BeagleBone Black

Lock-in Amplifier

Sena Gomashie – 357432
24-06-2019
Hanze University of Applied Sciences

Title page

Title:

BeagleBone black – Lock in amplifier

Made by:

Sena N. Gomashie
357432

Place & date:

Groningen
24-06-2019

Contractors:

SRON – Netherlands Institute for Space Research
Heino Smit – H.P.smit@sron.nl
Marcel Dijkstra – M.Dijkstra@sron.nl

Contact:

s.n.gomashie@st.hanze.nl
Phone: +31648552366

Github:

https://github.com/SNGomashie/BBBLockin

Preface

This report represents my six months of work on the BeagleBone Black Lock-in Amplifier. This project was commissioned by SRON Groningen under supervision of the Hanze University of Applied Sciences Groningen.

I have worked on the execution of this project from February to July 2019. I want to thank both Mr. Heino Smit and Mr. Marcel Dijkstra, whenever I would encounter hardships I could ask them for guidance and they would be glad to help. I want to thank them both for their support during this project, they were always thinking with me and I really appreciate that. I want to thank Mr. Dijkstra in particular for constantly thinking with me, and taking time out of his (free) days to help me. I also want to thank Joris van der Vlugt for helping me whenever he could.

Above all I want to thank SRON for their hospitality and for allowing me the opportunity to work for such an amazing institute with very nice people and great atmosphere.

Table of Contents

Summary

SRON is working on Hot Electron Bolometer arrays for the GUSTO balloon telescope. This telescope will fly around Antarctica at a height of 36 kilometers. Here it will measure electromagnetic radiation, which will help scientists with new research. SRON tests these Hot Electron Bolometers arrays with equipment they make themselves. One functionality that is missing from this test equipment is a lock-in amplifier. Lock-in amplifiers can extract a signal from a very noisy environment. SRON is also very interested in using the BeagleBone Black, or more specifically the Programmable Real-time Unit – Industrial Communications Sub System included in the BeagleBone Black' ARM processor, as a multi-channel lock-in amplifier. This leads to the objective of this project: **"To design and implement a digital lock-in amplifier on the Programmable Real time Unit – Industrial Communications Sub-System of the BeagleBone Black"** and our problem statement: **"How can a digital lock-in amplifier be implemented on the Programmable Real time Unit – Industrial Communications Sub-System?"**

The requirements for this project are to create a lock-in amplifier, capable of suppressing noise when the signal-to-noise ratio is between -20 and -30dB. It needs to be able to do this over 4 to 8 channels, with a resolution of 1mV. These requirements are discussed in the third chapter. The fourth chapter is all about understanding the lock-in amplifier algorithm and making recommendations on the hardware to be used in this project. This is done by first taking a mathematical approach, then simulating the algorithm in Python and MATLAB. These simulations will tell us what sample rate and integration time is required to extract a signal in a very noisy environment. We base our recommendations on the results of these simulations.

The decision was made to use the BeagleBone Black as microcontroller since the results from the simulation indicated that it is possible to use the BeagleBone Black for this purpose. The LTC1859 will be used as Analog-to-Digital converter since they have one of these on a development board at SRON and it is capable of sampling fast enough. DDS will be used to generate the reference waves for the lock-in amplifier. This will be done by one of the PRU cores.

The fifth chapter will explore the capabilities of the BeagleBone Black and the Programmable real-time units – Industrial Communications Sub System. This is followed by an explanation of how the reference signals are generated, how the signals are mixed, and the different filters available.

Both the lock-in amplifier, and the numerically controlled oscillator we use to generate the reference signals are tested during the evaluation. The reference signals generated by the numerically controlled oscillator have a SNR of around 50dB, and have a high frequency accuracy. The Lock-in amplifier is capable of suppressing noise in accordance to the requirements. It can sample a single channel at ~40000Hz and 4 channels at ~10000Hz. This is more than enough for our application, but allows the lock-in amplifier to also be used for other applications. These limitations come from the way the RPMsg buffer is read out, and the ADC which is not the fastest possible. Without these limitation, theoretical sampling rates of 50kHz+ are possible.

1. SRON Netherlands Institute for Space Research

1.1 General

SRON Netherlands Institute for space research was founded in 1983 by university groups as 'Stichting RuimteOnderzoek Nederland', changing their name to 'Netherlands Institute for Space Research' in 2005. SRON has a focus on four research topics: astrophysics, exoplanets, earth, and technology, with two expertise groups in: Instrument science and engineering. Working with some of the biggest space organizations worldwide like ESA, NASA, and JAXA has allowed SRON to be on the forefront of the newest research and developments in space research/exploration. The institute has two locations in the Netherlands, one in Groningen and one is Utrecht, although the one in Utrecht will move to Leiden soon.

1.2 Work

SRON is a big player, both nationally and internationally, on the space research playing field. Nationally they give counsel to the Dutch government and coordinate national contributions to international space missions. One of the biggest collaborators nationally is ASTRON, an institute focused on earth-based space research, in contrary to SRON which focusses mostly on space based research. SRON is internationally responsible for shaping expensive missions by defining science cases, developing and proto-typing technologies and implementing space-qualified instrumentation. They also contribute by delivering a great deal of scientific research papers and publications to the international community, helping international space communities further explore the earth, our atmosphere and the universe.

1.3 Organization

With the institution split in two locations, good cooperation and communication is very important. One way they do this is by having a video conference during the coffee break, they do this every other week on Monday. These conferences involve meeting new colleagues, presenting mentionable achievements and just general information about the state of everything. While SRON is a Dutch institute, it enjoys many nationalities within their research teams. A great example is the SRON directorate, being led by dr. Micheal Wise who is born and raised in the United States of America. The institute also has many international PhD students and employees. This makes SRON an multi-lingual company, speaking both Dutch and English within their company structure. With a possibility of enjoying conversations in other languages during coffee break.

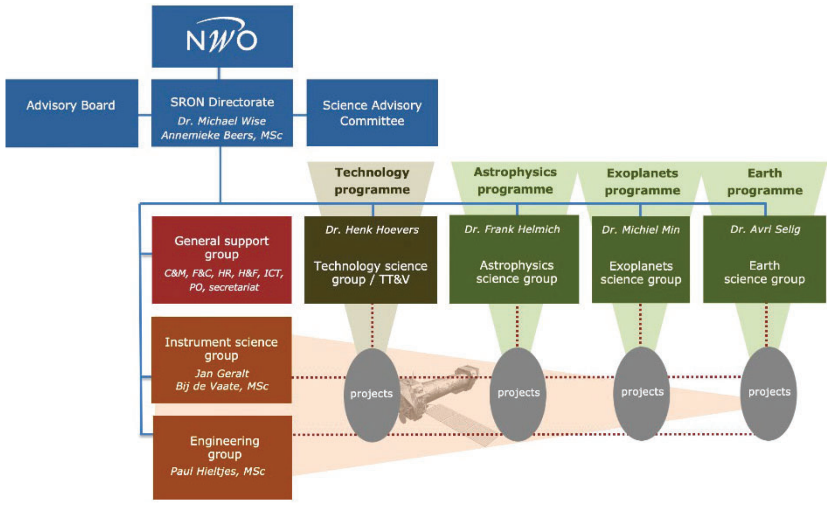


Figure 1: SRON Company organization

SRON is part of a Dutch government organization called NWO, which stands for Nederlandse organisatie voor Wetenschapelijk Onderzoek (Netherlands Organization for Scientific Research). This is an organization in charge of distributing funds to different scientific institutions, such as: ASTRON, CWI, NIOZ, etc. SRON has roughly 260 employees, spread across 7 divisions:

- Astrophysics
- Earth science
- Instrument science
- Engineering
- Exoplanets
- Technology science
- General support

Working for SRON will require at least a Masters or Bachelor, although they do hire people without these degrees for specific practical work such as soldering.

1.4 Company culture

Having only had the chance to visit the SRON location in Groningen, a conclusion can only be made on the culture within this location. Since their cooperation is very close, I would expect the same kind of culture in Utrecht but this is only a guess. SRON has the advantage of being a partially government funded institute with their only obligation being, having to focus on research from which the scientific community and the Dutch society can benefit. This allows there to be next to no financial pressure or outside influence which normally hinders the scientific process. This also allows them to take on multiyear projects such as building a satellite with big organizations, without having to stay profitable.

1.5 summary

SRON is a Dutch institute with a focus on space research in its many forms. It is part of the Dutch national organization NWO. From gathering data, to researching data, to creating high-tech sensors for telescopes, SRON does it all and has done it for years. Without financial pressure from shareholders, they are capable of doing scientific research in its purest form. All within an informal company culture which allows you to focus more on developing yourself and your work instead of small formalities.

2. Introduction

NASA will be launching a balloon mission in Antarctica called GUSTO. This is a telescope made to detect electromagnetic radiation. SRON will be delivering three Hot Electron Bolometer(HEB) arrays for GUSTO. These are devices capable of measuring photons using the properties of super conductive materials to make very accurate energy measurements. These measurements will help scientists discover more about the lifecycle of our galaxy and the formation of star formation clouds.

SRON tests these HEB arrays with equipment they also make themselves. One functionality missing in the test equipment is a lock-in amplifier. Lock-in amplifiers are devices capable of extracting a signal buried in noise orders of magnitude stronger than the original signal. It does this by modulating the signal to a known frequency and then extracting the original signal using mixing and filtering techniques.

SRON is also very interested in using the BeagleBone Black as lock-in amplifier since these are already employed within their organization as measurement systems. These microcomputers have a Programmable Real-time Unit- Industrial Communications Sub System. This includes two, 200 MHz, 32bit processing cores with access to all the peripherals of the AM355x System on Chip. Implementing the Lock-in amplifier on these programmable real-time units would allow the BeagleBone to function as a lock-in amplifier, while still being capable of doing other tasks.

The objective for this project is: **"To design and implement a digital lock-in amplifier on the Programmable Real-time Unit – Industrial Communications Sub-System of the BeagleBone Black"** This includes verifying the performance of the programmable real-time units and possibly designing a daughter board for the BeagleBone Black if time allows for this to happen.

This project has been done in four different phases: Analysis, design, implementation and evaluation. The analysis phase introduces the reader to the project, it does this by giving context and explaining the objective and problem statement. This is followed by a short discussion on the current state-of-the-art of lock-in amplifiers, including an introductory explanation on lock-in amplifiers. This information and information gained from meeting with the stakeholders will define the formal requirements. The design phase is all about finding out what is required to build a properly functioning lock-in amplifier. This is done by doing simulations to get familiar with the lock-in amplifier algorithm. Parameters for a properly functioning lock-in amplifier are extracted from these simulations. These can be used to verify if the BeagleBone Black is capable of functioning as a lock-in amplifier. Different options for microcontrollers, analog-to-digital converters and local oscillators will be discussed after these simulations to make recommendations for the reader. These recommendations will be used during the implementation of this project. The reader will be informed about the different features of the programmable real-time units and how they are combined to create a lock-in amplifier during the implementation phase. The last phase will be used to evaluate the performance of our product and will be used to give recommendations for future projects involving digital lock-in amplifiers or programmable real-time units.

3. Analysis

3.1 Context and objective

SRON is currently working on a balloon mission called 'GUSTO'. This is a balloon telescope made to detect electromagnetic radiation of 1,5; 1,9 and 4,7 terahertz to measure NI, CII and OI emission lines. This experiment will help scientists determine the lifecycle of our galaxy, the formation and deformation of star formation clouds and the behavior of gas streams close to the center of our galaxy. GUSTO will be launched from the south-pole and will circle the continent at a height of 36 kilometers. This mission started in February 2018 and is planned to launch in December 2021.

SRON will deliver three Hot Electron Bolometer(HEB) arrays for GUSTO. Hot Electron Bolometers are devices capable of measuring photons using the properties of super conductive materials to make very accurate energy measurements. The equipment made to test these HEB arrays is made in house by SRON. One functionality missing is the lock-in amplifier. A lock-in amplifier, also known as a phase sensitive detector, is an amplifier capable of recovering a signal buried in noise. It does this by modulating the signal to a known frequency and then applying mixing and filtering techniques to recover the original signal. SRON wants a digital lock-in amplifier implemented on the BeagleBone Black. They use the BeagleBone Black internally for their measurement systems. These open-source development boards have an AM335x ARM Cortex-A8 processor. Build into these processors is the Programmable Real-time Unit – Industrial Communication Sub-System(PRU-ICSS). This consists of two 32-bit Programmable Real time Units(PRU), 8 Kb data RAM per PRU, 12 Kb shared RAM and access to all the peripherals on the AM335x System-on-Chip. This includes but is not limited to the Multi-channel Serial Peripheral Interface(McSPI), General Purpose Input/Output(GPIO) and the Universal asynchronous receiver-transmitter(UART). The goal of this project is not only to create a properly functioning lock-in amplifier, SRON also want to test the capabilities of the PRU-ICSS.

This leads to our objective: **"To design and implement a digital lock-in amplifier on the Programmable Real time Unit – Industrial Communications Sub-System of the BeagleBone Black"**

This includes testing the suitability of the PRU-ICSS for real-time tasks and possibly designing a daughter board (Cape) for the BeagleBone Black if times allows it.

3.2 Problem statement

The problem statement is:

"How can a digital lock-in amplifier be implemented on the Programmable Real time Unit – Industrial Communications Sub-System?"

The problem statement has been split into several sub-questions that have to be answered before the objective can be reached:

- How do Lock-in amplifiers work?
- How does the PRU-ICSS work?
- How does communication between the ARM processor and PRU-ICSS work?
- Is the PRU-ICSS capable of functioning as a lock-in amplifier?
- Is the PRU-ICSS suitable for other real time tasks

3.3 State-of-the-art

This paragraph aims to give the reader a basic understanding of lock-in amplifiers, while discussing past and current advancements in the field.

Figure 2: Optical chopper scitec 340CD

A lock-in amplifier is a type of amplifier commonly used in engineering and science. It is an electrical instrument capable of recovering signal amplitude and phase from extremely noisy environments. It does this by modulating the signal from the experiment to a known frequency. This modulation can be done by, for example, interrupting a laser with a chopper (see Figure 2). After the modulation, homodyne/heterodyne detection is employed together with low-pass filtering (Figure 3) to recover the signal. Mixing the modulated signal with a reference signal will result in a signal with frequencies at the sum and difference of the two frequencies. When the frequencies of the modulated signal and reference signal are equal they will create a signal consisting of signals at DC and at two times the original frequency. A low-pass filter removes all frequencies above DC. The remaining signal will have the same amplitude as the original signal, as long as the two signals are in-phase. This type of lock-in amplifier can be seen in Figure 3-A. The bigger the phase difference, the more error will be introduced into the signal. The influence of phase difference can be eliminated by using a dual-phase lock-in amplifier. This type of lock-in amplifier is shown in Figure 3-B. Dual-phase lock-in amplifiers mix the modulated signal with two sinusoids, both at the reference frequency and one phase shifted by 90 degrees. This creates an In-phase and Quadrature signal, these are also known as I & Q or X & Y(To prevent confusion I & Q will be used for the duration of this report). This allows for measurement of not only the correct amplitude but also the phase difference.

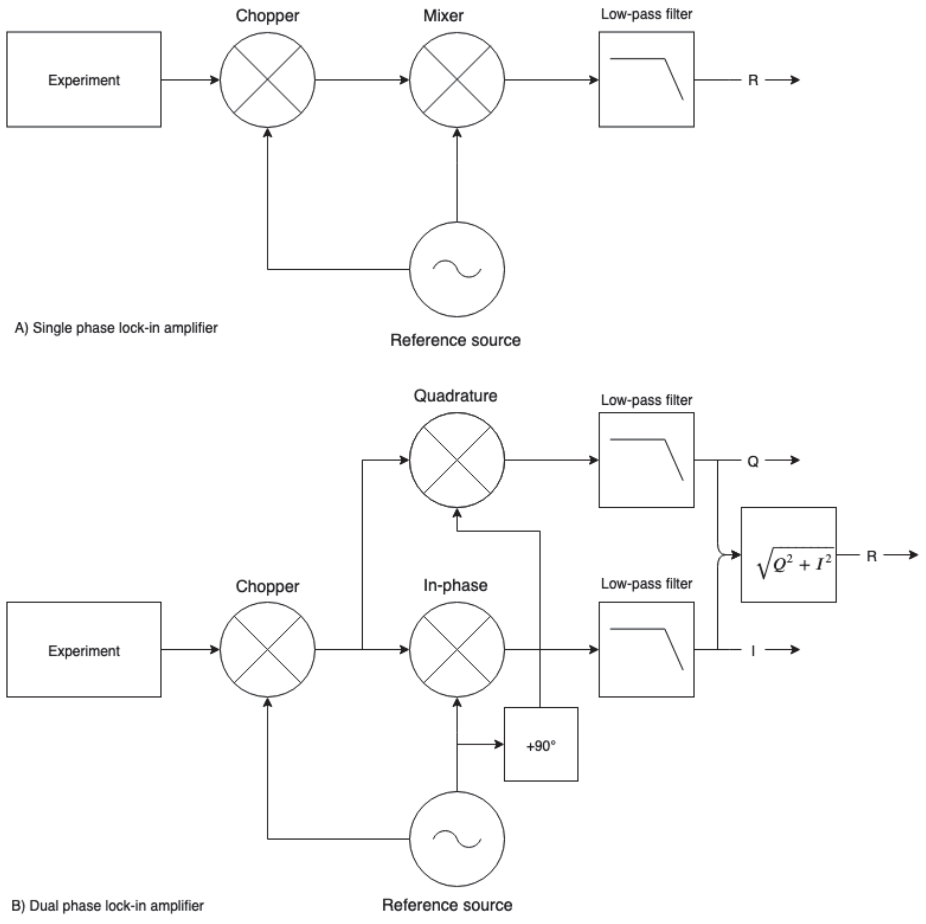


Figure 3: A single and dual phase lock-in amplifier black diagram

This technique was first described in the 1930s [1] [2] and commercialized in the mid-20th century in the form of an analog lock-in amplifier. These performed their jobs well but were sensitive to cross-talk and drifts caused by, for example, temperature changes. The transition from analog to digital lock-in amplifiers has gone hand in hand with increases in speed, resolution and linearity of Analog-to-Digital Converters(ADC). The advantage of digital lock-in amplifiers lies in the fact that digital signal processors(DSP) are less prone to errors caused by, for example, temperature changes. They are also capable of analyzing signals in multiple ways without increasing Signal-to-Noise Ratio(SNR). The only point where DSPs introduce error to the signal is when the signal gets digitized, introducing truncation errors. Digital lock-in amplifiers often generate their reference signal internally by means of a numerically controlled oscillator(NCO). The ADC will sample the signal from the chopper, the DSP will multiply it internally with a sample from the NCO and then apply a digital low-pass filter with a very low cut-off frequency. Analog lock-in amplifier and Digital lock-in amplifiers are the 2 main types of lock-in amplifiers. There are also several implementations which act as hybrids between the two. They, for example, leave the mixing on the analog side and only digitize afterwards. These come with the advantage of still being able to analyze and transmit the signal in different ways while simplifying the source code. The disadvantages are sensitivity to cross-talk and drift due to external elements.

3.4 Stakeholders

The stakeholder during this project was SRON National institute for space research. Several meetings were held with the contractors Mr. H. Smit and Mr. M. Dijkstra. Certain requirements came from these meetings and have been listed below in no particular order.

SRON showed great interest in the BeagleBone Black, in particular the PRU-ICSS included with the AM335x processor on the BeagleBone Black. This interested comes from the fact that they have been using BeagleBone' in their measurement systems, so they are commonly used within SRON. Using the PRU-ICSS would allow the BeagleBone to run other tasks while the Lock-in algorithm runs in the background. Not only are they interested in **1. The implementation of a digital lock-in amplifier on a microcontroller.** They want to find out if this can be done on the PRU-ICSS of the BeagleBone Black by **2. Testing if the PRU-ICSS is suitable for real-time tasks**, like acting as a lock-in amplifier. SRON wants the lock-in amplifier to **3. Have a dynamic reserve of 20 – 30dB** while keeping **4. A resolution of 1mV.** **4. The low-pass filters in the lock-in amplifier are allowed a maximum integration time of 1 minute.** The lock-in amplifier should be capable of doing this over **5. 4 to 8 channels** depending on what is realizable. The contractors have indicated early on that this could be quite a large project and that they do not necessarily expect to get a fully functioning end product. Later on the decision was made to not do any filtering on the BeagleBone Black. They also indicated that they wanted a daughter board (cape) for the PRU-ICSS lock-in amplifier. This should be designed and implemented only when time allows for it.

3.5 Formal requirements

#	Description
1	The implementation of a digital lock-in amplifier on a microcontroller
2	Testing if the PRU-ICSS is suitable for real-time tasks
3	Have a dynamic reserve of 20 – 30 dB
4	A resolution of 1mV
5	4 – 8 channels
6	The low-pass filters in the lock-in amplifier is allowed a maximum integration time of 1 minute.

4. Design

This chapter serves to advice and inform on design choices that have to be made while making a digital lock-in amplifier. There are three major design choices involved in making a digital lock-in amplifier: The type of ADC, the type of local oscillator(LO) and the type of microcontroller(MCU). If any good recommendations want to be made, a better understanding of the lock-in algorithm is required. This chapter will start by giving a detailed explanation on the lock-in algorithm, followed by simulations to predict the performance of digital lock-in amplifiers, continued with the different design choices and the considerations which have to be made during the project, and will end with a conclusion which will consist of a quick summary and recommendations.

4.1 The lock-in amplifier algorithm

4.1.1 Overview

A signal coming from an experiment is hidden in noise up-to four orders of magnitude larger than the desired signal. To analyze the desired signal, which will be referred to as the experiment signal from now on, amplification is required. A general amplifier will not only amplify the experiment signal and the noise within the amplifiers bandwidth but it will also inject noise into the output. The following examples from Microchip [3] and Stanford Research Systems [4] will serve as a guide to improve our understanding.

Consider an amplifier with input noise of $5\ nV/\sqrt{Hz}$, a gain of 30 dB (or 1000) and a bandwidth of $300\ kHz$. Feeding this amplifier a pure sine wave with an amplitude of $150\ nV$ at a frequency of $50\ kHz$ will result in bandwidth noise equal to the equation 4-1

$$\mathbf{5}\ \boldsymbol{nV} \times \mathbf{1000}\ \times \sqrt{\mathbf{300}\ \boldsymbol{kHz}} = \mathbf{2.7}\ \boldsymbol{mV} \qquad (4\text{-}1)$$

The experiment signal will have an amplitude of $150\ \mu V$ $(1000 * 150\ nV)$ after amplification. This is 18 times smaller than the bandwidth noise described in equation (4-1)

The amount of noise can be decreased by applying a band-pass filtered centered at $50\ kHz$. Such a filter would have a Q(quality factor of the filter) of 100 if it is a very high quality filter. Equation 4-2 shows how much noise is still being injected at such a Q factor.

$$\mathbf{5}\ \boldsymbol{nV}\ \times \mathbf{1000}\ \times \sqrt{\mathbf{50}\ \boldsymbol{kHz}/\boldsymbol{Q}} = \mathbf{111,8}\ \boldsymbol{\mu V} \qquad (4\text{-}2)$$

This is still a lot of noise relative to the experiment signal. Something else is required. Lock-in amplifiers are capable of reaching a Q as high as 10.000. This would reduce the noise in our signal to $11{,}2\ \mu V$, making it possible to do measurements on the experiment signal.

An implementation of a lock-in amplifier can be found in Figure 3, this figure illustrates a single-phase and dual-phase lock-in amplifier. Single-phase lock-in amplifiers require a phase shifter to align the reference signal with the experiment signal. If they are not in phase it will result in a decrease in magnitude of the signal of interest. Dual-phase lock-in amplifiers, as show in Figure 3-B, are not sensitive to these phase changes and allow for measurement of the phase difference between the experiment signal and reference signal.

4.1.2 Mathematical approach

The mathematical derivations below show the working principles of the dual-phase lock-in amplifier adapted from the SRS830 manual [4].

$$V_{exp} = V_{sig}\cos(\omega_{ref}t + \theta_{sig}) \qquad (4\text{-}3)$$

$$V_{cos} = V_{ref}\cos(\omega_{ref}t + \theta_{ref}) \qquad (4\text{-}4)$$

$$V_{sin} = V_{ref}\sin(\omega_{ref}t + \theta_{ref}) \qquad (4\text{-}5)$$

Equation 4-3 shows the experiment signal while equation 4-4 and 4-5 show the two reference signals. These need to be multiplied to create the in-phase (I) and quadrature (Q) components as shown in equation 4-6 and 4-7.

$$V_I = V_{exp} \times V_{cos} \qquad (4\text{-}6)$$

$$V_Q = V_{exp} \times V_{sin} \qquad (4\text{-}7)$$

$$V_I = V_{sig}\cos(\omega_{ref}t + \theta_{sig}) \times V_{ref}\cos(\omega_{ref}t + \theta_{ref}) \qquad (4\text{-}8)$$

$$V_I = \frac{1}{2}V_{sig}V_{ref}\cos(\theta_{sig} - \theta_{ref}) + \frac{1}{2}V_{sig}V_{ref}\cos(2\omega_{ref}t + \theta_{sig} + \theta_{ref}) \qquad (4\text{-}9)$$

$$V_I = \frac{V_{sig}V_{ref}}{2}\cos(\theta_{sig} - \theta_{ref}) \qquad (After\ the\ low\ pass\ filter) \qquad (4\text{-}10)$$

Equation 4-8(or 4-11) and 4-9(or 4-12) shows how to solve this multiplication. The multiplication results in equation 4-9(or 4-12) with twice the original frequency and one without a frequency. A low-pass filter will remove the component with twice the original frequency as can be seen in equation 4-10(4-13)

$$V_Q = V_{sig}\cos(\omega_{ref}t + \theta_{sig}) \times V_{ref}\sin(\omega_{ref}t + \theta_{ref}) \qquad (4\text{-}11)$$

$$V_Q = \frac{1}{2}V_{sig}V_{ref}\sin(\theta_{sig} - \theta_{ref}) + \frac{1}{2}V_{sig}V_{ref}\cos(2\omega_{ref}t + \theta_{sig} + \theta_{ref}) \qquad (4\text{-}12)$$

$$V_Q = \frac{V_{sig}V_{ref}}{2}\sin(\theta_{sig} - \theta_{ref}) \qquad (After\ the\ low\ pass\ filter) \qquad (4\text{-}13)$$

Assuming that the phase difference between the signal and reference is zero and that the outputs are passed through low-pass filters to remove the AC component. Resulting in these 2 equations.

$$V_I = \frac{V_{sig}V_{ref}}{2}\cos(\theta) \qquad (4\text{-}14)$$

$$V_Q = \frac{V_{sig}V_{ref}}{2}\sin(\theta) \qquad (4\text{-}15)$$

These outputs from the dual-phase lock-in amplifier can be converted to magnitude and phase difference between the experiment signal and reference signal using equations 4-16 and 4-17 respectively. The magnitude does not depend on the phase difference between the experiment signal and reference signal in a dual-phase lock-in amplifier.

$$\boldsymbol{Magnitude(R)(|V_{sig}|) = 2 \times \sqrt{{V_I}^2 + {V_Q}^2}} \quad \boldsymbol{(V_{ref} = 1V)} \qquad (4\text{-}16)$$

$$\boldsymbol{Phase(\angle V_{sig}) = \tan^{-1}(\frac{V_Q}{V_I})} \qquad (4\text{-}17)$$

4.1.3 Emulated approach

The algorithm could be implemented on either an Field Programmable Gate Array(FPGA) or DSP, but the contractor showed particular interest in the BeagleBone Black PRU-ICSS. Simulations will be ran in python and MATLAB in an attempt to emulate the functioning of a digital lock-in amplifier, to evaluate their performance, and to see what parameters are required to reach the requirements specified by the contractors. These simulations will be based on earlier works done by Sabyasachi Bhattacharyya[5] and Robert George Skillington [6].

The input for the simulations will be:

- Sample frequency
- Reference frequency
- Integration time
- Noise

The simulated components are:

- ADC
- Local oscillator
- Mixer
- Low-pass filter
- Lock-in amplifier magnitude output

The output of the simulated lock-in amplifier:

- Magnitude
- Phase
- SNR

The objectives of these simulations are:

1. Improve understanding of the lock-in amplifier
2. Evaluate the performance of the algorithm with different parameters at different SNR levels

Information to be gained from these simulations:

1. Is the implementation of lock-in amplifiers on microprocessors viable?
2. Is the implementation of lock-in amplifiers on the PRU-ICSS viable?
3. How does the sample frequency affect the results?
4. How does the integration time affect the results?

Our Python program starts with the generation of the experiment signal and two reference signals according to equation 4-3, 4-4 and 4-5. These have a constant frequency of $100\ Hz$ for a time period of 1 second. The sample frequency can be changed by increasing or decreasing the amount of samples in the time period of one second. The initial sample frequency is chosen to be $1000\ Hz$. The plots of these signals can be seen in Figure 4, with the left side showing the time domain plots and the right side the frequency domain plots.

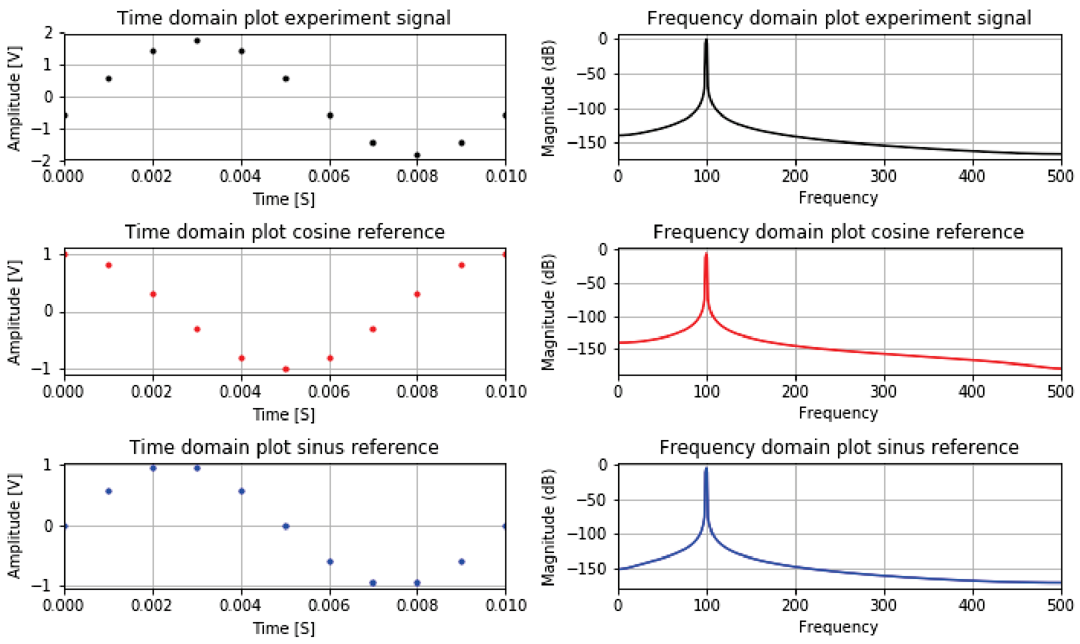


Figure 4: Time and frequency domain plots of the experiment signal and reference signals at 100Hz

The next step was implementing the mixer. This was done by multiplying the experiment with the reference cosine and the reference sinus in accordance to equation 4-6 and 4-7. In Python this is as easy as multiplying 2 arrays. Multiplying the experiment with the reference wave creates a signal with frequencies at the difference and sum of the frequencies of both signals according to equation 4-9 (4-12) . This can be seen in Figure 5, with the time domain plots on the left and frequency domain plots on the right.

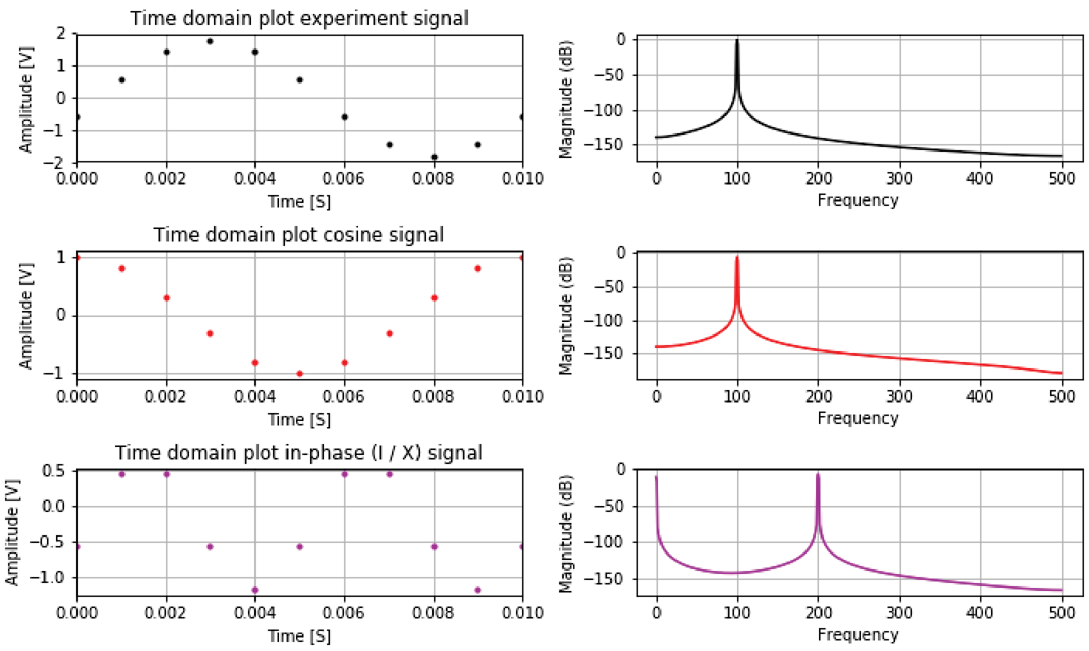


Figure 5: Time and frequency domain plots of quadrature multiplication

As you can see in the Fourier transform on the bottom right of Figure 5, there is one peak at the sum and one peak at the difference of the two frequencies resulting in a DC signal and a $200\ Hz$ signal. The I and Q lines require filtering to isolate the DC signal. A more in-depth discussion on filters can be found in the implementation. For now the numpy *mean* function is used to find the DC value. The *mean* function takes the average of a array. This will result in an moving average filter with an integration time as long as the time being evaluating. The output filters produce a single value for I and Q. These can be used to find the magnitude and phase. As long as there is no noise in either of the signals, the values should be the same values in the output as the input.

Variable	**Input**	**Output**
Amplitude	1,800	1,800
Phase delay	$\frac{\pi}{8} = 0{,}392699$	0,392699
Noise	0,0 dB	SNR: infinite dB

The table below shows that the output is not influenced by the phase difference between the reference and the experiment signal. These results will not be shown in future simulations.

Phase delay		**Amplitude**	
Input (radians)	**Output (radians)**	**Input (V)**	**Output (V)**
$\boldsymbol{\frac{\pi}{8} = 0.392699}$	0,392699	1,800	1,800
$\boldsymbol{2\pi = 6,283185}$	6,283185	1,800	1,800
$\boldsymbol{\pi = 3,141593}$	3,141593	1,800	1,800
$\boldsymbol{\frac{3\pi}{4} = 2,356194}$	2,356194	1,800	1,800

Random noise is being injected using Numpys *random randn* function to test the lock-in amplifiers noise rejection performance. This function generates random samples from normal (Gaussian) distribution. Our formal requirements state that it needs to reject noise up to an SNR of -20 to -30 db. This is done by increasing the amount of noise on a logarithmic scale from 10^{-1} to 10^{2}. Starting the simulation now would generate results that would vary because of the random nature of the noise being injected. 5000 samples are taken to come closer to normal distribution. The SNR and the magnitude is averaged and saved for each noise level. Signal quantization was used to simulate the ADC. The quantization was performed after the noise was added to the signal and change the number into a 16 bit integer.

Problems occurred during these simulations in Python, not only did they take such a long time that Python would crash. The results were not equal to earlier works [6] so the code was recreated in MATLAB. This was not hard since Pythons Scipy module is inspired by MATLAB functions. The results from these simulations were equal to earlier works and more consistent.

Figure 6 shows the output of the lock-in amplifier vs the SNR at the input. An error of 0.1% was reached at -20 dB At a sample frequency of $4000Hz$ with an integration time of 30 seconds.

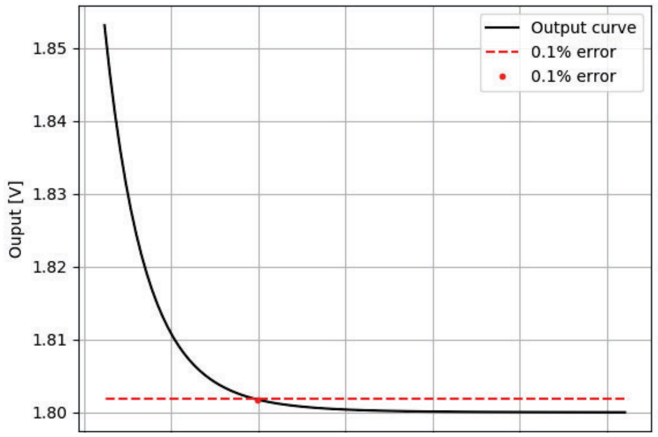


Figure 6: The lock-in amplifier output at different SNR @ 4000Hz sample frequency, 30 seconds integration time

4.1.4 Results

A dynamic reserve of 20 – 30 dB is required according to the formal requirements. This means that a signal with a SNR of -20 to -30 dB at the input should be retrievable. The requirements state that the maximum error allowed is 1 mV, this corresponds to 0.1% for a input signal with an amplitude of 1.8 V. The simulation was ran multiple times with different sample frequencies and integration times. These were changed manually because these simulations took quite a long time. The code can be found on this projects GitHub page under *Python/Simulations*. Both the Python code and the MATLAB code can be found there.

	Integration time				
Fs	**1 second**	**10 seconds**	**20 seconds**	**30 seconds**	**60 seconds**
1000	-3 dB	-15 dB	-16 dB	-16 dB	-22 dB
2000	-3 dB	-17 dB	-18 dB	-20 dB	
4000	-8 dB	-18 dB	-18 dB	-20 dB	
8000	-13 dB	-24 dB	-28 dB	-28 dB	
160000	-24 dB				

These results show that increasing the integration time results in an increase in dynamic range. An increase in sample frequency also results in an increase in dynamic reserve. Best results will be achieved with the longest integration time and the highest sample frequency. Increasing the integration time by an order of magnitude results in ~10 dB increase in dynamic reserve. Some of these results took so long to simulate that we neglected them since we already had good results for that sample rate or integration time.

4.2 Options

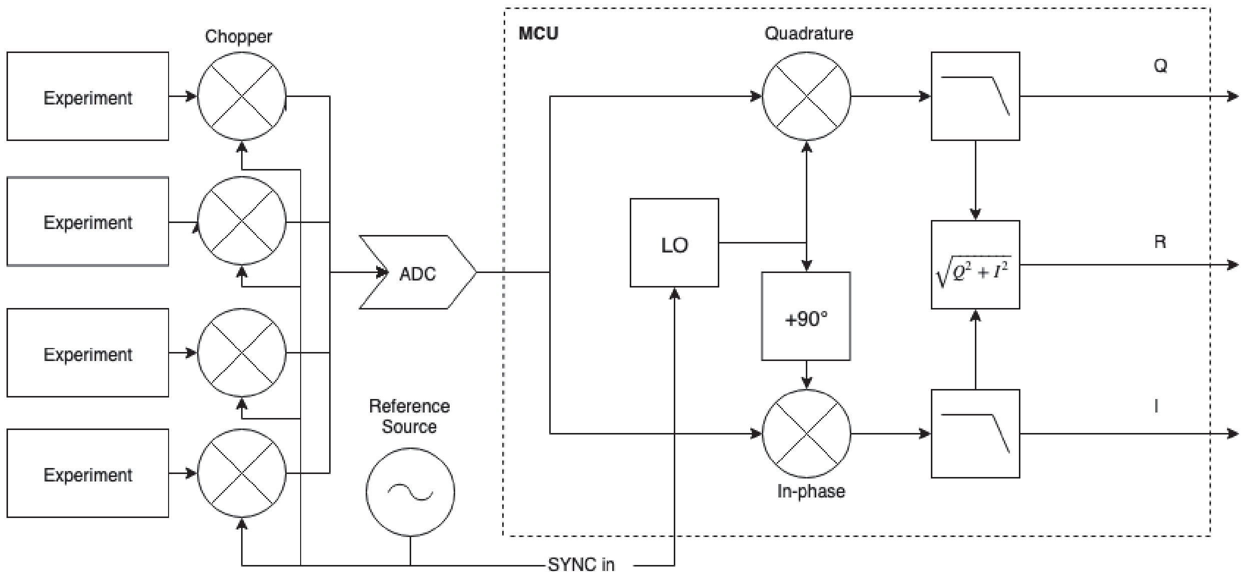


Figure 7: Functional diagram for a digital lock-in amplifier. (There is only one chopper in practice, multiple choppers are shown for visualization purposes only)

Figure 7 shows a functional diagram of a digital lock-in amplifier. A reference source excites both the copper and the local oscillator. This local oscillator generates both the sine an cosine waves required for quadrature multiplication. The multiplicand of this multiplication is the input from the ADC. The ADC digitizes the signal from the chopper and feeds this to the MCU. The MCU does the multiplication and filtering for the lock-in algorithm. The output of this can be transmitted out or used inside the MCU to calculate the magnitude and phase of the signal being analyzed.

Microcontroller

The microcontroller used should be capable of reading at least four channels of an ADC at a high sample rate, generating a reference wave from the sync input, and doing mathematical operations at a high rate. SRON has expressed great interest in the BeagleBone Black [3], specifically the PRU-ICSS included in the AM335x processor package. This is why the BeagleBone Black PRU-ICSS has received extra attention. Other commonly used options are FPGAs [4] and DSPs [5].

The BeagleBone Black is a low-cost, open-source development board running Linux. With its 1 Ghz AM335x ARM Cortex-A8 processor it can do up to two billion instructions per second. This allows developers to build complex applications with high-level software and low-level electronics. It supports many commonly used interfaces such as UART, SPI, I2C, CAN, Ethernet and more. SRON uses these boards in their measurement systems and are interested in using them for more applications. They are specifically interested in the PRU-ICSS included on the AM335x System-on-Chip (SoC) for real time operations. These are two 32-bit microprocessors encompassed in their industrial communication subsystem. These two 32-bit microprocessors run at 200 MHz, each with 8 KB of program RAM, 8 KB of data RAM and 12 KB of shared RAM. Using the Open Core Protocol (OCP) master port they can also access external Linux host memory allowing them to interface with all the peripherals the AM335x has. The PRU-ICSS has a 200 MHz clock and is capable of (5ns) single cycle instructions. The SPI chip included on the AM335x SoC can reach clock speeds of up to 48 MHz this allows for 20 ns clock pulses. 16 of these clock pulses are needed per sample. This results in 320 ns per sample, add the conversion time from the LTC1859 data sheet [9], which is ~4000ns. This adds up to ~4320ns per sample. Now each sample from the ADC needs to be multiplied by a sample from the reference wave which takes 1 clock cycles per multiplication for a total of 2 clock cycles. If the filtering is ignored for now, a theoretical sampling speed ~230kHz can be reached. With single cycle multiplications and additions to spare.

Field-Programmable-Gate-Arrays are semiconductor devices that allow you to program digital logic using hardware description languages such as VHDL or Verilog. FPGAs are very powerful and useful in many fields due to their parallel operation and programmable nature. This allows them to become basically anything as long as it is describable in digital logic gates. There are many options on the market with different amounts of configurable logic blocks. To decide how many logic blocks you need you can use software packages issued by FPGA manufactures such as: Quartus II 13.0, made by Intel, which comes with support for Altera Cyclone II/III/IV family FPGA's. This software package allows you to simulate the FPGA, using Model Sim™, without actually having a development board. Using languages such as TCL you can probe your code feeding it inputs and comparing/saving the outputs. Although programming logic gates allows for a great degree of freedom, it also introduces a new style of programming, since we are now programming parallel operations instead of sequential operations. It also requires a great amount of knowledge regarding digital logic.

Digital Signal Processors are microcontrollers capable of taking real-world signals and performing fast mathematical operations on these signals. Operations such as add, subtract, multiply, divide and math with floating point numbers have been built into the hardware and can be performed very quickly, and with great precision. These boards are often programmed using languages such as C, C++ and assembly. This allows for very low-level access to peripherals and makes them viable in a wide variety of applications. Big players in the DSP market, such as Microchip, Texas Instruments and NXP have a great variety of choices. Microchip has already implemented a Lock-in amplifier on one of their DSPs, the AN1115 from the dsPIC family [5].

Analog-to-Digital Converter

Figure 7 shows that an ADC is used to digitize the signals from the chopper. An ADC capable of reading at high frequencies, over 4-8 channel, with a low SNR, and with 16-bit resolution is required. ADCs have become a very big market so there are many different options available. Three have been selected to be discussed, the LTC1859, LTC2333 and the ADS8555.

The LTC2333 is a 16-bit, multiplexed 8 channel ADC with a maximum input range of ±12.5 V and a maximum sample frequency of 8000 ksps with an 94 dB SNR. The input range is programmable and can be set as 0-10.24 V, ±10.24V, 0-5.12V, ±5.12V, 0-12.5V, ±12.5V, 0-6.25V and ±6.25. The LTC2333 uses the SPI protocol to send and receive data and needs separate control pins to start conversions.

The ADS8555 has an 16-bit resolution with 6 channels and a maximum input range of ±12 V. It has two modes, parallel and serial. The Serial mode uses the SPI protocol to send and receive data, the parallel mode requires a software driver to send and receive data. The input range is programmable and depends on the reference voltage given to the ADC. The ADS8555 has a maximum sample rate in serial mode of 500 ksps and a SNR of 91 dB. The ADS8555 also requires separate control pins to start conversions.

The LTC1859 is a 16-bit, 8 channel ADC with a maximum input range of ±10 V and a maximum sample frequency of 100 ksps with an SNR of 87 dB. The input range is programmable and can be set a 0-5 V, ±5 V, 0-10 V and ±10 V. The LTC1859 uses the SPI protocol to send and receive data and needs control pins to start conversions. The LTC1859 is currently in use at SRON as a portable lab ADC and Mr. M. Dijkstra attempted to implement a

Lock-in amplifier with this ADC and an Arduino DUE, with minimum SNR because of jitter caused by timing inaccuracies.

Local oscillator

The chopper shown in Figure 7 spin at a frequency of ~100 Hz. Because the copper is a physical object rotating, slight drifts in frequency can occur. This would cause error in our sum and difference calculation unless our oscillator is capable of locking to the frequency of the chopper. To lock to the frequency a Phase locked loop(PLL) or Direct Digital Synthesis(DDS) can be used, these are techniques for following frequencies and generating waveforms at those frequencies. They can be implemented as either analog PLLs, DDS integrated circuits or DDS in software. DDS has long been recognized as a superior technology for generating high accuracy, frequency agile, low-distortion waveforms [6]. This is why PLLs will be ignored.

DDS integrated circuits are often implemented as a NCO with a Digital-to-Analog converter(DAC) as output. Implementations that output in a different format such as SPI exists but are less common. These chips need to be controlled by a microcontroller which tells them the frequency they should output. Using DDS on chip would require the microcontroller to sample/read one extra device every ADC sample.

DDS in software uses the same working principle as DDS integrated circuits, the only difference is that it is all done by the microcontroller. This would require the microcontroller to take some time to calculate the next sample. The speed of this calculation depends on the accuracy you want. Implementing DDS in software does allow for more customizability and makes prototyping different parameters easy. A DDS integrated circuit can always be chosen when the parameters required are clearly defined.

4.3 Conclusions

Simulations show that with an integration time of 30 seconds, a sample frequency of $4000\ Hz$ is required to reach an acceptable dynamic reserve. The dynamic reserve can be improved by either increasing the sample frequency of increasing the integration time. Implementing this algorithm on a MCU will require an ADC capable or sampling at least four times $4000\ Hz$, a MCU capable of doing fast calculations and a LO capable of generating sine wave samples at the same frequency.

There are many ADCs available with a wide range of performances, but choosing an ADC and designing a development board for it can take a lot of time to do right. SRON has a development board for the LTC1859 in their lab. The first prototypes will use this board to simplify the project, following the advice of Mr. Smit. A new board will be designed if time allows for it. This ADC can reach sample frequencies of up to 100 ksps which is orders of magnitude higher than the required sample frequency.

While there is a wide range of FPGAs and DSPs available on the market there is only one type of BeagleBone Black PRU-ICSS. There are plenty of FPGAs and DSPs capable of reaching the sample frequencies above the required sample frequency and doing calculations at a very high speed. The PRU-ICSS lacks this variety and has clear specifications. The theoretical sample frequency has been calculated to be ~230 kHz. This is at least two orders of magnitude larger than the sample frequency required, allowing for filtering to still be done. The Lock-in amplifier algorithm will be implemented on the PRU-ICSS because of its ability to sample fast, do quick calculations and because of the great interest shown by the contractors. Knowledge gained on the PRU-ICSS can be used for future projects within SRON.

A local oscillator is required to generate samples for the quadrature multiplication. DDS has long been recognized as a superior technology for generating high accuracy, frequency agile, low-distortion waveforms [6]. Since there are two cores available, one can be chosen to dedicate to generating the sine-wave and one core to sampling the ADC and executing the lock-in algorithm. This sine-wave can either be calculated/generated in the core itself or can be generated in a DDS integrated circuit and sampled by the core. Using a DDS IC would require us to make a development board for the chip, which would take a lot of time and is just not necessary. A sine-wave can be generated within the PRU core and transferred to the other core using shared memory. This gives a lot of flexibility in the generation of the waves, which is good for prototyping. If good results come from the sine-wave generated by the PRU core, a DDS integrated circuit can always be chosen with the same parameters as specified on the PRU core.

4.4 Morphological chart

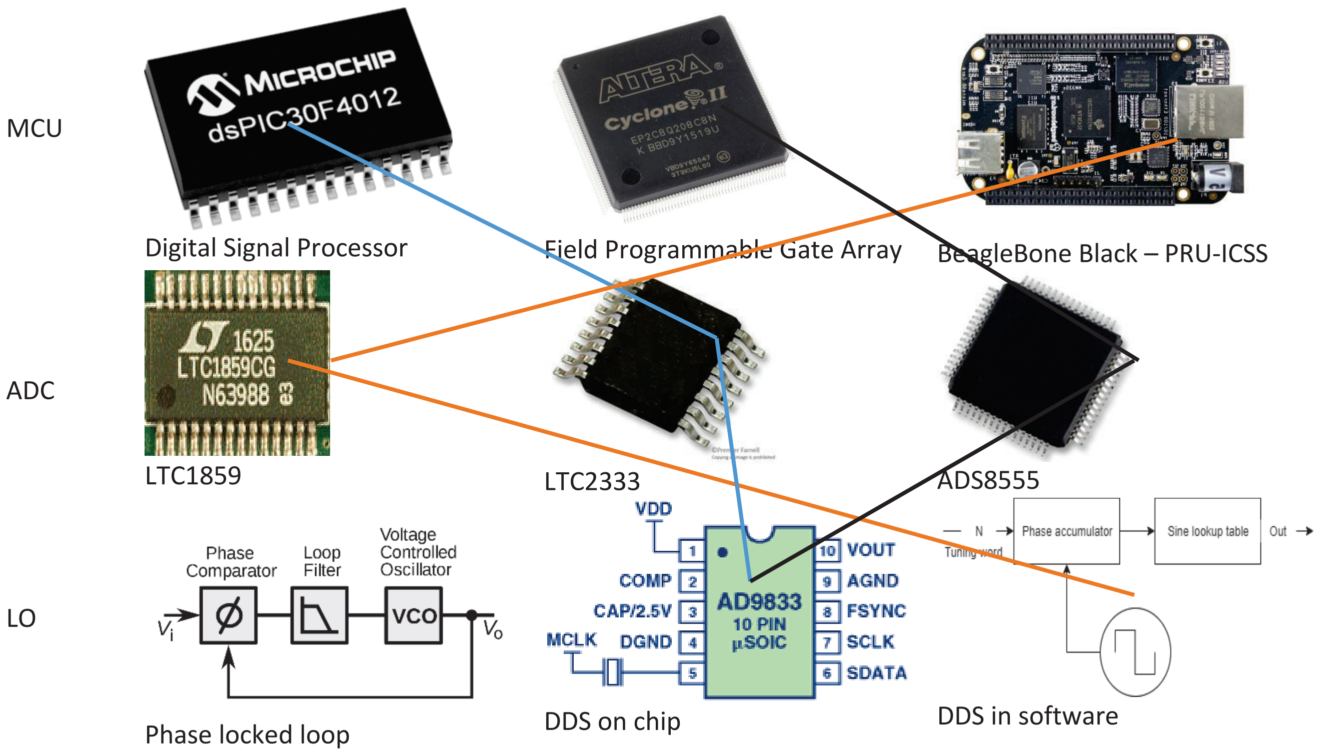


Figure 8: Morphological chart

Figure 8 show different paths that can be taken to achieve the objective. The text below explains the pros and cons of each path.

Blue path

The blue path uses an DSP together with the LTC2333 and a DSS integrated circuit. Choosing an DSP with floating point hardware, a very high clock speed and an SPI peripheral can result in a very high quality lock-in amplifier. Choosing the DSP can result in a lock-in amplifier capable of reaching higher sample frequencies and thus a better dynamic range than the PRU-ICSS. Choosing the blue path would require you to invest time in selecting a DSP, DDS and possibly a PCB with all of these components.

Black path

The black path uses an FPGA together with the ADC8555 and a DDS integrated circuit. There is a large variety of FPGAs available but each of these will require you to write your own drivers for the ADC and SPI. This becomes even more complicated because we are programming logic gates not sequential logic. This would require knowledge of the VHDL language and digital logic. A benefit of FPGAs is that they can be very fast and reliable because we are programming actual logic gates. Making them capable of reaching very high sample rates and thus a good dynamic reserve.

Orange path

The orange path uses the BeagleBone Black – PRU-ICSS, LTC1859 and DDS in software. This is also the path recommended because it uses the PRU-ICSS. The contractors showed a great interested in the PRU-ICSS not only for this project but also for future projects. The PRU-ICSS is capable of sampling fast enough to achieve the desired dynamic reserve.

4.5 Functional requirements

#	Functional requirements	Formal requirements
1	The PRU-ICSS is used to process the data	1, 2, 3
2	The LTC1859 is used to sample the experiment	3, 4, 5
3	DDS in software is used to generate the reference waves	3
4	Simulations show that a dynamic range of 20-30dB can be achieved with an integration time less than 1 minute.	6

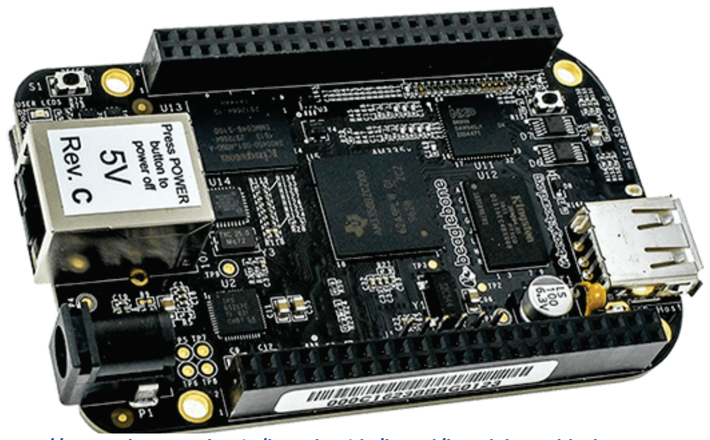
5. Implementation

This chapter will provide a detailed explanation on the implementation of a digital lock-in amplifier on the PRU-ICSS. This chapter does not serve as a tutorial on how to implement the digital lock-in amplifier algorithm on the BeagleBone Black. A quick start guide can be found on the GitHub page, this guide will help you from loading an image onto your SD card to uploading and running the lock-in amplifier algorithm. This chapter will discuss what the BeagleBone Black is, how to work with the BeagleBone Black, what needs to be configured on the BeagleBone Black to use the PRU-ICSS, how to compile programs for the PRU-ICSS, how to start/stop the PRU-ICSS, the features of the PRU-ICSS, the different peripherals the PRU-ICSS has access to, how to extract data from the PRU-ICSS, and finally how reference signals were generated and how the lock-in amplifier was implemented. All the code and examples can be found on this projects GitHub page.

5.1 BeagleBone Black

The BeagleBone Black is a cheap, open-source development board. SRON uses the BeagleBone Black in their measurement systems as acquisition processor. The specifications of the BeagleBone are listed below:

- Processor: AM335x 1 GHz ARM Cortex-A8
- RAM: 512 MB DDR3
- Storage: 4 GB 8-bit eMMC on-board flash storage (microSD slot)
- Remote: 2x PRU 32-bit microcontrollers
- USB for power, communication and host
- Ethernet
- HDMI
- 2x 46 muxed pin headers



https://www.electromaker.io/board-guide/board/beaglebone-black

Figure 9: BeagleBone Black Rev. C

The BeagleBone is capable of running Debian, Ubuntu, Android and much more. Linux kernel 4.14.94-ti-rt-r93 is used during this project, this was TI's the most recent real-time 4.14 release of the Linux kernel at the time. More information on the BeagleBone Black and how to get started can be found on their website [9] or on this projects GitHub page.

5.1.1 SSH and SFTP

The BeagleBone has HDMI capabilities but does not have a HDMI port. This means connecting a monitor, keyboard and mouse is not a possibility. Secure Shell (SSH) is a network protocol for operating network services securely over an unsecured network. SSH allows us to access the command line of the BeagleBone from any computer within the network. Windows users can use PuTTY [10] and Linux users can use openSSH on their terminal to access the BeagleBone Black. Windows users have to download and open putty, fill their BeagleBone IP address(192.168.7.2), click *open* and login with the default username and password(debian, temppwd). Linux user have to open their terminal and write: *SSH 192.168.7.2 –l debian –u temppwd.* Both of these actions will connect the user to the BeagleBone shell.

Almost everything is possible from the command-line, except file transfers. This is where SSH File Transfer Protocol (SFTP) comes in. Using WinSCP [11] on windows and the command line in Linux enables you to transfer files from the host PC to the BeagleBone Black. Windows users have to download and open WinSCP, this will prompt you to login. Once you logged in with the credentials from earlier, you will be able to see the BeagleBone file tree. Linux users have to open their terminal and write: *SFTP* debian@192.168.7.2 from here you can write *help* to see the different commands. This can be useful when you prefer an IDE such as Atom or Notepad++ over the code editors included in Linux, like VIM or Nano. The IDE Atom allows you to open a whole project from a remote desktop using the remote-ftp package [12]. This makes writing code/debugging a lot easier.

5.1.2 Device tree

Most computers have a BIOS to initialize their hardware. The BIOS is a piece of firmware loaded on ROM memory and comes pre-installed on most motherboards. Embedded systems such as the BeagleBone Black do not have a BIOS. Rather, to boot and configure the devices hardware, it uses files on the SD card or eMMC that describe the hardware. Every device running embedded Linux has these files. Older Linux versions required the modifications for different hardware to be done directly in the Linux source code. This caused proliferation of the Linux kernel customizations. Linus Torvalds became very unhappy with the amount of code that was being added directly to mainline Linux. The file containing all these hardware descriptions became around 4000 lines of code. New versions of Linux use the flattened device tree(FDT) model. The FDT is a data structure describing the hardware on a particular board. It is designed in such a way that it is readable for humans.

There are three types of device tree files: *.dts* which contain board level definitions, *.dtsi* which contain SOC level definitions and *.dtb* which is the binary you get after compiling a *.dts* file. *.dts* files often consist of many includes of *.dtsi* files. Compiling these DTS files can be done using the device tree compiler (DTC). Changes to the hardware description should be made to DTS files, these files should be compiled using the DTC and the resulting DTB files should be moved to the boot directory: */boot/dtbs/.*

One issue with this approach is that you have to reboot the whole device to change the hardware configuration. To solve this issue they introduced device tree overlays(DTO). These allow users to reconfigure their hardware during runtime. DTOs work together with *capemgr*; an in-kernel mechanism for dynamically loading device tree fragments from user space. This also allows the makers of capes(daughter boards for the BeagleBone black) to put a memory IC on their cape containing their device tree overlay. Loading the overlay automatically when plugged-in on the board.

Using the PRU-ICSS requires you to load the proper device tree overlay. Explanation how to do this can be found on this project's GitHub page under *Enabling the BeagleBone PRU*.

5.1.3 Pin mux-ing

The BeagleBone Black has two times 46 pin headers called P8 and P9. Almost each of these pins has multiple modes that can be controlled with the config-pin tool. Pins have to be configured in the right mode if the PRU-ICSS or another peripheral needs to drive them. A table of all the different modes can be found in Appendix B^1 – P8 header pin modes [6] & Appendix B^2 – P9 header pin modes [6], and explanation on how to use the tool can be found on this projects Github page under *Configure GPIO on startup*.

5.1.4 Compiler and linker

Texas Instruments(TI) has released the PRU Code Generation Tools (PRU-CGT) as part of Code Composer Studio V6. At the same time they released the PRU-CGT as CLI tool. This cool is capable of compiling C/C++ programs into a binary file the PRU can use as firmware. The basic workflow of the PRU-CGT can be seen in Figure 10. The PRU-CGT consists of the C compiler (clpru), the assembler(asmpru), the linker (lnkpru) and the output file generation tool (hexpru). This tool chain outputs an .out file which is the PRU firmware. To load this firmware onto one of the PRUs you have to move it to the directory */lib/firmware/* under the name *'am335x-pru0-fw'* or *'am335x-pru1-fw'* depending what core you want to load with the firmware. If you want to load a PRU with firmware you have to stop the PRU first.

A *makefile* can be found in Appendix E^1 – Makefile together with a description, and on this projects GitHub page under *Default files*. This *makefile* compiles, assembles and links your C program and moves the output file to the correct folder under the correct name.

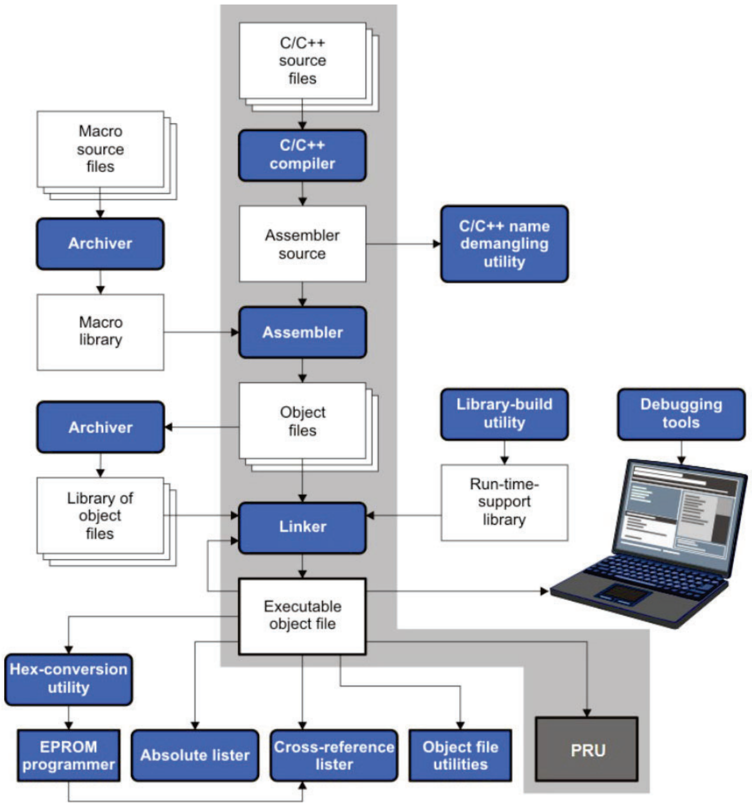


Figure 10: PRU-CGT flowchart

5.1.5 Remoteproc

The Remoteproc framework is a commonly used kernel module for communication with co-processors. It allows the ARM processor to power on, power off and load firmware to a remote processor while abstracting the hardware differences. The *sysfs* interface is used to specify the firmware to be loaded and to start and stop the remote processor. Figure 11 shows where the different parts of the framework are located.

When starting the PRU-ICSS the Remoteproc driver loads the *pru-rproc* module. This module checks if the PRU firmware binaries exist and if they are located in */lib/firmware/*. It also parses the firmware binaries looking for *.resource_table*. *.resource_table* is a section of the firmware that specifies the system resources the PRU will need during execution and will be explained later in this chapter. The *pru-rproc* module than configures all of the resources

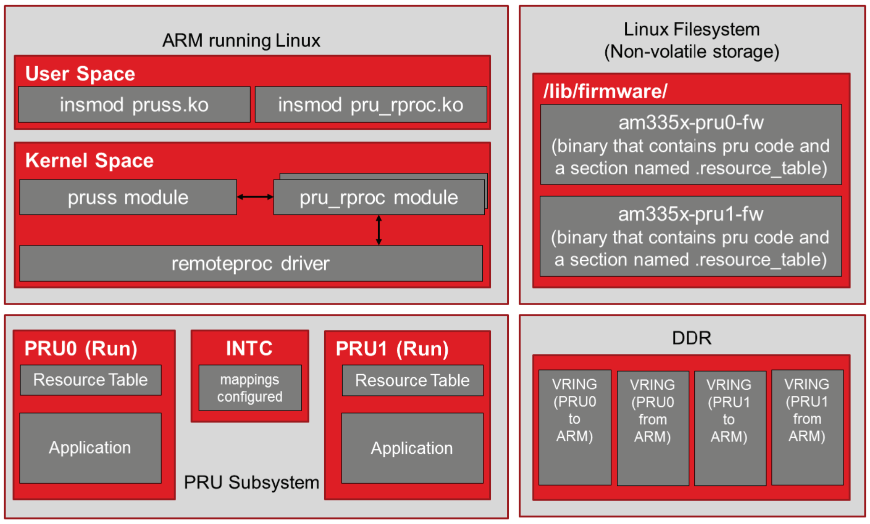


Figure 11: block diagram of the remoteproc protocol
http://processors.wiki.ti.com/index.php/PRU-ICSS_Remoteproc_and_RPMsg

requested by the firmware, loads the binary into the instruction RAM of the PRUs and loads the resource table into the PRUs data RAM. Finally *pru_rproc* instructs the PRU cores to begin execution.

5.2 Programmable Real-time Units

The PRU-ICSS consists of two 32-bit RISC cores, 12 kB shared memory , 8 kB data memory, 8 kB instruction memory, internal peripheral modules, and an interrupt controller. These two cores can work together or completely independent depending on the firmware loaded onto it. The firmware can be written in assembly, C or C++. TI has provided libraries to support developers, this includes memory mappings for peripheral registers. The programmable nature of the PRU, along with its access to pins, events and all SoC resources, provides flexibility in implementing fast real-time responses, specialized data handling operations, custom peripheral interfaces, and in offloading tasks from the other processor cores of the system-on-chip. Figure 12 shows the details of the PRU-ICSS. All communications to and from the PRU-ICSS go through a 32-bit interconnect bus called the interface/OCP Master port. More information about the PRU-ICSSs workings and peripherals can be found in the AM335x Technical Reference Manual [13] chapter 4. This paragraph serves to give the reader a basic understanding of the PRU-ICSS and the different peripherals used during this project. Header files have been created with the aim of making the usage and understanding of these peripherals easier. These files can be found on the projects GitHub page under the corresponding folder.

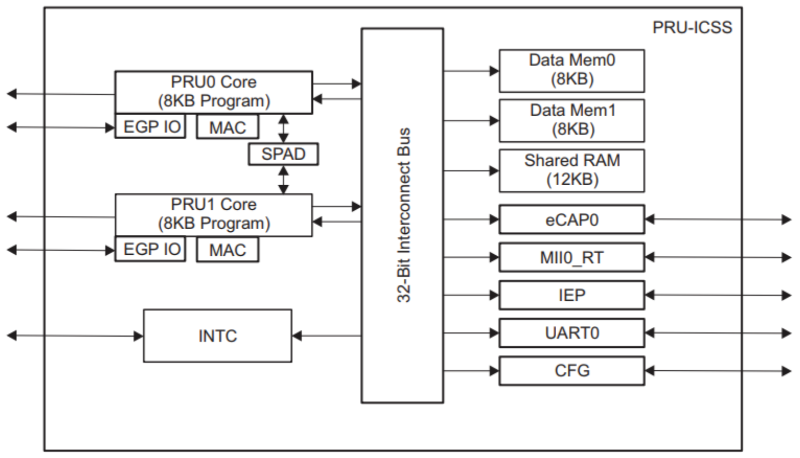


Figure 12: PRU-ICSS block diagram

5.2.1 Determinism

The PRUs are called programmable *real-time* units because they have been designed to function in a deterministic manner. All the assembly instructions available to the PRU are designed to complete in a set amount of time. Writes are always *fire & forget*, meaning that the core will not wait for a response it will fire the write and forget about it completely. The interrupt controller included in the PRU-ICSS is capable of sending and receiving interrupts to/from the PRUs, the ARM core and other peripherals. These interrupts are not capable of interrupting the running process on a PRU. The PRU has to be polling for the interrupt by reading register 31 bits [30:31]. The PRUs should not try to access devices outside of the PRU-ICSS if they want to be as deterministic as possible. This can cause undefined delays because of the Linux system trying to access other memory addresses. TI has defined many read latencies in an application report [16].

5.2.2 General Purpose Input and Output

Each PRU has 32 registers in its program RAM. Register 30 is the output register and register 31 is the input register. Not every pin is connected to a bit in the R[30:31] registers on the BeagleBone. Only bits [0:16] are connected to pin headers. There are four input modes and two output modes.

The input modes available are:

- Direct input
- 16-bit parallel capture
- 28-bit shift in

Direct input feeds the status of the input pins directly into PRU R31[0:16]. This is the default mode.

16-bit parallel capture mode captures 16 parallel bits on a rising or falling clock edge and saves them in R31[0:15]

28-bit shift in mode will continually capture and shift the input into R31[0:27] the shift rate is controlled by two cascaded divisors. Shift in mode also supports start bit detection and shift counter which sets R31[28] every 16 shifts.

The output modes available are:

- Direct output
- Shift out

Direct output feeds the contents of R30 directly to the output pins. This is the default mode.

Shift out mode shifts the contents of R30[0:15] into one of the two shadow registers. The contents of these registers in shifted out. The shift rate can be controlled by 2 cascaded divisors.

These modes can be set by writing into the appropriate registers. These can be found in chapter 4.5.9 of the technical reference manual [13].

5.2.3 Remote Processor Messaging

Communications with the PRU-ICSS is done using the Remoteproc and remote processor messaging (RPMsg). RPMsg is a message passing mechanism that requests resources through Remoteproc and builds on top of the existing virtio framework. Shared buffers are requested through the resource table mentioned in paragraph 4.1.5 and provided by remoteproc during the PRU firmware loading. The shared buffers are contained inside virtual ring buffers or vrings in the DDR memory. Two vrings are provided per PRU core, one is for PRU to ARM communications and the

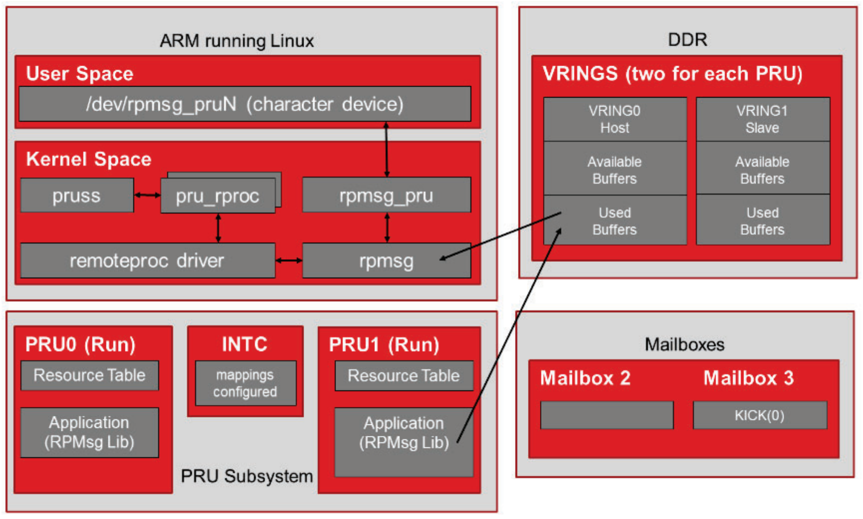


Figure 13: RPMsg block diagram

other is for ARM to PRU communications. These vrings are 512 bytes long with 16 of these bytes being a header. Interrupts are used to notify both the ARM and the PRU whenever messages are available in the shared buffer. Messages are received on the ARM side through character devices found in */dev/,* with their default names *''rpmgs-pru30/31".* TI has tried to abstract RPMsg on the PRU to a point where users can call pru_rpmsg_receive() or pru_rpmsg_send() to communicate with the ARM core.

A default resource table is available on this projects GitHub page under *Default files* or in Appendix F[1] – Resource Table together with a description. The resource table defines the interrupt mappings and vring data structures required for the PRU. These resources are initialized before the PRU core is started.

Sending a message through RPMsg is done by first initializing the underlying transport structures. The function *pru_rpmsg_init()* does this by taking the vrings and interrupt mappings provided by the resource table. Once this function is called, a channel can be created and used. Creating or destroying a channel is done with the function *pru_rpmsg_channel().* This function send a message to the ARM host letting it know a channel is to be created or destroyed. Messages can now be send from the ARM to the PRU and vice versa using the *pru_rpmsg_send()* function and the *pru_rpmsg_receive()* function. These functions require a source and a destination, which can only be provided by the ARM host. So before the PRU can send messages to the ARM, it will first have to receive a message from the ARM to determine the source of the message and the destination of the message to be send. All of these functions have four return values:

- PRU_RPMSG_NO_BUF_AVAILABLE; if these is no buffer available
- PRU_RPMSG_BUF_TOO_SMALL; if the buffer is too small
- PRU_RPMSG_INVALID_HEAD; if the head index returned is invalid
- PRU_RPMSG_SUCCES; if the message was successfully send

5.2.4 Debugging

One of the challenges with the PRU-ICSS is getting debug information out of it since it does not have the printf() function or any type of console. In this paragraph different methods of debugging the PRU are proposed.

The simplest method of debugging the PRU-ICSS is by connecting an external LED or using the on-board LEDs. Simply connect an LED through a breadboard and drive the LED by writing into R30. See the GPIO example on the GitHub page for an example on how to do this.

The second method is looking into the PRU-ICSS memory and registers. The internal registers can be accessed through the sysfs interface. Navigate to */sys/kernel/debug/remoteproc/remoteproc1/* and concatenate (cat) the regs file. This prints all the internal registers to the terminal. Accessing the PRU memory can be done using *'busybox devmem'* which is a tiny CLI utility that maps memory. Write *'sudo apt-get install busybox'* to install the utility. To read 4 bytes from the physical address 0xFFFFFFF write *'sudo busybox devmem 0xFFFFFFFF'*. A list of commonly used memory locations and offsets can be found in Appendix D.

Another method is to check the kernel log using 'dmesg'. Many errors, for example trying to load firmware without a resource table or the RPMsg message buffer being full, can be found here. Using 'dmesg –Hw' forces it to print in human readable form and to wait for new information. Having a terminal open with dmesg running can be very useful.

One of the most effective methods of extracting debug information is using UART over the serial port. This method is taken from the Arduino playbook and can be just as effective as using *printf()* to print debug information. Caution must be exercised since printing over the serial port can be a very slow process relative to other tasks.

5.2.5 Interrupt controller

The PRU-ICSS has its own interrupt controller(INTC) which serves as an interface between interrupts coming from all around the AM335x. These interrupts are called system events, the INTC can catch 64 of these system events. The system events available are specified in the TRM table 4-22. These system events have to be mapped to interrupt channels, the INTC has ten interrupt channels. Interrupts from these channels have to be mapped to host interrupts, the INTC also has ten of these. A graphical representation can be found in Figure 14.

Any of the 64 system events can be mapped to any of the 10 interrupt channels and multiple system events can be mapped to one interrupt channel. One system event should not be mapped to more than one interrupt channel this can cause undefined behavior. Any of the 10 interrupt channels can be mapped to any of the 10 host interrupts, but it is recommended to map interrupt channel *'Y'* to host interrupt *'Y'*. An interrupt channel should not be mapped to more than one host interrupt, this can also cause undefined behavior. Multiple interrupt channels can be mapped to a single host interrupt, but a channel cannot be mapped to multiple host interrupts, as this can cause undefined behavior. When multiple interrupt channels are connected to a single host interrupt their priority will be decided by their channel number. Lower channel numbers have a higher priority. The priority of multiple system events on a single channel is decided by the system event number. A lower number means higher priority.

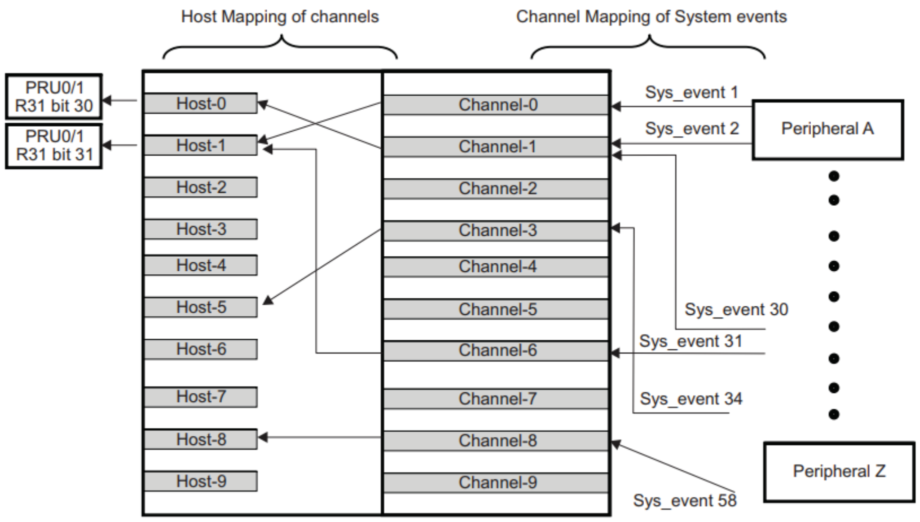


Figure 14: Interrupt mapping

Only two of the host interrupts are connected to the PRU-ICSS. These do not actually interrupt the PRU-ICSS, polling is required to detect an interrupt. Host interrupt 0 is connected to R31[30] (Internal register R31, bit 30) and host interrupt 1 is connected to R31[31] of both PRU0 and PRU1. Host interrupt 2-9 are for signaling the ARM interrupt controller or other peripherals such as EDMA. More information about the PRU-ICSS INTC can be found in the AM335x Technical Reference Manual [13] Chapter 4.4.2 page 225

5.2.6 Industrial Ethernet Peripheral

The Industrial Ethernet Peripheral(IEP) is a module capable of performing the hardware work for industrial Ethernet functions. This includes an industrial Ethernet timer with 8 compare events, industrial Ethernet sync generator and latch capture, industrial Ethernet watchdog timer, and a digital I/O port (DIGIO). Using the IEP for industrial Ethernet functions is out of the scope of this project. The IEP is useful because of its industrial Ethernet timer which features:

- A 200Mhz clock
- 32-bit count-up timer with overflow status bit
- Programmable increment value (1 – 16)
- Ten 32-bit capture registers
- Eight 32-bit compare registers
- System event for both capture and compare event

This allows it to either count until a capture events occurs or it can count and compare to a compare register so that it will generate an interrupt when the counter and the compare register are of equal value. More information can be found in the AM335x Technical Reference Manual [13] Chapter 4.4.3 page 233.

5.2.7 Enhanced Capture Module

The Enhanced Capture Module(eCAP) is a single capture channel that can be used for:

- Sample rate measurements
- Speed measurements
- Period/Duty cycle measurements

It features:

- 32-bit time base counter
- 4-event time-stamp registers
- Edge polarity selection for up to four sequenced time-stamp capture events
- Interrupt on either of the four events
- Single shot capture of up to four event time-stamps
- Continuous mode capture of time-stamps in a four deep circular buffer
- Absolute time-stamp capture
- Difference (Delta) mode time-stamp capture
- All of these resources dedicated to a single input pin
- When not used in capture mode the eCAP module can be configured as single channel PWM output

The eCAP module can capture falling or rising edges on its input pin. It can also generate PWM signals but that is outside of the scope of this project. During a capture event it can save the absolute timestamp since the eCAP module started running, the time since the last event, or the difference between two time stamps. This is useful for many applications such as tracking the period/frequency of a function generator using its sync output. This is a necessary functionality to track the frequency of the chopper so this mode will be worked out. Many modes are described and worked out in the AM335x Technical Reference Manual [13] chapter 15.3 page 2465. We use the Time Difference (Delta) Operation Rising Edge Trigger Example with some slight modifications (Page 2480 of the TRM). This example records the time between two rising edges and uses the four capture registers as a ring buffer to save the timestamps, as can be seen in Figure 15. The only modification to the register initializations provided by TI is the removal of the four register ring buffer so all the timestamps go to the same capture register. This allows us to create a single pointer to this register and thus easily read the value saved there.

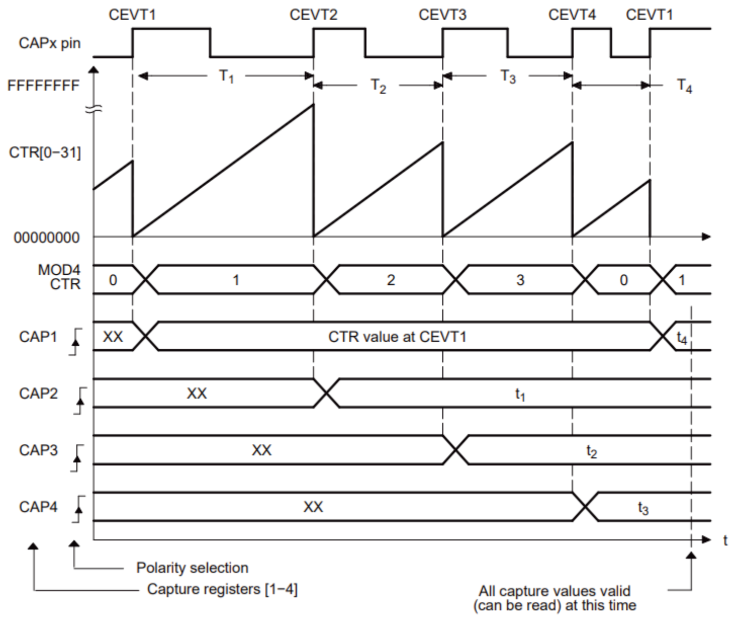


Figure 15: Time Difference (Delta) Operation Rising Edge Trigger Example

More information can be found in the AM335x Technical Reference Manual [13] Chapter 15.3 page 2465.

5.2.8 On-chip Analog-to-Digital Converter

The analog-to-digital converter included in the AM335x SOC is called the touchscreen controller and analog-to-digital converter subsystem (TSC_ADC_SS). This is an 8-channel general-purpose analog-to-digital converter with optional support for interleaving touchscreen conversion(TSC) for a 4-/5-/8-wire resistive panel. It has four configurations:

- 8 general-purpose ADC channels
- 4-wire TSC with 4 ADC channels
- 5-wire TSC with 3 ADC channels
- 8-wire TSC

Using a touchscreen is not necessary for this project so this will not be discussed. The TSC_ADC_SS will be used as an 8 channel general-purpose ADC. This ADC is capable of:

- 12-bit conversion
- Sampling rate as fast as 15 ADC clock cycles
- Single shot conversion
- Continuous conversion
- Sequence through all channels for conversion
- Programmable delay
- Programmable averaging

- Differential or single-ended
- Store data in FIFO buffers
- Progammable EDMA events
- Start/stop bit in register
- Interrupts

More information can be found in the AM335x Technical Reference Manual [13] Chapter 12 page 1828.

5.2.9 Multi-channel Serial Peripheral Interface

The AM335x SOC has a SPI peripheral called the Multi-channel Serial Peripheral Interface (McSPI). This is a general-purpose receive/transmit master/slave controller that can control four slaves devices or be controlled by one master device. It is capable of duplex, synchronous, and serial communication between a CPU and SPI device. It features:

- One word deep receive/transmit data registers
- Two DMA requests per channel
- Single interrupt line for multiple interrupt events
- Serial interface supports:
    - Full/half duplex
    - Multi-channel master or single channel slave
    - Programmable 1-32 bit transmit/receive shift operations
    - SPI world lengths continuous from 4-32 bits
- Four SPI channels
- SPI configuration per channel (clock, polarity, word length)
- Programmable clock

More information on the different configurations can be found in the AM335x Technical Reference Manual [13] in chapter 24 page 4883. Header files have been made with the aim of abstracting the use of the McSPI. One difference between these header files, and header files for previous peripherals is the fact that SPI devices can have many slightly different protocols. An ADC, for example, can require a conversion start pulse, this would require some form of bit-banging for that specific device. The header files do not include an device specific functions or definitions for this reason. They simplify the configuration of the McSPI peripheral.

LTC1859

The LTC1859 is used for this project. This is an 8-channel, 16-bit, 100ksps ADC with a programmable input range from 0-5V, +/- 5V, 0-10V, +/- 10V (both single ended and differential). This ADC operates in SPI mode 0, meaning a clock polarity of 0 (clock idles at 0 and each cycles consists of a pulse of 1), a clock phase of 0 (information gets clocked in on the leading edge and clocked out on the trailing edge). It has a maximum clock frequency of 20Mhz and clock data in with MSB first. The operating sequence for the LTC1859 can be found in its datasheet [9] or in Appendix H – LTC1859 operating sequence. Here you can also find the configuration for the input word. The operating sequence boils down to:

1. Send an 8-bit configuration word of the Serial Data In (SDI) (MOSI) line. This configuration word contains information about the MUX address, input range and power mode.

2. The ADC will start a conversion by setting the CONVST pin (pin 28 on the IC) high for 30ns, one conversion takes ~4000ns. The BUSY line on the ADC (pin 22 on the IC) will be low while a conversion is going on and high when nothing is going on.
3. The data will be clocked out on the next clock cycles after the conversion via the Serial Data Out(SDO)(MISO) line. This will only work while the RS pin(Pin 27 on the IC) is low. A new configuration word is clocked in at the same time.

Some form of bit-manipulation will be required since the McSPI module is not capable of changing the output of a pin arbitrarily. This has to be done to start the conversion (Setting CONVST to high) and optionally to read the output(Setting RD to low).

5.2.10 Multiply and Accumulate

Each PRU core has its own Multiplier with optional accumulation (MAC). This is a module capable of multiplying two 32-bit numbers into one 64-bit result. Since the PRU is a 32-bit processor, doing a 32-bit multiplication would be impossible (or at least take a long time). The MAC can do one of these multiplications in one clock cycle. Not only can it do 32-bit multiplication, it can do any multiplication making multiplication a single cycle instruction. Using the MAC requires setting a compiler flag *'hardware_mac=on'* this flag will tell the compiler to try and do every multiplication using the MAC. Not only making 32-bit multiplication possible but also increasing the speed for other multiplications.

5.2.11 Python PRU module

A Python module has been made to simplify the usage of the PRU-ICSS. Instead of having to start and stop the PRU manually using the sysfs interface or having to read the output of the RPMsg character device using 'cat' in the terminal, a simple python module can be written, include the module and start using the PRU-ICSS.

Functions have been included for the following functionalities:

- Status(*pru*); Returns the current status of the selected pru
- Start(*pru*); This method starts a PRU via the sysfs interface.
- Stop(*pru*); This method stops a PRU via the sysfs interface.
- Open_char(*pru*); This method creates a file object for the RPMsg character device.
- Close_char(*pru*); This method destroys a file object for the selected character device.
- Transmit(*pru, message*); This method transmits a message to the PRU-ICSS using RPMsg
- Receive(*pru, type, samples=0*); This method receives messages from the PRU-ICSS using RPMsg

The files for this module can be found on this projects GitHub page under *Python/PRU-ICSS*.

5.3 Digital Lock-in Amplifier

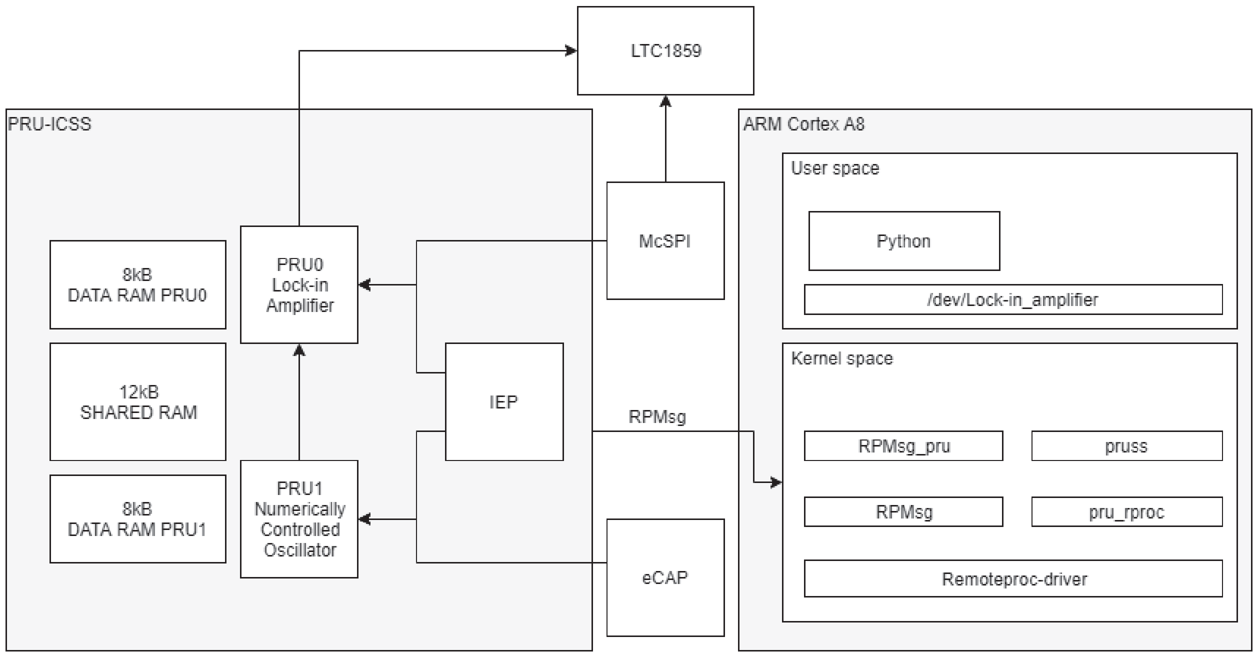


Figure 16: Block diagram of the Lock-in Amplifier Implementation on the PRU-ICSS

Figure 16 shows a block diagram describing a lock-in amplifier. The IEP module will define the sample frequency, interrupting both PRUs when a certain amount of time has passed. PRU1 will generate the sine and cosine waves for the quadrature multiplication, this will be done using a NCO. Using the period captured by the eCAP module, from the SYNC output of a function generator, a sine and cosine wave can be generated at the same frequency. How sine and cosine wave samples are generated from the period will be discussed in paragraph 5.3.1. The NCO will generate samples and place them in shared RAM every time the IEP generates an interrupt. When an interrupt is received by PRU0 it will sample the LTC1859 and wait for PRU1 to be done with generating the reference signals. PRU0 will grab these samples from the shared memory and do the quadrature multiplication. Filtering will be explained in paragraph 5.3.2. The results will be send to the ARM host processor via RPMsg. A flowchart can be found in Appendix F, and the code can be found on the projects GitHub page under *Lock-in.* The pin configuration for the lock-in amplifier can be found in Appendix C[2] – Lock-in pins.

5.3.1 Numerically Controlled Oscillator

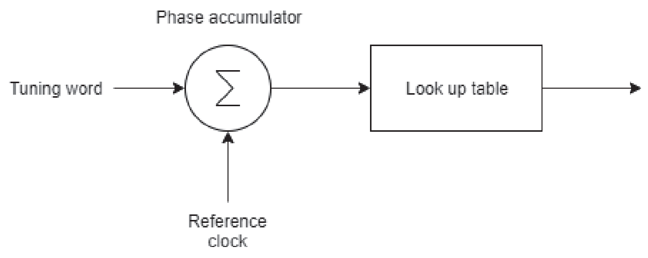


Figure 17: DDS Block diagram

Generating a sine and cosine wave can be a very computationally intensive tasks. This common task is often made less intensive by saving the values for an entire period in a look up table. Imagine having a look up table with 1000 samples that together represent one period of a sine wave. Stepping through this look up table with a step size of one and a frequency 100Hz will result in a sine wave with a frequency of 0.1Hz. Increasing the step size to two will result in a sine wave with a frequency of 0.5Hz. This means a sine wave with any frequency (in accordance to the sampling theorem) can be generated by changing the step size. The technique used for this is called Direct Digital Synthesis. The standard equation for Direct Digital Synthesis [14] states:

$$\boldsymbol{F_{out}} = \frac{\Delta\emptyset * \boldsymbol{F_s}}{\boldsymbol{2^N}} \quad \forall \quad \boldsymbol{F_{out}} \leq \frac{\boldsymbol{F_s}}{\boldsymbol{2}} \qquad (5\text{-}1)$$

$$F_{out} = Output\ frequency$$

$$\Delta\emptyset = Tuning\ word, step\ size$$

$$F_s = Sample\ frequency$$

$$N = The\ number\ of\ bits\ the\ tuning\ word$$

Figure 17 shows how the tuning word is being added to the phase accumulator on every clock cycle. The phase accumulator is the current index for the look up table and the tuning word is the step size calculated using equation 5-1. Equation 5-2 Can be used to calculate the Tuning word required for generating a certain frequency at a certain sample rate. The restraint in equation 5-1 comes from the sampling theorem. Periods will be used since the eCAP module captures the period from the function generator, simplifying the calculations (equation 5-2).

$$\Delta\emptyset = \frac{\boldsymbol{F_{out}} * \boldsymbol{2^N}}{\boldsymbol{F_s}} \rightarrow \Delta\emptyset = \frac{\boldsymbol{2^N} * \boldsymbol{T_s}}{\boldsymbol{T_{out}}} \qquad (5\text{-}2)$$

A look-up table with 4096 entries will have an N of 12. According to equation 5-3 This NCO will have a frequency resolution of 0,024 Hz at a sample frequency of 100 Hz. The frequency resolution can be improved by using fixed point numbers [15] instead of integer numbers as tuning word. This gives more bits in the tuning word. For example using a Q12.16 fixed point number as tuning word would allow a frequency resolution of $3{,}7 * 10^{-7}$ Hz at 100 Hz.

$$\boldsymbol{F_{res}} = \frac{\boldsymbol{F_s}}{\boldsymbol{2^N}} \qquad (5\text{-}3)$$

A Q12.16 fixed point number has a 12-bit integer part and a 16 bit fractional part. Only the integer part can be used as index for the look up table introducing error because an index of 1.5 does not exist in an array. The index values 1,5; 1,7; 1,9 represented as fixed point numbers would result in 3 outputs with index 1, unless interpolation is used.

Using interpolation enables us to take the fractional part into account for the result. The standard formula for linear interpolation is:

$$Y_{out} = X_1 + \frac{X_2 - X_1}{2^N} * t \qquad (5\text{-}4)$$

$$X_1 = LUT\ output\ at\ the\ integer\ index$$

$$X_2 = LUT\ output\ at\ the\ (integer\ index + 1)$$

$$N = number\ of\ fractional\ bits$$

$$t = the\ fractional\ part\ of\ the\ index$$

Figure 18 shows eight periods of a sine wave generated with interpolation and a sine wave generated without interpolation and their respective frequency plots. There are higher harmonics when not using interpolation although the time domain looks almost identical. This is because of the small inaccuracies caused by the rounding of the index, which is not necessarily causing distortion in the time domain but causes periodic inaccuracies resulting in distortion in the frequency domain.

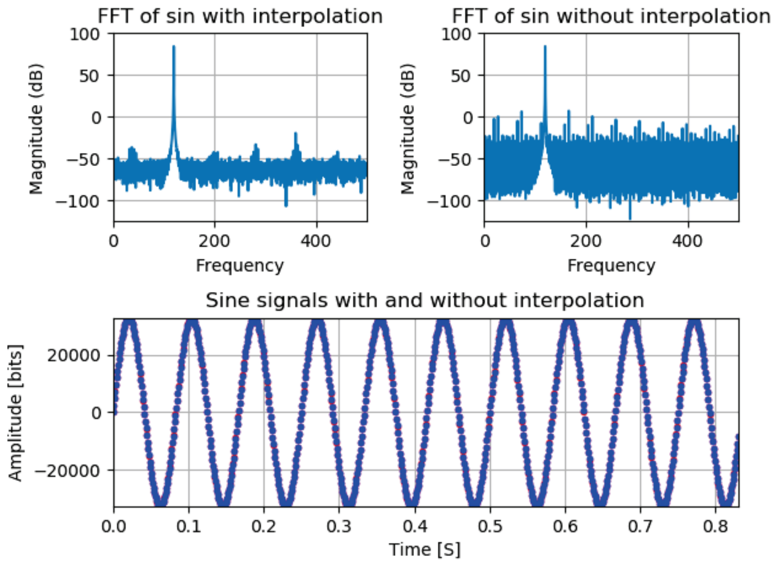


Figure 18: Time and frequency domain plots of the interpolated and not interpolated signal

5.3.2 Mixer

The output of the ADC is a 16-bit, two's complement number. This means that the voltages of +/- 5V are linearly mapped to the values +/- 32767. Equations 4-14 and 4-15 states that as long as the reference amplitude is 2V and the phase is 0° we should get the amplitude of the original experiment signal back. The problem here is that the PRU-ICSS is not able to generate a reference signal with an amplitude of 2, since it does not have any floating point hardware. This means that a fixed point number has to be used. A Q2.14 fixed point number was used for this purpose. Doing the multiplication with the scaled version of the 2V reference signal and then bit shifting the result to the right by 14 bits(or diving by 2^{14}) is the same as simply multiplying the ADC output with a 2V reference signal. A plot of the ADC output and reference signals can be found in Figure 19, these are both the time domain plot and frequency domain plots.

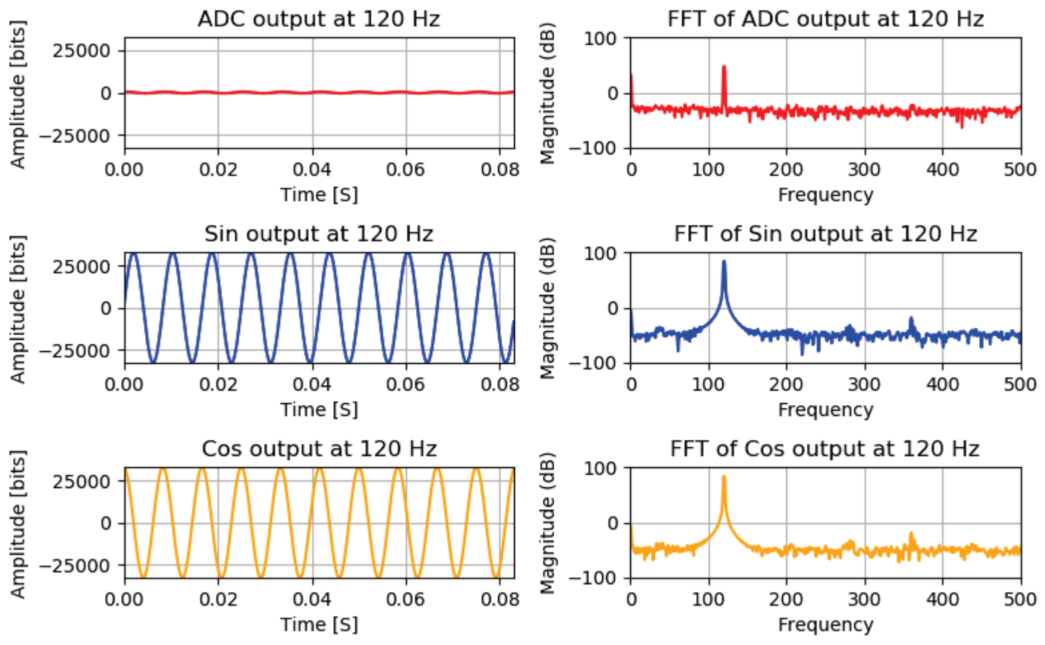


Figure 19: Time and frequency domain plots of the ADC input and the sine and cosine reference signals

The results of the mixing can be found in Figure 20. These plots show that both signals get a DC offset and are double the original frequency, as equation 4-12 predicted.

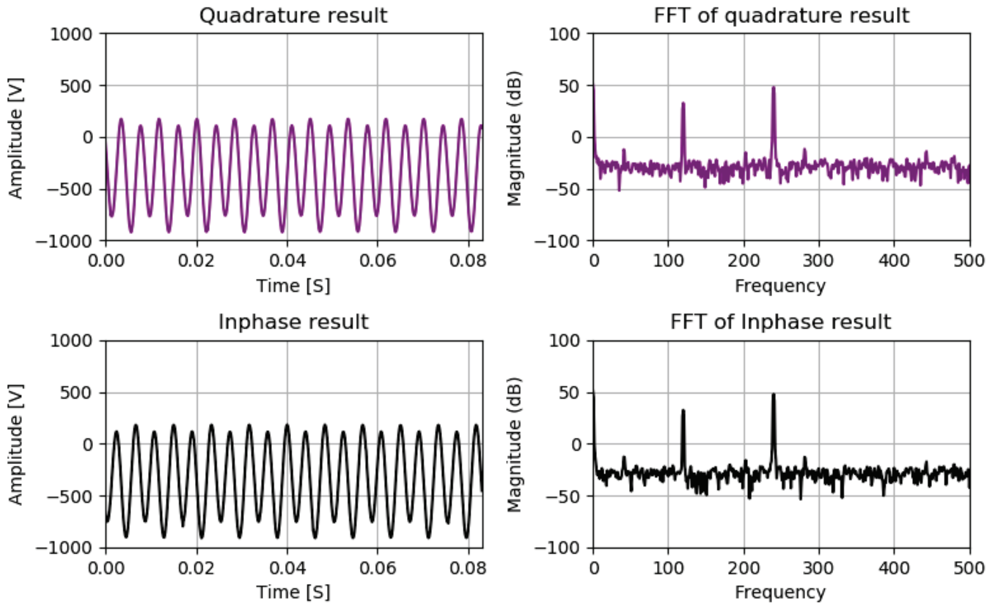


Figure 20: Time and frequency plots of the quadrature mixing

The oscillating component can be removed by low-pass filtering the signal, leaving only the DC component. Filters will be discussed in the next paragraph.

5.3.3 Filtering

The mixing has to be followed by a low-pass filter to remove the component with twice the original frequency and any other noise in the signal. The stopband attenuation of the filter will define how much noise can be rejected.

There are two types of filters in digital signal processing, Infinite Impulse Response (IIR) and Finite Impulse Response (FIR). The mathematical representations of these filters can be found in equations 5-4 and 5-5

$$\boldsymbol{FIR}: \boldsymbol{y}(\boldsymbol{n}) = \sum_{\boldsymbol{k}=0}^{\boldsymbol{N}} \boldsymbol{a}(\boldsymbol{k}) * \boldsymbol{x}(\boldsymbol{n}-\boldsymbol{k}) \qquad (5\text{-}5)$$

$$\boldsymbol{IIR}: \boldsymbol{y}(\boldsymbol{n}) = \sum_{\boldsymbol{k}=0}^{\boldsymbol{N}} \boldsymbol{a}(\boldsymbol{k}) * \boldsymbol{x}(\boldsymbol{n}-\boldsymbol{k}) + \sum_{\boldsymbol{j}=0}^{\boldsymbol{P}} \boldsymbol{b}(\boldsymbol{j}) * \boldsymbol{y}(\boldsymbol{n}-\boldsymbol{j}) \qquad (5\text{-}6)$$

These equations show one important difference between the two filters. The IIR uses the previous inputs and outputs to generate the new output in comparison to FIR filters that only uses the current and previous inputs.

FIR low-pass filter

FIR filters are called 'finite' because their impulse response returns to zero after some time, while IIR filters would keep ringing infinitely because of their previous inputs. FIR filters are easy to design and are always very stable with a constant phase delay. Almost any type of frequency response can be created with a FIR filter, but this comes at a price, namely, coefficients that take up memory. Some frequency responses are just not practical to implement as a FIR filter.

The most common technique for designing FIR filters is using the windowed-sinc method. It works by multiplying the sinc function and a window function such as the popular Blackman window(Figure 21), and then convolving the created filter kernel with the signal you want to filter. The sinc function would represent a brick-wall filter, which is impossible to create in practice because of the fact that it would require an infinite length. One solution would be to just truncate the signal, but this would cause excessive ripple in the pass and stop band. The windowed-sinc method prevents this excessive ripple by attenuating the sinc function, not completely cutting it.

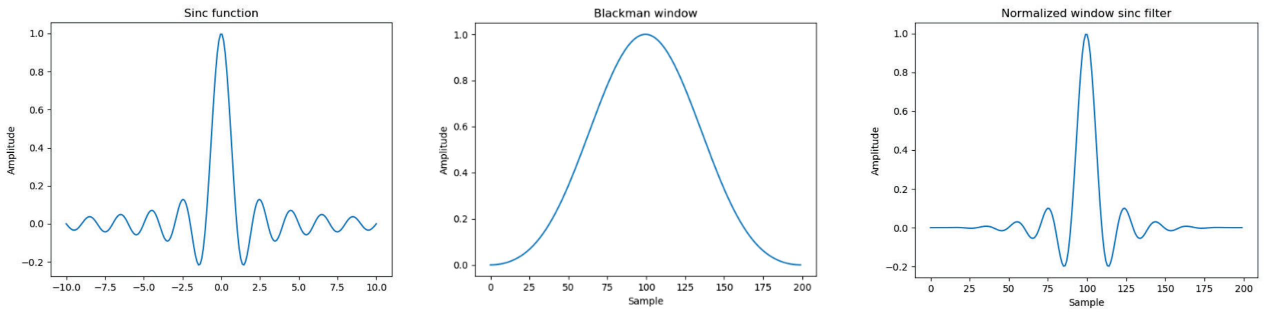
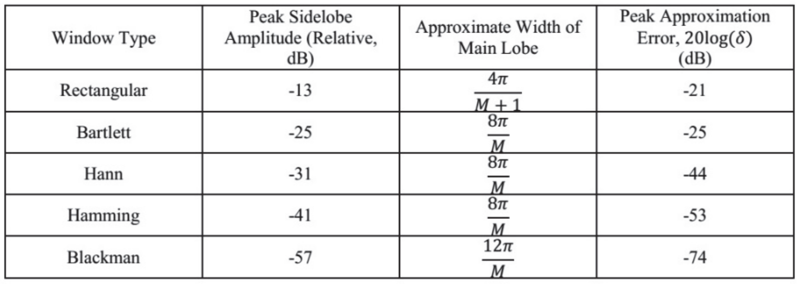


Figure 21: the windowed sinc method, visualed. Shows the sinc function, the Blackman window and the result of their multiplication

Designing your own filter kernel using the sinc function and a window is not necessarily hard, but the scipy Python module can greatly simply this task. Requiring you to specify the cut-off frequency, filter length and the type of window you want to use. The filter length defines the transition bandwidth, the bandwidth in which you go from the passband to the stopband. The filter length can be approximated with equation 5-6, where *M* is the filter length and *b* is the transition bandwidth as a fraction of the sample frequency.

$$\boldsymbol{b} \approx \frac{\boldsymbol{4}}{\boldsymbol{M}} \qquad (5\text{-}7)$$

Different windows exist which require different filter lengths to achieve certain transition bandwidths, and have different amounts of stop band attenuation. Figure 22 shows the different filter types with their stop band attenuation, and how to calculate their transition bandwidth more accurately. These are still approximations.



Window Type	Peak Sidelobe Amplitude (Relative, dB)	Approximate Width of Main Lobe	Peak Approximation Error, $20\log(\delta)$ (dB)
Rectangular	-13	$\frac{4\pi}{M+1}$	-21
Bartlett	-25	$\frac{8\pi}{M}$	-25
Hann	-31	$\frac{8\pi}{M}$	-44
Hamming	-41	$\frac{8\pi}{M}$	-53
Blackman	-57	$\frac{12\pi}{M}$	-74

Figure 22: Different window types with their stopband attenuation and transition bandwidth

Figure 23 shows the frequency response of FIR filters with a hamming window, a cut-off frequency of 10Hz and different numbers of taps.

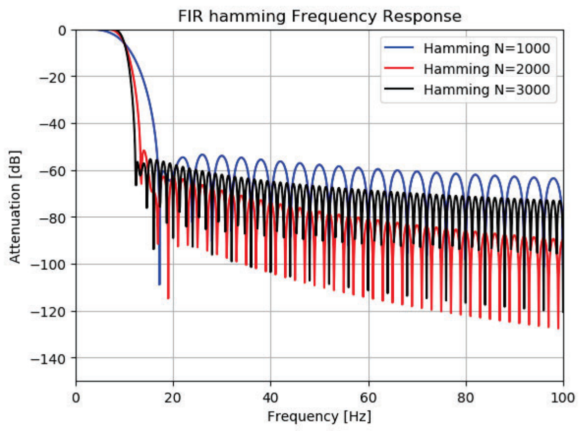


Figure 23: FIR filter with 1000, 2000, 3000 taps

IIR low-pass filter

IIR filters are called 'infinite' because their impulse response will keep ringing indefinitely. The advantage of using IIR filters is the fact that they require less coefficients to achieve a certain frequency response, they are also very fast to calculate. The disadvantage is that they can suddenly crash if not treated correctly, their phase delay is also not predictable, but because they require less coefficients for certain frequency response they will have less delay.

The most commonly used IIR filter design methods are Butterworth, Chebyshev(Type I & II) and Elliptic prototype methods. They are designed by first calculating the analog filter prototype and then transforming it to its discrete equivalent using bilinear z-transform. Each of these design methods has their own advantages and disadvantages. The process of designing these filters is automated by a lot of software packages such as MATLAB or Pythons *Scipy* module. These only require the cut-off frequency, samples frequency and the filter order.

Elliptic filters are known to have the fastest roll-off of all the filter prototypes, requiring the least amount of coefficients for a specific transition width. A disadvantage is the fact that it has quite a lot of ripple in both the pass and stop band.

Butterworth filters are often praised for their maximally flat passband and stopband, but this flatness comes with the disadvantage of a very slow roll-off. Requiring a high order for a small transition width.

The Chebyshev filter design method has two types. The difference between these types is the fact that type I has pass band ripple and type II has stopband ripple. Type I has a maximally flat stopband and a faster roll-off than Butterworth and Type II Chebyshev. Because of this, it is considered a good compromise between Butterworth and Elliptic. Type II has a maximally flat passband with a faster roll-off than Butterworth, but slower than Type I and is considered a good choice for DC measurement applications. The different frequency responses and their ripple and roll-off can be found in Figure 24. These are 6th order IIR filter with a 10Hz cut-off frequency and 40dB stopband attenuation.

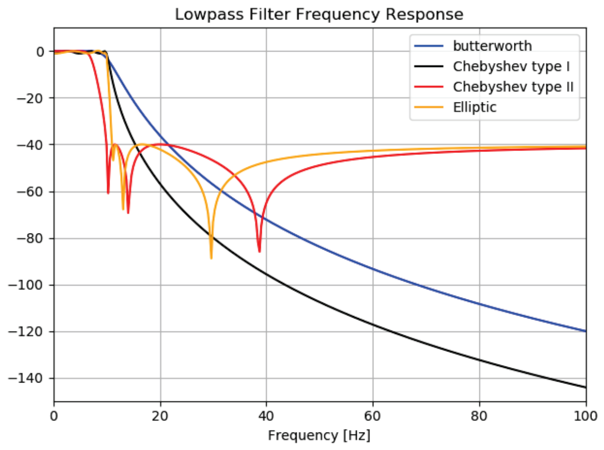


Figure 24: Different IIR filter design methods and their frequency responses

Moving Average Filter

The moving average filter is a special type of FIR filter. They are often used for smoothing out data or noise rejection. This is because they are very easy to implement and very fast in calculation. A moving average filter is defined by the transfer function in equation 5-7.

$$\boldsymbol{H(z)} = \frac{\boldsymbol{1}}{\boldsymbol{L}}\sum_{\boldsymbol{k=0}}^{\boldsymbol{L-1}} \boldsymbol{Z^{-k}} \qquad (5\text{-}8)$$

Equation 5-7 tells us that the numerator coefficients b_i and denominator coefficients a_i of an *L*-point moving average filter can be defined as

$$b_i = \left[\frac{1}{L}, \frac{1}{L}, \frac{1}{L}, \dots, \frac{1}{L}\right], i = 0, 1, 2, \dots, L-1$$

$$a_i = [1], i = 0$$

Figure 25 Shows the frequency response of moving average filters with different values for *L*. An increase in *L* causes the cut-off frequency to go down and increases the amount of attenuation in the stop-band, although there is a lot of ripple in the stopband. One disadvantage of the moving average filter is the fact that there is no direct relationship between *L* and the cut-off frequency. This means that choosing a value for *L* comes down to trial and error.

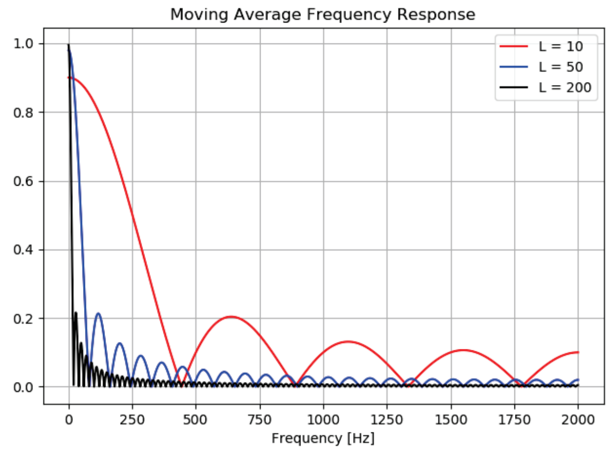


Figure 25: Frequency response of moving average filter with L=10, L=50, L = 200

6. Evaluation

6.1 Testing

Test plan local oscillator

PRU-ICSS Numerically Controlled Oscillator V1.0

Introduction

A NCO is used to generate the two reference signals needed for the quadrature multiplication. The performance has to be tested so the NCO can be characterized for future usage. The objective is to see if the NCO can generate sine waves when given a frequency or period. Another important factor is the amount of SNR and the frequency resolution of the NCO.

Features to be tested

- Ability to generate sine wave with a given frequency
- Signal-to-Noise ratio

Features not to be tested

- NaN

Approach

The first version of the algorithm is designed in Python, when this functions properly it can be replicated on the PRU-ICSS in C. This is done because the behavior of the Python interpreter is well defined compared to the behavior of the PRU-ICSS compiler. Python also has analytic tools included in modules such as Numpy and Scipy making testing a quick task. Debugging in Python is also easier than debugging on the PRU-ICSS because there is no need to recompile/upload the entire firmware every time changes are implemented. The performance of the NCO in Python has been tested and compared to the results of the NCO on the PRU-ICSS. This includes comparison of the recorded period, step-size, accumulator and the output. The accumulator is chosen to be 26 bits long with 10 index bits and 16 fractional bits.

Pass/fail criteria

#	Description	Criteria
1	Ability to generate sine wave with a given frequency	The dominant frequency will be calculated using the Fourier transform. This has to equal the given frequency. The frequency is given through the eCAP module and configured on the function generator.
2	Signal-To-Noise ratio	Compare the output of the NCO to a sinus of the same frequency but without quantization and harmonics.

Environmental requirements

- Python3 IDE (Atom used with the Hydrogen package)
- BeagleBone Black
- Access to the PRU-ICSS
- Code for the NCO in Python (GitHub under *Python/Pure_NCO.py*)
- Code for the NCO in PRU-ICSS C (GitHub under *NCO/Example*)

Testing

1. **Ability to generate sine wave with a given frequency**
The NCO programmed in Python will be used as reference. The performance achieved by the Python code is the performance desired for the PRU-ICSS. The code for the PRU-ICSS can be found on the GitHub page under *NCO/Example*.

Inputs:

- Sample frequency
- Desired frequency

Outputs:

- Quantized sine generated by Numpy *sin* function
- NCO output value
- Step-size (Incrementor)
- Accumulator
- Dominant frequency
- NCO resolution

Using different input parameters we will test the performance of the NCO in Python. The output of the NCO with a sample frequency of 1000Hz and a desired frequency of 100Hz can be found on the GitHub page under output files: *Py_NCO_out.out*

Sample frequency: 1000Hz Desired frequency: 100Hz	Sample frequency: 4000Hz Desired frequency: 100Hz	Sample frequency: 10000Hz Desired frequency: 100Hz
Step-size: 102	Step-size: 25	Step-size: 10
Dominant frequency: 100.00Hz	Dominant frequency: 100.00Hz	Dominant frequency: 100.00Hz
NCO resolution: 1,5 * 10^{-5} Hz	NCO resolution: 6 * 10^{-5} Hz	NCO resolution: 1,5 * 10^{-5} Hz
Sample frequency: 1000Hz Desired frequency: 200Hz	Sample frequency: 4000Hz Desired frequency: 200Hz	Sample frequency: 10000Hz Desired frequency: 200Hz
Step-size: 204	Step-size: 51	Step-size: 20
Dominant frequency: 200.00Hz	Dominant frequency: 200.00Hz	Dominant frequency: 200.00Hz
NCO resolution: 1,5 * 10^{-5} Hz	NCO resolution: 6 * 10^{-5} Hz	NCO resolution: 1,5 * 10^{-5} Hz

These same parameters will be used in the NCO on the PRU-ICSS, the only difference being that we feed the frequency through the eCAP module with a function generator. This should return exactly the same results as the NCO in Python did. The output files for this can be found on the GitHub page under output files: *BBB_NCO_out.out.* The NCO resolution will not be calculated here since the resolution is defined by equation 5-3 and does not vary.

Sample frequency: 1000Hz Desired frequency: 100Hz	Sample frequency: 4000Hz Desired frequency: 100Hz	Sample frequency: 10000Hz Desired frequency: 100Hz
Step-size: 102	Step-size: 25	Step-size: 10
Dominant frequency: 100.00Hz	Dominant frequency: 100.00Hz	Dominant frequency: 100.00Hz
Sample frequency: 1000Hz Desired frequency: 200Hz	Sample frequency: 4000Hz Desired frequency: 200Hz	Sample frequency: 10000Hz Desired frequency: 200Hz
Step-size: 204	Step-size: 51	Step-size: 20
Dominant frequency: 200.00Hz	Dominant frequency: 200.00Hz	Dominant frequency: 200.00Hz

Figure 26 Shows plots of the Fourier Transform of the Python NCO output, PRU-ICSS NCO output and the sine wave generated by the Numpy *sin* function.

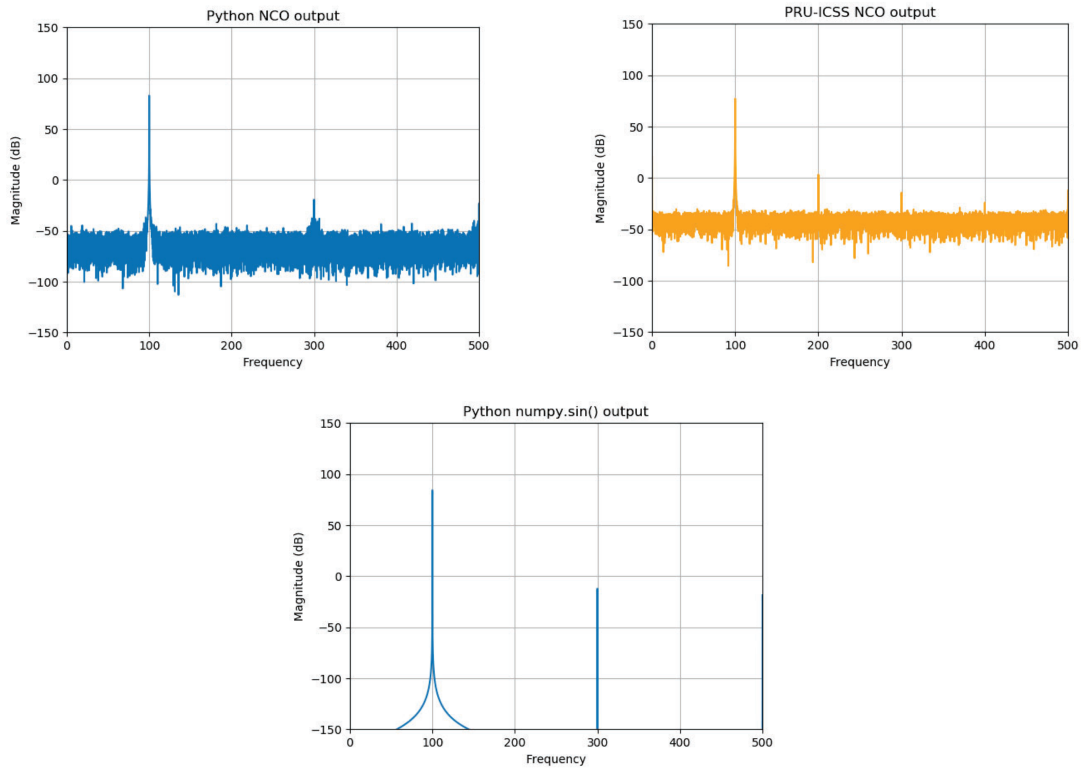


Figure 26: Python NCO output, PRU-ICSS NCO output and the Python *numpy.sin()* function

2. **Signal-to-Noise ratio**

All the higher harmonics in the reference wave will be multiplied with the signal from the chopper. The SNR of the NCO has to be as high as possible to prevent faulty multiplications. The harmonics could get multiplied into the 0Hz signal if there are too many harmonics. The SNR will be calculated in dB by taking the FFT of the PRU-ICSS output, and measuring the distance between the peak frequency and the harmonics. This is done with different LUT lengths and with/without interpolation.

Inputs:

- LUT size
- Interpolation?

Outputs:

- Signal-to-Noise Ratio

No interpolation LUT size: 256	No interpolation LUT size: 512	No interpolation LUT size: 1024
SNR: 20dB	SNR: 30dB	SNR:40dB
Interpolation LUT size: 256	Interpolation LUT size: 512	Interpolation LUT size: 1024
SNR: 50dB	SNR: 50dB	SNR: 50dB

Test plan Lock-in amplifier

PRU-ICSS Lock-in Amplifier V1.0

Introduction

This paragraph explains the activities performed as part of the testing of the BeagleBone Black – PRU-ICSS Lock-in Amplifier.

The objective of this project was to create a digital lock-in amplifier. The performance of this Lock-in amplifier has to be tested so the lock-in amplifier is properly characterized for future usage. The objective is to see how much noise the lock-in amplifier can reject before the output becomes unacceptable. This will be done with the highest possible sample frequencies and filter lengths using multiple channels. This will result in a clearly characterized lock-in amplifier.

Features to be tested

- Dynamic range of the lock-in amplifier, single channel
- Dynamic range of the lock-in amplifier, multi-channel

Features not to be tested

- Resolution

Approach

One PRU samples the ADC while the other core generates sine/cosine samples for the quadrature multiplication. This multiplication is done to generate the Quadrature and In-phase signals. These signals are transferred to the ARM core which in turn sends it to a computer for filtering. Python will be used to analyze the received signals. Two function generators will be used, one controlled by Python to increase the amount of noise and one outputting a constant signal. These signals are added together with a T-piece and divided over the 4 channels.

Pass/Fail criteria

#	Description	Criteria
1	Single channel: Dynamic range of the lock-in amplifier	A signal with variable magnitude will be injected. This signal should be completely removed after filtering.
2	Multi-channel: Dynamic range of the lock-in amplifier	A signal with variable magnitude will be injected. This signal should be completely removed after filtering. This should be done over multiple channels at the same time

Environmental requirements

- Python3 IDE
- BeagleBone Black
- Access to the PRU-ICSS
- (1 – 4) Function generators
- LTC1859 development board
- Code for the Lock-in amplifier in Python (GitHub under *Python/Pure_Lockin.py*)
- Code for the Lock-in amplifier in PRU-ICSS C (Github under *Lock-in*)

Testing

1. **Single channel: Dynamic range at different sample frequencies and integration times**
   The dynamic range of a lock-in amplifier is tested by injecting the signal to be measured with white noise. This noise should be completely filtered away at the output, leaving only the amplitude of the original signal.

   Inputs:

   - Sample frequency
   - Filter length
   - Noise frequency
   - Noise amplitude

   Outputs:

   - Lock-in amplifier output
   - Signal-to-Noise ratio

   A Function generator will generate a signal at the reference frequency. Another function generator will inject that signal with a signal at an arbitrary frequency. Changing the voltage of the injected signal will change the amount of SNR.

Reference frequency: 100Hz Noise bandwidth: 500Hz Sample frequency: 4000Hz FIR hamming low-pass length: 1000 FIR hamming low-pass cut-off: 1Hz Input amplitude: 75 mV	
Input SNR	Output
5dB	74.59 mV
3dB	74.72 mV
1dB	74.73 mV
-1dB	74.74 mV
-3dB	74.60 mV
-5dB	74.60 mV
-7dB	74.56 mV
-9dB	74.56 mV
-11dB	74.71 mV
-13dB	74.76 mV
-15dB	74.87 mV
-17dB	74.81 mV
-19dB	74.92 mV
-21dB	74.73 mV
-23dB	74.92 mV
-25dB	74.93 mV
-27dB	74.85 mV
-29dB	74.94 mV
-30dB	74.76 mV

Figure 27 Shows plots of the signal at different stages of the process

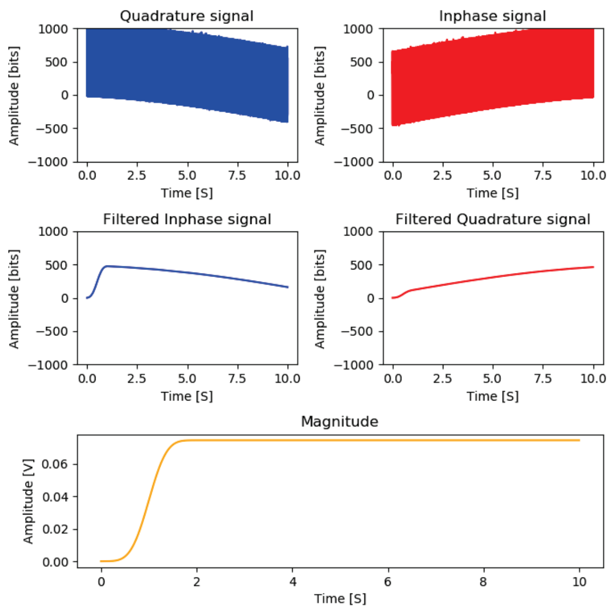


Figure 27: The Quadrature and In-phase signals, filtered versions and the output magnitude

2. Multi-channel: Dynamic range at different sample frequencies and integration times

The requirements for this project state that the lock-in amplifier should be capable of sampling 4 to 8 channels. This is done by doing the same test as in Test 1 but over multiple channels with multiple function generators. The crosstalk between channels was tested by blocking of 3 of the 4 channels with 50 Ω caps. Measurements of the open channel are taken to create a baseline. Then 5 V_{pp} white noise is forced into the 3 blocked channels and take another measurement of the first channel. No crosstalk was measured while doing these measurements. Or the crosstalk was so small that it was undiscernible from standard noise included in the signal.

Inputs:

- Sample frequency
- Filter length
- Noise frequency
- Noise amplitude

Outputs:

- Lock-in amplifier output
- Signal-to-Noise ratio

Channel 1 Reference frequency: 100 Hz Noise frequency: 370 Hz Sample frequency: 1000 Hz Input Amplitude: 75mV		Channel 4 Reference frequency: 100 Hz Noise frequency 370 Hz Sample frequency: 1000 Hz Input amplitude: 75 mV	
Input SNR	Output	Input SNR	Output
5dB	74.39 mV	5dB	74.55 mV
3dB	74.41 mV	3dB	74.57 mV
1dB	74.24 mV	1dB	74.40 mV
-1dB	74.25 mV	-1dB	74.40 mV
-3dB	74.35 mV	-3dB	74.50 mV
-5dB	74.37 mV	-5dB	74.52 mV
-7dB	74.40 mV	-7dB	74.55 mV
-9dB	74.30 mV	-9dB	74.46 mV
-11dB	74.45 mV	-11dB	74.60 mV
-13dB	74.43 mV	-13dB	74.59 mV
-15dB	74.45 mV	-15dB	74.58 mV
-17dB	74.59 mV	-17dB	74.72 mV
-19dB	74.47 mV	-19dB	74.64 mV
-21dB	74.44 mV	-21dB	74.59 mV
-23dB	74.50 mV	-23dB	74.67 mV
-25dB	74.57 mV	-25dB	74.74 mV
-27dB	74.51 mV	-27dB	74.64 mV
-29dB	74.44 mV	-29dB	74.58 mV
-30dB	74.55 mV	-30dB	74.70 mV

6.2 Results & Conclusions

6.2.1 PRU-ICSS

The PRU-ICSS is a real-time co-processor capable of offloading smaller tasks from the main ARM processor. It has access to all the memory locations in the AM335x SoC and its own interrupt controller which can interrupt the AM335x' Enhanced Direct Memory Access module or Interrupt controller. It has a fast and deterministic instruction set capable of single cycle addition, multiplication, bit shifts, and fire & forget writes. It can work together with the main ARM processor or it can function completely independently. The PRU-ICSS has a few limitations, namely the small amount (8 kB) of instruction RAM. This can fill up quickly when using large header files such as *math.h*, preventing you from using complex functions.

6.2.2 Numerically Controlled Oscillator

The numerically controlled oscillator algorithm designed for this project is made to be portable. It can be used to either follow a signal, to do FM modulation, or to simply generate a single tone. Since it calculates the tuning word for every single sample based on the period captured by the eCAP module, it is capable of tracking the frequency very accurately. Using an Look-up table size of 1024 and interpolation can generate sinusoids with an SNR of 50dB.

6.2.3 PRU-ICSS Lock-in amplifier

The PRU-ICSS lock-in amplifier algorithm outputs the Q & I signals for 1 to 4 channels. The sample frequency and the number of channels is defined by a command send from the ARM processor. The filtering can be done at any stage after the transmission from the PRU-ICSS. The lock-in amplifier bandwidth is partially decided by the output filters, these can be designed to be FIR or IIR filters of any order since Q & I signals are not filtered by the PRU-ICSS but are evaluated at a later stage where computational power is higher. The test results in paragraph 1 indicate that the PRU-ICSS Lock-in amplifier is capable of extracting a signal from an environment where the SNR is -30dB single channel and multi-channel. Single and multi-channel performance is comparable.

The resolution of the PRU-ICSS Lock-in amplifier is hard to define since the function generator data sheet states that there is always ~±1% amplitude error and ~±1 mVpp error for each kHz when recently calibrated and properly stored. This accuracy is already questionable for our required resolution of 1 mV. The function generators at SRON have not been calibrated since march 2014 so these specifications will be even worse. We make the assumption that we will achieve an accuracy of ~0.15 mV(= 5 / (2^{15})), since the function generator has a 16-bit DAC and the LTC1859 is a 16-bit ADC. We assume that the lock-in amplifier is functioning properly and not letting any noise through as long as the output is close enough to the set voltage with a margin of ~1 mV. We can also look at the plots and see a very stable signal.

6.2.4 Requirements

#	Formal requirement	Achieved
1	The implementation of a digital lock-in amplifier on a microcontroller	A digital lock-in amplifier has been partially implemented on a microcontroller. It lacks the filtering but outputs Q & I signals. Filtering in python on host PC
2	Testing the PRU-ICSS is suitable for real-time tasks	All the different peripherals the PRU-ICSS has access to have been tested and examples are available on this projects GitHub page
3	Have a dynamic reserve of 20 – 30 dB	The tests in paragraph 6.1 indicate that the lock-in amplifier has a dynamic reserve of 20-30 dB in single channel mode
4	A resolution of 1mV	We assume a resolution of 0.15 mV as explained in the paragraph 6.2.3
5	4 – 8 channels	We are capable of sampling 4 channels at ~10kHz reaching a dynamic range comparable to the single channel
6	The low-pass filters in the lock-in amplifier is allowed a maximum integration time of 1 minute.	The low pass filters are not implemented on the PRU-ICSS they are added in later stages of the measurement systems. These filters have an integration time of approximately 15-30 seconds

6.2.5 Header files

Header files have been created in an attempt to abstract all the different peripherals. These are available so that new users of the PRU-ICSS can understand how to talk to the different peripherals, and what needs to be configured before using the peripherals. The GitHub page for each peripheral includes programming aid telling you what registers have to be programmed in what order, the header files for that peripheral and an example program. Only the peripherals used during this project have been documented, and not every peripheral is used. The EDMA module is one of the peripherals not used, and not explained in this project. Mainly because this is not a peripheral within the PRU-ICSS but is accessible by the PRU-ICSS. An example of a peripheral not included in the PRU-ICSS but used for this project is the McSPI module.

6.2.6 Filters

A decision has been made to not do the filtering within the PRU-ICSS. There were two options, FIR and IIR filters, FIR filters required thousands of coefficients and thousands of previous inputs to be calculated and saved, this was impractical to implement on the PRU-ICSS. IIR filters could not reach a proper frequency response with a stable order number. Only leaving the moving average filter, but this filter did not give good results although the frequency response looked pretty good. We decided to do the filtering in later stages so to get the best performance.

6.2.7 Crosstalk, SNRs and more

The crosstalk between channels and the SNR before and after the quadrature multiplication have been evaluated. The crosstalk was tested by measuring a signal on one channel and having a 50 Ω cap on the other channels creating a base line. And then forcing 5Vpp noise into the other channels to see the influence. No cross-talk was measured during these tests. We also looked at the signal SNR before and after the quadrature multiplication. There was no significant change in SNR before and after the quadrature multiplication.

6.3 Recommendations and future visions

Improvements on the design proposed in this report are certainly possible. Being time limited caused the decisions that brought us to the final design. There were two main bottlenecks in this system: RPMsg, and the LTC1859 development board. RPMsg is not designed as a protocol for transferring large data sets. It is designed to send commands to and from the PRU-ICSS. It is recommended to either create a kernel module to take over the task of RPMsg or modify the existing framework to allow for bigger data packets to be send and faster transfer speeds. Removing this bottle neck could increase the sampling rate to 50+ kHz per channel over 4 channels.

The PRU-ICSS is capable of interfacing with the Enhanced Direct Memory Access (EDMA) peripheral of the AM335x. This module can do data transfers, offloading them from the processor. This module can also use interrupts to trigger a data transfer from for example the McSPI module. Using EDMA to transfer bigger chunks of data to the ARM data memory can improve the data throughput requiring the kernel module proposed earlier.

EDMA can also be used to automate the McSPI routine, receiving interrupts from the McSPI module and placing received data in PRU memory. Adding a DDS IC and transferring the reference signal samples from the DDS IC to PRU memory using EDMA can off-load reference signal generation to the DDS IC, making one more PRU core available for other tasks such as more Lock-in amplifier channels. This would require the design of a new ADC board with 2 ADCs and an DDS IC, but would improve the sample frequency significantly. A block-diagram of how this could is given in Figure 28.

Using a FPGA or DSP would remove the constraints placed by the usage of the PRU-ICSS. Allowing you to choose the perfect FPGA/DSP, ADC, DDS IC combination. One challenge would be: how to transfer the data to a computer or to the BeagleBone for further analysis. Implementing an Ethernet driver on an FPGA can become a very challenging and time consuming task.

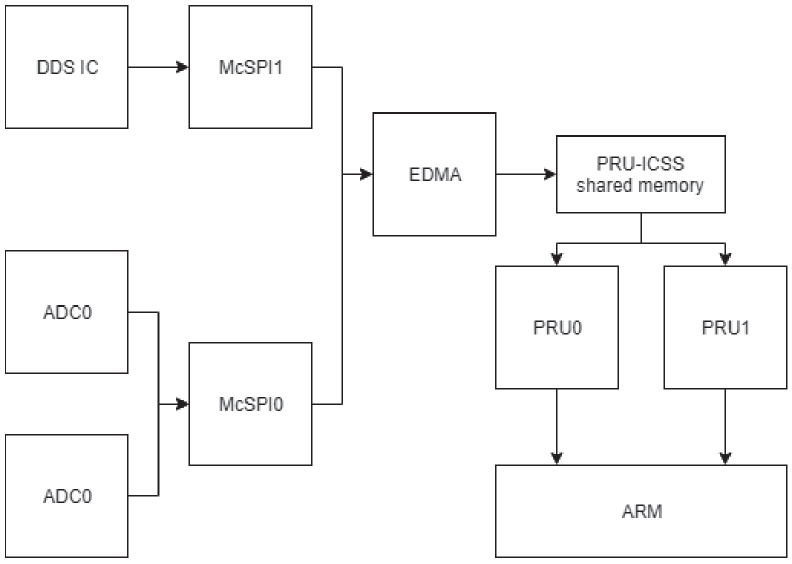


Figure 28: Enhanced PRU-ICSS lock-in amplifier

References

[1] W. Michels, "A Double Tube Vacuum Tube Voltmeter," Rev. Sci. Instrum., 1938.

[2] C. R. Cosens, "A balance-detector for alternating-current bridges," Proceedings of the Physical Society, 1941.

[3] D. Wenn, "Implementing Digital Lock-In Amplifiers Using the dsPIC DSC," Microchip Technology Inc., 2007.

[4] "MODEL SR830 DSP Lock-In Amplifier," Stanford Research Systems, Sunnyvale, 2011.

[5] S. Bhattacharyya, "Implementation of Digital Lock-in Amplifier," Journal of Physics, Gauhati, 2016.

[6] R. G. Skillington, "DSP Based Lock-in Amplifier," 2013.

[7] D. Molloy, Exploring BeagleBone, second edition, Indianapolis: Jogn Wiley & Sons, Inc., 2019.

[8] G. Macias-Bobadilla, J. Rodríguez-Reséndiz, G. Mota-Valtierra, G. Soto-Zarazúa, M. Méndez-Loyola and M. Garduño-Aparicio, "M. Dual-Phase Lock-In Amplifier Based on FPGA for Low-Frequencies Experiments.," Queretaro, 2016.

[9] Linear Technology Corporation, "LTC1857/LTC1858/LTC1859," Analog Devices.

[10] J. S. a. L. McHugh, "Signle-Chip Direct Digital Synthesis vs. the Analog PLL," Analog Devices, 1996.

[11] BeagleBoard, "BeagleBone Black," BeagleBoard, 15 05 2019. [Online]. Available: https://beagleboard.org/black. [Accessed 15 05 2019].

[12] S. Tatham, "Download PuTTY," PuTTY, [Online]. Available: https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html. [Accessed 15 05 2019].

[13] "WinSCP 5.15 download," WinSCP.net, [Online]. Available: https://winscp.net/eng/download.php. [Accessed 15 05 2019].

[14] icetee, "Atom.io remote-ftp," [Online]. Available: https://atom.io/packages/remote-ftp. [Accessed 15 05 2019].

[15] T. Instruments, *AM335x Techincal Reference Manual,* 2017.

[16] T. Instruments, "PRU Read Latencies," Texas Instruments, 2018.

[17] A. Devices, "Fundamentals of Direct Digital Synthesis".

[18] R. Yates, Fixed-Point Arithmetic: An Introduction, 2007.

Appendix A – Acronyms

HEB	**-**	Hot Electron Bolometer
PRU-ICSS	-	Programmable Real-time Unit – Industrial Communications SubSystem
PRU	-	Programmable Real-time Unit
McSPI	**-**	Multi-channel Serial Peripheral Interface
GPIO	-	General Purpose Input/Ouput
UART	**-**	Universal Asynchronous Receiver-ransmitter
ADC	-	Analog-to-Digital Converter
DAC	**-**	Digital-to-Analog Converter
DSP	-	Digital Signal Processor
FPGA	-	Field Programmable Gate Array
SNR	**-**	Signal-to-Noise Ratio
NCO	**-**	Numerically Controlled Oscillator
LO	-	Local Oscillator
MCU	**-**	MicroController Unit
SoC	**-**	System-on-Chip
DC	-	Direct Current
AC	-	Alternating Current
PLL	-	Phase-Locked Loop
DDS	**-**	Direct Digital Synthesis
RAM	**-**	Random Access Memory
IIR	**-**	Infinite Impulse Response
FIR	-	Finite Impulse Response
EDMA	-	Enhanced Direct Memory Access

Appendix B[1] – P8 header pin modes [6]

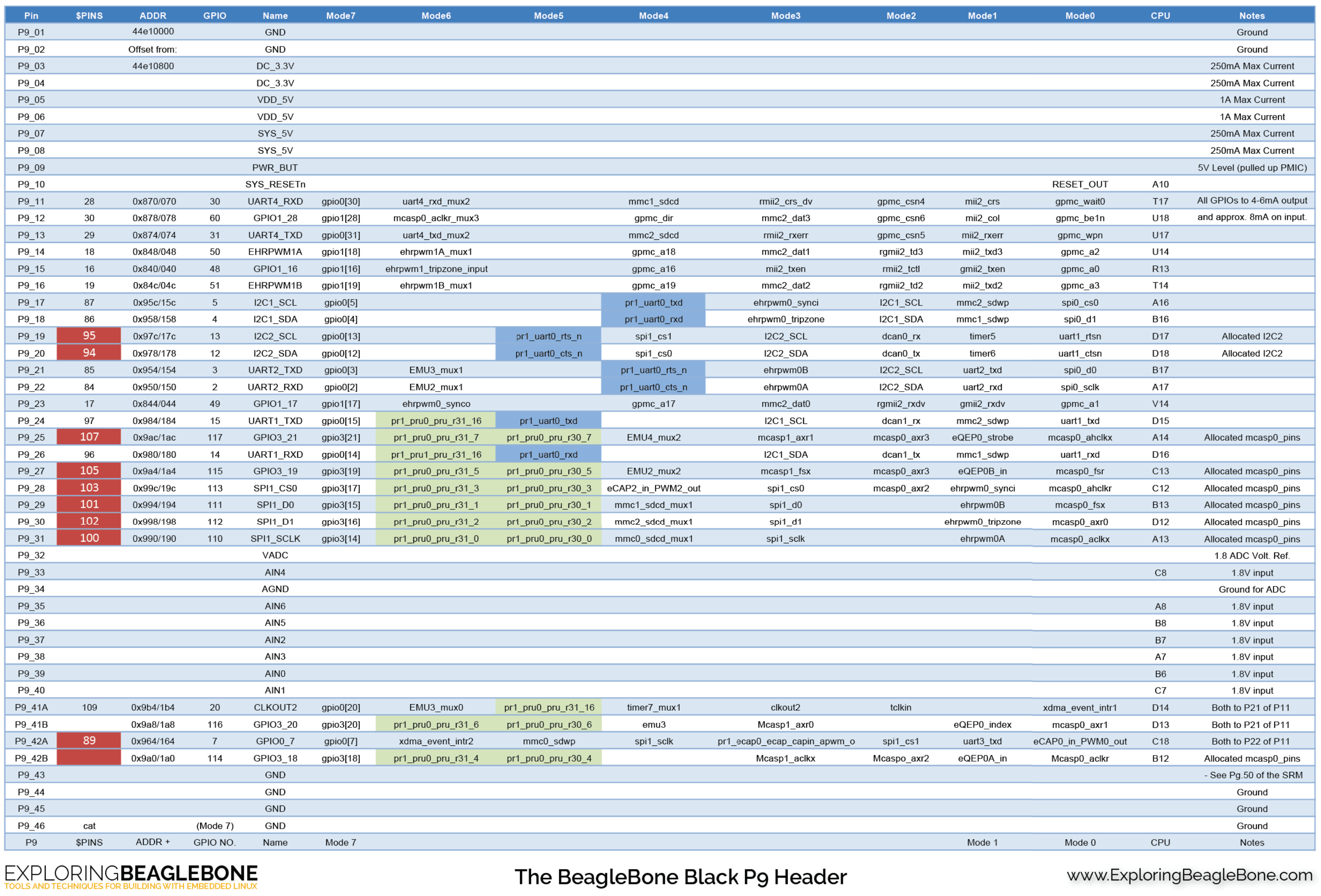
Pin	$PINS	ADDR	GPIO	Name	Mode7	Mode6	Mode5	Mode4	Mode3	Mode2	Mode1	Mode0	CPU	Notes
P8_01		Offset from:		DGND										Ground
P8_02		44e10800		DGND										Ground
P8_03	6	0x818/018	38	GPIO1_6	gpio1[6]						mmc1_dat6	gpmc_ad6	R9	Allocated emmc2
P8_04	7	0x81c/01c	39	GPIO1_7	gpio1[7]						mmc1_dat7	gpmc_ad7	T9	Allocated emmc2
P8_05	2	0x808/008	34	GPIO1_2	gpio1[2]						mmc1_dat2	gpmc_ad2	R8	Allocated emmc2
P8_06	3	0x80c/00c	35	GPIO1_3	gpio1[3]						mmc1_dat3	gpmc_ad3	T8	Allocated emmc2
P8_07	36	0x890/090	66	TIMER4	gpio2[2]					timer4		gpmc_advn_ale	R7	
P8_08	37	0x894/094	67	TIMER7	gpio2[3]					timer7		gpmc_oen_ren	T7	
P8_09	39	0x89c/09c	69	TIMER5	gpio2[5]					timer5		gpmc_be0n_cle	T6	
P8_10	38	0x898/098	68	TIMER6	gpio2[4]					timer6		gpmc_wen	U6	
P8_11	13	0x834/034	45	GPIO1_13	gpio1[13]	pr1_pru0_pru_r30_15		eQEP2B_in	mmc2_dat1	mmc1_dat5	lcd_data18	gpmc_ad13	R12	
P8_12	12	0x830/030	44	GPIO1_12	gpio1[12]	pr1_pru0_pru_r30_14		EQEP2A_IN	MMC2_DAT0	MMC1_DAT4	LCD_DATA19	GPMC_AD12	T12	
P8_13	9	0x824/024	23	EHRPWM2B	gpio0[23]			ehrpwm2B	mmc2_dat5	mmc1_dat1	lcd_data22	gpmc_ad9	T10	
P8_14	10	0x828/028	26	GPIO0_26	gpio0[26]			ehrpwm2_tripzone_in	mmc2_dat6	mmc1_dat2	lcd_data21	gpmc_ad10	T11	
P8_15	15	0x83c/03c	47	GPIO1_15	gpio1[15]	pr1_pru0_pru_r31_15		eQEP2_strobe	mmc2_dat3	mmc1_dat7	lcd_data16	gpmc_ad15	U13	
P8_16	14	0x838/038	46	GPIO1_14	gpio1[14]	pr1_pru0_pru_r31_14		eQEP2_index	mmc2_dat2	mmc1_dat6	lcd_data17	gpmc_ad14	V13	
P8_17	11	0x82c/02c	27	GPIO0_27	gpio0[27]			ehrpwm0_synco	mmc2_dat7	mmc1_dat3	lcd_data20	gpmc_ad11	U12	
P8_18	35	0x88c/08c	65	GPIO2_1	gpio2[1]	mcasp0_fsr			mmc2_clk	gpmc_wait1	lcd_memory_clk	gpmc_clk_mux0	V12	
P8_19	8	0x820/020	22	EHRPWM2A	gpio0[22]			ehrpwm2A	mmc2_dat4	mmc1_dat0	lcd_data23	gpmc_ad8	U10	
P8_20	33	0x884/084	63	GPIO1_31	gpio1[31]	pr1_pru1_pru_r31_13	pr1_pru1_pru_r30_13			mmc1_cmd	gpmc_be1n	gpmc_csn2	V9	Allocated emmc2
P8_21	32	0x880/080	62	GPIO1_30	gpio1[30]	pr1_pru1_pru_r31_12	pr1_pru1_pru_r30_12			mmc1_clk	gpmc_clk	gpmc_csn1	U9	Allocated emmc2
P8_22	5	0x814/014	37	GPIO1_5	gpio1[5]						mmc1_dat5	gpmc_ad5	V8	Allocated emmc2
P8_23	4	0x810/010	36	GPIO1_4	gpio1[4]						mmc1_dat4	gpmc_ad4	U8	Allocated emmc2
P8_24	1	0x804/004	33	GPIO1_1	gpio1[1]						mmc1_dat1	gpmc_ad1	V7	Allocated emmc2
P8_25	0	0x800/000	32	GPIO1_0	gpio1[0]						mmc1_dat0	gpmc_ad0	U7	Allocated emmc2
P8_26	31	0x87c/07c	61	GPIO1_29	gpio1[29]							gpmc_csn0	V6	
P8_27	56	0x8e0/0e0	86	GPIO2_22	gpio2[22]	pr1_pru1_pru_r31_8	pr1_pru1_pru_r30_8				gpmc_a8	lcd_vsync	U5	Allocated HDMI
P8_28	58	0x8e8/0e8	88	GPIO2_24	gpio2[24]	pr1_pru1_pru_r31_10	pr1_pru1_pru_r30_10				gpmc_a10	lcd_pclk	V5	Allocated HDMI
P8_29	57	0x8e4/0e4	87	GPIO2_23	gpio2[23]	pr1_pru1_pru_r31_9	pr1_pru1_pru_r30_9				gpmc_a9	lcd_hsync	R5	Allocated HDMI
P8_30	59	0x8ec/0ec	89	GPIO2_25	gpio2[25]	pr1_pru1_pru_r31_11	pr1_pru1_pru_r30_11				gpmc_a11	lcd_ac_bias_en	R6	Allocated HDMI
P8_31	54	0x8d8/0d8	10	UART5_CTSN	gpio0[10]	uart5_ctsn		uart5_rxd	mcasp0_axr1	eQEP1_index	gpmc_a18	lcd_data14	V4	Allocated HDMI
P8_32	55	0x8dc/0dc	11	UART5_RTSN	gpio0[11]	uart5_rtsn		mcasp0_axr3	mcasp0_ahclkx	eQEP1_strobe	gpmc_a19	lcd_data15	T5	Allocated HDMI
P8_33	53	0x8d4/0d4	9	UART4_RTSN	gpio0[9]	uart4_rtsn		mcasp0_axr3	mcasp0_fsr	eQEP1B_in	gpmc_a17	lcd_data13	V3	Allocated HDMI
P8_34	51	0x8cc/0cc	81	UART3_RTSN	gpio2[17]	uart3_rtsn		mcasp0_axr2	mcasp0_ahclkr	ehrpwm1B	gpmc_a15	lcd_data11	U4	Allocated HDMI
P8_35	52	0x8d0/0d0	8	UART4_CTSN	gpio0[8]	uart4_ctsn		mcasp0_axr2	mcasp0_aclkr	eQEP1A_in	gpmc_a16	lcd_data12	V2	Allocated HDMI
P8_36	50	0x8c8/0c8	80	UART3_CTSN	gpio2[16]	uart3_ctsn			mcasp0_axr0	ehrpwm1A	gpmc_a14	lcd_data10	U3	Allocated HDMI
P8_37	48	0x8c0/0c0	78	UART5_TXD	gpio2[14]	uart2_ctsn		uart5_txd	mcasp0_aclkx	ehrpwm1_tripzone_in	gpmc_a12	lcd_data8	U1	Allocated HDMI
P8_38	49	0x8c4/0c4	79	UART5_RXD	gpio2[15]	uart2_rtsn		uart5_rxd	mcasp0_fsx	ehrpwm0_synco	gpmc_a13	lcd_data9	U2	Allocated HDMI
P8_39	46	0x8b8/0b8	76	GPIO2_12	gpio2[12]	pr1_pru1_pru_r31_6	pr1_pru1_pru_r30_6		eQEP2_index		gpmc_a6	lcd_data6	T3	Allocated HDMI
P8_40	47	0x8bc/0bc	77	GPIO2_13	gpio2[13]	pr1_pru1_pru_r31_7	pr1_pru1_pru_r30_7	pr1_edio_data_out7	eQEP2_strobe		gpmc_a7	lcd_data7	T4	Allocated HDMI
P8_41	44	0x8b0/0b0	74	GPIO2_10	gpio2[10]	pr1_pru1_pru_r31_4	pr1_pru1_pru_r30_4		eQEP2A_in		gpmc_a4	lcd_data4	T1	Allocated HDMI
P8_42	45	0x8b4/0b4	75	GPIO2_11	gpio2[11]	pr1_pru1_pru_r31_5	pr1_pru1_pru_r30_5		eQEP2B_in		gpmc_a5	lcd_data5	T2	Allocated HDMI
P8_43	42	0x8a8/0a8	72	GPIO2_8	gpio2[8]	pr1_pru1_pru_r31_2	pr1_pru1_pru_r30_2		ehrpwm2_tripzone_in		gpmc_a2	lcd_data2	R3	Allocated HDMI
P8_44	43	0x8ac/0ac	73	GPIO2_9	gpio2[9]	pr1_pru1_pru_r31_3	pr1_pru1_pru_r30_3		ehrpwm0_synco		gpmc_a3	lcd_data3	R4	Allocated HDMI
P8_45	40	0x8a0/0a0	70	GPIO2_6	gpio2[6]	pr1_pru1_pru_r31_0	pr1_pru1_pru_r30_0		ehrpwm2A		gpmc_a0	lcd_data0	R1	Allocated HDMI
P8_46	41	0x8a4/0a4	71	GPIO2_7	gpio2[7]	pr1_pru1_pru_r31_1	pr1_pru1_pru_r30_1		ehrpwm2B		gpmc_a1	lcd_data1	R2	Allocated HDMI
P9 Header	cat $PINS	ADDR +	GPIO NO.	Name	Mode 7	Mode 6	Mode 5	Mode 4	Mode 3	Mode 2	Mode 1	Mode 0	CPU	

EXPLORING**BEAGLEBONE**
TOOLS AND TECHNIQUES FOR BUILDING WITH EMBEDDED LINUX

The BeagleBone Black P8 Header

www.ExploringBeagleBone.com

Appendix B^2 – P9 header pin modes [6]



Pin	$PINS	ADDR	GPIO	Name	Mode7	Mode6	Mode5	Mode4	Mode3	Mode2	Mode1	Mode0	CPU	Notes
P9_01		44e10000		GND										Ground
P9_02		Offset from:		GND										Ground
P9_03		44e10800		DC_3.3V										250mA Max Current
P9_04				DC_3.3V										250mA Max Current
P9_05				VDD_5V										1A Max Current
P9_06				VDD_5V										1A Max Current
P9_07				SYS_5V										250mA Max Current
P9_08				SYS_5V										250mA Max Current
P9_09				PWR_BUT										5V Level (pulled up PMIC)
P9_10				SYS_RESETn								RESET_OUT	A10	
P9_11	28	0x870/070	30	UART4_RXD	gpio0[30]	uart4_rxd_mux2		mmc1_sdcd	rmii2_crs_dv	gpmc_csn4	mii2_crs	gpmc_wait0	T17	All GPIOs to 4-6mA output
P9_12	30	0x878/078	60	GPIO1_28	gpio1[28]	mcasp0_aclkr_mux3		gpmc_dir	mmc2_dat3	gpmc_csn6	mii2_col	gpmc_be1n	U18	and approx. 8mA on input.
P9_13	29	0x874/074	31	UART4_TXD	gpio0[31]	uart4_txd_mux2		mmc2_sdcd	rmii2_rxerr	gpmc_csn5	mii2_rxerr	gpmc_wpn	U17	
P9_14	18	0x848/048	50	EHRPWM1A	gpio1[18]	ehrpwm1A_mux1		gpmc_a18	mmc2_dat1	rgmii2_td3	mii2_txd3	gpmc_a2	U14	
P9_15	16	0x840/040	48	GPIO1_16	gpio1[16]	ehrpwm1_tripzone_input		gpmc_a16	mii2_txen	rmii2_tctl	gmii2_txen	gpmc_a0	R13	
P9_16	19	0x84c/04c	51	EHRPWM1B	gpio1[19]	ehrpwm1B_mux1		gpmc_a19	mmc2_dat2	rgmii2_td2	mii2_txd2	gpmc_a3	T14	
P9_17	87	0x95c/15c	5	I2C1_SCL	gpio0[5]			pr1_uart0_txd	ehrpwm0_synci	I2C1_SCL	mmc2_sdwp	spi0_cs0	A16	
P9_18	86	0x958/158	4	I2C1_SDA	gpio0[4]			pr1_uart0_rxd	ehrpwm0_tripzone	I2C1_SDA	mmc1_sdwp	spi0_d1	B16	
P9_19	95	0x97c/17c	13	I2C2_SCL	gpio0[13]		pr1_uart0_rts_n	spi1_cs1	I2C2_SCL	dcan0_rx	timer5	uart1_rtsn	D17	Allocated I2C2
P9_20	94	0x978/178	12	I2C2_SDA	gpio0[12]		pr1_uart0_cts_n	spi1_cs0	I2C2_SDA	dcan0_tx	timer6	uart1_ctsn	D18	Allocated I2C2
P9_21	85	0x954/154	3	UART2_TXD	gpio0[3]	EMU3_mux1		pr1_uart0_rts_n	ehrpwm0B	I2C2_SCL	uart2_txd	spi0_d0	B17	
P9_22	84	0x950/150	2	UART2_RXD	gpio0[2]	EMU2_mux1		pr1_uart0_cts_n	ehrpwm0A	I2C2_SDA	uart2_rxd	spi0_sclk	A17	
P9_23	17	0x844/044	49	GPIO1_17	gpio1[17]	ehrpwm0_synco		gpmc_a17	mmc2_dat0	rgmii2_rxdv	gmii2_rxdv	gpmc_a1	V14	
P9_24	97	0x984/184	15	UART1_TXD	gpio0[15]	pr1_pru0_pru_r31_16	pr1_uart0_txd		I2C1_SCL	dcan1_rx	mmc2_sdwp	uart1_txd	D15	
P9_25	107	0x9ac/1ac	117	GPIO3_21	gpio3[21]	pr1_pru0_pru_r31_7	pr1_pru0_pru_r30_7	EMU4_mux2	mcasp1_axr1	mcasp0_axr3	eQEP0_strobe	mcasp0_ahclkx	A14	Allocated mcasp0_pins
P9_26	96	0x980/180	14	UART1_RXD	gpio0[14]	pr1_pru1_pru_r31_16	pr1_uart0_rxd		I2C1_SDA	dcan1_tx	mmc1_sdwp	uart1_rxd	D16	
P9_27	105	0x9a4/1a4	115	GPIO3_19	gpio3[19]	pr1_pru0_pru_r31_5	pr1_pru0_pru_r30_5	EMU2_mux2	mcasp1_fsx	mcasp0_axr3	eQEP0B_in	mcasp0_fsr	C13	Allocated mcasp0_pins
P9_28	103	0x99c/19c	113	SPI1_CS0	gpio3[17]	pr1_pru0_pru_r31_3	pr1_pru0_pru_r30_3	eCAP2_in_PWM2_out	spi1_cs0	mcasp0_axr2	ehrpwm0_synci	mcasp0_ahclkr	C12	Allocated mcasp0_pins
P9_29	101	0x994/194	111	SPI1_D0	gpio3[15]	pr1_pru0_pru_r31_1	pr1_pru0_pru_r30_1	mmc1_sdcd_mux1	spi1_d0		ehrpwm0B	mcasp0_fsx	B13	Allocated mcasp0_pins
P9_30	102	0x998/198	112	SPI1_D1	gpio3[16]	pr1_pru0_pru_r31_2	pr1_pru0_pru_r30_2	mmc2_sdcd_mux1	spi1_d1		ehrpwm0_tripzone	mcasp0_axr0	D12	Allocated mcasp0_pins
P9_31	100	0x990/190	110	SPI1_SCLK	gpio3[14]	pr1_pru0_pru_r31_0	pr1_pru0_pru_r30_0	mmc0_sdcd_mux1	spi1_sclk		ehrpwm0A	mcasp0_aclkx	A13	Allocated mcasp0_pins
P9_32				VADC										1.8 ADC Volt. Ref.
P9_33				AIN4									C8	1.8V input
P9_34				AGND										Ground for ADC
P9_35				AIN6									A8	1.8V input
P9_36				AIN5									B8	1.8V input
P9_37				AIN2									B7	1.8V input
P9_38				AIN3									A7	1.8V input
P9_39				AIN0									B6	1.8V input
P9_40				AIN1									C7	1.8V input
P9_41A	109	0x9b4/1b4	20	CLKOUT2	gpio0[20]	EMU3_mux0	pr1_pru0_pru_r31_16	timer7_mux1	clkout2	tclkin		xdma_event_intr1	D14	Both to P21 of P11
P9_41B		0x9a8/1a8	116	GPIO3_20	gpio3[20]	pr1_pru0_pru_r31_6	pr1_pru0_pru_r30_6	emu3	Mcasp1_axr0		eQEP0_index	mcasp0_axr1	D13	Both to P21 of P11
P9_42A	89	0x964/164	7	GPIO0_7	gpio0[7]	xdma_event_intr2	mmc0_sdwp	spi1_sclk	pr1_ecap0_ecap_capin_apwm_o	spi1_cs1	uart3_txd	eCAP0_in_PWM0_out	C18	Both to P22 of P11
P9_42B		0x9a0/1a0	114	GPIO3_18	gpio3[18]	pr1_pru0_pru_r31_4	pr1_pru0_pru_r30_4		Mcasp1_aclkx	Mcaspo_axr2	eQEP0A_in	Mcasp0_aclkr	B12	Allocated mcasp0_pins
P9_43				GND										- See Pg.50 of the SRM
P9_44				GND										Ground
P9_45				GND										Ground
P9_46	cat		(Mode 7)	GND										Ground
P9	$PINS	ADDR +	GPIO NO.	Name	Mode 7						Mode 1	Mode 0	CPU	Notes

EXPLORING**BEAGLEBONE**
TOOLS AND TECHNIQUES FOR BUILDING WITH EMBEDDED LINUX

The BeagleBone Black P9 Header

www.ExploringBeagleBone.com

Appendix C[1] – PRU pins

P9				P8			
GND	1	2	GND	GND	1	2	GND
3V3	3	4	3V3		3	4	
5V RAW	5	6	5V RAW		5	6	
5V	7	8	5V		7	8	
	9	10			9	10	
	11	12		PRU0 O 15	11	12	PRU0 O 14
	13	14			13	14	
	15	16		PRU0 I 15	15	16	PRU0 I 14
UART0 TX	17	18	UART0 RX		17	18	
UART0 RTS	19	20	UART0 CTS		19	20	PRU1 IO 13
UART0 RTS	21	22	UART0 CTS	PRU1 IO 12	21	22	
	23	24	PRU0 I 16		23	24	
PRU0 IO 7	25	26	PRU1 I 16		25	26	
PRU0 IO 5	27	28	PRU0 IO 3	PRU1 IO 8	27	28	PRU1 IO 10
PRU0 IO 1	29	30	PRU0 IO 2	PRU1 IO 9	29	30	PRU1 IO 11
PRU0 IO 0	31	32			31	32	
	33	34			33	34	
	35	36			35	36	
	37	38			37	38	
	39	40		PRU1 IO 6	39	40	PRU1 IO 7
PRU0 IO 6	41	42	PRU0 IO 4	PRU1 IO 4	41	42	PRU1 IO 5
GND	43	44	GND	PRU1 IO 2	43	44	PRU1 IO 3
GND	45	46	GND	PRU1 IO 0	45	46	PRU1 IO 1

Appendix C^2 – Lock-in pins

P9				P8			
GND	1	2	GND	GND	1	2	GND
3V3	3	4	3V3		3	4	
5V RAW	5	6	5V RAW		5	6	
5V	7	8	5V		7	8	
	9	10			9	10	
	11	12			11	12	
	13	14			13	14	
	15	16			15	16	
	17	18	SPI D0		17	18	
	19	20			19	20	
SPI D0	21	22	SPI CLK		21	22	
	23	24	UART0 TX		23	24	
CS	25	26	UART0 RX		25	26	
CONVST	27	28	PRU0 DEBUG		27	28	PRU1 DEBUG
RD	29	30			29	30	
BUSY	31	32			31	32	
	33	34			33	34	
	35	36			35	36	
	37	38			37	38	
	39	40			39	40	
	41	42	SYNC in		41	42	
GND	43	44	GND		43	44	
GND	45	46	GND		45	46	

Appendix D – Memory map

PRU-ICSS Memory offsets PRU Base address – 0x4A30_0000	
PRU0 Control register	0x0002_2000
PRU0 Debug register	0x0002_2400
PRU1 Control register	0x0002_4000
PRU1 Debug register	0x0002_4400
PRU0 Data RAM	0x0000_0000
PRU1 Data RAM	0x0000_2000
PRU-ICSS Shared RAM	0x0001_0000

PRU-ICSS INTC Memory offsets INTC Base offset – 0x0002_0000	
System event set register	0x0000_0020
System event clear register	0x0000_0024
System event enable register	0x0000_0028
System event disable register	0x0000_002C
Host interrupt enable register	0x0000_0034
Host interrupt disable resister	0x0000_0038
System event status [0:31]	0x0000_0200
System event status [32:63]	0x0000_0204
Channel map register [X]	0x0000_04[00:3C]
Host map register [X]	0x0000_080[0:8]

PRU-ICSS McSPI Memory offsets McSPI Base address – 0x4803_0000	
System config	0x0000_0110
System status	0x0000_0114
Module control	0x0000_0128
Channel[0:3] config	0x012c, 0x0140, 0x0154, 0x0168
Channel[0:3] control	0x0134, 0x0148, 0x015C, 0x0170
Interrupt status	0x0118
Interrupt enable	0x011C
RX[0:3]	0x013C, 0x0150, 0x0164, 0x0178
TX[0:3]	0x0138, 0x014C, 0x0160, 0x0178

PRU-ICSS UART Memory offsets UART Base address – 0x44E0_9000	
RX/TX register	0x0000_0000
FIFO control register	0x0000_0008
Line control register	0x0000_000C
Modem control register	0x0000_0010
Divisor LSB	0x0000_0020
Divisor MSB	0x0000_0024
Mode definition register	0x0000_0034

Power Reset Clock Memory offsets PRCM base address – 0x44E0_0000	
SPI clock management	0x0000_004C
Clock domain	0x0000_0400
ADC clock management	0x0000_04BC

PRU-ICSS IEP Memory offsets IEP Base offset – 0x0002_E000	
IEP global config	0x0000_0000
IEP global status	0x0000_0004
IEP count register	0x0000_000C
IEP compare [X]	0x0000_00[48:64]
IEP compare config	0x0000_0040

PRU-ICSS eCAP Memory offsets eCAP Base offset – 0x0003_0000	
eCAP counter register	0x0000_0000
eCAP capture register [X]	0x0000_00[08:14]
eCAP control register [X]	0x0000_00[28:2A]
eCAP interrupt enable	0x0000_002C
eCAP interrupt clear	0x0000_0030
eCAP interrupt status	0x0000_002E

PRU-ICSS ADC Memory offsets ADC Base address – 0x44E0_D000	
ADC control register	0x0000_0040
ADC step config[0:16]	0x0000_00[64:DC]
FIFO word count	0x0000_00E4
FIFO data	0x0000_0100
Step enable	0x0000_0054

Appendix E[1] – Makefile

```
ifdef PRU
    PRUN := $(PRU)
    PRU_ADDR := $(PRUN)+1
else
    PRUN := 0
    PRU_ADDR := 1
endif

# System paths for compiler and support package
PRU_CGT:= /usr/share/ti/cgt-pru
PRU_SUPPORT:= /usr/lib/ti/pru-software-support-package
PRU_BBB := /home/debian/BBBLockin

# Define current directory
MKFILE_PATH := $(abspath $(lastword $(MAKEFILE_LIST)))
CURRENT_DIR := $(notdir $(patsubst %/,%,$(dir $(MKFILE_PATH))))
PROJ_NAME=$(CURRENT_DIR)
FILE_NAME= Main
# Path to temporary folder
GEN_DIR:=gen

# Path to linker command file
LINKER_COMMAND_FILE=./AM335x_PRU.cmd

# Paths to libraries
LIBS=--library=$(PRU_SUPPORT)/lib/rpmsg_lib.lib
INCLUDE=--include_path=$(PRU_BBB)/include --include_path=$(PRU_CGT)/include --
include_path=$(PRU_SUPPORT)/include/ --
include_path=$(PRU_SUPPORT)/include/am335x/

# Stack & Heap size
STACK_SIZE=0x100
HEAP_SIZE=0x100

# Common compiler and linker flags (Defined in 'PRU Optimizing C/C++ Compiler
User's Guide)
CFLAGS=-v3 -O2 --c99 --float_operations_allowed=none --display_error_number --
endian=little --hardware_mac=on --printf_support=minimal --
obj_directory=$(GEN_DIR) --pp_directory=$(GEN_DIR) -ppd -ppa

# Linker flags (Defined in 'PRU Optimizing C/C++ Compiler User's Guide)
LFLAGS=--reread_libs --warn_sections --stack_size=$(STACK_SIZE) --
heap_size=$(HEAP_SIZE)

# Target file
TARGET=$(GEN_DIR)/$(PROJ_NAME).out

# Map file
MAP=$(GEN_DIR)/$(PROJ_NAME).map

# Source files
```

```
SOURCES=$(wildcard *.c)

#Using .object instead of .obj in order to not conflict with the CCS build process
OBJECTS=$(patsubst %,$(GEN_DIR)/%,$(SOURCES:.c=.object))

# Lookup PRU by address
ifeq ($(PRUN),0)
PRU_ADDR=remoteproc1
endif
ifeq ($(PRUN),1)
PRU_ADDR=remoteproc2
endif

# PRU sysfs interface directory
PRU_DIR=$(wildcard /sys/class/remoteproc/$(PRU_ADDR))
all: stop clean install start

stop:
    @echo "-    Stopping PRU $(PRUN)"
    @echo 'stop' | sudo tee -a $(PRU_DIR)/state > /dev/null || echo '-   Cannot stop $(PRUN)'

start:
    @echo "-    Starting PRU $(PRUN)"
    @echo 'start' | sudo tee -a $(PRU_DIR)/state > /dev/null || echo '-  Cannot start $(PRUN)'

install: $(TARGET)
    @echo '-    Copying firmware file $(GEN_DIR)/$(PROJ_NAME).out to /lib/firmware/am335x-pru$(PRUN)-fw'
    @sudo cp $(GEN_DIR)/$(PROJ_NAME).out /lib/firmware/am335x-pru$(PRUN)-fw

$(TARGET): $(OBJECTS) $(LINKER_COMMAND_FILE)
    @echo '-    Invoking: PRU Linker: $^'
    @$(PRU_CGT)/bin/clpru $(CFLAGS) -z -i$(PRU_BBB)/include -i$(PRU_CGT)/lib -i$(PRU_CGT)/include $(LFLAGS) -o $(TARGET) $(OBJECTS) -m$(MAP) $(LINKER_COMMAND_FILE) --library=libc.a $(LIBS)


$(GEN_DIR)/%.object: %.c
    @mkdir -p $(GEN_DIR)
    @echo '-    Invoking: PRU Compiler: $<'
    @$(PRU_CGT)/bin/clpru $(INCLUDE) $(CFLAGS) -fe $@ $< -D=PRUN=PRU$(PRUN)

.PHONY: all clean

clean:
    @echo '-    Clean'
    @rm -rf $(GEN_DIR)

# Includes the dependencies that the compiler creates (-ppd and -ppa flags)
-include $(OBJECTS:%.object=%.pp)
```

Appendix E^2 – Makefile explanation

Line #	**Explanation**
1-7	If the PRU number is defined select that PRU, otherwise select PRU 0 (default)
10-12	Compiler and support package system paths
15-18	Define the name of the directory the makefile is in
20	Define the name of the temporary folder
23	Define the path to the linker command file, tells the linker where in memory to put things
26-27	Define the paths to the libraries and includes
30-31	Define Stack and heap size
34-37	Compiler and Linker flags
40-43	Define the output file and the map file
46	Wildcard all the .c source files in the directory
52-57	Define the PRU address based on the PRU number defined in line 1-7
60	Wildcard the entire sysfs interface
61	First rule: Stop PRU, clean temporary folder, compile program, upload program, start PRU
63-65	This rule stops the PRU by writing 'stop' into the state file using the sysfs interface
67-69	This rule start the PRU by writing 'start' into the state file using the sysfs interface
71-73	Code is installed by copying the .out file into /lib/firmware under the name 'am335x-pruX-fw' where X is the number of the PRU (0/1)
75-83	Rules for compiling and linking the C code
87-89	Rule for cleaning the temporary folder

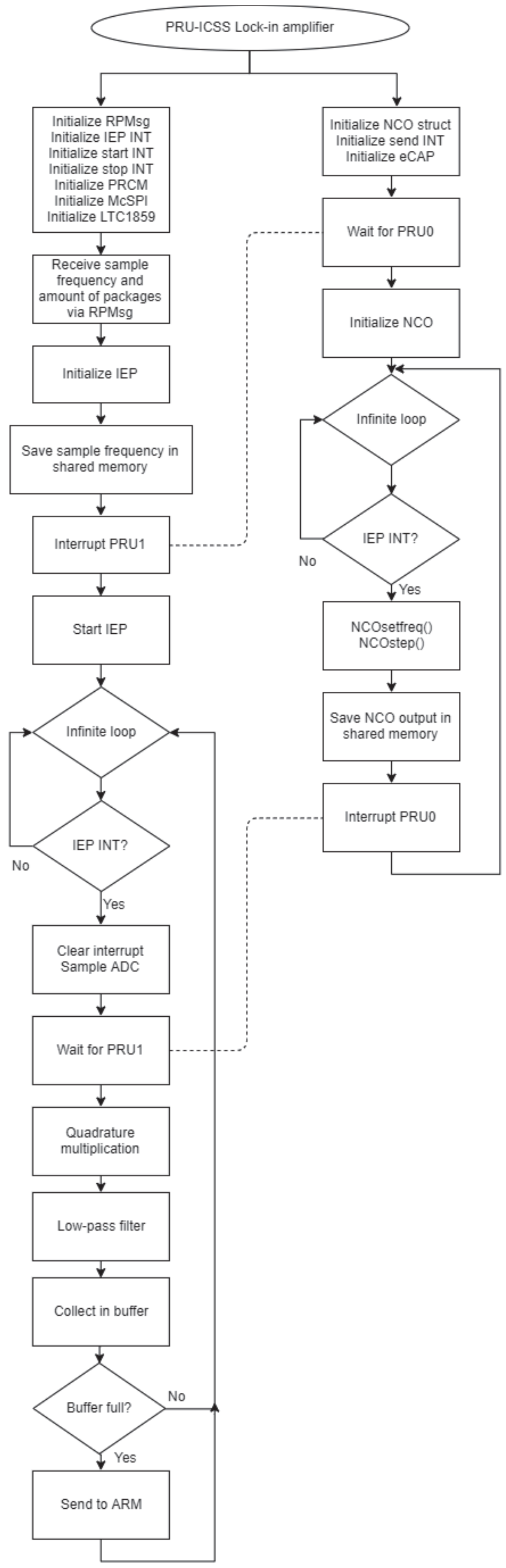
Appendix F[1] – Resource Table

```c
#ifndef _RSC_TABLE_PRU_H_
#define _RSC_TABLE_PRU_H_

#include <stddef.h> //needed for offset_of
#include <rsc_types.h>
#include "pru_virtio_ids.h"

/*
 * Sizes of the virtqueues (expressed in number of buffers supported,
 * and must be power of 2)
 */
#define PRU_RPMSG_VQ0_SIZE  16
#define PRU_RPMSG_VQ1_SIZE  16

/*
 * The feature bitmap for virtio rpmsg
 */
#define VIRTIO_RPMSG_F_NS   0       //name service notifications

/* This firmware supports name service notifications as one of its features */

#define RPMSG_PRU_C0_FEATURES   (1 << VIRTIO_RPMSG_F_NS)

/* Definition for unused interrupts */
#define HOST_UNUSED     255


#define TO_ARM_HOST         16
#define FROM_ARM_HOST           17
#define PRU0_TO_ARM_CHANNEL  2
#define PRU0_FROM_ARM_CHANNEL 0
#define PRU1_TO_ARM_CHANNEL  HOST_UNUSED
#define PRU1_FROM_ARM_CHANNEL HOST_UNUSED
#define HOST_INT            0x40000000
#define CHAN_NAME           "rpmsg-pru"
#define CHAN_DESC           "Channel 30"
#define CHAN_PORT           30
#define TO_ARM_CHANNEL PRU0_TO_ARM_CHANNEL
#define FROM_ARM_CHANNEL PRU0_FROM_ARM_CHANNEL

/* Mapping sysevts to a channel. Each pair contains a sysevt, channel. */
struct ch_map pru_intc_map[] = { {TO_ARM_HOST, TO_ARM_CHANNEL},
                 {FROM_ARM_HOST, FROM_ARM_CHANNEL},
};

struct my_resource_table {
    struct resource_table base;

    uint32_t offset[2]; /* Should match 'num' in actual definition */

    /* rpmsg vdev entry */
    struct fw_rsc_vdev rpmsg_vdev;
    struct fw_rsc_vdev_vring rpmsg_vring0;
    struct fw_rsc_vdev_vring rpmsg_vring1;

    /* intc definition */
    struct fw_rsc_custom pru_ints;
};

#pragma DATA_SECTION(resourceTable, ".resource_table")
#pragma RETAIN(resourceTable)
struct my_resource_table resourceTable = {
    1,  /* Resource table version: only version 1 is supported by the current driver */
    2,  /* number of entries in the table */
    0, 0,   /* reserved, must be zero */
    /* offsets to entries */
    {
        offsetof(struct my_resource_table, rpmsg_vdev),
        offsetof(struct my_resource_table, pru_ints),
    },

    /* rpmsg vdev entry */
    {
        (unsigned)TYPE_VDEV,                    //type
        (unsigned)VIRTIO_ID_RPMSG,              //id
        (unsigned)0,                            //notifyid
        (unsigned)RPMSG_PRU_C0_FEATURES,    //dfeatures
        (unsigned)0,                            //gfeatures
        (unsigned)0,                            //config_len
        (unsigned char)0,                             //status
        (unsigned char)2,                             //num_of_vrings, only two is supported
        { (unsigned char)0, (unsigned char)0 },             //reserved
        /* no config data */
    },
    /* the two vrings */
    {
        0,                      //da, will be populated by host, can't pass it in
        16,                     //align (bytes),
        PRU_RPMSG_VQ0_SIZE,     //num of descriptors
        0,                      //notifyid, will be populated, can't pass right now
        0                       //reserved
    },
    {
        0,                      //da, will be populated by host, can't pass it in
        16,                     //align (bytes),
        PRU_RPMSG_VQ1_SIZE,     //num of descriptors
        0,                      //notifyid, will be populated, can't pass right now
        0                       //reserved
    },

        {
            TYPE_CUSTOM, TYPE_PRU_INTS,
            sizeof(struct fw_rsc_custom_ints),
            { /* PRU_INTS version */
                0x0000,
                /* Channel-to-host mapping, 255 for unused */
                PRU0_FROM_ARM_CHANNEL, PRU1_FROM_ARM_CHANNEL, PRU0_TO_ARM_CHANNEL, PRU1_TO_ARM_CHANNEL, HOST_UNUSED,
                HOST_UNUSED, HOST_UNUSED, HOST_UNUSED, HOST_UNUSED, HOST_UNUSED,
                /* Number of evts being mapped to channels */
                (sizeof(pru_intc_map) / sizeof(struct ch_map)),
                /* Pointer to the structure containing mapped events */
                pru_intc_map,
            },
        },
    };

    #endif /* _RSC_TABLE_PRU_H_ */
```

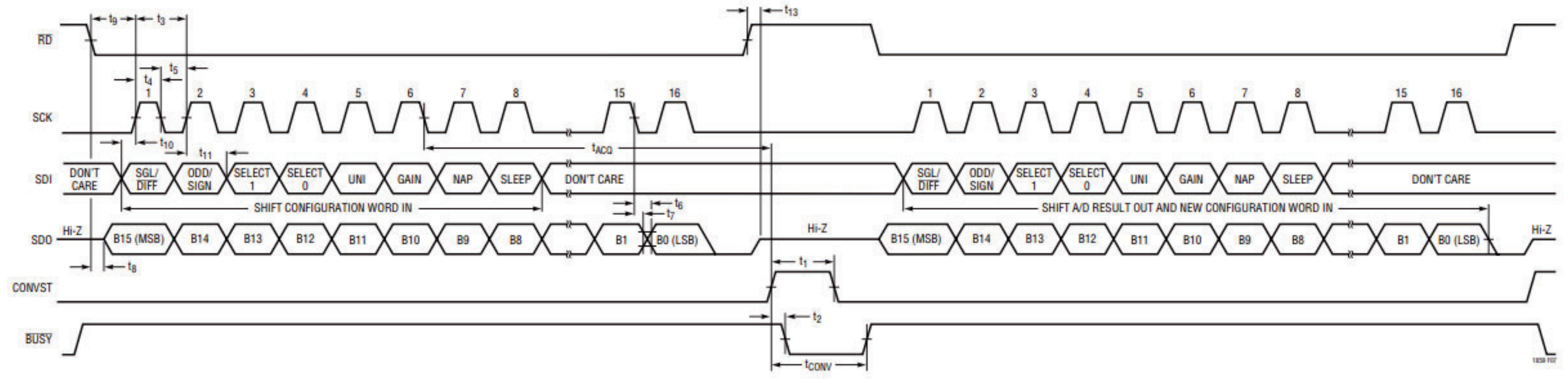
Appendix F^2 – Resource Table explanation

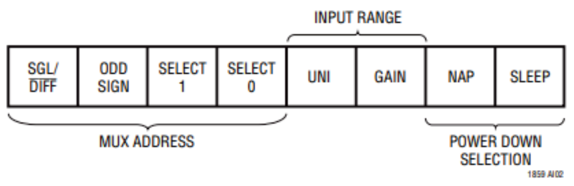
Line #	
12-13	Virtqueue sizes, can only be powers of 2
18-21	Defines required for RPMSG
24	Definiton for unused interrupt
27	System event for interrupt to ARM host = 16
28	System event for interrupt from ARM host = 17
29	Interrupt channel PRU0 to ARM host = 2
30	Interrupt channel PRU0 from ARM host = 0
31-32	Interrupt channel PRU1 from/to ARM host = UNUSED
33	Host interrupt definition (1 << 30)
34-36	Device character name
37-38	Actual channel definitions
41-43	System event -> channel mappings / Each system event-channel pair is contained in curly brackets {}
45-56	Defines all the resources requested by the resource table. (51)1 RPMSG virtual device, (52-53)2 RPMSG virtual ring buffers and (56)PRU interrupts.
61-100	Configure all the resources requested by the resource table
105-107	Channel -> host interrupt mappings / write channels in order of host interrupt priority (0-9)

Appendix G – Lock-in amplifier flowchart



Appendix H – LTC1859 operating sequence





MUX ADDRESS				SINGLE-ENDED CHANNEL SELECTION								
SGL/DIFF	ODD SIGN	SELECT 1	SELECT 0	0	1	2	3	4	5	6	7	COM
1	0	0	0	+								–
1	0	0	1			+						–
1	0	1	0					+				–
1	0	1	1							+		–
1	1	0	0		+							–
1	1	0	1				+					–
1	1	1	0						+			–
1	1	1	1								+	–

Table 2. Input Range Selection

UNI	GAIN	INPUT RANGE
0	0	±5V
1	0	0V to 5V
0	1	±10V
1	1	0V to 10V

Table 3. Power Down Selection

NAP	SLEEP	POWER DOWN MODE
0	0	Power On
1	0	Nap
X	1	Sleep