem is locating, we specified the original symbols as same as the sequences that shown in Figure 3.3, and then go through the compiling and executing. Therefore, we got the outputs of each computing kernel, and then display them in graph through Python programming. Moreover, we compare these graphic outputs with the waveforms what we've got in the chapter 3, and the results of comparison are shown in Table 4.2.

Table 4.2 The result of comparison between Simulink and OpenCL

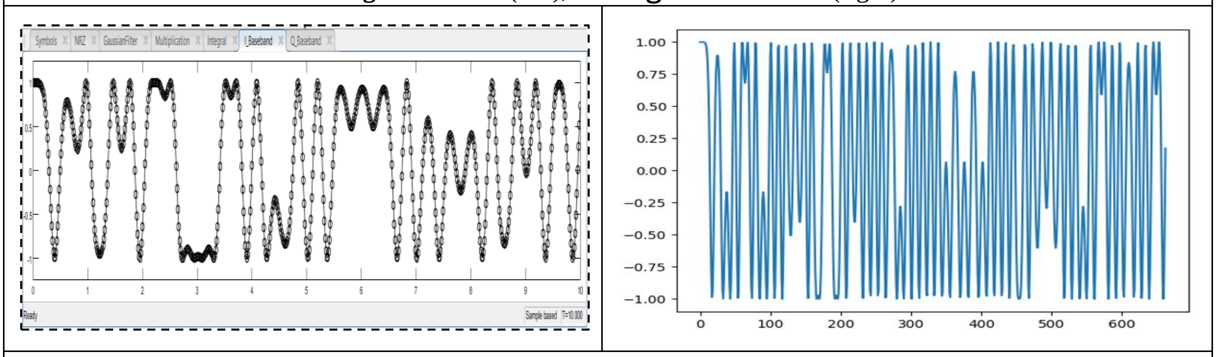
<table>
<tr><th>Simulink</th><th>OpenCL</th></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td colspan="2">**Bernoulli Binary Generator** block (left), `Symbols` kernel (right)</td></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td colspan="2">**Unipolar to Bipolar Converter** block (left), `BinaryNoneZero` kernel (right)</td></tr>
<tr><td>–</td><td>
</td></tr>
<tr><td colspan="2">– (left), `UpSample` kernel (right)</td></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td colspan="2">**FIR Filter** block (left), `Gaussian` kernel (right)</td></tr>
</table>



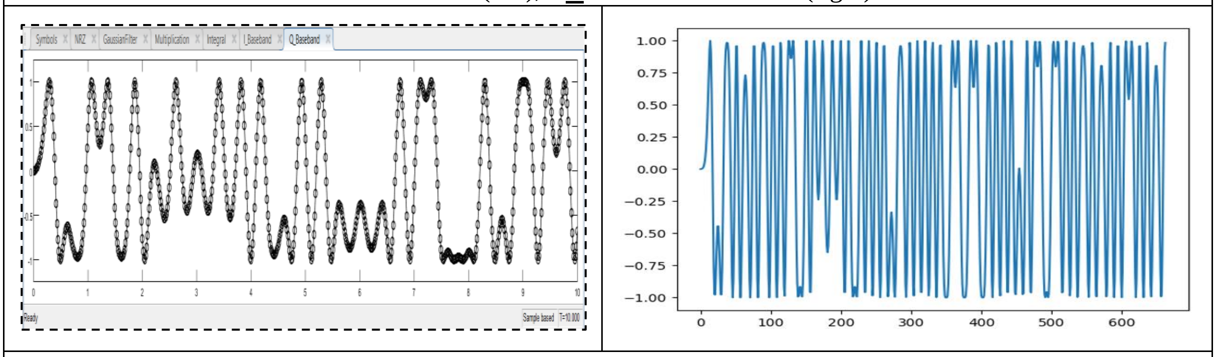
Argument Multiplying block (left), `Multiplication` kernel (right)



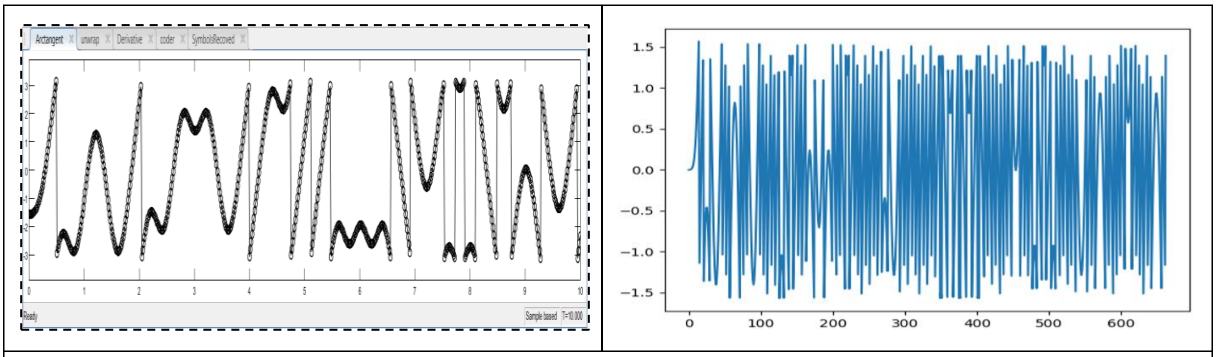
Integrator block (left), `Integrator` kernel (right)



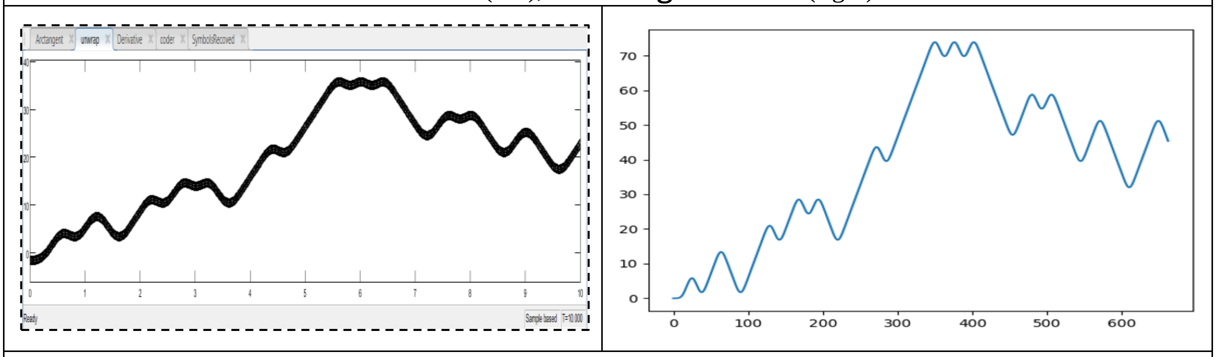
Cosin block (left), `I_Baseband` kernel (right)



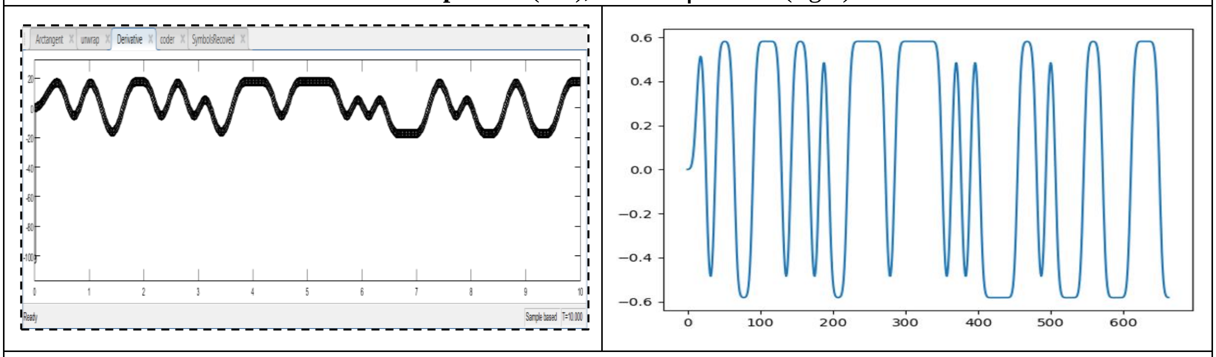
Sine block (left), `Q_Baseband` kernel (right)



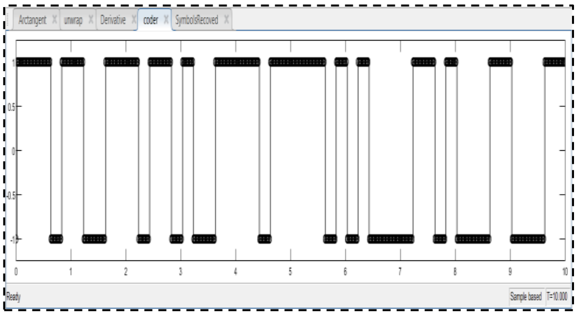
Atan2 block (left), `Arctangent` kernel (right)



Unwrap block (left), `Unwrap` kernel (right)

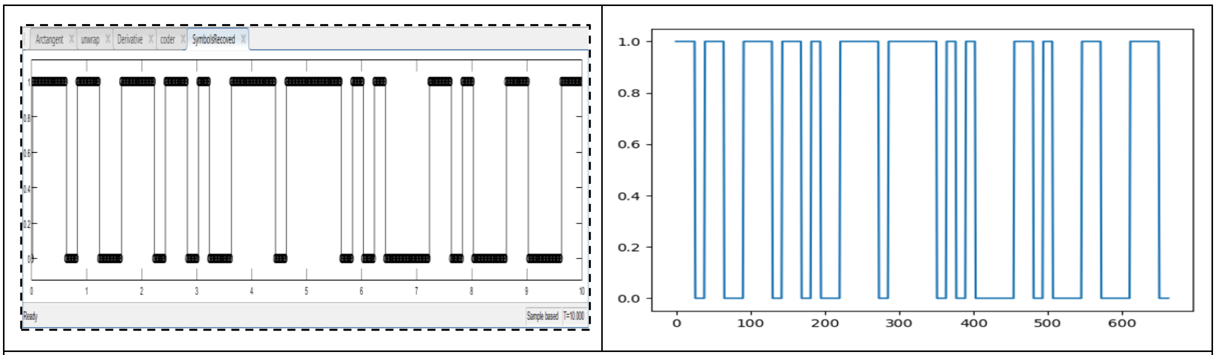


Discrete Derivative block (left), `Derivative` kernel (right)

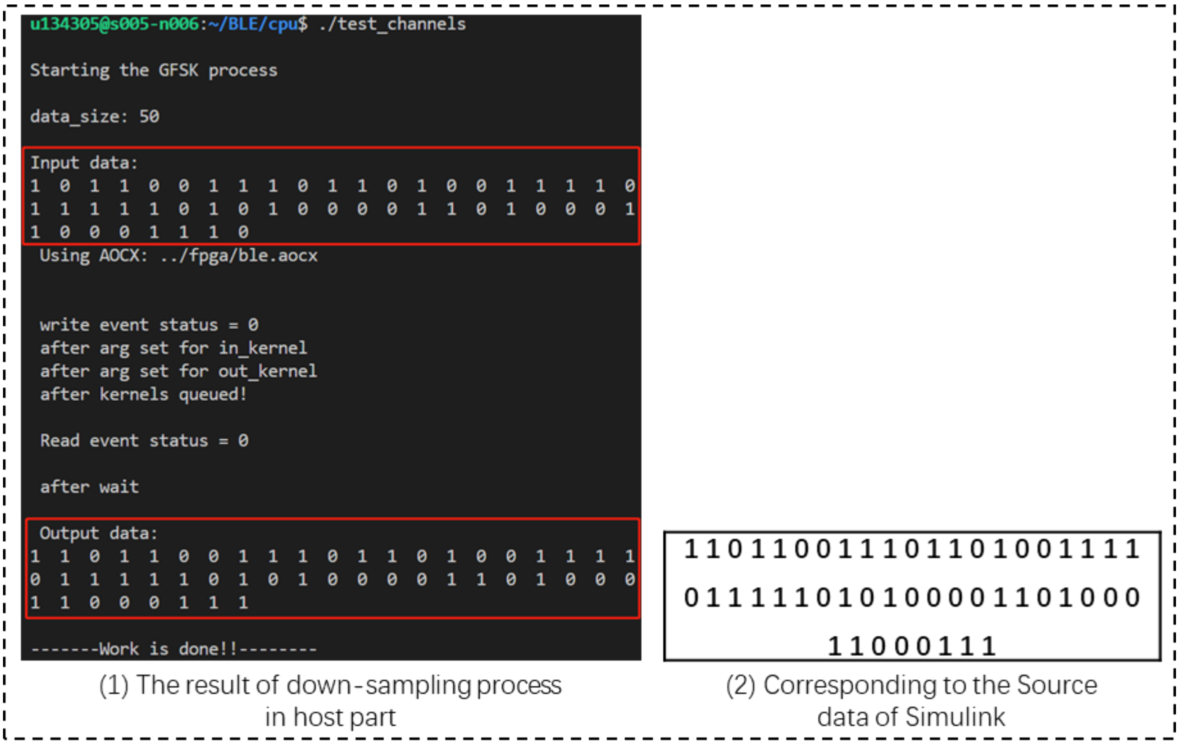


–

Coder block (left), − (right)



Bipolar to Unipolar Converter block (left), `Coder` kernel (right)



```
u134305@s005-n006:~/BLE/cpu$ ./test_channels

Starting the GFSK process

data_size: 50

Input data:
1  0  1  1  0  0  1  1  1  0  1  1  0  1  0  0  1  1  1  1  0
1  1  1  1  1  0  1  0  1  0  0  0  0  1  1  0  1  0  0  0  1
1  0  0  0  1  1  1  0
 Using AOCX: ../fpga/ble.aocx


 write event status = 0
 after arg set for in_kernel
 after arg set for out_kernel
 after kernels queued!

 Read event status = 0

 after wait

 Output data:
1  1  0  1  1  0  0  1  1  1  0  1  1  0  1  0  0  1  1  1  1
0  1  1  1  1  1  0  1  0  1  0  0  0  0  1  1  0  1  0  0  0
1  1  0  0  0  1  1  1

-------Work is done!!--------
```

(1) The result of down-sampling process in host part

```
1 1 0 1 1 0 0 1 1 1 0 1 1 0 1 0 0 1 1 1 1
0 1 1 1 1 1 0 1 0 1 0 0 0 0 1 1 0 1 0 0 0
1 1 0 0 0 1 1 1
```

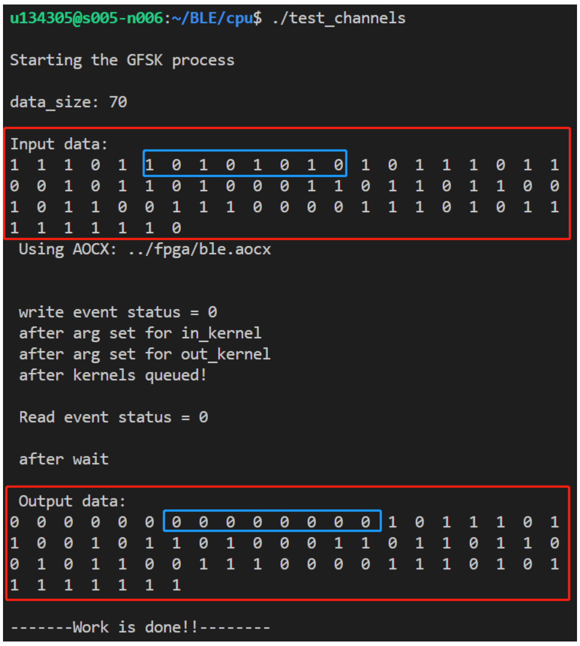
(2) Corresponding to the Source data of Simulink

Figure 4.7 The result of down-sampling process in Host part

From these results, except the down-sampling process not appeared in the Simulink, the rest are almost same. In this case, we found the problem is in the processing of `Down-sampling` kernel, it does not work through this algorithm. Finally, we removed the `Down-sampling` kernel in kernel part, and configured it within the host part. The position is locating at the second `for` loop which is reading data from kernel part within `main()` function, the detail is illustrated in the Appendix 1 Host Code, where just modified the condition $i = 0$ to $i = 7$, and $i{+}{+}$ to $i{+}{=}13$ in the `for` loop, and the result as showing Figure 4.7 (1). Besides, associating with the Figure 4.7 (2), we can see that the *output data* are same as the source data.

Furthermore, take considering into the synchronization of signal transmission in a real situation, we need to create a function to recognize the preamble code as well, for Bluetooth, it has explained in the Table 2.2. According to preamble format of adverting channel, the code of the computing unit in kernel `preamble` is shown below.

```
if ((samples[i] ==1 && samples[i+25] ==1 && samples[i+51] ==1 && samples[i+77]
   ==1 && samples[i+103] ==1 ) && (samples[i+12] ==0 && samples[i+38] ==0 &&
                                    samples[i+64] ==0 && samples[i+90] ==0 ))
{
      for (int j=0; j<GAUSSIAN_LEN*13*5; j++)
            acum=samples[j];
}
```



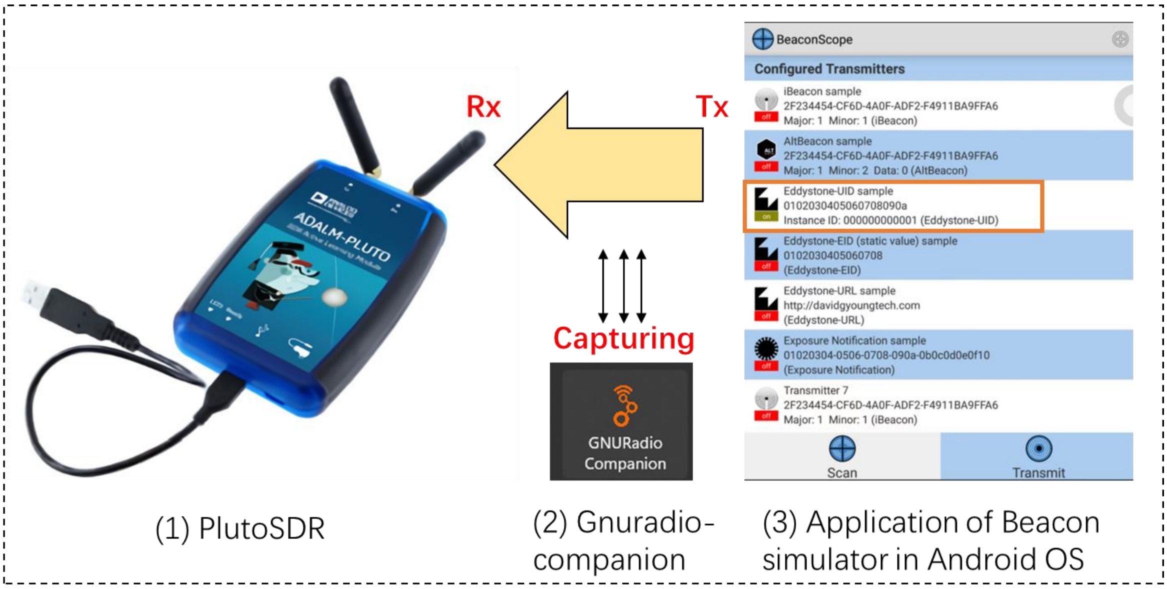
```
u134305@s005-n006:~/BLE/cpu$ ./test_channels

Starting the GFSK process

data_size: 70

Input data:
1  1  1  0  1  1  0  1  0  1  0  1  0  1  0  1  1  1  0  1  1
0  0  1  0  1  1  0  1  0  0  0  1  1  0  1  1  0  1  1  0  0
1  0  1  1  0  0  1  1  1  0  0  0  0  1  1  1  0  1  0  1  1
1  1  1  1  1  1  0
 Using AOCX: ../fpga/ble.aocx


 write event status = 0
 after arg set for in_kernel
 after arg set for out_kernel
 after kernels queued!

 Read event status = 0

 after wait

 Output data:
0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  1  1  1  0  1
1  0  0  1  0  1  1  0  1  0  0  0  1  1  0  1  1  0  1  1  0
0  1  0  1  1  0  0  1  1  1  0  0  0  0  1  1  1  0  1  0  1
1  1  1  1  1  1  1

-------Work is done!!--------
```

Figure 4.8 The final result of adding preamble kernel

Where we set a condition by `if` syntax to recognize the preamble sequence. Finally, after compiled and executed, the result is represented in Figure 4.8. We can see that the symbols have been eliminated before the preamble symbols, and even itself, which within the blue rectangle in this Figure. So, there is still a little not perfect, but except it, the all of functions in GFSK modulation and demodulation are almost realized. And finally, the whole kernel code we can check in Appendix 2 Kernel Code.

4.4 Verification with real BLE data stream

After finished the DSP design with OpenCL, the next we would verify it with the real BLE data stream. The first we need to capture some BLE data stream from air through the SDR device, so we build a capturing system to get this. And the architecture of this system should consist of the transmitter, receiver, and the data capturing tool, which are corresponding to the (3), (1), (2) respectively in Figure 4.8. We can see that here adopted the Beacon transmission method, exactly it is Eddystone-UID, and the packet format can see [39], obviously, we need to capture some BLE signal packets through the advertising channel. Regarding the PlutoSDR, it has transmitting and receiving channel, here we just utilize the latter one. As for the tool gnuradio-companion, remember the sample rate should be 13 MHz, because the GFSK modulated output is still in the up-sampling scope.



iBeacon sample
2F234454-CF6D-4A0F-ADF2-F4911BA9FFA6
Major: 1 Minor: 1 (iBeacon)

AltBeacon sample
2F234454-CF6D-4A0F-ADF2-F4911BA9FFA6
Major: 1 Minor: 2 Data: 0 (AltBeacon)

Eddystone-UID sample
0102030405060708090a
Instance ID: 000000000001 (Eddystone-UID)

Eddystone-EID (static value) sample
0102030405060708
(Eddystone-EID)

Eddystone-URL sample
http://davidgyoungtech.com
(Eddystone-URL)

Exposure Notification sample
01020304-0506-0708-090a-0b0c0d0e0f10
(Exposure Notification)

Transmitter 7
2F234454-CF6D-4A0F-ADF2-F4911BA9FFA6
Major: 1 Minor: 1 (iBeacon)

(1) PlutoSDR

(2) Gnuradio-companion

(3) Application of Beacon simulator in Android OS

Figure 4.9 The system structure of BLE data packet capturing

When accomplished the configuration of the capturing system, we can get some data stream from the air, then we unpack one of data stream via a tool called Jupyter Notebook, which based on the python kernel, and illustrated the result in Figure 4.10. We can see that there is a regular segment, and marked some column lines as the number 1 to 10 in this segment, because they perhaps are the available packet what we exactly want.

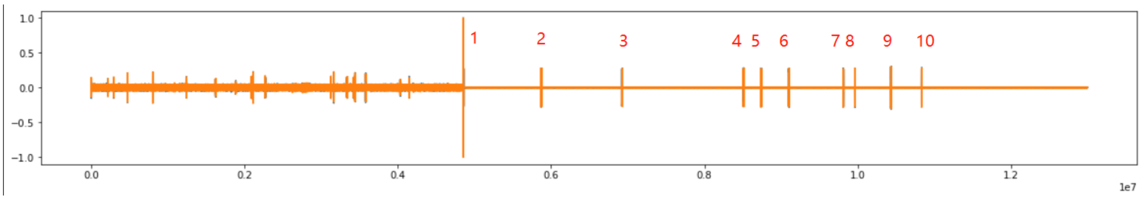


Figure 4.10 The stream of BLE data

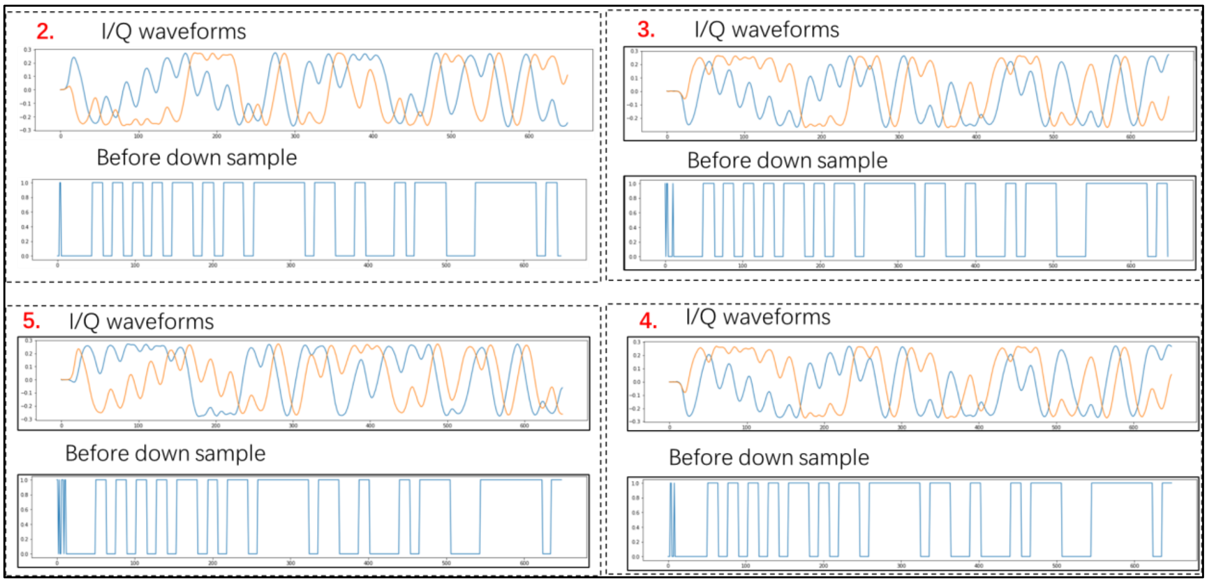


Figure 4.11 The Parts about the Preamble and Access address of BLE packet

Obviously, the signal segment before the number 1 might be the noise by the interference, in this case, we unpacked the packets of number 2, 3, 4, 5 from the whole data stream separately, then roughly located on the part of preamble and access address, the result we can see the Figure 4.11, where include the waveforms of I/Q and before the down sample. In the other word, we can preliminarily sure they are I/Q waveforms that is because we have had a clear perception about the GFSK modulation result from the subsection 4.3.2. On the other hand, these real data waveforms also can proof that they are almost correct about the GFSK modulation part of our theories, simulations and OpenCL programs. Then we made these packets go through the processes of demodulation with OpenCL in FPGA platform of Intel DevCloud, and they are same as the format in Figure 4.11. Hence we are sure in deep that these are the regular data stream instead of noise.

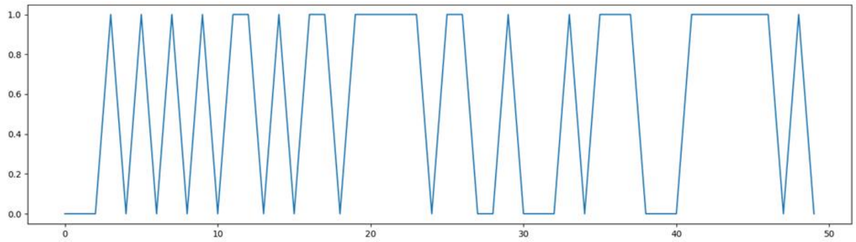


Figure 4.12 After down sample

To judge a packet if it exactly belongs to BLE, the importan thing is we need to check the preamble and access address at the beginning of a data packet, so we pick up one of them to finish the down sampling process. From the Figure 4.12, the sequence is ‘0001010101011010110111110110010001011100011111010’, then we can easily to located the preamble according to the Table 2.2. Regarding with the access address for advertising mode, it has a fixed pattern as ‘0x8E89BED6’ in hexadecimal, or ‘1000 1110 1000 1001 1011 1110 1101 0110’ in binary. But during the sequence which finished the down-sampling process, we didn’t yet find any corresponding structure of access address. Finally, we found the problem [41] is we need one more process to deal with the structure of access address which based on the normal format, that is flipping over the sequence from left to right, we can work the processes as below in the Matlab.

```
>> dec2bin(hex2dec('8E89BED6'),32)
      ans =
            '10001110100010011011111011010110'
>> fliplr(dec2bin(hex2dec('8E89BED6'),32))
      ans =
            '01101011011111011001000101110001'
```

Clearly, we can find the structure ‘01101011011111011001000101110001’ is existing in the sequence we captured, that is the access address of BLE packet. So, in the sequence what we’ve got in Figure 4.13, the preamble is referring to the ‘*Sequence1*’, and the access address is corresponding to ‘*Sequence2*’. Moreover, it also corresponds the relationship between the preamble and access address which shown in Table 2.2.

‘00 $\underset{\text{Seqence1}}{\underline{01010101}}$ $\underset{\text{Seqence2}}{\underline{01101011011111011001000101110001}}$ 11111010’↵

Figure 4.13 Captured sequence

Eventually, through unpacked and analyzed the data stream, we could definitely sure they are the available signals which transmission via the BLE. Meanwhile, put these data packet to go through our programs of OpenCL in FPGA platform of Intel DevCloud, it also worked out the same results after the modulation and demodulation.

CONCLUSION

From this project, we have an integral understanding about the BLE PHY, FPGA and OpenCL, we were adopting the methods which more efficiency, high productive, High-Performance Computing (HPC) and powerful portability in the development of digital signal processing. And these characteristics are the developing trends of communication system for now and afterward. Especially, the concept of parallel programming has shown an excellent capability for the data processing. As the complexity increasing rapidly of data computing unit, it can be a crucial factor for computing processing to improve the computing performance nowadays, and it’s more accommodative for the communicating technology development in the future. For the program verifying and debugging, to compare with the traditional design method, the remote platform is becoming an alternative option for the developers of FPGA programming, because it’s more economic and convenient. However, we have noticed that it will take a long time to compile the OpenCL kernel code by DevCloud, and for this project, it costs around two hours normally. Admittedly, it’s a drawback of OpenCL and DevCloud. Nevertheless, comparing with the above advantages, these shortages look like small and they are acceptable for a whole design as well, it’s no doubt they have a state-of-the-art attribution of technologic and concept.

Besides, according to the final verification, we could also sure the program of OpenCL about the GFSK modulation and demodulation for BLE signal transmission can be realized in FPGA platform of Intel DevCloud, and these processes are exactly the functions of BLE PHY. In a word, we have done the primary objective of this project. Even if there are some insufficiencies what we need to perfect, such as the preamble detecting, and moreover, maybe we also need to verify these in real FPGA platform instead of the remote method, so, the last but not the least, the work is still going on.

APPENDIX 1 HOST CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include "CL/opencl.h"
#include <string.h>
#include "aocl_utils.h"

static const char* kernel_name = "ble";
using namespace std;
using namespace aocl_utils;

static cl_platform_id platform = NULL;
static cl_device_id device = NULL;
static cl_context context = NULL;
static cl_program program = NULL;
static cl_kernel in_kernel = NULL;
static cl_kernel out_kernel = NULL;
static cl_command_queue fpga_to_host_queue = NULL;
static cl_command_queue host_to_fpga_queue = NULL;
static cl_int status = 0;

cl_mem input_buf;
cl_mem output_buf;

int data_size = 0;

bool init()
{
      if(!setCwdToExeDir()) {
            printf("init error\n");
            return false;
    }

      cl_uint num_platforms;
      // Get the OpenCL platform.
      status = clGetPlatformIDs(1, &platform, &num_platforms);
      if(platform == NULL) {
           printf("ERROR: Unable to find Intel(R) FPGA OpenCL platform.\n");
           return false;
      }

      // Query the available OpenCL devices.
      cl_uint num_devices;

      status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device,
                                                               &num_devices);
      if(status != CL_SUCCESS) {
            printf("Failed clGetDeviceIDs.\n");
            return false;
      }

      // Create the context.
      context = clCreateContext(NULL, 1, &device, NULL, NULL, &status);
      if(status != CL_SUCCESS) printf("Failed to create context");

      // Create the command queue.
      host_to_fpga_queue = clCreateCommandQueue(context, device,
                                  CL_QUEUE_PROFILING_ENABLE,  &status);
      if(status != CL_SUCCESS) printf("Failed to create command queue 1\n");

      fpga_to_host_queue = clCreateCommandQueue(context, device,
                                 CL_QUEUE_PROFILING_ENABLE, &status);
      if(status != CL_SUCCESS) printf("Failed to create command queue for
                                                               crc\n");

      std::string binary_file = getBoardBinaryFile("ble", device);
      static const char* ble_kernel_name = "../fpga/ble.aocx";
      printf("\n Using AOCX: %s\n\n", ble_kernel_name);
      program = createProgramFromBinary(context, ble_kernel_name, &device,
                                                                     1);
      if(status != CL_SUCCESS) {
            printf("Failed clCreateProgramWithBinary.\n");
            return false;
      }

    // Build the program that was just created.
      status = clBuildProgram(program, 0, NULL, "", NULL, NULL);
      if(status != CL_SUCCESS) printf("Failed to build program\n");

      const char *kernel_name_input = "Symbols";  // Kernel name, as defined
                                                           in the CL file
      in_kernel = clCreateKernel(program, kernel_name_input, &status);
      if(status != CL_SUCCESS) printf("Failed to create kernel 1\n");
      const char *kernel_name_output = "Output";
      out_kernel = clCreateKernel(program, kernel_name_output, &status);
      if(status != CL_SUCCESS) printf("Failed to create kernel 2\n");
      return true;
}

// Free the resources allocated during initialization
void cleanup()
{
      //free kernel/queue/program/context
      if(in_kernel)                 clReleaseKernel(in_kernel);
      if(out_kernel)                clReleaseKernel(out_kernel);
      if(host_to_fpga_queue)
      clReleaseCommandQueue(host_to_fpga_queue);
      if(fpga_to_host_queue)
      clReleaseCommandQueue(fpga_to_host_queue);
      if(program)             clReleaseProgram(program);
      if(context)             clReleaseContext(context);

      //free in/out buffers
      if(input_buf)                 clReleaseMemObject(input_buf);
      if(output_buf)                clReleaseMemObject(output_buf);
}

// The channel in case for creating the files to store the data
template <class TYPE>
class Channel
{
      TYPE* m_data;
      int m_rd;
      int m_wr;
      int m_size;

public:
      /**
      * Constructor
      */
      Channel(int size)
      {
            m_rd = 0;
            m_wr = 0;
            m_size = size;
            m_data = new TYPE[m_size];
      }

      virtual ~Channel()
      {
            delete [] m_data;
      }

      void write(TYPE v)
      {
           m_data[m_wr] = v;
           m_wr = (m_wr+1) % m_size;
      }

      TYPE read()
      {
           TYPE r = m_data[m_rd];
           m_rd = (m_rd+1) % m_size;
           return r;
      }
};
void filedump_int(char* f, Channel<int>& c)
{
      FILE* fp = fopen(f, "a");
      fprintf(fp, "%d\n", c.read());
      fclose(fp);
```

```
}

int dat_size = 0;
int dat_size_out = 0;

void Readfile(Channel<int>& c)
{
      int datalen=0;
      int data[65535];

      ifstream file("source.txt");

      while( ! file.eof() )

      file>>data[datalen++];
      dat_size = datalen;

      static int idx = 0;
      c.write(data[idx]);
      idx = (idx+1) % (sizeof(data)/sizeof(int));
      file.close();
}
int main()
{
    printf("\nStarting the GFSK process\n\n");

      Channel<int> input_ch=Channel<int>(1);
      Channel<int> output_ch=Channel<int>(1);

      Readfile(input_ch);
      dat_size_out=13*dat_size;

      printf("data_size: %d\n", dat_size);

    unsigned int *input = (unsigned int *)malloc(dat_size*sizeof(int));
    unsigned int *output = (unsigned int *)malloc(dat_size_out*sizeof(int));


    if (!input) printf("error in allocating input\n");
    if (!output) printf("error in allocating output\n");

    printf(" \nInput data: \n");
    for (int i =0; i < dat_size; ++i)
      {
            Readfile(input_ch);
            input[i] = input_ch.read();
            printf("%d  ", input[i]);
            filedump_int("input.txt",input_ch);
      }

    if (!init()) return false;
    input_buf = clCreateBuffer(context, CL_MEM_READ_ONLY, dat_size *
                                 sizeof(unsigned int), NULL, &status);
    if(status != CL_SUCCESS) printf( "Failed to create input buffer\n");

    output_buf = clCreateBuffer(context, CL_MEM_WRITE_ONLY, dat_size_out *
                                   sizeof(unsigned int), NULL, &status);
    if(status != CL_SUCCESS) printf( "Failed to create output_buf\n");

    cl_event write_event;
      cl_event finish_event;

    status = clEnqueueWriteBuffer(host_to_fpga_queue, input_buf, CL_FALSE,
            0, dat_size * sizeof(unsigned int), input, 0, NULL, &write_event);
    if(status!=CL_SUCCESS) printf("Failed to transfer input\n");

    clWaitForEvents(0, &finish_event);
    printf("\n write event status = %d ",status );

    unsigned argi = 0;
      cl_event in_kernel_event;
      cl_event out_kernel_event;

    status = clSetKernelArg(in_kernel, argi++, sizeof(cl_mem), &input_buf);
    if(status!=CL_SUCCESS) printf("Failed to set argument %d on in_kernel\n",
                                                                argi - 1);
    status = clSetKernelArg(in_kernel, argi++, sizeof(cl_int), (void *)
                                                              &dat_size);
      if(status!=CL_SUCCESS) printf("Failed to set argument %d on
                                                 in_kernel\n", argi - 1);
      printf("\n after arg set for in_kernel ");
    argi = 0;
    status = clSetKernelArg(out_kernel, argi++, sizeof(cl_mem), &output_buf);
    if(status!=CL_SUCCESS) printf("Failed to set argument %d on
                                              out_kernel\n", argi - 1);
      status = clSetKernelArg(out_kernel, argi++, sizeof(cl_int), (void *)
                                                          &dat_size_out);
      if(status!=CL_SUCCESS) printf("Failed to set argument %d on
                                                out_kernel\n", argi - 1);
      printf("\n after arg set for out_kernel");

      status = clEnqueueTask(host_to_fpga_queue, in_kernel, 1, &write_event,
                                                       &in_kernel_event);
      if(status!=CL_SUCCESS) printf("Failed to launch in_kernel\n");
      status = clEnqueueTask(fpga_to_host_queue, out_kernel, 0, NULL,
                                                      &out_kernel_event);
      if(status!=CL_SUCCESS) printf("Failed to launch out_kernel\n");
      printf("\n after kernels queued! ");

     status = clEnqueueReadBuffer(fpga_to_host_queue, output_buf, CL_FALSE,
      0, dat_size_out*sizeof(unsigned int), output, 1, &out_kernel_event,
                                                          &finish_event);
    printf("\n\n Read event status = %d \n ",status );
    if (status != CL_SUCCESS) printf("read error\n");
      //printf("\n status %d: CL_SUCCESS \n", status);

    clWaitForEvents(1, &finish_event);

    printf("\n after wait \n\n");
      printf(" Output data: \n");


    for (int i = 7; i< dat_size_out; i+=13)
      {
            // store the data to the external files
            output_ch.write(output[i]);
            filedump_int("output.txt",output_ch);

            //printf directly
            printf("%d  ", output[i]);
      }

      printf("\n\n-------Work is done!!--------\n\n");

      clReleaseEvent(in_kernel_event);
      clReleaseEvent(out_kernel_event);
      clReleaseEvent(finish_event);
      cleanup();
      return 0;
}
```

APPENDIX 2 KERNEL CODE

```
#pragma OPENCL EXTENSION cl_intel_channels : enable

channel int Source_ch;
channel int BinaryNoneZero_ch;
channel int Upsample_ch;
channel float Gaussian_ch;
channel float Multiple_ch;
channel float Integral_I_ch;
channel float Integral_Q_ch;
channel float I_Baseband_ch;
channel float Q_Baseband_ch;

channel float Arctangent_ch;
channel float Unwrap_ch;
channel float Derivative_ch;
channel int Coder_ch;
channel int Preamble_ch;

/*Instructs the compiler to omit logic that generates and dispatches
global, local, and group IDs into the compiled kernel*/
__attribute__((max_global_work_dim(0)))
__kernel  void  Symbols(__global  unsigned  int*  inputdata,  unsigned  int
dat_size)
{
	for (int i=0; i < dat_size; i++)
		write_channel_intel(Source_ch, inputdata[i]);
}

__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__kernel void BinaryNoneZero()
{
	while(1)
	{
		int v = read_channel_intel(Source_ch);

		if (v == 0)
			v = -1;
		write_channel_intel(BinaryNoneZero_ch, v);
	}
}

__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__kernel void UpSample()
{
	while(1)
	{
		int v = read_channel_intel(BinaryNoneZero_ch);
```

```
            for (int i=0; i<13; i++)
                  write_channel_intel(Upsample_ch, v);
      }
}

#define GAUSSIAN_LEN 28
constant float pi = M_PI_F;

__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__kernel void Gaussian()
{
float gaussian_values[GAUSSIAN_LEN];
float hmax = 1.5E6;
float ts = 1E-6;
float OSR = 8;

float acum = 0;

for (int i=0; i < GAUSSIAN_LEN; i++)
      {
                  int k = i - GAUSSIAN_LEN/2;

                   float v = (hmax*ts*k/OSR);
                  gaussian_values[i]=  exp(-(v*v));
                  acum += gaussian_values[i];
      }

for ( int i=0; i < GAUSSIAN_LEN; i++)
            gaussian_values[i] = gaussian_values[i] / acum;

float samples[GAUSSIAN_LEN];

while(1)
      {
            #pragma unroll
            for (int i=0; i < GAUSSIAN_LEN-1; i++)
                  samples[i] = samples[i+1]

            samples[GAUSSIAN_LEN-1]                =                (float)
read_channel_intel(Upsample_ch);

            float acum_ga = 0;

            #pragma unroll
            for (int i=0; i< GAUSSIAN_LEN; i++)
                  acum_ga += samples[i] * gaussian_values[i];
            write_channel_intel(Gaussian_ch, acum_ga);
      }
}
```

```
__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__kernel void Multiple()
{
      while(1)
      {
            float v = read_channel_intel(Gaussian_ch);
            float cv = v*(GAUSSIAN_LEN/5)*pi;
            write_channel_intel(Multiple_ch, cv);
      }
}

__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__kernel void Integrator()
{
      float cv = 0;
      while(1)
      {
            cv += read_channel_intel(Multiple_ch);
            write_channel_intel(Integral_I_ch, cv/(GAUSSIAN_LEN-1));
            write_channel_intel(Integral_Q_ch, cv/(GAUSSIAN_LEN-1));
      }
}

__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__kernel void I_Baseband()
{
      while(1)
      {
            float v = read_channel_intel(Integral_I_ch);
            float c_i = cos(v);
            write_channel_intel(I_Baseband_ch, c_i);
      }
}

__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__kernel void Q_Baseband()
{
      while(1)
      {
            float v = read_channel_intel(Integral_Q_ch);
            float c_q = sin(v);
            write_channel_intel(Q_Baseband_ch, c_q);
      }
}

__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__kernel void Arctangent()
```

```
{
      while(1)
      {
            float v1 = read_channel_intel(I_Baseband_ch);
            float v2 = read_channel_intel(Q_Baseband_ch);
            float cv = atan(v2/v1);
            write_channel_intel(Arctangent_ch, cv);
      }
}

__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__kernel void Unwrap()
{
      float samples[GAUSSIAN_LEN];
      while(1)
      {
            #pragma unroll
            for (int i=0; i < GAUSSIAN_LEN-1; i++)
                        samples[i] = samples[i+1];

            samples[GAUSSIAN_LEN-1] = read_channel_intel(Arctangent_ch);

            float acum = 0;

            #pragma unroll
            for (int i=0; i< GAUSSIAN_LEN; i++)
            {
                  if (((samples[i] - samples[i-1])<pi/2) && ((samples[i] –
                                                     samples[i-1])>-pi/2))
                        acum = samples[i];

                  for (int j=1; j< GAUSSIAN_LEN+1; j++)
                  {
                        if (((samples[i] - samples[i-1]) <-(2*j-1)*pi/2) &&
                                ((samples[i] - samples[i-1]) >-(2*j+1)*pi/2))
                        {
                              samples[i] += j*pi;
                              acum = samples[i];
                        }
                        else if (((samples[i] - samples[i-1]) >(2*j-1)*pi/2)
                             && ((samples[i] - samples[i-1]) <(2*j+1)*pi/2))
                        {
                              samples[i] -= j*pi;
                              acum = samples[i];
                        }
                  }
            }
            write_channel_intel(Unwrap_ch, acum);
      }
}
```

```
__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__kernel void Derivative()
{
      float samples[GAUSSIAN_LEN];
      while(1)
      {
            #pragma unroll
            for (int i=0; i < GAUSSIAN_LEN-1; i++)
                  samples[i] = samples[i+1];

            samples[GAUSSIAN_LEN-1] = read_channel_intel(Unwrap_ch);

            float acum = 0;

            #pragma unroll
            for (int i=0; i<GAUSSIAN_LEN; i++)
                  acum=samples[i]-samples[i-1];

            write_channel_intel(Derivative_ch, acum);
      }
}

__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__kernel void Coder()
{
      while(1)
      {
            float v = read_channel_intel(Derivative_ch);

            unsigned int acum=0;

            if (v>0)
                  acum=1;
            if (v<0)
                  acum=0;
            write_channel_intel(Coder_ch, acum);
      }
}

__attribute__((max_global_work_dim(0)))
__attribute__((autorun))
__kernel void Preamble()
{
      int samples[GAUSSIAN_LEN*13*5];

      while(1)
      {
            #pragma unroll
            for (int i=0; i < GAUSSIAN_LEN*13*5-1; i++)
                  samples[i] = samples[i+1];
            samples[GAUSSIAN_LEN*13*5-1] = read_channel_intel(Coder_ch);

            int acum = 0;

            #pragma unroll
            for (int i=0; i<GAUSSIAN_LEN*13*5; i++)
            {
                  if ((samples[i] ==1 && samples[i+25] ==1 && samples[i+51]
                            ==1 && samples[i+77] ==1 && samples[i+103] ==1 )
                && (samples[i+12] ==0 && samples[i+38] ==0 && samples[i+64]
                                                   ==0 && samples[i+90] ==0 ))
                  {
                        for (int j=0; j<GAUSSIAN_LEN*13*5; j++)
                              acum=samples[j];
                  }
            }
            write_channel_intel(Preamble_ch, acum);
      }
}

__attribute__((max_global_work_dim(0)))
__kernel void Output(__global unsigned int* outputdata, unsigned int
dat_size)
{
      for (int i=0; i<dat_size; i++)
      {
            outputdata[i] = read_channel_intel(Preamble_ch);
      }
}
```

BIBLIOGRAPHY

[1] Fangyan Li. "Multi-engine multi-level simulation for system specification validation and power consumption optimization," Electronics. Universit´e Nice Sophia Antipolis, 2016.

[2] J. Marcel, "New Trends and Forecasts in the 2021 Bluetooth Market Update | Bluetooth® Technology Website", *Bluetooth® Technology Website*, 2021. [Online]. Available: https://www.bluetooth.com/blog/new-trends-and-forecasts-in-the-2021-bluetooth-market-update/. [Accessed: 02- May- 2022].

[3] J. Losaw, "The Importance of Bluetooth in Product Development", *Enventys Partners*, 2020. [Online]. Available: https://enventyspartners.com/blog/bluetooth-product-development-importance/. [Accessed: 04- May- 2022].

[4] I. Grout, *Digital systems design with FPGAs and CPLDs*. Amsterdam: Newnes, 2008.

[5] H. Amano, *Principles and Structures of FPGAs*. Yokohama, Japan: Springer, 2018.

[6] M. Scarpino, *OpenCL in action*. Shelter Island, N.Y: Manning, 2013.

[7] J. Liu and M. Cai, "GFSK modulation for BLE baseband IC design," *2017 International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, 2017, pp. 1-2, doi: 10.1109/EDSSC.2017.8126464.

[8] X. Long et al., "Design of novel digital GFSK modulation and demodulation system for short-range wireless communication application," *2016 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, 2016, pp. 299-302, doi: 10.1109/EDSSC.2016.7785267.

[9] "Introduction to Bluetooth Low Energy (BLE)", *Argenox*. [Online]. Available: https://www.argenox.com/library/bluetooth-low-energy/introduction-to-bluetooth-low-energy-v4-0/. [Accessed: 05- May- 2022].

[10] J. Lee, Y. Su and C. Shen, "A Comparative Study of Wireless Protocols: Bluetooth, UWB, ZigBee, and Wi-Fi," *IECON 2007 - 33rd Annual Conference of the IEEE Industrial Electronics Society*, 2007, pp. 46-51, doi: 10.1109/IECON.2007.4460126.

[11] C. Del-Valle-Soto, L. Valdivia, R. Velázquez, L. Rizo-Dominguez and J. López-Pimentel, "Smart Campus: An Experimental Performance Comparison of Collaborative and Cooperative Schemes for Wireless Sensor Network", *Energies*, vol. 12, no. 16, p. 3135, 2019. Available: 10.3390/en12163135 [Accessed 6 May 2022].

[12] "Bluetooth® Low Energy (BLE) Physical Layer - Developer Help", *Microchipdeveloper.com*. [Online]. Available: https://microchipdeveloper.com/wireless:ble-phy-layer. [Accessed: 07- May- 2022].

[13] A. Ebrahimzadeh and A. Falahati, "Frequency Hopping Spread Spectrum Security Improvement with Encrypted Spreading Codes in a Partial Band Noise Jamming

Environment," *Journal of Information Security*, Vol. 4 No. 1, 2013, pp. 1-6. doi: 10.4236/jis.2013.41001.

[14] "BLE Advertising packet format | BLE Data packet format", *Rfwireless-world.com.* [Online]. Available: https://www.rfwireless-world.com/Terminology/BLE-Advertising-and-Data-Packet-Format.html. [Accessed: 10- May- 2022].

[15] Bluetooth core specification v5.0, *Specification of the Bluetooth System*, Bluetooth®, 2016

[16] H. Shuguang, C. Baoyong and W. Zhihua, "A Mixed-Loop CMOS Analog GFSK Modulator with Tunable Modulation Index," *in IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 6, pp. 547-551, June 2007, doi: 10.1109/TCSII.2007.891755.

[17] G R. Staszewski and P. Balasara, *All-digital frequency synthesizer in deep-submicron CMOS*. Hoboken, NJ: Wiley-Interscience, 2006, p. APPENDIX B.

[18] J. Proakis and M. Salehi, *Digital communications*, 5th ed. Boston: McGraw Hill, 2008.

[19] T. Turletti, "GMSK in a nutshell", *ResearchGate*, 2013. [Online]. Available: https://www.researchgate.net/publication/2575678_GMSK_in_a_nutshell. [Accessed: 06- May- 2022].

[20] T. ŠVEDEK, M. HERCEG, T. MATIĆ, "A Simple Signal Shaper for GMSK/GFSK and MSK Modulator Based on Sigma-Delta Look-up Table," *in Radioengineering*, vol.18, no.2, pp.230-237, June 2009.

[21] David Castells Rufas. "Scalable parallel architectures on reconfigurable platforms," PhD thesis, Universitat Autònoma de Barcelona (UAB), Spain, 2016.

[22] D. Castells Rufas, "RE: The preparing for interview at next Wednesday (11th May)", ganyong.mo@e-campus.uab.cat (May 9, 2022).

[23] A. Boutros and V. Betz, "FPGA Architecture: Principles and Progression," *in IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 4-29, Secondquarter 2021, doi: 10.1109/MCAS.2021.3071607.

[24] S. D. Pable and M. Hasan, "Performance analysis of FPGA interconnect fabric for ultra-low power applications," *in Proceedings of the 2011 International Conference on Communication, Computing & Security (ICCCS '11), association for Computing Machinery,* New York, NY, USA, 210–214, 2011, https://doi.org/10.1145/1947940.1947985.

[25] F. Umer, Z. Marrakchi and H. Mehrez, *Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization,* Springer New York, NY, 2012.

[26] G. Krishna and S. Roy, "Fundamentals of FPGA Architecture," *Advanced Engineering Technical and Scientific Publisher*, Ch. 2, pp. 12-30, 2017.

[27] R. Oshana, *DSP for Embedded and Real-Time Systems.* Newnes, 2012.

[28] F. Blaabjerg, *Control of Power Electronic Converters and Systems*. Academic Press, 2018.

[29] A. Munshi, “The OpenCL Specification Version: 1.2,” Khronos OpenCL Working Group, Nov.14, 2012. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf. [Accessed Apr.26, 2022].

[30] “The OpenCL Specification Version:3.0.11,” Khronos OpenCL Working Group, 2022. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_API.html. [Accessed Apr.26, 2022].

[31] L. Howes and A. Munshi, “The OpenCL Extension Specification Version 2.0,” Khronos OpenCL working Group, Feb.13, 2018. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-2.0-extensions.pdf. [Accessed Apr.30, 2022].

[32] "Migrating OpenCL™ FPGA Designs to SYCL*", *Intel*, Apr.14, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/migrate-opencl-fpga-designs-to-dpcpp/top.html. [Accessed: 01- May- 2022].

[33] “Intel® FPGA SDK for OpenCL™ Standard Edition Getting Started Guide”, *Intel*, Sept.24, 2018. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683678/18-1/standard-edition-getting-started-guide.html. [Accessed: 05- May- 2022].

[34] “Intel® DevCloud", *Intel*, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html. [Accessed: 17- Mar- 2022].

[35] “FPGA-Devcloud/main/Devcloud_Access_Instructions at master · intel/FPGA-Devcloud”, *GitHub*, 2022. [Online]. Available: https://github.com/intel/FPGA-Devcloud/tree/master/main/Devcloud_Access_Instructions#devcloud-access-instructions. [Accessed: 05- Apr- 2022].

[36] “gaussdesign (Gaussian FIR pulse-shaping filter design)”, *Mathworks*, 2022. [Online]. Available: https://uk.mathworks.com/help/signal/ref/gaussdesign.html. [Accessed: 08- Apr- 2022].

[37] “Practice Using Channels with OpenCL™ on Intel® FPGAs Exercise Manual”, *Intel.com*, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/customertraining/OLT/OpenCLChannels/lab.pdf. [Accessed: 10- May- 2022].

[38] Ana_R_intel1, "OpenCL kernel autorun feature", *intel.com*, 2019. [Online]. Available: https://community.intel.com/t5/FPGA-Wiki/OpenCL-kernel-autorun-feature/ta-p/735763. [Accessed: 03- Apr- 2022].

[39] “Altera SDK for OpenCL Programming Guide”, *Intel.com*, Dec.13, 2013. [Online]. Available: https://www.intel.com/content/dam/support/jp/ja/programmable/support-resources/bulk-container/pdfs/literature/hb/opencl-sdk/aocl-programming-guide.pdf. [Accessed: 10- May- 2022].

[40] H. Chen, P. Lin and C. Lin, "A Smart Roadside Parking System Using Bluetooth Low Energy Beacons", in Web, Artificial Intelligence and Network Applications, L. Barolli, M. Takizawa, F. Xhafa and T. Enokido, Ed. Cham: Springer International Publishing, 2019, pp. 471-480.

[41] D. Burnett, "All BLE guides are wrong (including this one)", UC Berkeley, 2018.