EDUCATIONAL TESTBED DESIGN FOR BASEBAND PREDISTORTION LINEARIZATION OF RF POWER AMPLIFIER

A Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Master of Science
in the Graduate School of The Ohio State University

By

Hong Tang, B.S.E.C.E

The Ohio State University
2008

Master’s Examination Committee: Approved by
Professor Patrick Roblin, Advisor
Professor Mohammed Ismail

Advisor
Graduate Program in
Electrical and Computer Engineering
ABSTRACT

The need for linearity is one of the major concerns in the design of modern power amplifiers. Nonlinearities distort the signals being amplified, resulting in imperfect reproduction of the amplified signal; splattering into adjacent channels and causing interference in the detection process. In this thesis, a solution will be proposed and integrated to minimize and clean the imperfect signal as well as remove the unwanted adjacent sideband tones.

Intermodulation effects of PA are identified as core cause of the degradation in linearization performance in a communication system. In general, there are two techniques considered to cancel intermodulation effects and improve linearization performances of RF Power Amplifiers. Firstly, is the Feed-forward technique, which allows the system to measure disturbance in a pre-defined way by using feedback and thus reacts to maintain the desired linearity of the system. Secondly, is the Predistortion technique; predistortion inversely models the PA’s amplitude and phase characteristics, and when combined with the amplifier, produces an overall system thereby canceling the nonlinearity of the amplifier and reducing the amplifier's distortion. Although in real life cases there are also memory effects of the PA due to thermal effect and/or charge storage,
in this thesis, we will only focus on linearization of a memoryless system using the predistortion approach.

Design and implement this memoryless predistortion linearizer will involve two parts. One is the digital part which consist the designing and implementing of the predistortion algorithm using Altera design environment and Stratix II FPGA board. Another is the analog part which requires the constructing of an analog testbed and testing of the linearizer with IQ modulator and RF power amplifier. The project was developed for deployment of the entire testbed in the ECE723 educational lab in ECE department and was tested by student groups.
DEDICATED TO MY PARENTS
ACKNOWLEDGMENTS

I would like to express my sincere thanks to my adviser, Professor Patrick Roblin, for his guidance, support, encouragement, and enthusiasm which made this thesis possible, and for all the teachings I receive from him.

I wish to thank Dr. Dominque Chaillot of C.E.A., France for all his help and mentoring. This work would not have been successfully completed without his invaluable help, mentoring and encouragement.

I am grateful to Professor Mohammed Ismail, his participation in my Master’s Examination Committee and his teachings throughout my years in OSU.

I also thank all my lab members in the Non-linear RF lab, Seok Joo Doo, Xian Cui, Jongsoo Lee, Inwon Suh, Venkatesh Balasubramanian, Xi Yang, and Shashank Mutha for their friendly support and help.

Finally, I wish to thank my parents for all their support and every effort they have put on me.
VITA

May 11, 1984. ................. Born – Shanggao, Jiangxi, China

2006. ...................... B.S. Electrical and Computer Engineering,
                       The Ohio State University

2008. ...................... Master of Science,
                       The Ohio State University

FIELDS OF STUDY

Major Field: Electrical and Computer Engineering


TABLE OF CONTENTS

Abstract .................................................................................................................. ii
Dedication ................................................................................................................. iv
Acknowledgments ...................................................................................................... v
Vita ............................................................................................................................... vi
List of Tables ........................................................................................................... ix
List of Figures .......................................................................................................... x
List of Abbreviations ................................................................................................. xiii
Chapters:

1. Introduction ......................................................................................................... 1
2. Memoryless Digital Predistortion Design and Implementation .................... 7
   2.1 Real-time Design and Implementation ......................................................... 9
      2.1.1 NIOS Interface .................................................................................. 9
      2.1.2 Multi-sine Generator ....................................................................... 11
      2.1.3 Linearization Algorithm .................................................................. 15
      2.1.4 Balancing Block ............................................................................. 19
      2.1.5 Limiter/Saturation Block .................................................................. 24
      2.1.6 Sign-Unsign Block .......................................................................... 26
   2.2 NIOS Integrate Design Environment ......................................................... 27
      2.2.1 IDE Functions and Design Flow ....................................................... 28
      2.2.2 IDE C/C++ Code ........................................................................... 30
   2.3 MATLAB Graphic User Interface ............................................................... 31
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Linearization Techniques</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Balancing Parameter Command Line.</td>
<td>23</td>
</tr>
<tr>
<td>2.2</td>
<td>Limiter Block Logic</td>
<td>25</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Real PA Characteristic</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Digital Predistortion Characteristic</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Linear PA Characteristic</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>System Diagram with Digital Predistorter</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Predistorter Module</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>NIOS Interface Block</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Multi-sine Algorithm</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Linearization Algorithm</td>
<td>18</td>
</tr>
<tr>
<td>2.6</td>
<td>Balancing Algorithm</td>
<td>21</td>
</tr>
<tr>
<td>2.7</td>
<td>Optimized Balancing Algorithm</td>
<td>24</td>
</tr>
<tr>
<td>2.8</td>
<td>Limiter Algorithm</td>
<td>26</td>
</tr>
<tr>
<td>2.9</td>
<td>Sign-Unsign Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>2.10</td>
<td>NIOS IDE Design Flow</td>
<td>29</td>
</tr>
<tr>
<td>2.11</td>
<td>MATLAB GUI</td>
<td>33</td>
</tr>
<tr>
<td>2.12</td>
<td>2-Tone Signal Testing</td>
<td>35</td>
</tr>
<tr>
<td>2.13</td>
<td>4-Tone Signal Testing</td>
<td>35</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>2.14</td>
<td>Limiter Block Testing Result</td>
<td>36</td>
</tr>
<tr>
<td>2.15</td>
<td>2-Tone Signal w/ Virtual PA</td>
<td>38</td>
</tr>
<tr>
<td>2.16</td>
<td>4-Tone Signal w/ Virtual PA</td>
<td>38</td>
</tr>
<tr>
<td>3.1</td>
<td>System Diagram with Analog and Digital Blocks</td>
<td>40</td>
</tr>
<tr>
<td>3.2</td>
<td>Baseband Stage One schematic</td>
<td>42</td>
</tr>
<tr>
<td>3.3</td>
<td>Baseband Stage Two/Stage Three Schematic</td>
<td>43</td>
</tr>
<tr>
<td>3.4</td>
<td>Baseband Stage Four Schematic</td>
<td>44</td>
</tr>
<tr>
<td>3.5</td>
<td>Baseband Circuit Board Layout</td>
<td>45</td>
</tr>
<tr>
<td>3.6</td>
<td>Baseband Circuit Arrangement</td>
<td>46</td>
</tr>
<tr>
<td>3.7</td>
<td>Board 1, Stage 1 and Stage 2</td>
<td>47</td>
</tr>
<tr>
<td>3.8</td>
<td>Board 2, Stage 3 and Stage 4</td>
<td>47</td>
</tr>
<tr>
<td>3.9</td>
<td>Testing Results of First Three Stages of Baseband Circuit</td>
<td>48</td>
</tr>
<tr>
<td>3.10</td>
<td>Testing Results of the Fourth Stage of Baseband Circuit</td>
<td>49</td>
</tr>
<tr>
<td>3.11</td>
<td>FPGA Board</td>
<td>50</td>
</tr>
<tr>
<td>3.12</td>
<td>AD8345 Input/Quadrature Modulator</td>
<td>51</td>
</tr>
<tr>
<td>3.13</td>
<td>Educational Testbed</td>
<td>52</td>
</tr>
<tr>
<td>4.1</td>
<td>Predistortion Linearization Testbed Testing System</td>
<td>55</td>
</tr>
<tr>
<td>4.2</td>
<td>Two-tone RF Output Signal</td>
<td>56</td>
</tr>
<tr>
<td>4.3</td>
<td>Two-tone RF output Linearization</td>
<td>57</td>
</tr>
<tr>
<td>4.4</td>
<td>Two-tone RF output Zoom View</td>
<td>57</td>
</tr>
<tr>
<td>4.5</td>
<td>Four-tone RF output Signal</td>
<td>58</td>
</tr>
<tr>
<td>4.6</td>
<td>RF PA Linearization for Two-tone Signal</td>
<td>59</td>
</tr>
<tr>
<td>4.7</td>
<td>RF PA Linearization for Two-tone Signal, Span = 5 MHz</td>
<td>59</td>
</tr>
</tbody>
</table>
4.8 RF PA Linearization for Four-tone Signal ................. 60
4.9 Proposed Adaptive Predistortion Linearization for 4-Band WCDMA . . 63
LIST OF ABBREVIATIONS

RF: Radio Frequency
PA: Power Amplifier
FPGA: Field Programmable Gate Array
W-CDMA: Wideband Code Division Multiple Access
IQ: In-phase Quadrature
IMD: Intermodulation Distortion
LUT: Look-Up Table
AM: Amplitude Modulation
PM: Phase Modulation
ADS: Advanced Design System
DSP: Digital Signal Processing
BSP: Board Support Package
ADC: Analog to Digital Converter
DAC: Digital to Analog Converter
ACPR: Adjacent Channel Power Ratio
USB: Upper Side Band
LSB: Lower Side Band
IBM: Input Baseband Modulation
OBM: output Baseband Modulation

QPSK: Quadrature Phase Shift Keying

QAM: Quadrature Amplitude Modulation

IDE: Integrate Development Environment

GUI: Graphic User Interface

LO: Local Oscillator
CHAPTER 1

INTRODUCTION

Active linearization has become an uprising area of interest in modern communication systems. Emphasis on higher data rates and spectral efficiency has driven the industry towards digital modulation techniques such as QPSK, 64 QAM, or multi-carrier configurations [7]. The concern of linearity thus became a major consideration when comes to design of any communication systems in general. Nonlinearities of the components within the system cause distortions in the transmitted signal and result in the generation of unwanted signals outside of the desired frequency band. In addition, interferences are placed inside and outside the adjacent channels in the allocated frequency band. Nowadays, as the bandwidth of the modulated signals become wider, radio frequency (RF) power amplifiers (PA) have become a serious bottleneck due to their nonlinear response [15].

A power amplifier is one of the most crucial components in communication systems. The Class A power amplifiers are highly linear components that meet the linearity requirement. However, they are inefficient in converting DC power to RF power. When efficiency is critical, higher efficient yet poor linearity PAs such as Class B, C, D, and E amplifiers must be used. Digital modulation results in signals with a
fluctuating envelope which generates intermodulation (IM) distortion from the power amplifiers [7]. Therefore, reliable linearization techniques need to be applied to those power amplifiers in order to achieve linearity for the entire system. With W-CDMA (wideband code division multiple access), the challenge becomes even greater, as it has to maintain linearity over a wider bandwidth while, at the same time, it has to support a higher peak-to-average power ratio of 3 to 12 dB. Various linearization techniques have already been developed by previous works; Table 1.1 shows some of the commonly used techniques [3].

<table>
<thead>
<tr>
<th>Technique</th>
<th>Correction (1)</th>
<th>Bandwidth (2)</th>
<th>Efficiency</th>
<th>Flexibility (3)</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analog Predistortion</td>
<td>3 to 5 dB</td>
<td>15 to 25 MHz</td>
<td>5 to 8%</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Cross-Cancellation</td>
<td>15 to 20 dB</td>
<td>10 to 20 MHz</td>
<td>10 to 12%</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Feed-Forward</td>
<td>30 dB</td>
<td>25 to 60 MHz</td>
<td>6 to 10%</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>Digital Predistortion</td>
<td>15 to 20 dB</td>
<td>15 to 20 MHz</td>
<td>12 to 14%</td>
<td>High</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Table 1.1: Linearization Techniques

Feed-forward technique is typically used for 2G baseband linearization, and requires the use of a second power amplifier for finding the linearization parameters [15]. Distortion cancellation in the feedforward amplifiers is based on the subtraction of two signals. The degree of cancellation is mainly determined by the amplitude and phase balances of the signals over the bandwidth of interest. However, due to the high peak-to-average ratio (PAR) of the error signal extracted at the first loop, the error amplifier is
easily saturated and the distortion generated by the error amplifier limits the error
cancellation capability of the feedforward amplifier. In addition, the large-size error
amplifier causes various problems of cost, heating and efficiency, etc. [5]

One other linearization technique worth mentioning is the Envelope Elimination
and Restoration (EER) technique. The envelope elimination and restoration technique
theoretically allows the implementation of highly efficient power amplifiers. In
particular, EER method presents the design of a switching power supply, a limiter, and
envelope detectors together to form a feedback system to linearize efficient but nonlinear
RF PAs [17]. The key remaining challenges for a successful implementation of the EER
technique is the efficient implementation of the switching power converter in charge of
amplifying the baseband envelope signal, since bandwidths in the order of several MHz
are expected for the envelopes to be tracked, hence requiring very high switching
frequencies and thus compromising efficiency [16].

Predistortion technique, on the other hand, requires the insertion of a nonlinear
module before the RF power amplifier. The nonlinear module, called the predistorter, has
the inverse response of the PA so the overall response at the output of the PA is linear [3].
An RF predistorter has the distinct advantage of being capable of handling moderate
bandwidths while continuously adjusting for component drift and power level changes
[7]. Compared to the feed-forward linearization, predistortion is a cost-saving technique.
Radio power amplifiers tend to become more non-linear as their power increases towards
their maximum rated output. Predistortion is a way to get more usable power from the
amplifier, without having to build a larger and more expensive amplifier.

The basic idea of baseband digital predistortion (DPD) system is to predistort a
baseband signal before modulation and amplification by the PA, so that the overall performance of the DPD and the PA system is linear [6]. In real life, the characteristic of a PA can be expressed by $V_{out} = f \cdot k \cdot V_{in}$, where $f$ represents the nonlinearity, see Figure 1.1,

![Figure 1.1: Real PA Characteristic](image)

This nonlinearity adversely affects the performance of the communication system. It causes in-band and inter-band distortions, as well as degrades the performance of the receiver in adjacent channels [8]. The function of the RF linearization predistorter is actually to add a predistortion signal before the power amplifier, which is exactly the inverse of the distortion (equal magnitude and negative in phase) caused by the power amplifier, expressed by the equation $V_{out} = f^{-1} \cdot k \cdot V_{in}$, with $f^{-1}$ representing the predistortion factor, see Figure 1.2,
When combining the predistorter with the power amplifier, the inverted signals cancel out the distortion resulting in an overall system with an ideal linear PA characteristic $V_{out} = k*V_{in}$. As shown in Figure below.

Nevertheless, as Figure 1.1 demonstrates, nonlinearities exhibit in the power amplifier cause it to eventually reach saturation. Hence, predistortion is effective up to the saturation point only, in other words, the predistorter can invert the PA characteristic only if its transfer characteristic is invertible [4].

In this thesis, we will try to implement such a baseband digital predistortion
linearization system that will be used for educational purposes in ECE723 lab. A memoryless DPD system is considered in our case. The entire system includes a Multi-tone generator, a predistorter module, DACs, a modulator, and a PA. The preference for this technique is due to the fact that increment in linearity of the RF power amplifier while improve the efficiency and reduce the size and cost of the broadcasting transmitter.

The organization of the thesis is as follow. We will demonstrate in Chapter 2 the design of our real-time digital predistorter along with the multi-tone generator. Testing and simulated results will also be illustrated in the end of the chapter. Chapter 3 introduces the baseband circuit construction and analysis as well as the whole educational testbed. Chapter 4 concludes this thesis with the overall system testing results and with a discussion of possible extension of this project.
CHAPTER 2

MEMORYLESS DIGITAL PREDISTORTION Design and Implementation

There are two parts to consider for the memoryless predistortion testbed, a digital predistorter and an analog part. The digital part is where the main predistortion algorithm is proposed and implemented. The analog part is the integration of hardware components and where the actual distortion of the signals is taking place. As mentioned in the previous chapter, we will thus send an inverse signal using our digital predistorter to cancel the PA distortion to effectively linearize the overall system. In this chapter, we will first express the general ideas of designing this memoryless RF predistorter, its logic and general logic. Then, we shall go into details about the design and implementation of the digital predistorter.

The digital predistorter is based on an Altera Stratix II S60 DSP development board. The detail of the FPGA board is introduced later in Chapter 3. The digital predistorter design can be categorized into three sections: Real-time design, NIOS Integrated Development Environment (IDE), and MATLAB graphic user interface (GUI). Real-time part of the design is conducted using Quartus II. Quartus II is a software tool
for analysis and synthesis of HDL designs, which enables the developer to compile their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer [9]. NIOS IDE allows us to integrate custom logic into the arithmetic logic unit (ALU) by handling the instruction as a macro in C/C++ and communicating with Altera NIOS II processor. Both Quartus II and NIOS IDE are part of the software design suite for Altera DSP boards. The MATLAB GUI is a user friendly interface developed to input and control our predistortion linearization parameters as well as retrieving data from the FPGA.

Real-time design can be further sorted into two parts, the multi-sine generator and the predistortion part, which we will discuss as we advance into the chapter. Figure 2.1 shows our digital predistorter as well as the overall system block diagram.

![Figure 2.1: System Diagram with Digital Predistorter](image)
2.1 Real-time Design and Implementation

Predistortion requires the insertion of a non-linear module before the RF power amplifier. This non-linear module, called the predistorter, is what we will discuss in this section. The entire predistorter module in real-time consists of the following blocks: NIOS interface, Multi-sine generator, Linearization block, Balancing block, Limiter/Saturation, and the Sign-Unsign block. Figure 2.2 shows the arrangement of each block in our predistorter module.

![Figure 2.2: Predistorter Module](image)

2.1.1 NIOS Interface

The NIOS interface is the foremost and essential block for the design. With the embedded builder, it allows us to specify the pin assignments for the Central Processing Unit (CPU). Therefore, enables the use of custom logics to communicate and exchange information with the CPU, in our case, the NIOS II processor to direct the FPGA performance.

NIOS configuration automates board support package (BSP) [12] to assign processor pins needed by the digital predistorter. Also, three PIOs are built to drive a
shared memory block for specification of all the parameters that apply to the multi-sine and the non-linearity correction purposes. The three PIOs are: 7-bit write address, 16-bit write data, and 4-bit write control. Shown in Figure 2.3.

![NIOS Interface Block](image)

**Figure 2.3: NIOS Interface Block**

Inside the Shared Memory Driver, a RAM is used to receive data and address from the NIOS configuration input. NIOS configuration will use these 3 PIOs to drive a total number of 58 output parameters to be employed later by different blocks such as the Multi-sine, Linearization, and Balancing. From Figure 2.3, write data [15..0] provides data values for each parameter output, write address [6..0] marks the address for each of the parameters, and write control [3..0] selects/decides the parameter employment by latter blocks.
2.1.2 Multi-sine Generator

The purpose of the Multi-sine generator is to generate a signal which could reach as high as 16 tones. The advantages of creating a digital multi-sine generator over choosing an external function generator for generating signals are obvious. By using the multi-sine generator, it is much easier to adjust the input parameters via the HP workstation, and at the same time saves a great deal of physical space for our testbed.

Consider a signal

\[ x = A \sin(\omega t + \phi) = I \cos(\phi) + Q \sin(\phi) \quad (2.1) \]

with \( I = A \sin(\omega t) \) and \( Q = A \cos(\omega t) \). This signal I (sine) and signal Q (cosine) is exactly what will be implemented by the multi-sine generator. With the help of an In-phase Quadrature (IQ) modulator to mix/add up the two signals, signal \( x \) can then be forward to the RF power amplifier.

A 16-tone multi-sine indicates that the signal is able to sum up to 16 arbitrary variables of relative frequencies, amplitudes, and phases. Hence, frequency parameters will firstly put through accumulators; the accumulators will integrate the frequencies and make them as an “address” for the Look-Up Table (LUT) to locate sine and cosine. Relative phases are added through adders to each frequency sent to the LUT. Out of the LUT, we can obtain our original \( I = \sin(\omega t + \theta) \) and \( Q = \cos(\omega t + \theta) \) signals. As shown in Figure 2.4.
Figure 2.4: Multi-sine Algorithm
Look-Up Table (LUT)

The LUT is a fundamental building block of the multi-sine generator. But it also has the tendency to occupy a large space of memory bits from the FPGA processor. An LUT itself alone is easy to come up with, but to construct it with an optimizing memory size can be a little tricky. We will demonstrate in the following a “smart” way of build the LUT block.

At first trial, the LUT blocks we built contains 4096 words of memory and 14 bit-width. The LUT stores one complete sine as illustrated. Same thing is done for cosine since they are identical signals with a 90° phase shift.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1016</td>
<td>8198</td>
</tr>
<tr>
<td>1024</td>
<td>8190</td>
</tr>
<tr>
<td>1032</td>
<td>8198</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2046</td>
<td>25</td>
</tr>
<tr>
<td>2047</td>
<td>13</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
</tr>
<tr>
<td>2049</td>
<td>-13</td>
</tr>
<tr>
<td>2050</td>
<td>-25</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3064</td>
<td>-8198</td>
</tr>
<tr>
<td>3072</td>
<td>-8190</td>
</tr>
<tr>
<td>3080</td>
<td>-8198</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>4093</td>
<td>-25</td>
</tr>
<tr>
<td>4094</td>
<td>-13</td>
</tr>
<tr>
<td>4095</td>
<td>0</td>
</tr>
</tbody>
</table>

We know that sinusoid signals are symmetric, for every point inside one period that passes over the center (180°). In our case, each point larger than the 2048th word is inversely symmetrical with the corresponding first half of the table values. For this second approach, we reduced the total LUT words of memory to one half, 2048 words and 12 bit-width. To negate the values in the first half of the LUT, the following logic can be implemented; which we call the one-half sine method:
Therefore, a complete sine signal is obtained via such method. Same thing is done for cosine signals.

Now let us take a look at one half sine figure. Again it is symmetrical. For the third approach we reduce the total words of memory by half once again, reducing it to one fourth of the original data points. For the new LUT, 1024 words of memory are used. Flip horizontally to complete a half sinusoid using now one fourth of the sinusoid signal, then apply the same method used in trial two. The logic is illustrated below. It is called the one-fourth sine method.

```plaintext
for 0 <= address <= 4096
  if address < 1024
    data = data;
  elseif 1024 <= address <= 2048
    counter = 0:1024, counter++;
    address = 1024 – counter;
    data = data;
  else
    data = 0 - data;
  end;
end;
```
Using the one-fourth sine method, we can verify that a total of 8% memory space is occupied from the processor rather than 36% with the first trial. When such, an optimum path is introduced, a great deal of resources is saved.

Out of the LUT, the unmodified I (sine) and Q (cosine) signals are found. Next, the signals are forwarding to the multipliers to multiply with them the incoming amplitude parameters from the NIOS interface. We then respectively sum up the Is and Qs to obtain the final \( I = A \sin(\omega t + \theta) \) and \( Q = A \cos(\omega t + \theta) \) outputs. As shown in Figure 2.4.

### 2.1.3 Linearization Algorithm

Linearization block is a core block for the predistortion module. It is in charge of producing a non-linear signal to compensate the Intermodulation Distortion (IMD) caused by RF PAs in communication systems. In this section, we will concentrate on the development of the linearization theory and its real-time part algorithm.

#### Linearization Theory

Remember the output signal I and Q we obtained using our multi-sine generator,

\[
x = A \sin(\omega t + \phi) = I \cos(\phi) + Q \sin(\phi) \quad (2.1)
\]

Now, with the predistorter, we can inversely modify the phase and amplitude of the input signal to compensate for the phase shifts and amplitude distortions introduced by the PA in our analog testbed. The output of the linearizer then should be,

\[
x = AB \sin(\omega t + \phi + \psi) = I_{pd} \cos(\phi) + Q_{pd} \sin(\phi) \quad (2.2)
\]

where one can easily verify \( I_{pd} \) and \( Q_{pd} \) are given by

\[
\begin{bmatrix}
I_{pd} \\
Q_{pd}
\end{bmatrix} = \begin{bmatrix}
B \cos(\psi) - B \sin(\psi) \\
B \sin(\psi) & B \cos(\psi)
\end{bmatrix} = \begin{bmatrix}
\alpha(t) - \beta(t) \\
\beta(t) & \alpha(t)
\end{bmatrix} * \begin{bmatrix}
I \\
Q
\end{bmatrix} \quad (2.3)
\]
A 3rd order memoryless algorithm uses the envelope \( E \)

\[
E^2 = I^2 + Q^2 \quad (2.4)
\]

to evaluate the predistortion coefficients \( \alpha \) and \( \beta \)

\[
\alpha(t) = 1 + \alpha_3 E^2 \quad (2.5)
\]
\[
\beta(t) = 1 + \beta_3 E^2 \quad (2.6)
\]

This nonlinear equation is used to cancel the intermodulation generated by the 3rd order memoryless nonlinearities. In our case, the coefficients \( \alpha_3 \) and \( \beta_3 \) are both real numbers due to the memoryless characteristic. They are used to minimize the Adjacent Channel Power Ratio (ACPR) and obtain optimization for multi-sine excitations.

Expanding equation (2.3) further and we have

\[
I_{pd} = I + \alpha_3 I E^2 - Q - \beta_3 Q E^2
\]
\[
Q_{pd} = I + \beta_3 I E^2 + Q + \alpha_3 Q E^2
\]

\[
I_{pd} = (I - Q) + (\alpha_3 I E^2 - \beta_3 Q E^2) \quad (2.7)
\]
\[
Q_{pd} = (I + Q) + (\beta_3 I E^2 + \alpha_3 Q E^2) \quad (2.8)
\]

thus, the linearization algorithm can be given by following equation,

\[
I_{pd} = I_{diff} + I_{minus} \quad (2.9)
\]
\[
Q_{pd} = Q_{sum} + Q_{minus} \quad (2.10)
\]

where,

\[
I_{diff} = I - Q
\]
\[
Q_{sum} = I + Q
\]
\[
I_{minus} = \alpha_3 I E^2 - \beta_3 Q E^2
\]
\[
Q_{minus} = \beta_3 I E^2 + \alpha_3 Q E^2
\]
The parameters $I_{\text{diff}}$, $I_{\text{minus}}$, $Q_{\text{sum}}$, and $Q_{\text{minus}}$ are the terms we will be implementing in the following real-time design.

**Real-Time Linearization Algorithm**

The Linearization block involves 4 inputs $I$, $Q$, $\alpha_3$, and $\beta_3$, and 2 outputs $I_{pd}$ and $Q_{pd}$. I and Q outputs from the multi-sine generator we created in previous section are injected to the input of the Linearization block. I and Q signals multiplied with itself to obtain $I^2$ and $Q^2$. Respectively, Q is subtracted from I, and I is added to Q with adder/substrate to find parameters $I_{\text{diff}}$ and $Q_{\text{sum}}$. Note that they will not be used until a few clock cycles later. Forwarding the resulting $I^2$ and $Q^2$ to an adder, thus obtain the envelop $E^2$.

$E^2$ is then respectively multiplied with I, Q, $\alpha_3$, and $\beta_3$, as shown in Figure 2.5. Four outputs result from these multiplication, $\alpha_3 I E^2$, $\beta_3 Q E^2$, $\beta_3 I E^2$, and $\alpha_3 Q E^2$, satisfying the terms we need to use in equations (2.7) and (2.8). $\alpha_3$ and $\beta_3$ are the linearization parameters incoming from the NIOS interface. While I and Q are the same signals as the ones placed in the algorithm before, they must be synchronized because of the real-time nature of the calculation. Next, $\beta_3 Q E^2$ is subtracted to $\alpha_3 I E^2$ to get $I_{\text{minus}}$, and $\beta_3 I E^2$ is added to $\alpha_3 Q E^2$ to get $Q_{\text{minus}}$. This way the desired parameters $I_{\text{diff}}$, $I_{\text{minus}}$, $Q_{\text{sum}}$, and $Q_{\text{minus}}$ are determined. Hence, combine them together with adders to obtain the final result as shown in equations (2.9) and (2.10). The results are directly forwarded to the outputs $I_{pd}$ and $Q_{pd}$ of the block, thus, concluding the design for our Linearization block. One important note to keep in mind when design in real-time, all signals have to be synchronized to prevent operation failure. For a complete illustration of the linearization algorithm, please see Figure 2.5.
Figure 2.5: Linearization Algorithm
2.1.4 Balancing Block

The next element in the module is the Balancing block. This block is used for the purpose of correction of unbalancing distortions in IQ modulator. This IQ modulator which is a crucial component placed before the RF power amplifier, performs the mixture of the I and the Q signals to obtain a RF input (signal $x$ from Equation (2.1)) to be amplified by the PA. A more detailed introduction of the IQ modulator will be given in Chapter 3. However, in real life, such a modulator is not ideal due to limitations in the technology. Degradation in accuracy of the I and Q vectors due to noise, and offsets in Amplitude Modulation (AM) and Phase Modulation (PM) are all significant factors which contribute to the unbalancing of the IQ modulator. Hence, in this section, we will consider the adjustment of two parameters, the amplitude ($\Delta A$) and the phase ($\Delta \phi$), to correct the imbalance of the IQ modulator.

To balance the amplitude and phase respectively, we use the following K-Modeling equation [15], which is actually similar to equation (2.3),

\[
\begin{bmatrix}
I_{ba} \\
Q_{ba}
\end{bmatrix} = \begin{bmatrix}
(1 + \frac{\Delta A}{2})\cos(\frac{\Delta \phi}{2}) & (1 + \frac{\Delta A}{2})\sin(\frac{\Delta \phi}{2}) \\
(1 - \frac{\Delta A}{2})\sin(\frac{\Delta \phi}{2}) & (1 - \frac{\Delta A}{2})\cos(\frac{\Delta \phi}{2})
\end{bmatrix} \begin{bmatrix}
I_{in} \\
Q_{in}
\end{bmatrix}
\text{(2.11)}
\]

Expand equation (2.11) and we have

\[
I_{ba} = (1 + \frac{\Delta A}{2})\cos(\frac{\Delta \phi}{2}) * I_{in} + (1 + \frac{\Delta A}{2})\sin(\frac{\Delta \phi}{2}) * Q_{in}
\text{(2.12)}
\]

\[
Q_{ba} = (1 - \frac{\Delta A}{2})\sin(\frac{\Delta \phi}{2}) * I_{in} + (1 - \frac{\Delta A}{2})\cos(\frac{\Delta \phi}{2}) * Q_{in}
\text{(2.13)}
\]
We will apply the above equations to construct the balancing algorithm.

The Balancing block contains 4 inputs and 2 outputs. $I_{in}$ and $Q_{in}$ are received from the previous Linearization block. $\Delta A$ and $\Delta \phi$ are parameters transmitted from the foremost NIOS interface. First, $\Delta A$ and $\Delta \phi$ are subtracted to an Arithmetic Right Shift (ARS) to get the coefficients $\frac{\Delta A}{2}$ and $\frac{\Delta \phi}{2}$. $\frac{\Delta A}{2}$ is subtracted by the constant 1 and added by 1 respectively to obtain the expression $1 + \frac{\Delta A}{2}$ and $1 - \frac{\Delta A}{2}$. In the same clock moment, $\frac{\Delta \phi}{2}$ is placed through a LUT (identical to the one used in Multi-sine algorithm) to acquire $\sin(\frac{\Delta \phi}{2})$ and $\cos(\frac{\Delta \phi}{2})$ signals. The sine and cosine signals are then multiplied with $1 + \frac{\Delta A}{2}$ and $1 - \frac{\Delta A}{2}$ respectively to obtain the four terms inside the matrix in equation (2.11), as demonstrated in Figure 2.6.

These four terms are fed to four multipliers with multiplication to $I_{in}$ and $Q_{in}$. Again, $I_{in}$ and $Q_{in}$ have to be synchronized with delays before the multiplication. The results are then forwarded to two adders to determine the final balanced I and Q outputs. Please see Figure 2.6 for the design.
Figure 2.6: Balancing Algorithm
Alternative Approach

Nevertheless, all the Balancing algorithm is constructed in real-time part, which without a doubt will cause memory usage in the processor, especially with the insertion of the LUT block. Although the one-fourth sine method (mentioned in section 2.1.2) is applied, the LUT will seize a share of the memory space regardless. In order to optimize the design algorithm and save resource, an alternative approach of constructing the Balancing block is explored.

Take a look at equation (2.12) and (2.13), we can rewrite the equations as,

\[
I_{pd} = \text{Param}_A I_{in} + \text{Param}_B Q_{in}
\]

\[
Q_{pd} = \text{Param}_C I_{in} + \text{Param}_D Q_{in}
\]

with

\[
\text{Param}_A = (1 + \frac{\Delta A}{2}) \cos(\frac{\Delta \varphi}{2}) \tag{2.14}
\]

\[
\text{Param}_B = (1 + \frac{\Delta A}{2}) \sin(\frac{\Delta \varphi}{2}) \tag{2.15}
\]

\[
\text{Param}_C = (1 - \frac{\Delta A}{2}) \sin(\frac{\Delta \varphi}{2}) \tag{2.16}
\]

\[
\text{Param}_D = (1 - \frac{\Delta A}{2}) \cos(\frac{\Delta \varphi}{2}) \tag{2.17}
\]

One can easily notice that Param A, B, C, and D are no more than a simple mathematic expression with coefficients \(\Delta A\) and \(\Delta \varphi\). Therefore, equation (2.14) – (2.17) can be reconstructed in C program inside NIOS IDE. We will demonstrate the role of NIOS IDE in Section 2.2. For balancing purpose, a math.h library is included and the programming commands are shown in Table 2.1, as well as included in the C program in Appendix A.
```c
#include "math.h"

static int deltaA, deltaPhi;
static int Param_A, Param_B, Param_C, Param_D;

static void Balancing(void)
{
    float radian_Phi;
    radian_Phi = (float)(deltaPhi*2*3.141592653589/1024);  // because deltaPhi is a 10 bit definition => 1024 LSB means 360deg
    Param_A = (signed int)((2048+((deltaA-2048)/2))*cos(radian_Phi/2));
    Param_B = (signed int)((2048+((deltaA-2048)/2))*sin(radian_Phi/2));
    Param_C = (signed int)((2048-((deltaA-2048)/2))*sin(radian_Phi/2));
    Param_D = (signed int)((2048-((deltaA-2048)/2))*cos(radian_Phi/2));
}
```

Table 2.1: Balancing Parameter Command Line

Thus, the balancing parameters are built via C program calculation in NIOS IDE and deployed to the NIOS interface, shown in Figure 2.3. We can modify the design in Figure 2.6 by removing the Look-Up Table and various ALU blocks. An optimized algorithm is restructured, shown in Figure 2.7. This new Balancing block contains 6 inputs and 2 outputs with a much simpler real-time algorithm. Hence complete the implementation for the balancing module.
2.1.5 Limiter/Saturation Block

The function of the Limiter block is to limit the dynamic range of the transmitting signal to prevent saturation. The algorithm for the Limiter is relatively simple compared with previous blocks in the predistorter module. Inside the Limiter, an active range is set to force the transmitting signals to travel within the range. In other words, signal entering and living is restricted within a certain peak-to-peak value; it can neither exceed the maximum nor below the minimum. Thus, signal passing through the block is stabilized between the allowable limits to avoid system errors caused by the extremely high impulse. The logic of the Limiter can be summarized in Table 2.2,
for all signal input
    if input > maximum
        output = maximum;
    elseif input < minimum
        output = minimum;
    else
        output = input;
    end;
end;

Table 2.2: Limiter Block Logic

The Limiter involves two 18-bit inputs I[17..0], Q[17..0] and two 14-bit outputs I'[13..0] and Q'[13..0]. The algorithms applied to I and Q signals are identical. We insert two comparators to verify a signal; one compares the maximum value and one compares the minimum value. If the comparator answer is true (out ==1), the result is consequently injected to a multiplexer to determine the final output for the Limiter block. This is shown in Figure 2.8.

The tricky part designer must be aware here is that Quartus is a bit based design tool. Hence, bit base numbers are used instead of decimal numbers. An example is this 18 to 14 bit Limiter implemented. Because the 14-bit output, we are limiting the signal in a range of +/-8191 (2^14 – 1 = 16384 / 2 -1 = 8191). For the comparator, the maximum value is set at 8191, and the minimum value is set at 253953, which is -8191 in an 18 based numerical system. For the multiplexer, the maximum value is set at 8191, and the minimum value is set at 8193, which is -8191 in a 14 based numerical system.
The same algorithm can be applied to limit the signal amplitude to any bit width. As a matter of fact, multiple Limiter blocks are usually placed within a system module in many cases to achieve better results.

### 2.1.6 Sign-Unsign Block

The Sign-Unsign Block is the last block in the predistortion module before going to the analog part, and it is the utterly simplest block within the digital system. The block is used to adjust a signed signal to unsigned as it moves through. We can be easily verify this goal by insertion of a NOT logic gate to the Most Significant Bit (MSB) of the input. The flip-flops are used as delays to synchronize the signals. Please see Figure 2.9.
Outputs from the Sign-Unsign block are directly forwarded to the analog part of the testbed. Thus, with the completion of this block concludes our design for the digital predistorter module. In subsequent sections, we will demonstrate the operation of the NIOS IDE and development of a MATLAB GUI, as well as our testing result for this predistorter module.

2.2 NIOS Integrate Design Environment

NIOS II processor is the second generation software configurable processor designed to target Altera FPGA devices. The NIOS II integrated development environment (IDE) is the primary development tool for the NIOS II family of embedded processors [11]. With a workstation, an Altera FPGA, and a Joint Test Action Group (JTAG) download cable (such as an USB cable), we can construct programs to control and communicate with the NIOS II processor system. All software development tasks can be accomplished within the NIOS IDE, including editing, building, and debugging programs [11].
The NIOS II IDE provides three main functions for our predistorter development:

- Manage the linearization parameters for real-time part
- Allow the use of custom instructions (in C/C++) to program FPGA performance
- Enable the use of custom peripherals (GUI) to integrate with the processor

### 2.2.1 IDE Functions and Design Flow

In Section 2.1, we have demonstrated the creation of predistorter module to eliminate intermodulation distortion for RF PA. But, it is up to the NIOS IDE to provide the rest of the design process. The IDE provides a C/C++ runtime environment for program building; therefore we can integrate our custom logic in C/C++ instructions to handle the real-time part as well as fine-tune the system hardware (FPGA) to meet performance goals [13].

The NIOS program consists of two projects: an application project and a library project. The application project is the C/C++ source code constructed by user, and the library project consists of the BSP automated device component configuration files. NIOS IDE combines these two projects to build an Executable and Linked Format (.elf) File. The executable .elf file is then loaded to a 32 Mb RAM which connects to the NIOS II processor. Hence, the NIOS processor will communicate with the program stored in the RAM to determine the FPGA behaviors. The program design flow is shown in Figure 2.10.
Aside from the functions to specifying the FPGA performance, NIOS IDE also enables us to create a custom peripheral to modify settings that control its behaviors. Although the IDE itself contains an interface, it is neither user friendly nor satisfies our design specifications. The custom peripheral, in our case, a MATLAB GUI is developed to monitor and command the FPGA for predistortion purposes. A more detailed demonstration of the MATLAB GUI will be presented in Section 2.3.
2.2.2 IDE C/C++ Code

NIOS IDE is a C/C++ program based platform in general. It allows us to create, modify, build, run, and debug the user program similar to any other C compilers. Thus, the executable .elf file mentioned above is the basis for the C language settings. The C/C++ program can also be modified and rebuilt without the recompilation of the real-time part. We can reconstruct the application project as long as the NIOS interface is unaltered, making it more flexible to amends. In this section, we will briefly go over the C code and its hierarchy.

The C code consists of a main function and multiple subroutines. The main function is used to call the subroutines as it encounters various “cases”, where the subroutines are the fundamental elements of the program. The subroutines in the program can be further categorized into 3 functions: initialization, write PIO, and read outputs. Initialization subroutine initializes all PIOs and parameters to the processor to be used for linearization uses. The PIO subroutines describe the parameter pattern and assign them to the MATLAB interface. Read output subroutines retrieve the data stored in FPGA and place them in the GUI. Additional subroutine is inserted such as the balancing parameter expressions listed in Table 2.1 to save processor memory. With combination of all those functions, an application program is implemented to compensate our real-time predistorter module. To view the full source C code, please turn to Appendix A.
2.3 MATLAB Graphic User Interface

In order to control the linearization parameters of our digital predistorter, a MATLAB graphic user interface is developed. The construction of the MATLAB GUI consists of two parts, the MATLAB code and the graphic skin. The MATLAB runs in association with the C/C++ application in NIOS IDE. In the code, separate functions are built to state the relation between the .elf stored in the RAM (see Appendix B for the MATLAB code). Hence, the code enables the sending and receiving of data from the PIOs of the processor. In the MATLAB skin, we will place “Callbacks” to call out these functions and put them to use. A demonstration of the MATLAB GUI is shown in Figure 2.11.

In Figure 2.11, we can summarize the GUI from left to right into five portions: associated with initialization, display, multi-sine control, unbalancing correction, and linearization correction. The red “Reboot” button in the initialization window allow us to directly upload the executable .elf file to NIOS. The test buttons monitor the proper working of the program and device; we are able to verify the FPGA board response such as the 7 segment display and LED display by clicking these four test buttons. “Write PIOs” and “Init NIOS” buttons write the initial values to the PIOs. NIOS’s answers will be displayed in the display window. The multi-sine control enables us to send multi-sine parameters to the processor. By using the checkbox options, we can add and remove tones to the generated signal. The rest can be further divided into three parts: frequency control, amplitude control, and phase control, which allows us to write and modify the relative frequencies, amplitudes, and phases separately for each tone. An “Absolute Amplitude” was also inserted to control the overall amplitude of the signal. The
unbalancing correction will adjust the amplitude and phase parameters to correct the unbalanced offset in the IQ modulator (as explain in Section 2.1.4). The linearization correction will allows us to enter linearization coefficients to correct the nonlinearity of the power amplifier and associated distortions, which is the core feature of our project. The read buttons will allow the program to read the data stored in the FPGA and displays it on the GUI.

Having described the MATLAB GUI, we turn now toward the implementation of the digital predistorter. In the last section of this chapter, we shall test our design to demonstrate that it works properly.
Figure 2.11: MATLAB GUI
2.4 Memoryless Predistorter Testing

In this section, we shall present our test result for our real-time predistorter. We will test three algorithms in the predistorter module, the multi-sine generator, the limiter, and the linearization algorithm. The testing will be conducted by using an oscilloscope and a spectrum analyzer.

2.4.1 Multi-sine Testing Result

First, we put the multi-sine generator to the test. Using the MATLAB GUI created, we generate a 2-tone signal and a 4-tone signal. Figure 2.12 shows the 2-tone signal created, with \( \text{Frequency}_1 = 7000 \text{ lsb} \) \((1 \text{ lsb} = 1.5 \text{ KHz})\), \( \text{Frequency}_2 = 8000 \text{ lsb} \), and Absolute Amplitude \( = 2000 \). Figure 2.12 shows (a) the oscilloscope view of the 2-tone signal in the time domain; and (b) the spectrum analyzer view of the signal in frequency domain. In the latter a 2 tone signal is clearly displayed. Figure 2.13 shows the example of a 4-tone signal, with \( \text{Frequency}_1 = 7000 \text{ lsb} \), \( \text{Frequency}_2 = 8000 \text{ lsb} \), \( \text{Frequency}_3 = 9000 \text{ lsb} \), \( \text{Frequency}_4 = 10000 \text{ lsb} \), and an Absolute Amplitude of 2000. Like in the previous case, Figure 2.13 (a) shows the signal in the time domain and 2.13 (b) the signal in frequency domain. By the testing results shown in Figure 2.12 and 2.13, we can easily verify the multi-sine generator works as desired. For a specific outgoing signal, one can modify the parameters of the multi-sine generator and number of its tones using the MATLAB GUI.
Figure 2.12: 2-Tone Signal Testing

(a) Time Domain          (b) Frequency Domain

Figure 2.13: 4-Tone Signal Testing

(a) Time Domain          (b) Frequency Domain
2.4.2 Limiter Testing Result

The main function of the Limiter is to prevent errors due to excess signal amplitude by limiting the dynamic range of the signal. In this section, we first test the result by inputting a signal with low amplitude through the Limiter block, as shown in the left figure in Figure 2.14. Figure 2.14 (a) shows a transmitting 8-tone signal with amplitude that is within the dynamic range of the Limiter. Next, a high amplitude signal is tested for comparison with the previous result. Figure 2.14 (b) shows the 8-tone signal with amplitude exceeding the dynamic range; the signal spectrum is conserved, but with some noise spread in to the lower and upper side band of the signal. This provides evidence that this Limiter block is working properly.

Figure 2.14: Limiter Block Testing Result
2.4.3 Linearization Testing Result

As explained in previous sections, the purpose of the Linearization block is to compensate the distortion caused by the RF power amplifier. For this virtual testing, no physical PAs will be used; we shall verify the block’s ability to linearize a signal with a “virtual PA”. This “virtual PA” is in fact another Linearization block placed before the predistortion module. We will use it to act as a distorter to distort signals, mimicking the behavior of a power amplifier.

Figure 2.15 shows the testing result for a 2-tone signal. As is clearly indicated, Figure 2.15 (a) shows a lower sideband and an upper sideband tones caused by the non-linear distortion of the virtual PA. The two nonlinear tones are located at -2.3 MHz and +2.3 MHz relative to the center frequency at 11.44 MHz. Figure 2.15 (b) shows the successful removal of the spectral regrowth on the lower and upper side band simultaneously yielding about 51 dBc ACPR with the use of the linearization module. Similar to the 2-tone excitations, Figure 2.16 shows a 4-tone signal. Figure 2.16 (a) is the result obtained without adjustment, a total of six inter-band distortion tones (3 lower sidebands at -4, -5.4, -6.9 MHz and 3 upper sidebands at +4, +5.4, +6.9 MHz relative to the center frequency at 12.94 MHz) appeared for the 4-tone signal. As seen in Figure 2.16(b), we are able to reduce the non-linear tones by about 46 dBc ACPR. These testing results show that the Linearization block is working as desired.
(a) before linearization                                   (b) after Linearization

Figure 2.15: 2-Tone Signal w/ Virtual PA

(a) before linearization                                   (b) after Linearization

Figure 2.16: 4-Tone Signal w/ Virtual PA,
2.5 Summary

In Chapter 2, we have introduced the design and implementation of a third order memoryless digital predistorter. We have discussed in details each of the essential block that makes up the predistorter module, the assembly of the C/C++ application in the NIOS IDE and the MATLAB Graphic User Interface used by the digital predistorter. In the last part of this Chapter, we have validated our design by verifying some of the core functions of the predistorter. In conclusion, a memoryless digital predistorter is successfully created. In Chapter 3, we will demonstrate the construction of the analog part including the baseband circuit, IQ modulator, power amplifier as well as other components which constitute the testbed.
CHAPTER 3

ANALOG BASEBAND CIRCUIT AND TESTBED

As discussed earlier, a memoryless predistortion testbed can be categorized into a digital part and an analog part. The previous chapter demonstrated the successful completion of the digital predistorter. In this chapter, the construction of the analog circuit as well as the integration of the testbed shall be discussed. In Figure 3.1, the arrangement of the analog circuit is presented.

![System Diagram with Analog and Digital Blocks](image)

Figure 3.1: System Diagram with Analog and Digital Blocks
3.1 Analog Circuit Construction

As shown in Figure 3.1, the analog part contains the integration of three elements: a basedband circuit, an IQ modulator, and an RF power amplifier. Signals output from the D/A converter of the FGPA are directly feed to the baseband circuits. The four-stage analog baseband circuits were developed for the adaptation of the signals between the FPGA board and the IQ modulator. The first three stages increase the voltage level of the signals coming from the D/A converters and provide DC offset controls to manually adjust the LO leakage in the IQ modulator. The last analog board stage includes additional differential DC offset, change the mode of the IQ signals from single to complementary outputs to drive the differential IQ modulator [15]. The IQ modulator uses the IQ signals to generate a digitally modulated RF signal which is fed to the power amplifier.

3.1.1 Baseband Circuit

The analog baseband circuit consists of 2 identical set (one set for I and one set for Q signals) of four stages, which contains operational amplifiers, and components such as resistors, capacitors and potentiometers to increase the voltage gain and to remove DC offsets. We are grateful to Dr. Dominique Chaillot for developing the schematic and layout of the circuit.

Stage One

The first stage is built based upon an Analog Device (AD) 8009 operational amplifier (op-amp). The op-amp is operating at a voltage supply of +5 V to -5 V. The gain for this op-amp stage is:
Gain = \frac{R_2}{R_1} = \frac{500\,\Omega}{100\,\Omega} = 5

loaded to 50 ohms impedance. In addition, capacitors are placed in various locations for the filtering and decoupling of the power supplies, as shown in Figure 3.2.

Figure 3.2: Baseband Stage One Schematic

Note that the first stage was originally designed to serve two main purposes. One was to change differential inputs to single ended output for the old Texas Instrument (TI) FPGA board. The other purpose was for voltage gain. Since the group is now using the Altera FPGA board for this project, we no longer need the second input SMA in this stage.
Stage Two and Three

Stage two and Stage three are two identical stages based on the AD8009 op-amp. The needs for these two stages arise from the necessity for adjusting the gain and DC offset. The insertion of Stage three is used for production of additional voltage gains and additional DC offset control. The maximum gain available output amplitude can reach ±1.6 volts when driving 50ohms, coding for a full scale input signal. At minimum gain, the output reduces to ±200 mV under the same condition. The offset may be set between ±500 mV and even beyond, depending on the signal [14]. Capacitors are applied for filtering and decoupling purposes. Please refer to Figure 3.3 for detailed schematic.

Figure 3.3: Baseband Stage Two/Stage Three Schematic
Stage Four

The last stage of the baseband circuit, stage four is built based upon an AD 8132 op-amp. The op-amp operates at a supply voltage of +5 V to -5 V, and the gain for this stage equals to 1. The main function of stage four is to provide differential outputs needed to operate the IQ modulator. In addition, it provides the means to control the differential offset to minimize leakage from the local oscillator (LO). Again, various capacitors are placed to filter and decouple the power suppliers. Please refer to Figure 3.4 for detailed schematic.

![Figure 3.4: Baseband Stage Four Schematic](image)

Layout and Actual Circuits

The baseband circuit layout is constructed using Advanced Design System (ADS). The circuit layout consists of a single layer and three tiers with each tier corresponding to
one stage of the circuits. Figure 3.5 shows the board layout in ADS. The first tier represents the first stage, with a fixed voltage gain equals to 5. The second tier is used twice, for stages two and three respectively; it allows the adjustment of gain and the control of DC offset. Because stage two and stage three are identical, instead of inserting of an additional tier to the circuit board, a duplicate board is fabricated to place the twin stages. Uni-polar output from the intermediate tier is applied to the uni-polar input of the last tier to obtain differential outputs needed for the IQ modulator. The LO leakage can be minimized by adjusting the differential offset in this level. The maximum tension applied to the modulator are very much lower than those necessary for the uni-polar output (2nd tier) for applications such IBL or OBL [14].

![Figure 3.5: Baseband Circuit Board Layout](image)

**First Tier:**
Stage 1

**Second Tier:**
Stage 2 & 3

**Third Tier:**
Stage 4

Figure 3.5: Baseband Circuit Board Layout
As mentioned previously, the baseband circuit consists of two sets of circuits for IQ signals, hence, a single set of the circuit contains the location of two boards in series. With signals received from the output of the FPGA in-board D/A converter and forward to a modulator. The arrangement of the circuit boards is shown in Figure 3.6.

![Baseband Circuit Arrangement](image)

Figure 3.6: Baseband Circuit Arrangement

The actual circuits are fabricated based on duroid micro-strip boards with substrate $\varepsilon_r = 2.22$. These boards are made on our premises and with the resources of the MILES laboratory. We thank Mr. Xian Cui for helping with fabricating the board. For adjustment of voltage gain and adjustment of DC offset, potentiometers are used in stage two, stage three, and stage four. SMA connectors are placed at the beginning and the end of each stage; they can be used to validate the proper operation of the stages. These circuits are shown in Figure 3.7 and Figure 3.8.
3.1.2 Baseband Circuit Testing

As is seen in Figure 3.9, a two-tone multi-sine signal is generated for testing the first three stages of the circuit. Figure 3.9 (a) shows the output signal from the end of the D/A converter of FPGA board before feeding to the baseband circuits. Figure 3.9 (b) shows the signal as it travels through stage one of the baseband circuits. The gain of this
stage equals to 5, hence the output signal is significantly larger than the input signal. Figures 3.9 (c) and 3.9 (d) display the two-tone signal as it transmitted through stage two and stage three of the baseband circuit, respectively. The signal strength can be verified using the potentiometers placed on the board. Fig 3.9 (c) shows the signal with voltage gain higher than 1 and Fig 3.9 (d) shows the signal with gain less than 1.

Figure 3.9: Testing Results of First Three Stages of Baseband Circuit
Figure 3.10 shows the signal output from the fourth stage of baseband circuit; differential outputs are obtained in this stage. Fig 3.10 (a) illustrates the dual outputs obtained in time domain: the two signals are in equal magnitude and exact inverse in phase. Fig 3.10 (b) shows the differential output in X-Y coordinate. Fig 3.10 (c) and Fig 3.10 (d) display the spectrum of a one-tone output signal from the positive and negative terminals respectively.

Figure 3.10: Testing Results of the Fourth Stage of Baseband Circuit
3.2 Testbed Integration

With the completion of the baseband circuits, we are ready to proceed with the integration of the entire educational testbed. As illustrated in Figure 3.1, the overall testbed will consists of the integration of the FPGA board, Baseband circuits, an IQ modulator, and an RF power amplifier.

3.2.1 FPGA Board and IQ Modulator

This educational testbed is built centered on the Altera Stratix II S60 DSP (FPGA) development board. This development board includes two in-board D/A converters and two A/D converters. The D/A converters are 14 bits and the A/D converters are 12 bits. The system runs on a 100 MHz clock which can be distributed via synchronizations to the external components. See Figure 3.11.

![FPGA Board Diagram](image-url)

Figure 3.11: FPGA Board
This IQ modulator features two mixers that accept two different baseband frequencies as its input and up-convert them at its output using the local oscillator input. The manipulation of frequency performed by the IQ modulator can be used to move signals between bands, and to encode them [10].

The Analog Device 8345 Input/Quadrature modulator used in this educational testbed is a silicon, monolithic RF modulator with operating frequencies from 700 MHz to 2700 MHz. It operates at a power supply of +5 V with Phase Quadrature error of 0.3 degrees at 2140 MHz and amplitude balance of 0.1 dB. The baseband inputs must remain within a range of 1.2 V peak-to-peaks.

Figure 3.12: AD8345 Input/Quadrature Modulator
3.2.2 Overall System Testbed

The overall system testbed is demonstrated in Figure 3.13. As shows in the picture, an RS232 cable is used for direct communications between the workstation and the FPGA board.

![Figure 3.13: Educational Testbed](image)

3.3 Summary

In this chapter, we have demonstrated the construction of a four-stage analog baseband circuitry. We presented in details the schematics and functions as well as
verification results for each of the baseband stages. Hence, a final RF predistortion linearization educational testbed was assembled by combining the digital predistorter with the analog part. In next chapter, we will conduct the final testing of this predistortion testbed. Measurement results will be presented and analyzed.
CHAPTER 4

LINEARIZATION RESULT AND CONCLUSION

With the completion of the educational baseband predistortion linearization testbed in Chapter 2 and 3, we are now ready to use it. In this chapter, the testbed along with our $3^{rd}$ order predistortion correction algorithm is put to the test with the linearization of an RF power amplifier. The amplified multi-tone signal can thus be linearized at the RF amplifier output, delivering up to 50 dBc ACPR.

4.1 RF Predistortion Linearization Results

The performance of the educational testbed is tested using the resources of the MISES laboratory. An ESG4438C signal generator is used for the generation of an RF input signal with frequency at 900 MHz and amplitude of -6 dBm to drive the vector modulator. The RF power amplifier 8447B we use in the measurement is taken from the educational lab. The 8447B Hewlett-Packard amplifier operates at frequencies from 0.4 to 1.3 GHz, with a gain of 22 dB and on a maximum of 10 V DC. The testing results are
displayed on an HP 8592A spectrum analyzer. The entire system setup is shown in Figure 4.1.

![Predistortion Linearization Testbed Testing System](image)

**Figure 4.1: Predistortion Linearization Testbed Testing System**

### 4.1.1 Testbed RF Output Results

First, being tested are the RF signals output from the IQ modulator before entering the power amplifier. The performance of the RF output is investigated using the multi-tone signal generated by the FPGA multi-sine generator. Figure 4.2 shows a two-tone signal with frequency set up at 3000 lsb and 3500 lsb using the multi-sine controller (in MATLAB GUI), and an absolute amplitude of 2000. The vertical scale is about 10
dB per division, and the horizontal scale is about 1.5 MHz per division. Figure 4.2 (a) shows the two tone signal with the desired upper sideband (USB), the LO leakage and unbalanced lower sideband (LSB) terms; the LSB and USB are about -4.5 MHz and +4.5 MHz relative to the LO. Figure 4.2 (b) shows the reduced LO leakage obtained by adjusting the DC offset in the baseband circuit. The LSB term are reduced to a minimum, leaving only the USB term for testing.

![Diagram](image)

(a) before adjustment  (b) after adjustment

Figure 4.2: Two-tone RF Output Signal

The RF output signals will sometimes experience nonlinearities due to the IQ modulator itself. Figure 4.3 demonstrates the success removal of the 3rd order nonlinearities using the designed predistorter, with linearization parameter $\alpha_3 = 246$ and $\beta_3 = 409$. Figure 4.4 (a) shows a zoom in of the signal with a span of 10 MHz. Figure 4.4 (b) shows a zoom out view of the signal with a span of 100 MHz. As evidenced in
the graph, a highly linear signal can be obtained by slightly reducing the gain in the baseband stage.

Figure 4.3: Two-tone RF output Linearization

(a) before linearization  (b) after linearization

Figure 4.4: Two-tone RF output Zoom View

(a) with 10 MHz span  (b) with 100 MHz span
A four-tone signal is also generated in addition to the two-tone example. With Frequency$_1 = 3000$ lsb, Frequency$_2 = 3500$ lsb, Frequency$_3 = 4000$ lsb, Frequency$_4 = 4500$ lsb, and an Absolute Amplitude of 2000. The resulting output is displayed in Figure 4.5. Both a reduced leakages and a single side band are observed.

![IQ Imbalance and LO Leakage](image)

(a) before adjustment  
(b) after adjustment

Figure 4.5: Four-tone RF output Signal

**4.1.2 RF Power Amplifier Linearization**

Finally, the performance of the educational predistortion testbed is investigated by connecting it to the HP 8447B power amplifier. A two-tone signal is being tested, with Frequency$_1 = 3000$ lsb (1 lsb = 1.5 KHz), Frequency$_2 = 3500$ lsb, and absolute amplitude = 2000. As shown in Figure 4.6, the vertical scale is about 10 dB per division, and the horizontal scale is about 1 MHz per division. With the lower sideband completely removed using the balance correction, we can focus only on the upper sideband. Figure 4.6 (a) shows non-linear tones caused by the intermodulation distortion in the power amplifier. The lower and upper non-linear tones are located at -1.2 MHz.
and +1.2 MHz relative to the center of the two tone signal. As is seen in Figure 4.6 (b), setting the linearization parameter $\alpha_3 = 90$ and $\beta_3 = 150$, the spectral regrowth on both side are completely eliminated simultaneously (for about 51 dBc ACPR). Figure 4.7 display the same two-tone signal with a span of 5 MHz.

![Figure 4.6: RF PA Linearization for Two-tone Signal](image)

(a) before linearization    (b) after linearization

Figure 4.6: RF PA Linearization for Two-tone Signal

![Figure 4.7: RF PA Linearization for Two-tone Signal, Span = 5 MHz](image)

(a) before linearization    (b) after linearization

Figure 4.7: RF PA Linearization for Two-tone Signal, Span = 5 MHz
Again, an additional four-tone signal is being tested. The results are shown in Figure 4.8, with Frequency\(_1\) = 3000 lsb, Frequency\(_2\) = 3500 lsb, Frequency\(_4\) = 4000 lsb, Frequency\(_4\) = 4500 lsb, and an Absolute Amplitude of 3000. With linearization parameter \(\alpha_3 = 106\) and \(\beta_3 = 230\), the non-linear side tones are completely removed (for about 50 dBC ACPR).

![Figure 4.8: RF PA Linearization for Four-tone Signal](image)

(a) before linearization  
(b) after linearization

4.2 Conclusion and Future Work

In this thesis we have focus on the development of an educational purpose baseband predistortion linearization testbed to reduce interferences generated by RF power amplifiers. Contributions were made during the process in several areas and are reviewed below.
A first group of contribution is the real time multi-sine generator. We presented theoretically and experimentally the design of the digital multi-sine generator using the Look-Up Table (LUT). We then proposed a resource saving one-fourth sine method that reduces the total resources of the FPGA significantly.

A second group of contribution is related to the correction of unbalancing of the IQ modulator and our memoryless predistortion algorithm. We presented a linear K-model equation providing the means of correlating together the amplitude ($\Delta A$) and the phase ($\Delta \phi$) of the baseband RF signals in the communication system. We extract this K-model algorithm to modify the IQ imbalance of the IQ modulator. Via such steps, we are able to reduce the IQ imbalance for the LSB and USB terms respectively. Our predistortion algorithm is centered on a 3rd order memoryless linearization theory. An FPGA testbed is developed to test this theory. As demonstrated experimentally, we can obtain simultaneous cancellation of the 3rd order LSB and USB interband spectral regrowth for a multi-tone signal. The straightforward and simplicity nature of this memoryless predistortion theory also make it extremely suitable for educational purposes.

A third group of contribution is concerned with the use of the RS232 port for direct communication between the FPGA and computer workstation. By establishing custom PIOs and specifying pins in the Altera NIOS II interface, we are able to directly control the FPGA performance using only an RS232 cable, as well as sending and retrieving data from the FPGA testbed.

A fourth group of contribution is the development of the MATLAB graphic user interface. The MATLAB GUI provides an easy-to-use platform for users to enter inputting multi-sine, IQ unbalance correction, and nonlinear correction parameters. As
demonstrated, information read from the FPGA can also be displayed via this friendly human machine interface.

As evidenced in previous sections, our proposed baseband linearization algorithm and educational testbed were successfully completed. However, certain limitations and constraints do exist in the system. For example, the maximum absolute amplitude for the two-tone multi-sine signal can only be set at 2000. Amplitudes set higher than 2000 will result in a less effective cancellation of the intermodulation tones. One possible assumption of this phenomenon is that high amplitude signals lead to the trig of memory effect in the PA and/or in the system. As our linearization algorithm is only targeted for memoryless cases, the predistorter is not capable of eliminating non-linear tones including memory effect. A feasible solution to resolve the problem may lead to the modification of the limiter block for further restriction of the amplitude range.

Another extension of the present work is the design and the implementation of the new predistortion linearization theory taking the memory effect into consideration. As discussed in the introduction, current memoryless predistortion schemes can be upgraded to effectively linearize wider bandwidth (WCDMA) signals in 3G standard. In MISES laboratory, works have been published for a frequency selective predistortion 2 band WCDMA signals [2], allowing independent linearization of each band and their interactions. A 128 tone OFDM generator with 20 MHz bandwidth and BPSK, QPSK modulation is already constructed for targeting a 4 band predistortion linearization experiment. Adaptive skin using the Fast Fourier Transform (FFT) theory for the 4 band predistortion is developing in process as right now.
Figure 4.9: Proposed Adaptive Predistortion Linearization for 4-Band WCDMA
APPENDIX A

NIOS IDE C Programming Code

/**************************************************************************
 * NIOS C Program for user interface
 * Module      : NIOS_II
 * Program     : NiosCSoft.c
 * **************************************************************************/

#include "alt_types.h"
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include "system.h"
#include "sys/alt_irq.h"
#include "altera_avalon_pio_regs.h"
#include "altera_avalon_uart_regs.h"
#include "math.h"

#define UART1_BAUD 19200

static void WritePIOs(void);
static void WritePulse(void);
static void WriteBlocPIOs(void);
static void WriteOptions(void);
static void WriteOptions_part(void);
static void Initialize(void);
static void ReadOutputsA(void);
static void ReadOutputsB(void);
static void ReadOutputsC(void);

volatile int edge_capture;
static alt_8 Data[6], PIO[7], NumBloc[2], Bloc[512], Buffer[512];
static int Options;
static int A_Balancing, Phi_Balancing;
static int Unbal_A, Unbal_B, Unbal_C, Unbal_D, Shift_bit;
static int Alpha_2nd_order, Beta_2nd_order, Alpha_3rd_order, Beta_3rd_order, Alpha_4th_order, Beta_4th_order, Alpha_5th_order, Beta_5th_order, A_absolute;

static void Pong2PC(void)
{ printf("Nios: OK (%dBds)",UART1_FREQ/(1+IORD_ALTERA_AVALON_UART_DIVISOR(UART1_BASE)));
}

static void WriteOptions_part(void)
{ int Nbr_Tone;
  if ((Options & 0x0001)!=0 ) { Amplitude_int_A=Amplitude_A; Nbr_Tone=1; }
  else Amplitude_int_A=0; //=1st tone
  if ((Options & 0x0002)!=0 ) { Amplitude_int_B=Amplitude_B; Nbr_Tone=Nbr_Tone+1; }
  else Amplitude_int_B=0; //=2nd tone
  if ((Options & 0x0004)!=0 ) { Amplitude_int_C=Amplitude_C; Nbr_Tone=Nbr_Tone+1; }
  else Amplitude_int_C=0; //=3rd tone
  if ((Options & 0x0008)!=0 ) { Amplitude_int_D=Amplitude_D; Nbr_Tone=Nbr_Tone+1; }
  else Amplitude_int_D=0; //=4th tone
  if ((Options & 0x0010)!=0 ) { Amplitude_int_E=Amplitude_E; Nbr_Tone=Nbr_Tone+1; }
  else Amplitude_int_E=0; //=5th tone
  if ((Options & 0x0020)!=0 ) { Amplitude_int_F=Amplitude_F; Nbr_Tone=Nbr_Tone+1; }
  else Amplitude_int_F=0; //=6th tone
  if ((Options & 0x0040)!=0 ) { Amplitude_int_G=Amplitude_G; Nbr_Tone=Nbr_Tone+1; }
  else Amplitude_int_G=0; //=7th tone
}
if ((Options & 0x0080)!=0 ) { Amplitude_int_H=Amplitude_H; Nbr_Tone=Nbr_Tone+1; } else Amplitude_int_H=0; //8th tone 

if ((Options & 0x0100)!=0 ) { Amplitude_int_I=Amplitude_I; Nbr_Tone=Nbr_Tone+1; } else Amplitude_int_I=0; //9th tone 

if ((Options & 0x0200)!=0 ) { Amplitude_int_J=Amplitude_J; Nbr_Tone=Nbr_Tone+1; } else Amplitude_int_J=0; //10th tone 

if ((Options & 0x0400)!=0 ) { Amplitude_int_K=Amplitude_K; Nbr_Tone=Nbr_Tone+1; } else Amplitude_int_K=0; //11th tone 

if ((Options & 0x0800)!=0 ) { Amplitude_int_L=Amplitude_L; Nbr_Tone=Nbr_Tone+1; } else Amplitude_int_L=0; //12th tone 

if ((Options & 0x1000)!=0 ) { Amplitude_int_M=Amplitude_M; Nbr_Tone=Nbr_Tone+1; } else Amplitude_int_M=0; //13th tone 

if ((Options & 0x2000)!=0 ) { Amplitude_int_N=Amplitude_N; Nbr_Tone=Nbr_Tone+1; } else Amplitude_int_N=0; //14th tone 

if ((Options & 0x4000)!=0 ) { Amplitude_int_O=Amplitude_O; Nbr_Tone=Nbr_Tone+1; } else Amplitude_int_O=0; //15th tone 

if ((Options & 0x8000)!=0 ) { Amplitude_int_P=Amplitude_P; Nbr_Tone=Nbr_Tone+1; } else Amplitude_int_P=0; //16th tone 

if (Nbr_Tone==1) { Shift_bit=7; 
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Shift_bit); IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 61); WritePulse( ); 
} 
if (Nbr_Tone==2) { Shift_bit=8; 
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Shift_bit); IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 61); WritePulse( ); } 
if ((Nbr_Tone>2) && (Nbr_Tone<5) ) { Shift_bit=9; 
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Shift_bit); IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 61); WritePulse( ); 
} 
if ((Nbr_Tone>4) && (Nbr_Tone<9) ) { Shift_bit=10; 
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Shift_bit); IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 61); WritePulse( ); 
} 
if (Nbr_Tone>8) { Shift_bit=11; 
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Shift_bit); IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 61); WritePulse( ); } 

static void WriteOptions(void) 
{ int Pattern; 
sscanf(Data,"%XX",&Pattern); //Conversion String => enter hexadecimal 
if (!strcmp(PIO,"Pio060")) {printf("Nios: 0x%0s --> OPTIONSPIO",Data); 
Options=Pattern; WriteOptions_part( ); } 
}


else  {printf("Nios: PIO %s unknowned in writing ",PIO); usleep(500000); }
}

static void WritePulse(void)
{
    IOWR_ALTERA_AVALONPIO_DATA(W_CONTROL_BASE, 0);
    IOWR_ALTERA_AVALONPIO_DATA(W_CONTROL_BASE, 2);
    IOWR_ALTERA_AVALONPIO_DATA(W_CONTROL_BASE, 3);
    IOWR_ALTERA_AVALONPIO_DATA(W_CONTROL_BASE, 2);
    IOWR_ALTERA_AVALONPIO_DATA(W_CONTROL_BASE, 0);
}

static void Initialize(void)
{
    IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_int_A);
    IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 1);  WritePulse();
    IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_int_B);
    IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 4);  WritePulse();
    IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_int_C);
    IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 7);  WritePulse();
    IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_int_D);
    IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 10); WritePulse();
    IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_int_E);
    IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 13); WritePulse();
    IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_int_F);
    IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 16); WritePulse();
    IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_int_G);
    IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 19); WritePulse();
    IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_int_H);
    IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 22); WritePulse();
    IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_int_I);
    IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 25); WritePulse();
    IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_int_J);
    IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 28); WritePulse();
    IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_int_K);
    IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 31); WritePulse();
    IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_int_L);
    IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 34); WritePulse();
    IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_int_M);
    IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 37); WritePulse();
    IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_int_N);
    IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 40); WritePulse();
    IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_int_O);
    IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 43); WritePulse();
    IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_int_P);
    IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 46); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_A);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 0); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_B);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 3); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_C);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 6); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_D);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 9); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_E);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 12); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_F);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 15); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_G);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 18); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_H);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 21); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_I);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 24); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_J);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 27); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_K);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 30); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_L);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 33); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_M);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 36); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_N);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 39); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_O);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 42); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_P);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 45); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_A);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 2); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_B);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 5); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_C);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 8); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_D);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 11); WritePulse();
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_E);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 14); WritePulse();
static void Balancing(void)
{
    float radian_Phi;
    radian_Phi = (float)(Phi_Balancing*2*3.141592653589/1024);  // because
    Phi_balancing is a 10 bit definition => 1024 lsb means 360deg
    Unbal_A = (signed int)((2048+((A_Balancing-2048)/2))*cos(radian_Phi/2));
    Unbal_B = (signed int)((2048+((A_Balancing-2048)/2))*sin(radian_Phi/2));
    Unbal_C = (signed int)((2048-((A_Balancing-2048)/2))*sin(radian_Phi/2));
    Unbal_D = (signed int)((2048-((A_Balancing-2048)/2))*cos(radian_Phi/2));
}

static void ReadOutputsA(void)


```c
{
    sprintf(Bloc,"%s%5X",Bloc,Frequency_A);       // PIO001
    sprintf(Bloc,"%s%5X",Bloc,Amplitude_int_A);    // PIO002
    sprintf(Bloc,"%s%5X",Bloc,Phase_A);           // PIO003
    sprintf(Bloc,"%s%5X",Bloc,Frequency_B);       // PIO004
    sprintf(Bloc,"%s%5X",Bloc,Amplitude_int_B);   // PIO005
    sprintf(Bloc,"%s%5X",Bloc,Phase_B);           // PIO006
    sprintf(Bloc,"%s%5X",Bloc,Frequency_C);       // PIO007
    sprintf(Bloc,"%s%5X",Bloc,Amplitude_int_C);   // PIO008
    sprintf(Bloc,"%s%5X",Bloc,Phase_C);           // PIO009
    sprintf(Bloc,"%s%5X",Bloc,Frequency_D);       // PIO010
    sprintf(Bloc,"%s%5X",Bloc,Amplitude_int_D);   // PIO11
    sprintf(Bloc,"%s%5X",Bloc,Phase_D);           // PIO012
    sprintf(Bloc,"%s%5X",Bloc,Frequency_E);       // PIO013
    sprintf(Bloc,"%s%5X",Bloc,Amplitude_int_E);   // PIO014
    sprintf(Bloc,"%s%5X",Bloc,Phase_E);           // PIO015
    sprintf(Bloc,"%s%5X",Bloc,Frequency_F);       // PIO016
    sprintf(Bloc,"%s%5X",Bloc,Amplitude_int_F);   // PIO017
    sprintf(Bloc,"%s%5X",Bloc,Phase_F);           // PIO018
    sprintf(Bloc,"%s%5X",Bloc,Frequency_G);       // PIO019
    sprintf(Bloc,"%s%5X",Bloc,Amplitude_int_G);   // PIO020
    sprintf(Bloc,"%s%5X",Bloc,Phase_G);           // PIO021
    sprintf(Bloc,"%s%5X",Bloc,Frequency_H);       // PIO022
    sprintf(Bloc,"%s%5X",Bloc,Amplitude_int_H);   // PIO023
    sprintf(Bloc,"%s%5X",Bloc,Phase_H);           // PIO024
    printf("Nios: 0x%s",Bloc); }

static void ReadOutputsB(void)
{
    sprintf(Bloc,"%s%5X",Frequency_I);       // PIO085
    sprintf(Bloc,"%s%5X",Bloc,Amplitude_int_I); // PIO086
    sprintf(Bloc,"%s%5X",Bloc,Phase_I);       // PIO087
    sprintf(Bloc,"%s%5X",Bloc,Frequency_J);   // PIO088
    sprintf(Bloc,"%s%5X",Bloc,Amplitude_int_J); // PIO089
    sprintf(Bloc,"%s%5X",Bloc,Phase_J);       // PIO090
    sprintf(Bloc,"%s%5X",Bloc,Frequency_K);   // PIO091
    sprintf(Bloc,"%s%5X",Bloc,Amplitude_int_K); // PIO092
    sprintf(Bloc,"%s%5X",Bloc,Phase_K);       // PIO093
    sprintf(Bloc,"%s%5X",Bloc,Frequency_L);   // PIO094
    sprintf(Bloc,"%s%5X",Bloc,Amplitude_int_L); // PIO095
    sprintf(Bloc,"%s%5X",Bloc,Phase_L);       // PIO096
    sprintf(Bloc,"%s%5X",Bloc,Frequency_M);   // PIO097
    sprintf(Bloc,"%s%5X",Bloc,Amplitude_int_M); // PIO098
    sprintf(Bloc,"%s%5X",Bloc,Phase_M);       // PIO099
```

70
static void ReadOutputsC(void)
{
    static char *Bloc = "\000";
    sprintf(Bloc, "%s\%s5X", Bloc, (unsigned int)Alpha_2nd_order);  // PIO114
    sprintf(Bloc, "%s\%s5X", Bloc, (signed int)Alpha_3rd_order);  // PIO118
    sprintf(Bloc, "%s\%s5X", Bloc, (signed int)Beta_3rd_order);  // PIO119
    sprintf(Bloc, "%s\%s5X", Bloc, (signed int)Beta_4th_order);  // PIO121
    sprintf(Bloc, "%s\%s5X", Bloc, (signed int)Beta_5th_order);  // PIO127
    sprintf(Bloc, "%s\%s5X", Bloc, Frequency_A);  // PIO040
    sprintf(Bloc, "%s\%s5X", Bloc, Frequency_O);  // PIO043
    sprintf(Bloc, "%s\%s5X", Bloc, Frequency_P);  // PIO046
    sprintf(Bloc, "%s\%s5X", Bloc, Frequency_W);  // PIO049
    printf("Nios: 0x%s", Bloc); }

static void WritePIOs(void)
{ int Pattern;
  sscanf(Data, "%X", &Pattern);
  if       (!strcmp(PIO,"Pio001")) {printf("Nios answer : 0x%s in Frequency_A ",Data);
    Frequency_A=Pattern; IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Frequency_A); IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 0);
    WritePulse(); }
else if (!strcmp(PIO,"Pio004")) {printf("Nios answer : 0x%s in Frequency_B ",Data);
Frequency_B=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Frequency_B); IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 3);
WritePulse();
}
else if (!strcmp(PIO,"Pio007")) {printf("Nios answer : 0x%s in Frequency_C ",Data);
Frequency_C=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Frequency_C); IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 6);
WritePulse();
}
else if (!strcmp(PIO,"Pio010")) {printf("Nios answer : 0x%s in Frequency_D ",Data);
Frequency_D=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Frequency_D); IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 9);
WritePulse();
}
else if (!strcmp(PIO,"Pio013")) {printf("Nios answer : 0x%s in Frequency_E ",Data);
Frequency_E=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Frequency_E); IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 12);
WritePulse();
}
else if (!strcmp(PIO,"Pio016")) {printf("Nios answer : 0x%s in Frequency_F ",Data);
Frequency_F=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Frequency_F); IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 15);
WritePulse();
}
else if (!strcmp(PIO,"Pio019")) {printf("Nios answer : 0x%s in Frequency_G ",Data);
Frequency_G=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Frequency_G); IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 18);
WritePulse();
}
else if (!strcmp(PIO,"Pio022")) {printf("Nios answer : 0x%s in Frequency_H ",Data);
Frequency_H=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Frequency_H); IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 21);
WritePulse();
}
else if (!strcmp(PIO,"Pio025")) {printf("Nios answer : 0x%s in Frequency_I ",Data);
Frequency_I=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Frequency_I); IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 24);
WritePulse();
}
else if (!strcmp(PIO,"Pio028")) {printf("Nios answer : 0x%s in Frequency_J ",Data);
Frequency_J=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Frequency_J); IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 27);
WritePulse();
}
else if (!strcmp(PIO,"Pio031")) {printf("Nios answer : 0x%s in Frequency_K ",Data);
    Frequency_K=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,Frequency_K);
    IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 30);
    WritePulse();
}
else if (!strcmp(PIO,"Pio034")) {printf("Nios answer : 0x%s in Frequency_L ",Data);
    Frequency_L=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,Frequency_L);
    IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 33);
    WritePulse();
}
else if (!strcmp(PIO,"Pio037")) {printf("Nios answer : 0x%s in Frequency_M ",Data);
    Frequency_M=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,Frequency_M);
    IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 36);
    WritePulse();
}
else if (!strcmp(PIO,"Pio040")) {printf("Nios answer : 0x%s in Frequency_N ",Data);
    Frequency_N=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,Frequency_N);
    IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 39);
    WritePulse();
}
else if (!strcmp(PIO,"Pio043")) {printf("Nios answer : 0x%s in Frequency_O ",Data);
    Frequency_O=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,Frequency_O);
    IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 42);
    WritePulse();
}
else if (!strcmp(PIO,"Pio046")) {printf("Nios answer : 0x%s in Frequency_P ",Data);
    Frequency_P=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,Frequency_P);
    IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 45);
    WritePulse();
}
else if (!strcmp(PIO,"Pio023")) {printf("Nios answer : 0x%s in Amplitude_H ",Data); Amplitude_H=Pattern; WriteOptions_part(); }
else if (!strcmp(PIO,"Pio026")) {printf("Nios answer : 0x%s in Amplitude_I ",Data); Amplitude_I=Pattern; WriteOptions_part(); }
else if (!strcmp(PIO,"Pio029")) {printf("Nios answer : 0x%s in Amplitude_J ",Data); Amplitude_J=Pattern; WriteOptions_part(); }
else if (!strcmp(PIO,"Pio032")) {printf("Nios answer : 0x%s in Amplitude_K ",Data); Amplitude_K=Pattern; WriteOptions_part(); }
else if (!strcmp(PIO,"Pio035")) {printf("Nios answer : 0x%s in Amplitude_L ",Data); Amplitude_L=Pattern; WriteOptions_part(); }
else if (!strcmp(PIO,"Pio038")) {printf("Nios answer : 0x%s in Amplitude_M ",Data); Amplitude_M=Pattern; WriteOptions_part(); }
else if (!strcmp(PIO,"Pio041")) {printf("Nios answer : 0x%s in Amplitude_N ",Data); Amplitude_N=Pattern; WriteOptions_part(); }
else if (!strcmp(PIO,"Pio044")) {printf("Nios answer : 0x%s in Amplitude_O ",Data); Amplitude_O=Pattern; WriteOptions_part(); }
else if (!strcmp(PIO,"Pio047")) {printf("Nios answer : 0x%s in Amplitude_P ",Data); Amplitude_P=Pattern; WriteOptions_part(); }
else if (!strcmp(PIO,"Pio003")) {printf("Nios answer : 0x%s in Phase_A ",Data); Phase_A=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_A);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 2); WritePulse(); }
else if (!strcmp(PIO,"Pio006")) {printf("Nios answer : 0x%s in Phase_B ",Data); Phase_B=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_B);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 5); WritePulse(); }
else if (!strcmp(PIO,"Pio009")) {printf("Nios answer : 0x%s in Phase_C ",Data); Phase_C=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_C);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 8); WritePulse(); }
else if (!strcmp(PIO,"Pio012")) {printf("Nios answer : 0x%s in Phase_D ",Data); Phase_D=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_D);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 11); WritePulse(); }
else if (!strcmp(PIO,"Pio015")) {printf("Nios answer : 0x%s in Phase_E ",Data); Phase_E=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_E);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 14); WritePulse(); }
else if (!strcmp(PIO,"Pio018")) {printf("Nios answer : 0x%s in Phase_F ",Data); Phase_F=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Phase_F);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 17); WritePulse();
    else if (!strcmp(PIO,"Pio021")) {printf("Nios answer : 0x%0s in Phase_G ",Data);
        Phase_G=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_G);
    
    IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 20); WritePulse();
    else if (!strcmp(PIO,"Pio024")) {printf("Nios answer : 0x%0s in Phase_H ",Data);
        Phase_H=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_H);
    
    IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 23); WritePulse();
    else if (!strcmp(PIO,"Pio027")) {printf("Nios answer : 0x%0s in Phase_I ",Data);
        Phase_I=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_I);
    
    IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 26); WritePulse();
    else if (!strcmp(PIO,"Pio030")) {printf("Nios answer : 0x%0s in Phase_J ",Data);
        Phase_J=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_J);
    
    IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 29); WritePulse();
    else if (!strcmp(PIO,"Pio033")) {printf("Nios answer : 0x%0s in Phase_K ",Data);
        Phase_K=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_K);
    
    IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 32); WritePulse();
    else if (!strcmp(PIO,"Pio036")) {printf("Nios answer : 0x%0s in Phase_L ",Data);
        Phase_L=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_L);
    
    IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 35); WritePulse();
    else if (!strcmp(PIO,"Pio039")) {printf("Nios answer : 0x%0s in Phase_M ",Data);
        Phase_M=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_M);
    
    IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 38); WritePulse();
    else if (!strcmp(PIO,"Pio042")) {printf("Nios answer : 0x%0s in Phase_N ",Data);
        Phase_N=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_N);
    
    IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 41); WritePulse();
    else if (!strcmp(PIO,"Pio045")) {printf("Nios answer : 0x%0s in Phase_O ",Data);
        Phase_O=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Phase_O);

IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 44); WritePulse();
else if (!strcmp(PIO,"Pio048")) {printf("Nios answer : 0x%s in Phase_P ",Data);
Phase_P=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Phase_P);

IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 47); WritePulse();
else if (!strcmp(PIO,"Pio049")) {printf("Nios answer : 0x%s in A_Balancing ",Data);
A_Balancing=Pattern; Balancing();
else if (!strcmp(PIO,"Pio050")) {printf("Nios answer : 0x%s in Phi_Balancing ",Data);
Phi_Balancing=Pattern; Balancing();
else if (!strcmp(PIO,"Pio051")) {printf("Nios answer : 0x%s in Alpha_2 ",Data);
Alpha_2nd_order=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Alpha_2nd_order);

IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 52); WritePulse();
else if (!strcmp(PIO,"Pio052")) {printf("Nios answer : 0x%s in Beta_2 ",Data);
Beta_2nd_order=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Beta_2nd_order);

IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 53); WritePulse();
else if (!strcmp(PIO,"Pio053")) {printf("Nios answer : 0x%s in Alpha_3 ",Data);
Alpha_3rd_order=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Alpha_3rd_order);

IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 54); WritePulse();
else if (!strcmp(PIO,"Pio054")) {printf("Nios answer : 0x%s in Beta_3 ",Data);
Beta_3rd_order=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Beta_3rd_order);

IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 55); WritePulse();
else if (!strcmp(PIO,"Pio055")) {printf("Nios answer : 0x%s in Alpha_4 ",Data);
Alpha_4th_order=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Alpha_4th_order);

IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 56); WritePulse();
else if (!strcmp(PIO,"Pio056")) {printf("Nios answer : 0x%s in Beta_4 ",Data);
Beta_4th_order=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Beta_4th_order);

IOWR_ALTERA_AVALON_PIO_DATA(W_ADDRESS_BASE, 57); WritePulse();
else if (!strcmp(PIO,"Pio057")) {printf("Nios answer : 0x%s in Alpha_5 ",Data);
Alpha_5th_order=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,
Alpha_5th_order; IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 58); WritePulse();
else if (!strcmp(PIO,"Pio058")) {printf("Nios answer : 0x%s in Beta_5 ",Data);
Beta_5th_order=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,Beta_5th_order);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 59); WritePulse();
else if (!strcmp(PIO,"Pio059")) {printf("Nios answer : 0x%s in A_absolute ",Data);
A_absolute=Pattern; IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE,A_absolute);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 60); WritePulse();
else if (!strcmp(PIO,"Pio060")) {printf("Nios answer : 0x%s Options",Data);
Options=Pattern; WriteOptions_part();
else {printf("Nios: PIO %s income entry!",PIO); usleep(5000); }
}

static void WriteBlocPIOs(void)
{
int Pattern;
int Num, k;
scanf(NumBloc,"%X",&Num);
if (Num==1) {Bloc[0]="0"; 
strcat(Bloc,Buffer); printf("Nios: %s 0x%s --> Pios",NumBloc,Buffer);
if (Num==6) {k=0;
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Frequency_A=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_A);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 0); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Amplitude_A=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Amplitude_A);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 1); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Phase_A=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_A);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 2); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Frequency_B=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_B);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 3); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Amplitude_B=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Amplitude_B);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 4); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Phase_B=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_B);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 5);
WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Frequency_C=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_C);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 6); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Amplitude_C=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Amplitude_C);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 7); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Phase_C=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_C);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 9); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Frequency_D=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_D);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 8);
WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Amplitude_D=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Amplitude_D);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 10); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Phase_D=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_D);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 11);
WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Frequency_E=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_E);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 12); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Amplitude_E=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_E);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 13); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Phase_E=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Phase_E);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 14);
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Frequency_F=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Frequency_F);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 15); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Amplitude_F=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_F);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 16);
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Phase_F=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Phase_F);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 17); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Frequency_G=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Frequency_G);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 18); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Amplitude_G=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_G);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 19); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Phase_G=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Phase_G);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 20);
WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Frequency_H=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Frequency_H);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 21); WritePulse();
sscanf(Bloc+k,%"5X",&Pattern); k=k+5; Amplitude_H=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_H);
IOWR_ALTERA_AVALONPIO_DATA(W_ADRESS_BASE, 22); WritePulse();
sscanf(Bloc+k,%"5X",&Pattern); k=k+5; Phase_H=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Phase_H);
IOWR_ALTERA_AVALONPIO_DATA(W_ADRESS_BASE, 23);

WritePulse();
sscanf(Bloc+k,%"5X",&Pattern); k=k+5; Frequency_I=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Frequency_I);
IOWR_ALTERA_AVALONPIO_DATA(W_ADRESS_BASE, 24); WritePulse();
sscanf(Bloc+k,%"5X",&Pattern); k=k+5; Amplitude_I=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_I);
IOWR_ALTERA_AVALONPIO_DATA(W_ADRESS_BASE, 25); WritePulse();

sscanf(Bloc+k,%"5X",&Pattern); k=k+5; Phase_I=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Phase_I);
IOWR_ALTERA_AVALONPIO_DATA(W_ADRESS_BASE, 26);

WritePulse();
sscanf(Bloc+k,%"5X",&Pattern); k=k+5; Frequency_J=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Frequency_J);
IOWR_ALTERA_AVALONPIO_DATA(W_ADRESS_BASE, 27); WritePulse();

sscanf(Bloc+k,%"5X",&Pattern); k=k+5; Amplitude_J=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_J);
IOWR_ALTERA_AVALONPIO_DATA(W_ADRESS_BASE, 28); WritePulse();

sscanf(Bloc+k,%"5X",&Pattern); k=k+5; Phase_J=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Phase_J);
IOWR_ALTERA_AVALONPIO_DATA(W_ADRESS_BASE, 29);

WritePulse();
sscanf(Bloc+k,%"5X",&Pattern); k=k+5; Frequency_K=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Frequency_K);
IOWR_ALTERA_AVALONPIO_DATA(W_ADRESS_BASE, 30); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Amplitude_K=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_K);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 31); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Phase_K=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Phase_K);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 32);

WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Frequency_L=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Frequency_L);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 33); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Amplitude_L=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_L);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 34);

WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Phase_L=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Phase_L);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 35);

WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Frequency_M=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Frequency_M);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 36); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Amplitude_M=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Amplitude_M);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 37);

WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Phase_M=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Phase_M);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 38);

WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Frequency_N=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Frequency_N);
IOWR_ALTERA_AVALONPIO_DATA(W_ADDRESS_BASE, 39); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Amplitude_N=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Amplitude_N);
IOWR_ALTERA_AVALON_PIO_DATA
(W_ADRESS_BASE, 40); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Phase_N=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_N);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 41);

WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Frequency_O=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_O);
IOWR_ALTERA_AVALON_PIO_DATA
(W_ADRESS_BASE, 42); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Amplitude_O=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Amplitude_O);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 43);

WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Phase_O=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_O);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 44);

WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Frequency_P=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Frequency_P);
IOWR_ALTERA_AVALON_PIO_DATA
(W_ADRESS_BASE, 45); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Amplitude_P=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Amplitude_P);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 46);

WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Phase_P=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, Phase_P);
IOWR_ALTERA_AVALON_PIO_DATA(W_ADRESS_BASE, 47);

WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; A_Balancing=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, A_Balancing);
IOWR_ALTERA_AVALON_PIO_DATA
(W_ADRESS_BASE, 48); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Phi_Balancing=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Phi_Balancing);
IOWR_ALTERA_AVALONPIO_DATA

(W_ADDRESS_BASE, 49); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Alpha_2nd_order=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Alpha_2nd_order);
IOWR_ALTERA_AVALONPIO_DATA

(W_ADDRESS_BASE, 52); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Beta_2nd_order=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Beta_2nd_order);
IOWR_ALTERA_AVALONPIO_DATA

(W_ADDRESS_BASE, 53); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Alpha_3rd_order=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Alpha_3rd_order);
IOWR_ALTERA_AVALONPIO_DATA

(W_ADDRESS_BASE, 54); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Beta_3rd_order=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Beta_3rd_order);
IOWR_ALTERA_AVALONPIO_DATA

(W_ADDRESS_BASE, 55); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Alpha_4th_order=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Alpha_4th_order);
IOWR_ALTERA_AVALONPIO_DATA

(W_ADDRESS_BASE, 56); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Beta_4th_order=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Beta_4th_order);
IOWR_ALTERA_AVALONPIO_DATA

(W_ADDRESS_BASE, 57); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Alpha_5th_order=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Alpha_5th_order);
IOWR_ALTERA_AVALONPIO_DATA

(W_ADDRESS_BASE, 58); WritePulse();
sscanf(Bloc+k,"%5X",&Pattern); k=k+5; Beta_5th_order=Pattern;
IOWR_ALTERA_AVALONPIO_DATA(W_DATA_BASE, Beta_5th_order);
IOWR_ALTERA_AVALONPIO_DATA

(W_ADDRESS_BASE, 59); WritePulse();
sscanf(Bloc+k,"%5X", &Pattern); k=k+5; A_absolute=Pattern;
IOWR_ALTERA_AVALON_PIO_DATA(W_DATA_BASE, A_absolute);
IOWR_ALTERA_AVALON_PIO_DATA
(W_ADRESS_BASE, 60); WritePulse();
    sscanf(Bloc+k,"%5X", &Pattern); k=k+5; Options=Pattern; WriteOptions_part();
}
}

/* static void TestLEDs(void)
{
    int LedPattern;
    printf("Nios: Test des leds");
    Write2LCD("Test Leds'n"); Write2LCD(Data);
    sscanf(Data,"%X", &LedPattern);                        //Conversion String => enter hexadecimal
    IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, LedPattern>>8);
    usleep(200000);
    IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, LedPattern);
    usleep(200000);
    IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, 0x00);
}

static void handle_button_interrupts(void* context, alt_u32 id)
{
    volatile int* edge_capture_ptr = (volatile int*) context;                        //Cast context to
    edge_capture's type. It is important to keep this to avoid compiler optimization issues.
    *edge_capture_ptr =
    IORD_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE);                        //Store
    the value in the Button's edge capture register in *context.
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0);
} //Reset the Button's edge capture register.

static void init_button_pio()
{
    void* edge_capture_ptr = (void*) &edge_capture;                        //Recast the
    edge_capture pointer to match the alt_irq_register() function prototype.
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0xf);
    //Enable all 4 button interrupts.
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(BUTTON_PIO_BASE, 0x0);
    //Reset the edge capture register.
    alt_irq_register( BUTTON_PIO_IRQ, edge_capture_ptr, handle_button_interrupts );
} //Register the interrupt handler.

static void TestButtons(void)
{
    alt_u8 buttons_tested;
    alt_u8 all_tested;
    int last_tested;
    Write2LCD("Test buttons\n"); Write2LCD("Click buttons !");
    printf("Nios: Appuie sur tous les boutons !");
}
init_button_pio();
buttons_tested=0x0; all_tested=0xf; edge_capture=0; last_tested=0xffff;
while (buttons_tested!=all_tested)
{ if (last_tested==edge_capture) continue;
else
{ last_tested=edge_capture;
switch (edge_capture)
{ case 0x1: buttons_tested=buttons_tested | 0x1; Write2LCD("Buttons 1 OK ");
break;
    case 0x2: buttons_tested=buttons_tested | 0x2; Write2LCD("Buttons 2 OK ");
break;
    case 0x4: buttons_tested=buttons_tested | 0x4; Write2LCD("Buttons 3 OK ");
break;
    case 0x8: buttons_tested=buttons_tested | 0x8; Write2LCD("Buttons 4 OK ");
break;
}}}
//IOWR_ALTERA_AVALON_PIO_IRQ_MASK(BUTTON_PIO_BASE, 0x0); // Disable button interrupts for anything outside this loop.
return; /*

int main()
{ int Commande;
IOWR_ALTERA_AVALON_UART_DIVISOR(UART1_BASE, UART1_FREQ/UART1_BAUD+.5);
while(1)
{
// Write2LCD("nNios ready\n\n");
    scanf("%C",&Commande);
    switch(Commande)
{ case 'a': scanf("%4s",Data); TestLEDs(); break;
case 'b': TestButtons(); break;
case 'c': TestSevenSeg(); break;
case 'd': Pong2PC(); break;
case 'e': scanf("%5s %6s",Data,PIO); WritePIOs(); Initialize(); Balancing(); break;
case 'f': scanf("%1s %s",NumBloc,Buffer); WriteBlocPIOs(); Initialize(); break;
case 'g': ReadOutputsA(); break;
case 'h': ReadOutputsB(); break;
case 'i': ReadOutputsC(); break;
case 'l': scanf("%5s %6s",Data,PIO); WriteOptions(); Initialize(); break;
default: break;}}

85
APPENDIX B

MATLAB Code for Graphic User Interface

function varargout=IHMNios(varargin);
%Begin initialization code - DO NOT EDIT
ui_State=1;
ui_State=struct('ui_Name',mfilename,'ui_Singleton',ui_Singleton,'ui_OpeningFcn',
@IHMNios_OpeningFcn,'ui_OutputFcn',@IHMNios_OutputFcn,'ui_LayoutFcn',[],'ui_Callback',[]);
if nargin & isstr(varargin{1}); gui_State.ui_Callback=str2func(varargin{1}); end;
if nargout; [varargout{1:nargout}]=gui_mainfcn(gui_State, varargin{:}); end;
% try disp(varargin{1}); end;

%-------------------------------------------------------------------------------------------------------------------------------------
function IHMNios_OpeningFcn(hObject, eventdata, handles, varargin);
global s1
handles.output= hObject;
LogBook('LogBook Nios','g',1,handles);
s1=instrfind('Tag','Nios'); if isempty(s1);
s1=serial('COM1','BaudRate',19200,'Tag','Nios','InputBufferSize',512,'OutputBufferSize',512,'TimeOut',10); end;
if strmatch(s1.status,'closed'); fopen(s1); LogBook('Opening COM1','g',0,handles);end;
handles.Extra.hSerial=s1; guidata(hObject, handles);
PBLoadInitFile(hObject, eventdata, handles, handles);

function varargout=IHMNios_OutputFcn(hObject, eventdata, handles);
varargout{1}=handles.output;

%Callbacks----------------------------------------------------------------------------------------------------------------------------
function PBReBootNios(hObject, eventdata, handles);
!NiosShell&
LogBook('Reboot Nios','g',1,handles);
function PBPingNios(hObject, eventdata, handles);
s1=handles.Extra.hSerial; fprintf(s1,'%s','d ');
tic; while s1.BytesAvailable==0 & toc<2; end;
if s1.BytesAvailable~0;
    pause(0.2);
    S=fread(s1,s1.BytesAvailable); SRresponse=char(S'); LogBook(SRresponse,'g',0,handles);
else; LogBook('Nios is not responding!',',r',0,handles); end;

function PBLoadInitFile(hObject, eventdata, handles);
[NomPio,Min,Max,Defaut,NomPioQuartus,Comment]=textread(get(handles.EdLoadInitFile,'String'),'%s%d%d%d%s%s');
handles.Extra.Comment=Comment;
guidata(handles.figure1, handles);

% for k=1:length(Min);
for k=1:60;
    hEdPio=eval(['handles.Ed' NomPio{k}]); set(hEdPio,'String',num2str(Defaut(k)));
    set(hEdPio,'TooltipString',num2str(Min(k)+(Min(k)<0)) ' ?' num2str(Max(k)));
    try hSlidPio=eval(['handles.Slid' NomPio{k}]);
        set(hSlidPio,'Min',Min(k)+(Min(k)<0)); set(hSlidPio,'Max',Max(k));
        set(hSlidPio,'Value',Defaut(k));
    end;
end;

for k=0:15; try Bit=bitget(str2num(get(handles.EdPio060,'String')),k+1); set(eval(['handles.Options' num2str(k)]),'Value',Bit); end; end;
LogBook(['Load Init IHM parameter' get(handles.EdLoadInitFile,'String')],'g',0,handles);
set(handles.PBWritePios,'FontWeight','Bold');
set(handles.PBWriteBlocPios,'FontWeight','Bold');

function PBSaveInitFile(hObject, eventdata, handles);
Comment=handles.Extra.Comment;
for k=1:length(Min); hEdPio=eval(['handles.Ed' NomPio{k}]);
    Default(k)=str2num(get(hEdPio,'String'));
    fid=fopen(get(handles.EdSaveInitFile,'String'),'w');
    for k=1:length(Min); fprintf(fid,'%s %d %d %d %s %s %s %s %s',NomPio{k},Min(k),Max(k),Defaut(k),NomPioQuartus{k},Comment{k}); end;
    fclose(fid);
    LogBook(['Save IHM parameter' get(handles.EdSaveInitFile,'String')],'g',0,handles);

function PBWritePios(hObject, eventdata, handles);
for k=1:60; Data=str2num(get(eval(['handles.Ed' NomPio{k}]),'String'));
Data=Data+(Data<0)*2*abs(Min(k)); Send2Nios(['e ' dec2hex(Data) ' ' NomPio{k}],handles); end;
set(handles.PBWritePios,'FontWeight','Normal');
set(handles.PBWriteBlocPios,'FontWeight','Normal');

function PBWriteBlocPios(hObject, eventdata, handles);
Bloc='';
for k=1:60; Data=str2num(get(eval(['handles.Ed' NomPio{k}]),'String'));
Data=Data+(Data<0)*2*abs(Min(k)); Bloc=[Bloc dec2hex(Data,5)]; end;
set(handles.PBWritePios,'FontWeight','Normal');
set(handles.PBWriteBlocPios,'FontWeight','Normal');

function PBWritePios(hObject, eventdata, handles);
for k=1:60; Data=str2num(get(eval(['handles.Ed' NomPio{k}]),'String'));
Data=Data+(Data<0)*2*abs(Min(k)); Send2Nios(['f 1 ' Bloc(1:50) ' '],handles);
DataPios=Send2Nios(['f 2 ' Bloc(51:100) ' '],handles);
DataPios=Send2Nios(['f 3 ' Bloc(101:150) ' '],handles);
DataPios=Send2Nios(['f 4 ' Bloc(151:200) ' '],handles);
DataPios=Send2Nios(['f 5 ' Bloc(201:250) ' '],handles);
DataPios=Send2Nios(['f 6 ' Bloc(251:300) ' '],handles);
set(handles.PBWritePios,'FontWeight','Normal');
set(handles.PBWriteBlocPios,'FontWeight','Normal');

function PBReadOutputsA(hObject, eventdata, handles);
for k=61:84; set(eval(['handles.Ed' NomPio{k}]),'String',''); end;
DataPios=Send2Nios('g ',handles); DataPios=reshape(DataPios,5,length(DataPios)/5)';
for k=61:84; Data=hex2dec(DataPios(k-60,:)); Data=Data-2*abs(Min(k))*(Data>=abs(Min(k)));
set(eval(['handles.Ed' NomPio{k}]),'String',num2str(Data)); end;

function PBReadOutputsB(hObject, eventdata, handles);
for k=85:108; set(eval(['handles.Ed' NomPio{k}]),'String',''); end;
DataPios=Send2Nios('h ',handles); DataPios=reshape(DataPios,5,length(DataPios)/5)';
for k=85:108; Data=hex2dec(DataPios(k-84,:)); Data=Data-2*abs(Min(k))*(Data>=abs(Min(k)));
set(eval(['handles.Ed' NomPio{k}]),'String',num2str(Data)); end;

function PBReadOutputsC(hObject, eventdata, handles);
for k=109:128; set(eval(['handles.Ed' NomPio{k}]),'String',''); end;
function EdPio(hObject, eventdata, handles);
Pio=get(hObject,'Tag'); Pio=['Pio' Pio(end-2:end)]; hSlideAssocie=eval(['handles.Slid' Pio]);
SData=get(hObject,'String'); try Data=str2num(SData); catch set(hObject,'ForegroundColor','r'); return; end;
if Data<=get(hSlideAssocie,'Max'); set(hSlideAssocie,'Value',Data);
set(hObject,'String',num2str(Data)); WritePio(hObject,handles);
else set(hObject,'ForegroundColor','r');
end;
end;

function SlidPio(hObject, eventdata, handles);
set(hObject,'Enable','off');
Pio=get(hObject,'Tag'); Pio=['Pio' Pio(end-2:end)]; hEditAssocie=eval(['handles.Ed' Pio]);
set(hEditAssocie,'String',num2str(round(get(hObject,'Value'))));
WritePio(hObject,handles);
set(hObject,'Enable','on');
end;

function CBOptions(hObject, eventdata, handles);
WriteOptions(hObject,handles);
end;

function Quit(hObject, eventdata, handles);
LogBook('Closing COM1','g',0,handles); fclose(instrfind('Tag','Nios')); close;}

%%%%SubRoutines---------------------------------------------------------------

function WriteOptions(hObject,handles);
OptionsBits=0;
for k=0:15; try OptionsBits=bitset(OptionsBits,k+1,get(eval(['handles.Options' num2str(k)]),'Value'));
end; end;
Tag=get(handles.EdPio060,'Tag'); NumPio=str2num(Tag(end-2:end));
Pio=NomPio{NumPio};
set(handles.EdPio060,'ForegroundColor','g');
Data=str2num(get(handles.EdPio060,'String'));
Data=Data+(Data<0)*2*abs(Min(NumPio));
Send2Nios(['l ' dec2hex(Data) ' ' Pio],handles);
function WritePio(hEdit,handles);
Tag=get(hEdit,'Tag'); NumPio=str2num(Tag(end-2:end)); Pio=NomPio{NumPio};
set(hEdit,'ForegroundColor','g');
Data=str2num(get(hEdit,'String')); Data=Data+(Data<0)*2*abs(Min(NumPio));
Send2Nios(['e ' dec2hex(Data) ' ' Pio],handles);

function SReponse=Send2Nios(Commande,handles);
s1=handles.Extra.hSerial; fprintf(s1,'%s',Commande);
tic; while s1.BytesAvailable==0 & toc<5; end;
pause(.2);
if s1.BytesAvailable; S=fread(s1,s1.BytesAvailable); SReponse=char(S);
LogBook(SReponse,'g',0,handles); else LogBook('Nios is not responding!','r',0,handles); end;

function Send2Memory(Commande,handles);
global s1
s1=handles.Extra.hSerial; fprintf(s1,'%s',Commande);
function LogBook(Texte,Couleur,Reset,handles);
if Reset; set(handles.LiLogBook,'ForegroundColor',Couleur,Reset,handles);
set(handles.LiLogBook,'String',{Texte datestr(now)}); set(handles.LiLogBook,'Value',2);
else    ListeReponse=get(handles.LiLogBook,'String');
Index=get(handles.LiLogBook,'Value');
    ListeReponse{end+1}=Texte; Index=Index+1;
    set(handles.LiLogBook,'ForegroundColor',Couleur);
set(handles.LiLogBook,'String',ListeReponse); set(handles.LiLogBook,'Value',Index); end;
BIBLIOGRAPHY


91


